Andreas Krause, Fanny Yang

Introduction to Machine Learning

SPRING 2025

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Machine Learning
Department of Computer Science

Compilation date: **June 18, 2025**

Table of Notations

Abbreviations	
w.r.t.	with respect to
e.g.	for example
i.e.	that is
i.i.d.	independent and identically distributed
SVD	singular value decomposition
p.s.d.	positive semi-definite
p.d.	positive definite
CDF	cumulative distribution function
PMF	probability mass function
PDF	probability density function
LLN	law of large numbers
CLT	central limit theorem
GD	gradient descent
SGD	stochastic gradient descent
CV	cross-validation
ANN	artificial neural network
MLE	maximum likelihood estimation
MAP	maximum a posteriori estimation
Linear Algebra	
$A, B, C, \ldots$	matrices
$\mathbb{R}^{m \times n}$	the set of m-by-n matrices
$\boldsymbol{x}, \boldsymbol{y}, \ldots$	vectors
x_i or $x[i]$ or $[\boldsymbol{x}]_i$	the i-th component of the vector $\boldsymbol{x}$
$\boldsymbol{x}_i$	the i-th vector in a collection of vectors
$\text{span}(\mathcal{S})$	span of vectors in $\mathcal{S}$
$\text{range}(A)$	range of the matrix A
$\ker(A)$	kernel of the matrix A
$\text{rank}(A)$	rank of the matrix A
$\det(A)$	determinant of the square matrix A
$\mathbf{0}$	the zero vector
$\mathbf{0}_{m \times n}$	an $m \times n$ matrix filled with zeros
A^{-1}	inverse of the invertible matrix A
$A^\top$	transpose of A

$\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ or $\boldsymbol{u}^\top \boldsymbol{v}$	inner product of the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$
$\|\cdot\|_2$	Euclidean norm
$\|\cdot\|_p$	the p-norm
$\perp$	perpendicular
$\mathcal{S}^\perp$	orthogonal complement of $\mathcal{S}$
δ_{ij}	Kronecker delta function
$\sigma_i(A)$	the i-th largest singular value of A
$\lambda_i(A)$	the i-th largest eigenvalue of A
$x \mapsto f(x)$	description of a function that maps x to $f(x)$
I	Identity matrix
$A \succ 0$	Matrix A is p.d.
$A \succeq 0$	Matrix A is p.s.d.
$A^\dagger$	Moore-Penrose pseudo-inverse
$\mathrm{tr}(A)$	trace of the square matrix A
Π_X	projection matrix onto range(X)

Analysis

$Df(\boldsymbol{x})$	derivative of f at the point $\boldsymbol{x}$
$Df(\boldsymbol{x})[\boldsymbol{v}]$	derivative of f at the point $\boldsymbol{x}$ applied to $\boldsymbol{v}$
$\frac{\partial f}{\partial x_i}$	partial derivative of f with respect to x_i
$\nabla f(\boldsymbol{x})$	gradient of f at the point $\boldsymbol{x}$
$D^2 f(\boldsymbol{x})$	Hessian of f at the point $\boldsymbol{x}$
$\nabla^2 f(\boldsymbol{x})$	Another notation for Hessian of f at the point $\boldsymbol{x}$
$f \circ g$	composition of f and g
$o(\cdot)$	little-o notation
$\mathcal{O}(\cdot)$	big-O notation

Probability

Ω	sample space
$\mathcal{F}$	family of events
$\mathbb{P}$	probability function
$(\Omega, \mathcal{F}, \mathbb{P})$	probability space
X, Y	random variables
$\mathbb{P}_X$	distribution/law of the random variable X
F_X	cumulative distribution function of the random variable X
p_X	probability mass or probability density function of the random variable X
$\mathbb{E}[X]$	expected value of X
$\mathbb{E}[X \mid Y = y]$	conditional expectation of X given $Y = y$
$\mathbb{E}[X \mid Y]$	conditional expectation of X given Y
$\mathbb{E}_X[f(X, Y)]$	expectation of f w.r.t. the randomness of X
$\mathbb{E}_{X\|y}[f(X, Y)]$	expectation of f w.r.t. the conditional distribution of $X \mid Y = y$
$\mathrm{Var}(X)$	variance of X
$\mathrm{Var}_X(f(X, Y))$	variance of f w.r.t. the randomness of X
$\mathrm{Var}_{X\|y}(f(X, Y))$	variance of f w.r.t. the conditional distribution of $X \mid Y = y$

$\mathrm{Cov}(X, Y)$	covariance of X and Y
$\mathcal{N}(\mu, \sigma)$	normal distribution with mean μ and variance σ^2
$\mathrm{Unif}(S)$	uniform distribution on the set S
$\mathrm{Exp}(\lambda)$	exponential distribution with parameter λ
$\mathrm{Ber}(q)$	Bernoulli distribution with parameter q
$\mathrm{Cat}(p_1, p_2, \ldots, p_k)$	categorical distribution with parameters $p_1, p_2, \ldots, p_k$
$\mathrm{Binom}(q, n)$	binomial distribution with parameters q and n
$\mathrm{Poisson}(\lambda)$	Poisson distribution with parameter λ

Linear Regression

$\boldsymbol{x}$	feature vector
y	target/label/output
$\boldsymbol{X}$	feature matrix
F	function class
$f(\boldsymbol{x})$	predictor function evaluated at $\boldsymbol{x}$
$f_{\boldsymbol{w}}$	the linear function corresponding to the vector $\boldsymbol{w}$
$\ell(f(\boldsymbol{x}), y)$	loss at observation $(\boldsymbol{x}, y)$ under predictor f
$\boldsymbol{w}$	weight vector
w_0	bias term/intercept
$L(f)$	average training loss of the predictor f
$\hat{\boldsymbol{w}}$	minimizer of training loss
$\phi(\boldsymbol{x})$	feature map of $\boldsymbol{x}$
Φ	feature matrix with feature maps as rows

Optimization

$\boldsymbol{w}^t$	estimate at iteration t of an algorithm
η, η_t	step size
$\|A\|_{\mathrm{op}}$	operator norm of A (the largest singular value)
$\lambda_{\max}, \lambda_{\min}$	largest and smallest eigenvalue
κ	condition number
ξ	remainder term in Taylor expansion

Model Selection and Regularization

f^*	ground truth function
$\widehat{f}_{\mathcal{D}}$	the *model* (scientist's estimator for f^*) based on dataset $\mathcal{D}$
$\widehat{f}$	general notation for a model
$L(f; \mathcal{D})$	training error of function f on dataset $\mathcal{D}$
$L\left(\widehat{f}; \mathbb{P}_{X,Y}\right)$	generalization error of function f
$\mathbb{P}$	joint probability distribution of (X, Y) on the space $\mathcal{X} \times \mathcal{Y}$

Classification

$\mathcal{X}$	space of data points (usually $\mathbb{R}^d$)
$\mathcal{Y}$	set of all possible labels/classes
$d(\boldsymbol{z}, H)$	signed distance of the point $\boldsymbol{z}$ from hyperplane H

Neural Networks

$\boldsymbol{\Theta}$	parameters of a neural network
$f(\boldsymbol{x}; \boldsymbol{\Theta})$	output of a NN with parameters $\boldsymbol{\Theta}$ at point $\boldsymbol{x}$
$\boldsymbol{W}^{(l)}$	weight matrix of the l-th layer
$\boldsymbol{b}^{(l)}$	bias vector of the l-th layer
$\boldsymbol{z}^{(l)}$	pre-activations of the l-th layer
$\boldsymbol{h}^{(l)}$	(post-activation) hidden units of the l-th layer
$\varphi(\cdot)$	(non-linear) activation function
$L(\boldsymbol{\Theta}; \mathcal{D})$	average training loss of a NN with parameters $\boldsymbol{\Theta}$ on dataset $\mathcal{D}$
$l(\boldsymbol{\Theta}; \boldsymbol{x}_i, y_i)$	loss of a NN with parameters $\boldsymbol{\Theta}$ at the point $(\boldsymbol{x}_i, y_i)$
$\nabla_{\boldsymbol{W}} l(\boldsymbol{\Theta}; \boldsymbol{x}_i, y_i)$	gradient of the training loss w.r.t the weights $\boldsymbol{W}$

Probabilistic Modeling & Inference

Θ	space of parameters for a parametric model
$\mathbb{P}_{X;\theta}$	parameterized distribution of X with parameter θ
$\{\mathbb{P}_{X;\theta}\}_{\theta \in \Theta}$	parameterized family of distributions of X with parameter θ
$\mathbb{P}_X(\cdot; \theta)$	probability function of X with parameter θ
$F_X(\cdot; \theta)$	CDF of X with parameter θ
$p_X(\cdot; \theta)$	PMF/PDF of X with parameter θ
$\mathbb{P}_X^*$	true distribution of X
$\widehat{\mathbb{P}}_X$	estimated distribution of X

Contributors

This set of notes is the result of the efforts of the following contributors:

- The contents of the *Preliminaries* are an accumulation of the "Math Recap" tutorials throughout the last years, and is written, expanded, and refined by Riccardo Zuliani, Tobias Wegel, and Mohammad Reza Karimi.
- Chapter 4 on *Linear Regression* is based on the notes written by Seyedmorteza 2021 and is edited, expanded, and refined by Kai Lion, Mohammad Reza Karimi, Julia Kostin and Fanny Yang.
- Chapter 5 *Optimization* is written by Gizem Yüce, Kai Lion, and Mohammad Reza Karimi and edited and refined by Tobias Wegel and Fanny Yang.
- Chapter 6 on *Model Selection* is written by Georgios Gavrilopoulos and expanded, edited and refined by Fanny Yang, Julia Kostin and Xinyu Sun.
- Chapter 7 on *Bias-Variance tradeoff and regularization* is written by Georgios Gavrilopoulos and Daniel Yang, and are expanded, edited, and refined by Fanny Yang, Tobias Wegel and Xinyu Sun.
- Chapter 8 on *Classification* is written by Georgios Gavrilopoulos, Tobias Wegel and Julia Kostin and reviewed and edited by Fanny Yang.
- Chapter 9 on *Kernels* is written by Julia Kostin, Tobias Wegel, Kirill Brilliantov and reviewed and edited by Fanny Yang.
- Chapter 10 on *Neural Networks* is written by Georgios Gavrilopoulos and is reviewed and edited by Harun Mustafa, Xinyu Sun, Andreas Krause, Julia Kostin, Tobias Wegel, Alex Shevchenko, Dimitri von Rütte, and Sidak Pal Singh.
- Chapter 11 and Chapter 12 on *Principal Component Analysis* and *Clustering* is written by Giorgio Racca and Thomas Out and is reviewed and edited by Harun Mustafa and Andreas Krause.

- Chapter 13 on *Probabilistic modeling and inference* is written by Daniel Yang and is reviewed and edited by Julia Kostin, Tobias Wegel and Fanny Yang.
- Chapter 14 on *Gaussian Mixture Models* is written by Mojmir Mutny, and Andreas Krause.

We thank Lenart Treven, Andisheh Amrollahi, Shyam Sundhar Ramesh and Lejs Behric for their valuable feedback and edits.

Contents

Part I

Preliminaries

1

Linear Algebra

In this chapter we review the most important concepts, ideas, definitions, and results of linear algebra that are needed in this course. We do not take an abstract standpoint, and only consider the Euclidean space $\mathbb{R}^n$ with its standard basis.

Roadmap

In Section 1.1 we recall the notion of linear subspace, linear independence, and span. In Section 1.2 we review some algebraic facts about matrices. Most importantly, we revisit matrix-vector multiplication, subspaces related to a matrix, and notions of rank and nullity. Section 1.3 starts with the additional structure of the Euclidean space: inner product. This is followed by notions of orthogonality and orthogonal matrices. The pinnacle of this section is the singular value decomposition, which gives a complete geometric characterization of "how a linear map works." After orthogonality, we build intuition on how to project a vector onto a given subspace in Section 1.4. We will first solve this problem for the case where the subspace is explained by a set of linearly independent vectors, and then give a complete solution for a general set of vectors in Section 1.4.1, where we introduce the notion of pseudo-inverse. We then review eigenvalues and eigenvectors in Section 1.5, which are vital to understand the dynamic behavior of a linear map, used in analyzing optimization algorithms. In Section 1.6, we construct the multivariate analogue of quadratic functions and describe the important notion of positive-definiteness. We will describe geometric objects such as ellipses (or hyperellipses) using these functions, and create the foundation for understanding the second order derivatives. Lastly, in Section 1.7 we introduce the notion of trace, which assigns a number to a square matrix, and is a useful tool both in derivations and understanding quadratic functions.

Learning Objectives

After reading this chapter you should know

- how to understand matrix-vector multiplication via linear combinations.
- what is rank, and how it is related to the kernel of a matrix.
- what is a p-norm and how inner product is related to different norms.
- how to check if a matrix is orthogonal.
- what is SVD, and how to interpret it geometrically.
- how to orthogonally project a vector onto a subspace spanned by a set of linearly independent vectors.

- what is pseudo-inverse, how to compute it based on SVD, and how to construct a projection matrix by it.
- what is eigen-decomposition and how it is related to SVD for symmetric matrices.
- what is a p.s.d. matrix, how to interpret positive-definiteness geometrically, and different ways to verify if a matrix is positive.
- how to create (hyper-)ellipses in multiple dimensions via p.s.d. matrices.
- what is trace, and how is it related to eigenvalues.
- the trace trick.

1.1 Vector Space Notions

Definition 1.1 (Linear subspace)**.** A nonempty set $\mathcal{V} \subseteq \mathbb{R}^n$ is a linear subspace of $\mathbb{R}^n$ if for any two vectors $\boldsymbol{u}_1, \boldsymbol{u}_2 \in \mathcal{V}$ and two scalars $\alpha, \beta \in \mathbb{R}$, it holds that $\alpha\boldsymbol{u}_1 + \beta\boldsymbol{u}_2 \in \mathcal{V}$.

Definition 1.2 (Linear independence, Dimension)**.** A set of vectors $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ in $\mathbb{R}^n$ are linearly independent if no non-trivial linear combination of them is the zero vector. That is, if

$$\sum_{i=1}^{k} c_i \boldsymbol{v}_i = \boldsymbol{0},$$

then $c_1 = \cdots = c_k = 0$. The *dimension* of a linear subspace $\mathcal{V}$ is the size of a maximal linearly independent subset of $\mathcal{V}$.

Definition 1.3 (Span)**.** The *span* of a set of vectors $\mathcal{S} = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_k\}$, with $\boldsymbol{a}_i \in \mathbb{R}^n$ for $i = 1, \ldots, k$, is the set of all possible linear combinations of them:

$$\operatorname{span}(\mathcal{S}) = \left\{ \sum_{i=1}^{k} c_i \boldsymbol{a}_i : c_i \in \mathbb{R}, \quad i = 1, \ldots, k \right\}.$$

Notice that the span of a set of vectors is a linear subspace, and, if $\boldsymbol{a}_i$ are linearly independent, their span has dimension k.

1.2 Matrix algebra

A matrix $A \in \mathbb{R}^{m \times n}$ defines a *linear map* between the Euclidean spaces $\mathbb{R}^n$ and $\mathbb{R}^m$, i.e., it maps a column vector $\boldsymbol{x} \in \mathbb{R}^n$ to the column vector $A\boldsymbol{x} \in \mathbb{R}^m$. We write x_i (or sometimes $x[i]$) to denote the i-th component of the vector $\boldsymbol{x}$.

An important observation about matrix-vector multiplication is that the vector $\boldsymbol{b} = A\boldsymbol{x}$ is a linear combination of the columns of A:

$$\boldsymbol{b} = A\boldsymbol{x} = \sum_{i=1}^{n} x_i \boldsymbol{a}_i,$$

where $\boldsymbol{a}_i \in \mathbb{R}^m$ denotes the i-th column of A. Similarly, we can understand matrix-matrix multiplication in this way. Let $B \in \mathbb{R}^{n \times p}$.

Then the matrix product $AB \in \mathbb{R}^{m\times p}$, consists of linear combinations of the columns of A:

$$\begin{bmatrix} & A & \end{bmatrix} \underbrace{\begin{bmatrix} | & & | \\ \boldsymbol{b}_1 & \cdots & \boldsymbol{b}_p \\ | & & | \end{bmatrix}}_{B} = \begin{bmatrix} | & & | \\ A\boldsymbol{b}_1 & \cdots & A\boldsymbol{b}_p \\ | & & | \end{bmatrix}.$$

Example 1.4 (Outer Product)**.** Given two vectors $\boldsymbol{u} \in \mathbb{R}^m$ and $\boldsymbol{v} \in \mathbb{R}^n$, their outer product is the $m \times n$ matrix $\boldsymbol{u}\boldsymbol{v}^\top$, which contains repeated copies of $\boldsymbol{u}$ multiplied by the elements of $\boldsymbol{v}$:

$$\boldsymbol{u}\boldsymbol{v}^\top = \begin{bmatrix} | & & | \\ v_1\boldsymbol{u} & \dots & v_n\boldsymbol{u} \\ | & & | \end{bmatrix} \in \mathbb{R}^{m\times n}.$$

Notice that here, we interpret a vector in $\mathbb{R}^n$ as an $n \times 1$ matrix.

Definition 1.5 (Image, Kernel, Rank)**.** The *image* or *range* of a matrix A, denoted as $\text{range}(A)$, is the span of its columns. The *kernel* or *null space* of a matrix $A \in \mathbb{R}^{m\times n}$ is

$$\ker(A) = \{\boldsymbol{x} \in \mathbb{R}^n : A\boldsymbol{x} = \boldsymbol{0}\}.$$

Both range and kernel are linear subspaces. The *rank* of a matrix is the dimension of its range.

Example 1.6 (Rank of some small matrices)**.** Let the matrices A and B be defined as

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 \\ 0.5 & 1 \end{bmatrix}.$$

Then $\text{rank}(A) = 2$ and $\text{rank}(B) = 1$. For any two nonzero vectors, one can verify that their outer product is always a matrix of rank one.

The kernel and image are tied together via the following theorem:

Theorem 1.7 (Rank-Nullity)**.** *Let $A \in \mathbb{R}^{m\times n}$, then*

$$\dim(\ker(A)) + \text{rank}(A) = n.$$

Definition 1.8 (Determinant)**.** The determinant of an $n \times n$ matrix A is defined as the (signed) volume of the parallelepiped made out of the columns of A.[1]

If the columns of A are linearly dependent, the parallelepiped is degenerate and has no volume, hence, the determinant is zero.

We state the following facts without proof. The reader who is not familiar with these facts is urged to prove them, or consult any textbook on linear algebra, e.g., Trefethen and Bau 1997.

- $\text{rank}(A) = \text{rank}(A^\top)$. Hence, for a matrix $A \in \mathbb{R}^{m\times n}$, $\text{rank}(A) \leq \min\{m, n\}$. We say A has *full rank* if $\text{rank}(A) = \min\{m, n\}$.
- The linear map defined by $A \in \mathbb{R}^{m\times n}$ is *injective* (i.e., $A\boldsymbol{x} = A\boldsymbol{y}$ only if $\boldsymbol{x} = \boldsymbol{y}$) if and only if $\ker(A) = \{\boldsymbol{0}\}$.

[1] A computational definition of the determinant of a matrix is not needed in this course. However, it should be clear by this definition that

$$\det\left(\begin{bmatrix} a_1 & & 0 \\ & \ddots & \\ 0 & & a_n \end{bmatrix}\right) = a_1 a_2 \cdots a_n.$$

- The linear map defined by $A \in \mathbb{R}^{m\times n}$ is *surjective* (i.e., $\text{range}(A) = \mathbb{R}^m$) if and only if $\text{rank}(A) = m$.
- The linear map defined by A is *bijective* (i.e., it is surjective and injective) if and only if one of the following equivalent conditions are true:
  - A is square and $\ker(A) = \{\mathbf{0}\}$;
  - A is square and $\text{rank}(A) = m$;
  - A is square and $\det(A) \neq 0$.

Definition 1.9 (Invertible matrix)**.** A square matrix $A \in \mathbb{R}^{n\times n}$ is *invertible* (or *nonsingular*) if there exists a square matrix $B \in \mathbb{R}^{n\times n}$ such that

$$AB = BA = I.$$

Here, I is the identity matrix. If this is the case, then B is the (unique) *inverse* of A, and it is denoted by $B = A^{-1}$. Equivalently, A square matrix is invertible if and only if the linear map defined by it is bijective. A non-invertible matrix is said to be *singular*.

Bonus Material

Let $A \in \mathbb{R}^{n\times n}$ be a square invertible matrix and let $\boldsymbol{b} = (b_1, \ldots, b_n) \in \mathbb{R}^n$ be arbitrary. As A is invertible, $\boldsymbol{b}$ is in the range of A, and hence, can be expressed as a linear combination of columns of A:

$$\boldsymbol{b} = \sum_{i=1}^{n} c_i \boldsymbol{a}_i.$$

Hence, we can numerically represent the same vector by either b_i (in the Euclidean basis) or c_i (in the basis formed by columns of A). The coordinates b_i and c_i are related to each other via A and A^{-1}. That is, if b_i are given, we can obtain c_i by a matrix multiplication by A^{-1}:

$$\begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} = A^{-1} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix},$$

and, if c_i are given, we can get b_i by multiplying by A. Thus, A^{-1} can be viewed as a change of basis operator, converting the representation of a vector in Euclidean basis into the representation in the basis of columns of A.

1.3 Geometric Constructs

Definition 1.10 (Euclidean inner product and norm)**.** The *Euclidean inner product* of two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ is defined as

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \boldsymbol{u}^\top \boldsymbol{v} = \sum_{i=1}^{n} u_i v_i,$$

where $u_i, v_i \in \mathbb{R}$ are the i-th component of $\boldsymbol{u}$ and $\boldsymbol{v}$ respectively. The *Euclidean norm* of a vector $\boldsymbol{v} \in \mathbb{R}^n$ is defined as

$$\|\boldsymbol{v}\|_2 = \sqrt{\langle \boldsymbol{v}, \boldsymbol{v} \rangle}.$$

We will sometimes drop the subscript 2 when referring to the Euclidean norm and write $\|\cdot\|$ instead of $\|\cdot\|_2$.

The Euclidean inner product can be used to define the angle between vectors:

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \|\boldsymbol{v}\| \, \|\boldsymbol{u}\| \cos \angle(\boldsymbol{u}, \boldsymbol{v}),$$

where $\angle(\boldsymbol{u}, \boldsymbol{v})$ is the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$. See the Cauchy-Schwarz inequality (Theorem 1.12) below.

Remark. Recall that in Section 1.2, we observed that matrix-vector multiplication can be seen as taking a linear combination. Here we present another point of view. Let $A \in \mathbb{R}^{m \times n}$ and $\boldsymbol{x} \in \mathbb{R}^n$, and denote by $\boldsymbol{a}^i \in \mathbb{R}^n$ the ith row of A. Then

$$A\boldsymbol{x} = \begin{bmatrix} \langle \boldsymbol{a}^1, \boldsymbol{x} \rangle \\ \vdots \\ \langle \boldsymbol{a}^m, \boldsymbol{x} \rangle \end{bmatrix}.$$

When facing with a matrix-vector product, sometimes this understanding is useful and sometimes the former.

There are norms other than the Euclidean norm,[2] that have different geometric meanings and are interesting for us in different occasions. A rich class of norms, that include the Euclidean norm as a special case, is the class of p-norms.

[2] In general, a *norm* is any function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ that satisfies these properties:

- $\|\boldsymbol{x}\| \geq 0$ and $\|\boldsymbol{x}\| = 0$ iff $\boldsymbol{x} = \boldsymbol{0}$.
- $\|\alpha \boldsymbol{x}\| = |\alpha| \, \|\boldsymbol{x}\|$ for all $\alpha \in \mathbb{R}$.
- $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$.

Definition 1.11 (p-norm)**.** Let $p \in [1, \infty)$ be a real number. The p-norm of a vector $\boldsymbol{x} \in \mathbb{R}^n$ is defined as

$$\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p},$$

where x_i is the i-th entry of $\boldsymbol{x}$. The *infinity norm* is also defined as

$$\|\boldsymbol{x}\|_\infty = \max_{i=1,\ldots,n} |x_i|.$$

Figure 1.1 shows the *unit-norm balls* in $\mathbb{R}^2$, i.e., the sets $\{\boldsymbol{x} \in \mathbb{R}^2 : \|\boldsymbol{x}\|_p \leq 1\}$, for different values of p.

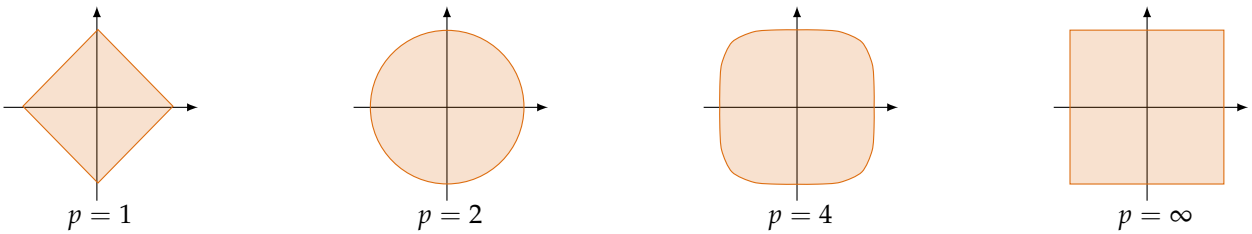


Figure 1.1: Unit-norm balls in $\mathbb{R}^2$ for different p-norms.

Theorem 1.12 (Hölder and Cauchy-Schwarz inequalities)**.** *Let $p, q \in [1, \infty]$ with $1/p + 1/q = 1$. Then for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ we have that*

$$|\langle \boldsymbol{x}, \boldsymbol{y} \rangle| \leq \|\boldsymbol{x}\|_p \, \|\boldsymbol{y}\|_q. \tag{1.1}$$

For $q = p = 2$, this inequality is known as Cauchy-Schwarz inequality.

We now move back to the Euclidean space with its usual norm and inner product. First, we deal with the notion of orthogonality, which is central to the discussions that follows.

Definition 1.13 (Orthogonality)**.** Two vectors $\boldsymbol{v}, \boldsymbol{u} \in \mathbb{R}^n$ are *orthogonal*, denoted by $\boldsymbol{v} \perp \boldsymbol{u}$, if $\langle \boldsymbol{v}, \boldsymbol{u} \rangle = 0$.

If two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ are orthogonal, then

$$\begin{aligned} \|\boldsymbol{x}+\boldsymbol{y}\|^2 &= \langle \boldsymbol{x}+\boldsymbol{y}, \boldsymbol{x}+\boldsymbol{y} \rangle \\ &= \langle \boldsymbol{x}, \boldsymbol{x} \rangle + \langle \boldsymbol{x}, \boldsymbol{y} \rangle + \langle \boldsymbol{y}, \boldsymbol{x} \rangle + \langle \boldsymbol{y}, \boldsymbol{y} \rangle \\ &= \langle \boldsymbol{x}, \boldsymbol{x} \rangle + \langle \boldsymbol{y}, \boldsymbol{y} \rangle \\ &= \|\boldsymbol{x}\|^2 + \|\boldsymbol{y}\|^2 . \end{aligned}$$

The reader can confirm that this is indeed the *Pythagoras theorem*.

A subspace $\mathcal{S} \subset \mathbb{R}^n$ is orthogonal to subspace $\mathcal{R} \subset \mathbb{R}^n$ if every vector in $\mathcal{S}$ is orthogonal to every vector in $\mathcal{R}$.

Definition 1.14 (Orthogonal complement)**.** Let $\mathcal{S}$ be a subspace of $\mathbb{R}^n$. The orthogonal complement of $\mathcal{S}$ is the set $\mathcal{S}^\perp$ whose elements are orthogonal to every element of $\mathcal{S}$:

$$\mathcal{S}^\perp = \{\boldsymbol{x} \in \mathbb{R}^n \mid \langle \boldsymbol{x}, \boldsymbol{y} \rangle = 0, \text{ for all } \boldsymbol{y} \in \mathcal{S}\}.$$

Exercise 1.15. Prove that the orthogonal complement of any set is a linear subspace.

Definition 1.16 (Orthonormal basis)**.** A set of vectors $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$ in $\mathbb{R}^n$ are *orthonormal* if they are pairwise orthogonal, i.e., $\langle \boldsymbol{q}_i, \boldsymbol{q}_j \rangle = 0$ for $i \neq j$, and have unit norm, i.e., $\|\boldsymbol{q}_i\| = 1$ for $i = 1, \ldots, m$. In a more compact form,

$$\langle \boldsymbol{q}_i, \boldsymbol{q}_j \rangle = \delta_{ij} := \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

A set of n orthonormal vectors in $\mathbb{R}^n$ forms a basis and is called an *orthonormal basis*.

If $\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n\}$ form an orthonormal basis of $\mathbb{R}^n$, then any vector $\boldsymbol{v} \in \mathbb{R}^n$ can be expressed as

$$\boldsymbol{v} = \sum_{i=1}^{n} \left(\boldsymbol{v}^\top \boldsymbol{q}_i \right) \boldsymbol{q}_i .$$

Remark. By changing the parenthesis, we see that the equation above can also be written as

$$\boldsymbol{v} = \sum_{i=1}^{n} \left(\boldsymbol{q}_i \boldsymbol{q}_i^\top \right) \boldsymbol{v} .$$

This new equation is very different from the previous one: each term in the sum is the application of the *matrix* $\boldsymbol{q}_i \boldsymbol{q}_i^\top$ to the vector $\boldsymbol{v}$. Recall that this matrix is the outer product of $\boldsymbol{q}_i$ with itself, and

hence is a rank 1 matrix. We will see later that this matrix is the orthogonal projection matrix (see Definition 1.23) on the direction q_i.

Definition 1.17 (Orthogonal matrix)**.** An *orthogonal matrix* U is a real square matrix whose columns are orthonormal. Equivalently, U is orthogonal iff $U^\top U = I$.

Exercise 1.18. Let U be an orthogonal matrix. Argue why $UU^\top = I$. Moreover, prove that an orthogonal matrix, preserves the Euclidean inner product and norm:

$$\langle x, y \rangle = \langle Ux, Uy \rangle .$$

and

$$\|Ux\| = \|x\| .$$

Exercise 1.19. Let $x_1, \ldots, x_n$ be vectors in $\mathbb{R}^m$. Consider the matrix $X \in \mathbb{R}^{m \times n}$ whose columns are $x_1, \ldots, x_n$.

1. Represent the elements of the matrix $X^\top X$ using the vectors $x_1, \ldots, x_n$.
2. Let $n \leq m$ and $x_1, \ldots, x_n$ be orthonormal. What does the linear map $b \mapsto XX^\top b$ do?

Solution. 1. The entries of the square matrix $X^\top X \in \mathbb{R}^{n \times n}$ correspond to the inner products of all pairs of x_i:

$$\left(X^\top X\right)_{ij} = \left\langle x_i, x_j \right\rangle .$$

2. The linear operator $XX^\top$ is an orthogonal projection matrix (see Definition 1.23) on the range of X:

$$XX^\top b = X \begin{bmatrix} \langle x_1, b \rangle \\ \vdots \\ \langle x_n, b \rangle \end{bmatrix} = \sum_{i=1}^{n} \langle x_i, b \rangle \, x_i .$$

If $m = n$ then $XX^\top b = b$. Figure 1.2 shows an example of an orthogonal projection from $\mathbb{R}^3$ to a 2-dimensional plane.

□

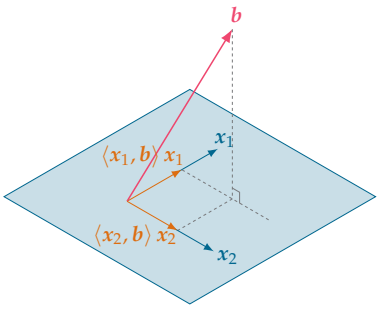


Figure 1.2: Projection of a vector $b \in \mathbb{R}^3$ on the plane spanned by $x_1, x_2 \in \mathbb{R}^3$.

Understanding what a matrix does as a linear map can be complicated. It is oftentimes useful to *decompose* the matrix into simpler building blocks. The singular value decomposition is such a tool that reveals a lot of geometric facts about the linear map.

Theorem 1.20 (Singular Value Decomposition)**.** *Any matrix $A \in \mathbb{R}^{m \times n}$ can be decomposed as the product $A = U\Sigma V^\top$, where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal, and Σ is an $m \times n$ diagonal matrix with non-negative diagonal entries. This factorization is called a* singular value decomposition (SVD) *of A.*

The diagonal entries of Σ, denoted by $\sigma_i(A) := \Sigma_{ii}$, are called the *singular values* of A, columns of V are called the *right singular vectors* of A, and columns of U are called the *left singular vectors* of A. Often, we assume that the singular values are sorted decreasingly, that is, $\sigma_1 \geq \sigma_2 \geq \cdots$. See Figure 1.3 for a schematic description of SVD. Also, it is sometimes easier to consider a *reduced* SVD, see Figure 1.4.

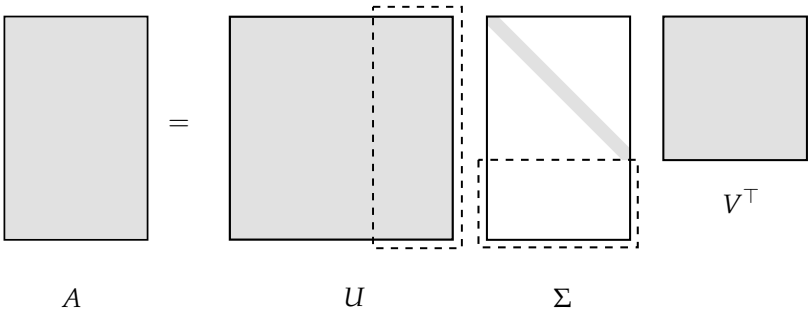


Figure 1.3: SVD of the matrix A, as defined in Theorem 1.20. Sometimes this is called the *full SVD*. The white parts are zeros, and shaded areas are the parts that can have nonzero numbers. The part of U that is marked with dashed line has no information, as it corresponds to the dashed part of Σ, which is all zeros.

The singular value decomposition has the following properties:

- The rank of A is equal to the number of nonzero singular values.
- The left singular vectors of A that correspond to nonzero singular values form an orthonormal basis (see Definition 1.16) for $\text{range}(A)$.
- The right singular vectors of A that correspond to zero singular values form an orthonormal basis for $\ker(A)$.

The relation between left and right singular vectors is simple: $A\boldsymbol{v}_i = \sigma_i \boldsymbol{u}_i$. Moreover, it is left as an exercise to see that the SVD of A can also be written as the sum

$$A = \sum_{i=1}^{r} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top, \tag{1.2}$$

where r is the number of nonzero singular values of A ($= \text{rank}(A)$).

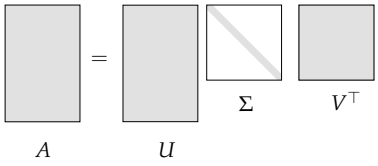


Figure 1.4: Another way to write down the SVD. In this way, Σ becomes a square matrix, but U will not be square. However, it still holds that the columns of U are orthonormal; $U^\top U = I$. This is called the *reduced SVD*.

Bonus Material

Equation 1.2 shows how we can decompose X into a sum of r rank-1 matrices (recall that the outer product $\boldsymbol{u}_i \boldsymbol{v}_i^\top$ is of rank 1). Each matrix is determined by a pair of directions (i.e., $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$) and a magnitude σ_i. If we decide to truncate the sum at a value $k < r$, then we obtain a *rank-k approximation* of the matrix A

$$A \approx \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top.$$

The error of this approximation is related to the magnitude of the remaining singular values σ_i for $i = k+1, \ldots, r$.

Remark. The SVD theorem also has a geometric interpretation:

> The image of the unit sphere under a linear transformation is always a hyperellipse.

A *hyperellipse* is generalization of a sphere, which can be obtained by stretching the unit sphere in $\mathbb{R}^m$ by some factors $\sigma_1, \ldots, \sigma_m$ is some orthogonal direction $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m \in \mathbb{R}^m$. If we choose $\|\boldsymbol{u}_i\| = 1$, then the vectors $\sigma_i \boldsymbol{u}_i$ are the *principal semiaxes* of the hyperellipse.

If we apply the linear mapping defined by $A \in \mathbb{R}^{m \times n}$ to the unit

sphere of $\mathbb{R}^n$, the SVD theorem tells us that the result is a hyperellipse. Moreover, the left singular vectors $\boldsymbol{u}_i$ of A are the directions of the principal semiaxes of the hyperellipse and the associated singular values σ_i are the magnitudes of the corresponding semiaxes (see Figure 1.5 for an example where $n = m = 2$).

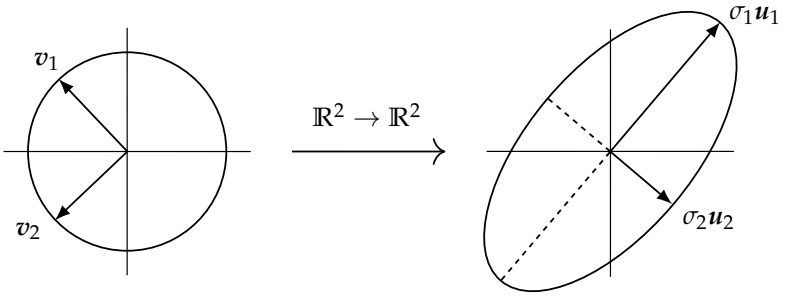


Figure 1.5: Image of the unit sphere under a linear transformation from $\mathbb{R}^2$ to $\mathbb{R}^2$.

1.4 Projection Matrices

Definition 1.21 (Projection matrix)**.** A *projection matrix* is a square matrix P that satisfies $P^2 = P$.

The definition is equivalent to saying that a projection sends a vector in its range to itself. One can think about a general projection as the shadow of an object on a plane, when the sun (or a parallel light source) is shining with an angle. Notice that the definition does not necessarily imply that P projects points orthogonally. The direction of the projection can be easily deduced by drawing the line that connects $\boldsymbol{v}$ to $P\boldsymbol{v}$, i.e., the vector $P\boldsymbol{v} - \boldsymbol{v}$ (see Figure 1.6). By applying the projection matrix P to this vector, we get

$$P(P\boldsymbol{v} - \boldsymbol{v}) = P^2\boldsymbol{v} - P\boldsymbol{v} = \boldsymbol{0},$$

which means that $P\boldsymbol{v} - \boldsymbol{v} \in \ker(P)$. This shows that the direction of the projection is always described by a vector in $\ker(P)$.

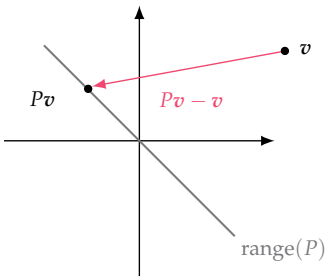


Figure 1.6: Projection of the vector $\boldsymbol{v}$ using the projection matrix P.

Exercise 1.22. Let $P \in \mathbb{R}^{n\times n}$ be a projection matrix. Then the *complementary projection* $Q \in \mathbb{R}^{n\times n}$ of P is given by

$$Q = I - P. \tag{1.3}$$

First, prove that Q is a projection matrix. Then, show that $\text{range}(Q) = \ker(P)$. With a similar argument, show that $\ker(Q) = \text{range}(P)$.

Solution. Notice that $(I - P)^2 = I - 2P + P^2 = I - 2P + P = I - P$. Hence, Q is a projection matrix. Consider a vector $\boldsymbol{u} \in \ker(P)$. Then

$$(I - P)\boldsymbol{u} = \boldsymbol{u} - P\boldsymbol{u} = \boldsymbol{u};$$

therefore, any vector in $\ker(P)$ is also in $\text{range}(I - P)$ and

$$\ker(P) \subseteq \text{range}(I - P). \tag{1.4}$$

Next, let $\boldsymbol{x} \in \text{range}(I - P)$, that is

$$\boldsymbol{x} = (I - P)\boldsymbol{v} = \boldsymbol{v} - P\boldsymbol{v}. \tag{1.5}$$

for some v. Applying the projection matrix operator to x and using the equivalence in Equation 1.5 we have

$$Px = Pv - P^2v = Pv - Pv = 0,$$

where we have used the fact that $P^2 = P$. We can conclude that $x \in \ker(P)$ and that

$$\text{range}(I - P) \subseteq \ker(P). \tag{1.6}$$

Combining Equation 1.4 with Equation 1.6 we obtain

$$\text{range}(I - P) = \ker(P). \qquad \square$$

Definition 1.23 (Orthogonal projection)**.** A projection matrix is an *orthogonal projection* if its range and kernel are orthogonal (see Figure 1.7). We usually write Π for orthogonal projection matrices.

Let $\mathcal{V}$ be a linear subspace of $\mathbb{R}^n$. Then we know that any $x \in \mathbb{R}^n$ can be uniquely decomposed as $x = x_{\mathcal{V}} + x_{\mathcal{V}^\perp}$, where $x_{\mathcal{V}} \in \mathcal{V}$, $x_{\mathcal{V}^\perp} \in \mathcal{V}^\perp$.[3] The (linear) map that takes x and outputs $x_{\mathcal{V}}$ is indeed an orthogonal projection (why?), and we denote it by $\Pi_{\mathcal{V}}$.

Definition 1.24 (Orthogonal projection onto a subspace)**.** Let $\mathcal{V}$ be a linear subspace of $\mathbb{R}^n$. The orthogonal projection $\Pi_{\mathcal{V}}$ whose range is $\mathcal{V}$ is called the *orthogonal projection onto* $\mathcal{V}$, and satisfies

$$\Pi_{\mathcal{V}}(x) = x_{\mathcal{V}}.$$

For a matrix X, we abuse the notation and write Π_X to denote $\Pi_{\text{range}(X)}$. This is indeed the orthogonal projection matrix on the span of columns of X.

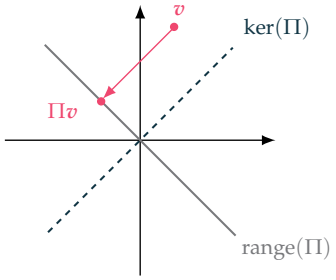


Figure 1.7: Projection of a vector v using the orthogonal projection Π.

[3] This is the so-called *orthogonal decomposition* of x.

Exercise 1.25. Show that a projection matrix Π is orthogonal if and only if Π is symmetric, that is, $\Pi = \Pi^\top$.

Solution. We only prove the "$\Leftarrow$" part, and leave the other direction to the reader.

Assume Π is a projection matrix with $\Pi = \Pi^\top$. Let us take two arbitrary vectors $x, y \in \mathbb{R}^n$ in the range and kernel of Π respectively. Our goal is to prove that $x \perp y$. We can write $x = \Pi v$ and $y = (I - \Pi)w$ for some $v, w \in \mathbb{R}^n$ (see Exercise 1.22). We then have:

$$\begin{aligned}
\langle x, y \rangle &= \langle \Pi v, (I - \Pi)w \rangle \\
&= v^\top \Pi^\top (I - \Pi)w \\
&= v^\top \Pi (I - \Pi)w && \text{as } \Pi \text{ is symmetric} \\
&= v^\top (\Pi - \Pi^2)w \\
&= 0 && \text{as } \Pi \text{ is a projection.}
\end{aligned}$$

We conclude that $\ker(\Pi) \perp \text{range}(\Pi)$ and hence, Π is an orthogonal projection. $\square$

Exercise 1.26. Consider n linearly independent vectors $a_1, \ldots, a_n \in \mathbb{R}^m$ ($n \leq m$). What is the orthogonal projection matrix onto $\text{span}(\{a_1, \ldots, a_n\})$?

Solution. Consider the matrix $A \in \mathbb{R}^{m \times n}$ whose columns are given by $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. It is clear that $\text{range}(A) = \text{span}(\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\})$, and as $\boldsymbol{a}_i$ are linearly independent, A is full-rank. Assume that $\boldsymbol{y} \in \mathbb{R}^m$ is the orthogonal projection of $\boldsymbol{v}$ on $\text{range}(A)$. Of course $\boldsymbol{y} \in \text{range}(A)$, thus, we can express $\boldsymbol{y}$ as $\boldsymbol{y} = A\boldsymbol{x}$ for some $\boldsymbol{x} \in \mathbb{R}^n$. As the projection is orthogonal,

$$\boldsymbol{y} - \boldsymbol{v} \perp \text{range}(A),$$

or equivalently

$$\begin{aligned}
\boldsymbol{y} - \boldsymbol{v} \perp \boldsymbol{a}_i\,, \quad \forall\, i \quad &\Leftrightarrow \quad \boldsymbol{a}_i^\top (A\boldsymbol{x} - \boldsymbol{v}) = 0\,, \quad \forall\, i \\
&\Leftrightarrow \quad \underbrace{\begin{bmatrix} — & \boldsymbol{a}_1^\top & — \\ & \vdots & \\ — & \boldsymbol{a}_n^\top & — \end{bmatrix}}_{A^\top} (A\boldsymbol{x} - \boldsymbol{v}) = 0 \\
&\Leftrightarrow \quad A^\top (A\boldsymbol{x} - \boldsymbol{v}) = 0 \\
&\Leftrightarrow \quad A^\top A\boldsymbol{x} = A^\top \boldsymbol{v}
\end{aligned}$$

It is easy to see that $A^\top A$ is invertible.[4] Using this fact we can conclude that

$$x = (A^\top A)^{-1} A^\top \boldsymbol{v} \implies \boldsymbol{y} = A\boldsymbol{x} = A(A^\top A)^{-1} A^\top \boldsymbol{v}.$$

Thus, the projection matrix is $A(A^\top A)^{-1} A^\top$. □

To summarize, we showed in Exercise 1.26 that if A is full-rank,

$$\Pi_A = A(A^\top A)^{-1} A^\top.$$

Notice that if the columns of A were orthonormal, i.e., $A^\top A = I$, then $\Pi_A = AA^\top$. This is the same result that we had in Exercise 1.19.

Finally, an important property of orthogonal projections is that the projection $\Pi_{\mathcal{V}}(\boldsymbol{x})$ of some point $\boldsymbol{x}$ onto a linear subspace $\mathcal{V} \subseteq \mathbb{R}^n$ is the point in $\mathcal{V}$ that is closest to $\boldsymbol{x}$ in Euclidean norm, as specified in Exercise 1.27.

Exercise 1.27. Let $\mathcal{V} \subseteq \mathbb{R}^n$ be a linear subspace. Show that for any $\boldsymbol{x} \in \mathbb{R}^n$ it holds that

$$\Pi_{\mathcal{V}}(\boldsymbol{x}) = \arg\min_{\boldsymbol{v} \in \mathcal{V}} \|\boldsymbol{x} - \boldsymbol{v}\|_2\,.$$

Solution. Let $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{v} \in \mathcal{V}$. By definition, $\boldsymbol{x} - \Pi_{\mathcal{V}}(\boldsymbol{x})$ is orthogonal to any vector in $\mathcal{V}$, and hence it is also orthogonal to $\boldsymbol{v} - \Pi_{\mathcal{V}}(\boldsymbol{x})$ that is also an element in $\mathcal{V}$ by linearity. Therefore, the squared Euclidean distance between $\boldsymbol{x}$ and $\boldsymbol{v}$ can be decomposed into

$$\|\boldsymbol{x} - \boldsymbol{v}\|_2^2 = \|\boldsymbol{x} - \Pi_{\mathcal{V}}(\boldsymbol{x})\|_2^2 + \|\Pi_{\mathcal{V}}(\boldsymbol{x}) - \boldsymbol{v}\|_2^2$$

by the Pythagorean Theorem. Minimizing the right-hand side with respect to $\boldsymbol{v} \in \mathcal{V}$ is achieved by $v^* = \Pi_{\mathcal{V}}(\boldsymbol{x})$, as the first term is independent of $\boldsymbol{v}$ and the second term is non-negative. This proves the claim. □

Combining Exercise 1.27 with Exercise 1.26 will become vital in Chapter 4 for Theorem 4.1.

[4] Let $A = U\Sigma V^\top$ be the SVD of A. Notice that since $n \leq m$,

$$\Sigma = \left[\begin{array}{ccc} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ \hline & \boldsymbol{0}_{(m-n)\times n} & \end{array}\right] \in \mathbb{R}^{m \times n}$$

is a tall matrix, where $\boldsymbol{0}_{(m-n)\times n} \in \mathbb{R}^{(m-n)\times n}$ is a matrix full of zeros. Moreover, since A is full-rank, $\sigma_i > 0$. We therefore have that

$$\begin{aligned}
A^\top A &= V\Sigma^\top U^\top U\Sigma V^\top \\
&= V\Sigma^\top \Sigma V^\top \\
&= V \begin{bmatrix} \sigma_1^2 & & \\ & \ddots & \\ & & \sigma_n^2 \end{bmatrix} V^\top,
\end{aligned}$$

which is an invertible matrix.

1.4.1 Pseudo-Inverse

A useful construct related to orthogonal projections is the Moore-Penrose pseudo-inverse. Conceptually, the pseudo-inverse can be seen as a generalization of *inverse* for arbitrary matrices. But we will see that the pseudo-inverse of a matrix is closely related to the orthogonal projection on the range of that matrix. We first bring a formal definition of what properties should the pseudo-inverse satisfy, and then we give a computationally friendly way to describe it.

Definition 1.28 (Pseudo-inverse)**.** Let A be a real matrix. The pseudo-inverse of A is a matrix $A^\dagger$ satisfying the following properties:

$$\begin{aligned} &(i) && AA^\dagger A = A, \\ &(ii) && A^\dagger AA^\dagger = A^\dagger, \\ &(iii) && (AA^\dagger)^\top = AA^\dagger, \\ &(iii) && (A^\dagger A)^\top = A^\dagger A. \end{aligned}$$

It turns out that the pseudo-inverse always exists and is unique.

Theorem 1.29. *The pseudo-inverse $A^\dagger$ of A also satisfies:*

$$\begin{aligned} &(iv) && (A^\top)^\dagger = (A^\dagger)^\top, \\ &(v) && (AA^\top)^\dagger = (A^\dagger)^\top A^\dagger, \\ &(vi) && A^\dagger \boldsymbol{x} = 0 \;\Leftrightarrow\; \boldsymbol{x}^\top A = 0 \;\Leftrightarrow\; A^\top \boldsymbol{x} = 0. \end{aligned}$$

The pseudo-inverse can be computed via SVD easily:

Theorem 1.30. *The pseudo-inverse of a real matrix $A \in \mathbb{R}^{m\times n}$ with SVD $A = U\Sigma V^\top$ is the matrix*

$$A^\dagger = V\Sigma^\dagger U^\top \in \mathbb{R}^{n\times m},$$

where $\Sigma^\dagger$ is given by

$$\Sigma^\dagger = \left[\begin{array}{ccc|c} \sigma_1^{-1} & & & \\ & \ddots & & \mathbf{0}_{r\times(m-r)} \\ & & \sigma_r^{-1} & \\ \hline & \mathbf{0}_{(n-r)\times r} & & \mathbf{0}_{(n-r)\times(m-r)} \end{array}\right] \in \mathbb{R}^{n\times m},$$

and $\sigma_1, \sigma_2, \ldots, \sigma_r$ are the nonzero singular values of A.[5]

[5] If $r = n$, matrix $\Sigma^\dagger$ does not contain the terms $\mathbf{0}_{(n-r)\times r}$ and $\mathbf{0}_{(n-r)\times(m-r)}$; similarly, if $r = m$ $\Sigma^\dagger$ does not contain $\mathbf{0}_{r\times(m-r)}$ and $\mathbf{0}_{(n-r)\times(m-r)}$.

We now bring the main theorem of this section, that the pseudo-inverse of a matrix is closely related to the orthogonal projection on the range.

Theorem 1.31 (Projection by pseudo-inversion)**.** *Let $A \in \mathbb{R}^{m\times n}$ be a real matrix. Then $\Pi = AA^\dagger$ is the orthogonal projection onto* range(A)*. Consequently, the matrix $Q = I - AA^\dagger$ is the orthogonal projection onto* $\ker(A)$.

Proof. Any vector $\boldsymbol{x} \in \mathbb{R}^m$ can be orthogonally decomposed as $\boldsymbol{x} = \boldsymbol{v} + \boldsymbol{u}$, where $\boldsymbol{u} \in \text{range}(A)$ and $\boldsymbol{v} \in \text{range}(A)^\perp$. Since $\boldsymbol{u} = A\boldsymbol{r}$ for some $\boldsymbol{r} \in \mathbb{R}^n$, we have (by property (i) of Definition 1.28) that

$$AA^\dagger \boldsymbol{u} = AA^\dagger A\boldsymbol{r} = A\boldsymbol{r} = \boldsymbol{u}.$$

Since $\boldsymbol{v} \in \text{range}(A)^\perp$, we have (by property (vi) of Theorem 1.29) that:

$$\begin{aligned} & \langle \boldsymbol{v}, A\boldsymbol{s} \rangle = 0 \ \ \forall \boldsymbol{s} \in \mathbb{R}^n, \\ \iff & \boldsymbol{s}^\top A^\top \boldsymbol{v} = 0 \ \ \forall \boldsymbol{s} \in \mathbb{R}^n, \\ \iff & A^\top \boldsymbol{v} = 0, \\ \iff & A^\dagger \boldsymbol{v} = 0. \end{aligned}$$

As a result, $AA^\dagger \boldsymbol{x} = AA^\dagger \boldsymbol{v} + AA^\dagger \boldsymbol{u} = \boldsymbol{u}$, and $\Pi = AA^\dagger$ is the orthogonal projection to range(A). Using Exercise 1.22, we can prove that $Q = I - AA^\dagger$ is the projection onto $\ker(A)$. □

Exercise 1.32. Let A be a matrix. Prove the following equalities regarding the pseudo-inverse:

- $A^\dagger = (A^\top A)^\dagger A^\top$
- $A^\top = A^\top A A^\dagger$.

1.5 Eigenvectors and Eigenvalues

Definition 1.33 (Eigenvector and Eigenvalue)**.** An n-dimensional nonzero vector $\boldsymbol{v}$ is an *eigenvector* of the $n \times n$ matrix A if it satisfies

$$A\boldsymbol{v} = \lambda \boldsymbol{v}, \tag{1.7}$$

for some scalar λ, called the *eigenvalue* associated to $\boldsymbol{v}$. In other words, the eigenvector $\boldsymbol{v}$ is a direction in the Euclidean space which is merely scaled by the linear mapping defined by A. The eigenvalue λ represents the scaling factor.

Notice that Equation 1.7 has a nonzero solution in $\boldsymbol{v}$ if and only if there exists some λ such that

$$(A - \lambda I)\boldsymbol{v} = \boldsymbol{0},$$

or equivalently, if

$$\det(A - \lambda I) = 0. \tag{1.8}$$

Equation Equation 1.8 is referred to as the *characteristic equation* of A, and $\det(A - \lambda I)$ is the *characteristic polynomial*.

While the characteristic equation (which is a polynomial) might have complex roots, if A is symmetric, all its roots become real:

Theorem 1.34. *If $A \in \mathbb{R}^{n \times n}$ is a real symmetric matrix then all of its eigenvalues are real and its eigenvectors can be chosen to form an orthonormal basis of the Euclidean space $\mathbb{R}^n$.*

We saw that *any* matrix has an SVD. If the study of eigenvectors and eigenvalues is of interest, there is another useful decomposition that shows this information. However, this decomposition only exists under specific assumptions. For example, if A is a square $n \times n$ matrix with n linearly independent eigenvectors, then A has a so-called *eigen-decomposition* or *spectral decomposition*, which is written as

$$A = Q\Lambda Q^{-1}.$$

Here, Q is a square $n \times n$ matrix whose columns are the n eigenvectors of A and Λ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues. If A is symmetric, according to Theorem 1.34, Q will be orthonormal, and hence the decomposition looks like

$$A = Q\Lambda Q^{\top}.$$

The relation between singular values and eigenvalues of a matrix $A \in \mathbb{R}^{n\times n}$ is given by[6]

$$\sigma_i(A)^2 = \lambda_i(AA^{\top}) = \lambda_i(A^{\top}A).$$

[6] Here, $\lambda_i(B)$ is the ith eigenvalue of B.

1.6 Quadratic forms

Definition 1.35 (Quadratic form)**.** Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix. The *quadratic form* induced by A is defined as the function $f : \mathbb{R}^n \to \mathbb{R}$ with $f(\boldsymbol{x}) = \boldsymbol{x}^{\top}A\boldsymbol{x}$.

Notice that the quadratic form $f(\boldsymbol{x})$ can be expressed as a polynomial of degree 2 in terms of the components of $\boldsymbol{x}$:

$$f(\boldsymbol{x}) = \boldsymbol{x}^{\top}A\boldsymbol{x} = \langle \boldsymbol{x}, A\boldsymbol{x}\rangle = \sum_{i,j} a_{i,j}x_i x_j \ , \quad i,j = 1,\dots,n \ ,$$

where $x_i \in \mathbb{R}$ is the i-th component of the vector $\boldsymbol{x}$ and $a_{i,j} \in \mathbb{R}$ are the entries of the matrix A.

Definition 1.36. A symmetric matrix with real entries $A \in \mathbb{R}^{n\times n}$ is *positive (semi-) definite*, or p.d. (p.s.d.), if and only if its induced quadratic form is positive (non-negative) for every nonzero $\boldsymbol{x} \in \mathbb{R}^n$, i.e.,

$$\begin{aligned} A \in \mathbb{R}^{n\times n} \text{ is p.s.d} &\iff \boldsymbol{x}^{\top}A\boldsymbol{x} \geq 0 \ \ \forall \boldsymbol{x} \in \mathbb{R}^n, \\ A \in \mathbb{R}^{n\times n} \text{ is p.d} &\iff \boldsymbol{x}^{\top}A\boldsymbol{x} > 0 \ \ \forall \boldsymbol{x} \in \mathbb{R}^n, \ \boldsymbol{x} \neq 0 \ . \end{aligned}$$

Example 1.37. Let

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \ , \quad A_2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \ , \quad A_3 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \ . \tag{1.9}$$

It is easy to verify that A_1 is positive definite; A_2 is positive semi-definite and A_3 is nondefinite (i.e., neither positive semi-definite or negative semi-definite). The quadratic forms induced by A_1, A_2, A_3 in Equation 1.9, are

$$\begin{aligned} f_1(\boldsymbol{x}) &= x_1^2 + x_2^2 \ , \\ f_2(\boldsymbol{x}) &= x_2^2 \ , \\ f_3(\boldsymbol{x}) &= x_2^2 - x_1^2 \ , \end{aligned} \tag{1.10}$$

where $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^\top$ and $x_1, x_2 \in \mathbb{R}$. Figure 1.8 shows the plots of the quadratic functions defined in Equation 1.10.

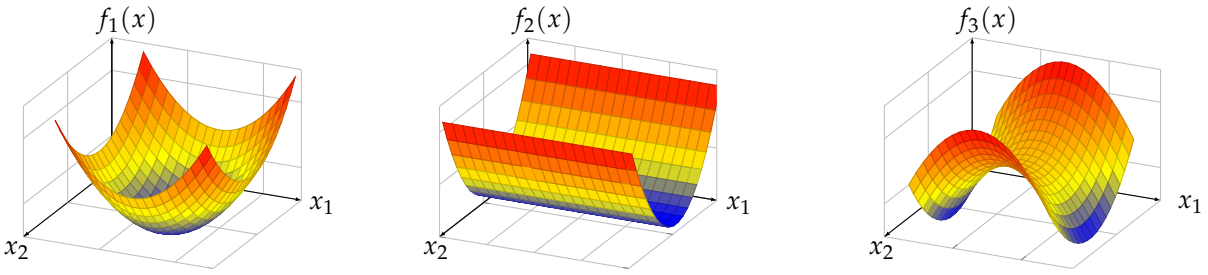


Figure 1.8: Plots of the quadratic forms described in Equation 1.10.

Positivity of a matrix can be read-off from its eigenvalues:

Theorem 1.38. *A symmetric matrix with real entries is positive (semi-) definite if and only if all its eigenvalues are positive (nonnegative).*

Similar to the real numbers, where for any real $a \in \mathbb{R}$, we have $a^2 \geq 0$, the same argument holds for matrices, with some tweaks.

Theorem 1.39. *Let $A \in \mathbb{R}^{m \times n}$. Then the matrices $AA^\top$ and $A^\top A$ are symmetric and positive semi-definite. Moreover, if $m \leq n$ and $\operatorname{rank}(A) = m$, then $AA^\top$ is also positive definite.*

Proof. Consider the quadratic form induced by $AA^\top$,

$$\boldsymbol{x}^\top AA^\top \boldsymbol{x} = (A^\top \boldsymbol{x})^\top (A^\top \boldsymbol{x}) \,, \quad \boldsymbol{x} \in \mathbb{R}^m \,. \tag{1.11}$$

Let $\boldsymbol{y} = A^\top \boldsymbol{x}$. Then the quadratic form in Equation 1.11 can be rewritten as:

$$\boldsymbol{x}^\top AA^\top \boldsymbol{x} = \boldsymbol{y}^\top \boldsymbol{y} = \|\boldsymbol{y}\|^2 \geq 0.$$

This proves the first statement of Theorem 1.39 for $AA^\top$, the proof for $A^\top A$ is analogous.

To prove the second statement of the theorem, we need to show that if $\operatorname{rank}(A) = m$ and $\boldsymbol{x} \neq \boldsymbol{0}$, then $\boldsymbol{y} \neq \boldsymbol{0}$. This is always true except if $\boldsymbol{x} \in \ker(A^\top)$; therefore, we need to prove that if $\operatorname{rank}(A^\top) = m$ then $\ker(A^\top)$ is trivial ($= \{\boldsymbol{0}\}$). By the rank-nullity theorem (Theorem 1.7), we have that

$$\operatorname{rank}(A^\top) + \dim(\ker(A^\top)) = m$$

therefore

$$\ker(A^\top) = \{\boldsymbol{0}\} \iff \operatorname{rank}(A^\top) = m \iff \operatorname{rank}(A) = m,$$

which is satisfied by assumption. This proves the second statement of the theorem. □

In real numbers, every non-negative number has a square root; similar argument holds for p.s.d. matrices. However, notice that the square root is not unique in general, but there exists a specific square root which is unique if the matrix is p.d.

Theorem 1.40 (Cholesky Decomposition)**.** *Every positive (semi-) definite matrix $A \in \mathbb{R}^{n\times n}$ can be factored as*

$$A = LL^\top , \tag{1.12}$$

where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix with positive (nonnegative) entries on its diagonal. The factorization given in Equation 1.12 is referred to as the Cholesky decomposition *of A and it is unique if A is positive definite.*

In the following example we use the Cholesky decomposition to characterize a set described by a quadratic inequality.

Example 1.41. Suppose $A \in \mathbb{R}^{n\times n}$ is a positive definite matrix and let $\boldsymbol{c} \in \mathbb{R}^n$. We will investigate the shape of the set

$$\mathcal{E} = \left\{\boldsymbol{x} \in \mathbb{R}^n : (\boldsymbol{x}-\boldsymbol{c})^\top A^{-1}(\boldsymbol{x}-\boldsymbol{c}) \le 1\right\}.$$

To obtain a better characterization of $\mathcal{E}$, consider the Cholesky decomposition of A,

$$A = LL^\top.$$

Since A is positive definite, the determinant of L is positive[7] and L is invertible. As a result, we can write:[8]

$$A^{-1} = (L^{-1})^\top L^{-1}.$$

Let $\boldsymbol{y} = L^{-1}(\boldsymbol{x}-\boldsymbol{c})$. Using $\boldsymbol{y}$, we can write:

$$(\boldsymbol{x}-\boldsymbol{c})^\top A^{-1}(\boldsymbol{x}-\boldsymbol{c}) = (\boldsymbol{x}-\boldsymbol{c})(L^{-1})^\top L^{-1}(\boldsymbol{x}-\boldsymbol{c}) = \boldsymbol{y}^\top\boldsymbol{y} = \|\boldsymbol{y}\|^2.$$

Notice that $\boldsymbol{x} = L\boldsymbol{y} + \boldsymbol{c}$; therefore, the set $\mathcal{E}$ can be equivalently described by:

$$\mathcal{E} = \{L\boldsymbol{y} + \boldsymbol{c} : \boldsymbol{y} \in \mathbb{R}^n,\ \|\boldsymbol{y}\| \le 1\}. \tag{1.13}$$

Using Equation 1.13 and the remark after the SVD theorem (Theorem 1.20) we conclude that $\mathcal{E}$ is a hyperellipse with center c.

[7] We used the fact that the determinant of a triangular matrix is the product of the entries on its diagonal.

[8] We note that for an invertible matrix B, we have $(B^{-1})^\top = (B^\top)^{-1}$.

1.7 Trace

Definition 1.42 (Trace)**.** The *trace* of a square matrix is the sum of its diagonal entries.

We now state the main properties of the trace.

1. The trace is a *linear functional*, i.e., for all square matrices $A, B \in \mathbb{R}^{n\times n}$ and all $c \in \mathbb{R}$ we have that

$$\operatorname{tr}(A+B) = \operatorname{tr}A + \operatorname{tr}B, \quad \operatorname{tr}(cA) = c\cdot\operatorname{tr}A.$$

2. For all $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times m}$ we have that

$$\operatorname{tr}(AB) = \operatorname{tr}(BA).$$

3. The trace is invariant under *cyclic permutations*, i.e., for matrices A, B, C, D with compatible dimensions,

$$\operatorname{tr}(ABCD) = \operatorname{tr}(BCDA) = \operatorname{tr}(CDAB) = \operatorname{tr}(DABC).$$

Notice that arbitrary permutations are not allowed.

4. As a corollary of 3, for any square matrix A and any invertible matrix P of appropriate dimensions, we have that

$$\operatorname{tr} A = \operatorname{tr}(P^{-1}AP) = \operatorname{tr}(APP^{-1}).$$

5. Let Π_A be the orthogonal projection on the range(A), then $\operatorname{tr}(\Pi_A) = \operatorname{rank}(A)$.

6. The trace of a square matrix is the sum of its eigenvalues

$$\operatorname{tr} A = \sum_{i=1}^{n} \lambda_i,$$

where $A \in \mathbb{R}^{n\times n}$ and λ_i is the ith eigenvalue of A. (Remember that the determinant of a square matrix is the product of its eigenvalues $\det(A) = \prod_{i=1}^{n} \lambda_i$.)

Using the trace we can define an inner product on $\mathbb{R}^{m\times n}$ as

$$\langle A, B\rangle = \operatorname{tr}(A^\top B) = \sum_{i,j} a_{ij}b_{ij}\,,$$

and a matrix norm as

$$\|A\|_F = \sqrt{\langle A, A\rangle} = \sqrt{\sum_{i,j} a_{ij}^2}\,. \tag{1.14}$$

The norm in Equation 1.14 is referred to as the *Frobenius norm* of A.

The following exercise needs knowledge of probability theory. In case you are not comfortable with probability, leave the exercise here, and after reading the rest of the notes, come back and try to solve it.

Exercise 1.43. Let $\varepsilon \in \mathbb{R}^n$ be a random vector with mean μ and covariance Σ. Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix and consider the quadratic form $\varepsilon^\top A\varepsilon$. What is the expected value $\mathbb{E}\left[\varepsilon^\top A\varepsilon\right]$ in terms of μ, Σ, and A?

Solution. Using the properties of the trace and the fact that $\varepsilon^\top A\varepsilon \in \mathbb{R}$, we have

$$\varepsilon^\top A\varepsilon = \operatorname{tr}\left(\varepsilon^\top A\varepsilon\right) = \operatorname{tr}\left(A\varepsilon\varepsilon^\top\right). \tag{1.15}$$

This is often called the *trace trick*. Hence, the expected value can now be computed as:

$$\begin{aligned}
\mathbb{E}\left[\varepsilon^\top A\varepsilon\right] &= \mathbb{E}\left[\operatorname{tr}\left(A\varepsilon\varepsilon^\top\right)\right], && \triangleright \text{From (1.15)}\\
&= \operatorname{tr}\left(\mathbb{E}\left[A\varepsilon\varepsilon^\top\right]\right), && \triangleright \operatorname{tr} \text{ and } \mathbb{E} \text{ are linear}\\
&= \operatorname{tr}\left(A\mathbb{E}\left[\varepsilon\varepsilon^\top\right]\right), && \triangleright A \text{ is deterministic, } \mathbb{E} \text{ linear}\\
&= \operatorname{tr}\left(A(\Sigma + \mu\mu^\top)\right), && \triangleright \mathbb{E}\left[\varepsilon\varepsilon^\top\right] = \Sigma + \mu\mu^\top\\
&= \operatorname{tr}(A\Sigma) + \operatorname{tr}\left(A\mu\mu^\top\right), && \triangleright \operatorname{tr} \text{ is linear}\\
&= \operatorname{tr}(A\Sigma) + \mu^\top A\mu. && \triangleright \text{Cyclic permuation}
\end{aligned}$$

We conclude that $\mathbb{E}\left[\varepsilon^\top A\varepsilon\right] = \operatorname{tr}(A\Sigma) + \mu^\top A\mu$. □

2

Analysis

This chapter will briefly recap the most important concepts from multivariate analysis, and recall the asymptotic notation.

Roadmap

We start by recalling what is a derivative of a multivariate function in Section 2.1. There, we introduce the gradient of a real-valued function, its relation to the derivative, and its geometric meaning. In Section 2.2, we review the chain rule, and use it to derive the directional derivative and derivative of the inverse function. This rule is computationally important for us later when learning about neural networks. Section 2.3 reviews the first-order necessary conditions of optimality. This is vital for understanding how and why many optimization algorithms work. In Section 2.4 and Section 2.5 we introduce the idea of approximating a function via its first and second order derivatives. This includes introducing the Hessian, which relates to quadratic form reviewed before in Section 1.6. Lastly, we recall the asymptotic notation in Section 2.6, which we will use to express how good an approximation is, and also use it to specify how our algorithms scale with input size later in the course.

Learning Objectives

After reading this chapter you should know

- what is the derivative and Jacobian, and how to compute the Jacobian for a differentiable function.
- what is the gradient of a function, its relation to derivative, and its geometric meaning that points to the steepest ascent direction.
- how to use chain rule to compute the derivative of the inverse of a function.
- what are critical points and what is their relation to optimization.
- how to construct a first- and second-order approximation to a function.
- how to interpret $f(x) = \mathcal{O}(g(x))$ and $f(x) = o(g(x))$ as $x \to a$.

2.1 *Multivariate Derivatives*

Let $f : \Omega \subset \mathbb{R}^n \to \mathbb{R}^m$ be a map between Euclidean spaces, where Ω is some open subset of $\mathbb{R}^n$. Formally, if there exists a unique linear

map $Df(\boldsymbol{x}_0) : \mathbb{R}^n \to \mathbb{R}^m$ that fulfills

$$\lim_{\|\boldsymbol{h}\| \to 0} \frac{\|f(\boldsymbol{x}_0 + \boldsymbol{h}) - f(\boldsymbol{x}_0) - Df(\boldsymbol{x}_0)[\boldsymbol{h}]\|}{\|\boldsymbol{h}\|} = 0, \tag{2.1}$$

we say that f is differentiable at $\boldsymbol{x}_0$ and call $Df(\boldsymbol{x}_0)$ the derivative of f at $\boldsymbol{x}_0$.

When $m = n = 1$, the derivative $Df(x_0) = f'(x_0)$ intuitively corresponds to the slope of the graph of f at the point x_0. Moreover, it induces the *best linear approximation* of f around x_0, which is defined as

$$g(x) = f(x_0) + f'(x_0) \cdot (x - x_0),$$

and it corresponds geometrically to the tangent line at the graph of f at the point $(x_0, f(x_0))$. Similarly, for any $m, n \geq 1$, the derivative $Df(\boldsymbol{x}_0)$ induces the *tangent plane* of f at $(\boldsymbol{x}_0, f(\boldsymbol{x}_0)) \in \mathbb{R}^{m+n}$, which is defined by the equation

$$g(\boldsymbol{x}) = f(\boldsymbol{x}_0) + Df(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0),$$

and can still be thought of as the best linear approximation of f around the point $\boldsymbol{x}_0$.

If $Df(\boldsymbol{x})$ exists for all $\boldsymbol{x} \in \Omega$, we call f differentiable. If additionally $Df(\boldsymbol{x})$ is continuous in $\boldsymbol{x}$, we call f continuously differentiable. From now on, we will only consider continuously differentiable maps f and omit any further case distinctions.

Note that because $Df(\boldsymbol{x}_0)$ is a linear map, we can interpret it as a matrix which we call the *Jacobian* of f. If $f_1, \ldots, f_m$ are the components of the vector-valued function f, then the entries of the Jacobian are $[Df(\boldsymbol{x}_0)]_{i,j} = \frac{\partial f_i}{\partial x_j}(\boldsymbol{x}_0)$, i.e.,

$$Df(\boldsymbol{x}_0) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\boldsymbol{x}_0) & \cdots & \frac{\partial f_1}{\partial x_n}(\boldsymbol{x}_0) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\boldsymbol{x}_0) & \cdots & \frac{\partial f_m}{\partial x_n}(\boldsymbol{x}_0) \end{bmatrix},$$

The Jacobian of a function f at a point $\boldsymbol{x}_0$ is sometimes also denoted by $\mathbf{J}_f(\boldsymbol{x}_0)$. If f is real-valued, the Jacobian $Df(\boldsymbol{x}_0)$ becomes a row vector:

$$Df(\boldsymbol{x}_0) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\boldsymbol{x}_0) & \cdots & \frac{\partial f}{\partial x_n}(\boldsymbol{x}_0) \end{bmatrix}.$$

The transpose of this row vector, considered as a vector, is called the *gradient* of f at $\boldsymbol{x}_0$ and is often denoted as $\nabla f(\boldsymbol{x}_0)$. Note that it follows that

$$Df(\boldsymbol{x}_0)[\boldsymbol{h}] = \langle \nabla f(\boldsymbol{x}_0), \boldsymbol{h} \rangle. \tag{2.2}$$

Among various important properties of the gradient, the following are particularly relevant to optimization, in which context they will repeatedly appear:

1. The gradient $\nabla f(\boldsymbol{x}_0)$ of f at $\boldsymbol{x}_0$ is a vector that points towards the direction in which f increases the most locally around $\boldsymbol{x}_0$.

Likewise, $-\nabla f(\boldsymbol{x}_0)$ points towards the direction in which f decreases the most locally around $\boldsymbol{x}_0$. That is why the gradient is sometimes called the *steepest ascent* direction.

2. The above property also implies that $\nabla f(\boldsymbol{x}_0)$ is orthogonal to the *level set* of f at $\boldsymbol{x}_0$, defined as $\{\boldsymbol{x} \in \Omega \mid f(\boldsymbol{x}) = f(\boldsymbol{x}_0)\}$. In two dimensions, this corresponds to lines called *contour lines*, as Figure 2.1 shows.

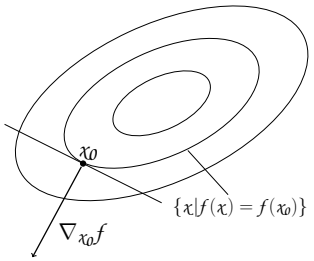


Figure 2.1: The gradient of f is orthogonal to its level sets.

Bonus Material

This bonus part discusses differentiation with respect to matrices. Whereas this is a topic of independent interest, in the context of these notes it will only be useful for understanding Subsection 10.3.3.

The Jacobian generalizes the notion of the derivative to higher dimensions by showing us how to differentiate vector-valued functions $f = (f_1, \dots, f_k)$ with respect to vector variables $\boldsymbol{x} = (x_1, \dots, x_n)$. Later, when we discuss neural networks, we will also need to differentiate linear vector-valued functions with respect to matrices. In other words, let $f(\boldsymbol{x}; \mathbf{W}) = \mathbf{W}\boldsymbol{x}$ be a linear function that depends on a parameter $\mathbf{W} \in \mathbb{R}^{k \times n}$. What is then

$$\frac{\partial f}{\partial \mathbf{W}}?$$

This would generally result in three-dimensional objects called *tensors*. We will use tensors in Chapter 10, but in a different context. Regarding differentiation with respect to matrices, we are only going to apply it to linear functions. Hence, for the sake of simplicity, we will try to avoid tensors.

To do this, we compute and use derivatives with respect to matrices as follows:

Let $\text{vec}(\mathbf{W}) \in \mathbb{R}^{kn}$ be the vector resulting from the matrix $\mathbf{W}$ after we place its rows one after the other. If we replace $\mathbf{W}$ by $\text{vec}(\mathbf{W})$, we can think of $\frac{\partial f}{\partial \mathbf{W}}$ as $\frac{\partial f}{\partial\, \text{vec}(\mathbf{W})}$. The latter is equal to the matrix

$$\frac{\partial f}{\partial\, \text{vec}(\mathbf{W})} = \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \frac{\partial f_1}{\partial w_{12}} & \cdots & \frac{\partial f_1}{\partial w_{kn}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial w_{11}} & \frac{\partial f_k}{\partial w_{12}} & \cdots & \frac{\partial f_k}{\partial w_{kn}} \end{bmatrix} \in \mathbb{R}^{k \times kn}.$$

Notice that $f_i(\boldsymbol{x}) = \mathbf{W}_{(i)}^\top \boldsymbol{x}$, where $\mathbf{W}_{(i)}$ denotes the i-th row of $\mathbf{W}$. Hence, f_i depends on no row of $\mathbf{W}$ apart from the i-th one. Thus, the above matrix can be written in short form as

$$\frac{\partial f}{\partial\, \text{vec}(\mathbf{W})} = \begin{bmatrix} \boldsymbol{x}^\top & \mathbf{0}_n & \dots & \mathbf{0}_n \\ \mathbf{0}_n & \boldsymbol{x}^\top & \dots & \mathbf{0}_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_n & \mathbf{0}_n & \dots & \boldsymbol{x}^\top \end{bmatrix} \in \mathbb{R}^{k \times kn},$$

where $\mathbf{0}_n = (0, \dots, 0)$ is the n-dimensional zero row-vector.

Identifying $\frac{\partial f}{\partial \mathbf{W}}$ with $\frac{\partial f}{\partial \operatorname{vec}(\mathbf{W})}$ causes a dimension inconsistency that will lead to problematic equations when we want to multiply $\frac{\partial f}{\partial \mathbf{W}}$ with other derivatives, like in the chain rule (see Section 2.2). To avoid this, we will need to remove the identically zero entries and define

$$\frac{\partial f}{\partial \mathbf{W}} := \begin{bmatrix} \boldsymbol{x}^\top \\ \boldsymbol{x}^\top \\ \vdots \\ \boldsymbol{x}^\top \end{bmatrix} = \begin{bmatrix} x_1 & \dots & x_n \\ x_1 & \dots & x_n \\ \vdots & \ddots & \vdots \\ x_1 & \dots & x_n \end{bmatrix} \in \mathbb{R}^{k \times n}. \tag{2.3}$$

Moreover, to avoid this dimension inconsistency, we will often need to execute the multiplication row-wise, namely,

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{bmatrix} \odot \begin{bmatrix} \boldsymbol{x}^\top \\ \boldsymbol{x}^\top \\ \vdots \\ \boldsymbol{x}^\top \end{bmatrix} := \begin{bmatrix} \alpha_1 \boldsymbol{x}^\top \\ \alpha_2 \boldsymbol{x}^\top \\ \vdots \\ \alpha_k \boldsymbol{x}^\top \end{bmatrix} \in \mathbb{R}^{k \times n}.$$

2.2 Chain Rule

The chain rule is one of the fundamental laws of multivariate analysis and a handy tool to prove some further properties of the derivative as well as the basis for deep learning.

Theorem 2.1 (Chain Rule). *Let $f : \Omega \subset \mathbb{R}^n \to \mathbb{R}^m$ and $g : \Omega' \subset \mathbb{R}^k \to \mathbb{R}^n$ be differentiable functions, where Ω and Ω' are open and* $\operatorname{range}(g) \subset \Omega$. *Then it holds for $\boldsymbol{x}_0 \in \Omega'$ that*[1]

$$D(f \circ g)(\boldsymbol{x}_0) = (Df)(g(\boldsymbol{x}_0)) \circ Dg(\boldsymbol{x}_0).$$

[1] The notation $\circ$ means the composition of two maps. So $f \circ g(x) = f(g(x))$.

Note that because $(Df)(g(\boldsymbol{x}_0))$ and $Dg(\boldsymbol{x}_0)$ are linear, their composition is also linear. Composition of linear maps in matrix form becomes matrix multiplication, and thus, the formula can be written in terms of Jacobians as

$$D(f \circ g)(\boldsymbol{x}_0) = (Df)(g(\boldsymbol{x}_0))Dg(\boldsymbol{x}_0).$$

Also note that $(Df)(g(\boldsymbol{x}_0))$ is *not* equal to the differential of $f \circ g$ evaluated at $\boldsymbol{x}_0$, $D(f \circ g)(\boldsymbol{x}_0)$, but rather the differential of f evaluated at $g(\boldsymbol{x}_0)$.

We will omit the proof of Theorem 2.1, note however that this follows directly from the definition Equation 2.1 of the derivative.

Example 2.2 (Directional Derivative). Let $f : \mathbb{R}^n \to \mathbb{R}$ and consider the function $\phi_v : \mathbb{R} \to \mathbb{R}^n$ with $\phi_v = \boldsymbol{x}_0 + t\boldsymbol{v}$. This function traces a line in the direction $\boldsymbol{v}$ passing through $\boldsymbol{x}_0$. Recall that the *directional derivative* $\partial f / \partial \boldsymbol{v}$ of f at $\boldsymbol{x}_0$ in direction $\boldsymbol{v} \in \mathbb{R}^n$ is defined as the derivative, at $t = 0$, of the function $g(\cdot\,; \boldsymbol{v}) : \mathbb{R} \to \mathbb{R}$, with $g(t; \boldsymbol{v}) = (f \circ \phi_v)(t) = f(\boldsymbol{x}_0 + t\boldsymbol{v})$. In other words,

$$\frac{\partial f}{\partial \boldsymbol{v}}(\boldsymbol{x}_0) := D(f \circ \phi_v)(0) = \lim_{t \to 0} \frac{f(\boldsymbol{x}_0 + t\boldsymbol{v}) - f(\boldsymbol{x}_0)}{t} \in \mathbb{R}.$$

Using the chain rule, we can show that the directional derivative in direction of $v \in \mathbb{R}^n$ corresponds to $Df(x_0)[v]$:

$$\frac{\partial f}{\partial v}(x_0) = D(f \circ \phi_v)(0) = (Df)(\phi_v(0)) \underbrace{D\phi_v(0)}_{=v} = Df(x_0)[v].$$

In particular, by the definition of the gradient,

$$\frac{\partial f}{\partial v}(x_0) = \langle \nabla f(x_0), v \rangle . \tag{2.4}$$

The chain rule also directly implies what the Jacobian of the inverse of a bijective $f : \Omega \subset \mathbb{R}^n \to \mathbb{R}^m$ looks like.

Corollary 2.3 (Jacobian of the Inverse)**.** *Assume that f and f^{-1} are differentiable. Then it holds for any $y_0 \in \text{range}(f)$ that*

$$Df^{-1}(y_0) = (Df)^{-1}(f^{-1}(y_0)).$$

Proof. Applying the chain rule from Theorem 2.1 to

$$I = D(\text{id})(y_0) = D(f \circ f^{-1})(y_0) = (Df)(f^{-1}(y_0))Df^{-1}(y_0)$$

yields the result. □

2.3 Extremal Points

From calculus, we learned how to find the minimum of a (differentiable) function $f : \mathbb{R} \to \mathbb{R}$; we searched through the points that $f'(x) = 0$, and knew that the minimizer is going to be one of them. In this section, we want to make this intuition more precise, and applicable to multivariate functions.

Suppose we are given a function $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$, defined on an open subset Ω of $\mathbb{R}^n$. First, we have to agree on different notions of optimality, that is, what do we mean if we say "the minimum/-maximum of f". One can think of two types of optimality: We call $x_0 \in \Omega$ a *local minimizer* of f, if $f(x_0) \leq f(x)$ for all x close enough to x_0.[2] The value $f(x_0)$ is called a *local minimum* of f. Local maximizer and local maximum are defined analogously. We call x_0 a *global minimizer* of f if $f(x_0) \leq f(x)$ for all $x \in \Omega$. In an optimization problem, these minimizers are the ones that we are hoping to find. However, sometimes the optimization problem might be so hard that we would be happy with finding a local minimizer. Note that, importantly, a global minimizer is also a local minimizer.[3]

The following theorem gives a necessary condition for being a local minimizer.[4]

Theorem 2.4 (Necessary Condition for Local Optimality)**.** *Let $\Omega \subseteq \mathbb{R}^n$ be open and $f : \Omega \to \mathbb{R}$ a continuously differentiable function. If $x_0 \in \Omega$ is a local minimizer, then*

$$\nabla f(x_0) = \mathbf{0}.$$

[2] That is, if there exists an $\varepsilon > 0$, such that for all x that are ε-close to x_0 ($\|x - x_0\| < \varepsilon$) we have $f(x_0) \leq f(x)$.

[3] Please take note that the discussion in this subsection is applicable to "unconstrained" optimization, where the domain of the function to be optimized is $\mathbb{R}^n$, or the case where the domain of the function is an open set. Recall that an open subset Ω of $\mathbb{R}^n$ has the property that for all $x \in \Omega$, there is a small-enough ball around x that is contained in Ω.

[4] The same theorem holds for local maximizers.

Therefore, when searching for local/global minima, we only have to search within the set of critical (or stationary) points *defined as* $\{\boldsymbol{x} \in \Omega \mid \nabla f(\boldsymbol{x}) = \mathbf{0}\}$.

Proof. Note that if $\boldsymbol{x}_0$ is a local minimizer of f, it also must be a local minimizer of any of the functions $\psi_v(t) = f(\boldsymbol{x}_0 + t\boldsymbol{v})$ defined for all $\boldsymbol{v} \in \mathbb{R}^n$. As these new functions are $\mathbb{R} \to \mathbb{R}$, we know from calculus that $\frac{d\psi_v}{dt}(0) = 0$. Note that $\frac{d\psi_v}{dt}(0) = \frac{\partial f}{\partial \boldsymbol{v}}(\boldsymbol{x}_0)$. As seen in equation Equation 2.4, this implies that for all $\boldsymbol{v} \in \mathbb{R}^n$

$$\nabla f(\boldsymbol{x}_0)^\top \boldsymbol{v} = \frac{d\psi_v}{dt}(0) = 0.$$

This in turn implies that $\nabla f(\boldsymbol{x}_0) = \mathbf{0}$. □

Note that the converse is not true. If the gradient of a function is zero, the function does not necessarily have a local minimum/maximum.

Theorem 2.4, together with the property of the gradient that it always points towards the direction of the steepest ascent, motivates the *gradient descent* algorithm–one of the most popular optimization methods used in machine learning.

2.4 Taylor Expansions

It was already mentioned that the derivative is the best linear approximation of a function. This intuitive concept can be formulated rigorously by using the first-order Taylor expansion, as the following theorem shows.

Theorem 2.5 (First-order Taylor Expansion)**.** *Let $\Omega \subset \mathbb{R}^n$ be open and $f : \Omega \to \mathbb{R}$ be a continuously differentiable function. Let $\boldsymbol{x}_0 \in \Omega$. Then, it holds for all $\boldsymbol{x}$ in an open neighbourhood of $\boldsymbol{x}_0$ that*

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_0) + \langle \nabla f(\boldsymbol{x}_0), \boldsymbol{x} - \boldsymbol{x}_0 \rangle + o\left(\|\boldsymbol{x} - \boldsymbol{x}_0\|\right)$$

as $\boldsymbol{x} \to \boldsymbol{x}_0$.[5] *In other words, the approximation error $\xi(\boldsymbol{x} - \boldsymbol{x}_0) = f(\boldsymbol{x}) - (f(\boldsymbol{x}_0) + \langle \nabla f(\boldsymbol{x}_0), \boldsymbol{x} - \boldsymbol{x}_0 \rangle)$ satisfies $\lim_{x \to x_0} \frac{\xi(\boldsymbol{x} - \boldsymbol{x}_0)}{\|\boldsymbol{x} - \boldsymbol{x}_0\|} = 0$.*

[5] See Section 2.6 for the definition of the little-o notation.

2.5 Second-order Derivatives, Hessian

Just like the first-order derivative is the best linear approximation, the second-order derivative is the best quadratic approximation. In higher dimensions, quadratic forms (see Section 1.6) are used to express quadratic functions.

Let $f : \Omega \subset \mathbb{R}^n \to \mathbb{R}$ be a real-valued function. We learned that Df is the best linear approximation for f at any point. We say f is twice differentiable at $\boldsymbol{x}_0 \in \Omega$ if there exists a quadratic form

$D^2 f(\boldsymbol{x}_0)$ such that

$$\begin{aligned} f(\boldsymbol{x}) = f(\boldsymbol{x}_0) \\ &+ \langle \nabla f(\boldsymbol{x}_0), \boldsymbol{x} - \boldsymbol{x}_0 \rangle \\ &+ \frac{1}{2} (\boldsymbol{x} - \boldsymbol{x}_0)^\top D^2 f(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0) \\ &+ o(\|\boldsymbol{x} - \boldsymbol{x}_0\|^2) \quad \text{as} \quad \boldsymbol{x} \to \boldsymbol{x}_0. \end{aligned}$$

This quadratic form is called the second derivative of f at $\boldsymbol{x}_0$. As any quadratic form can be represented by a symmetric matrix, it turns out that the matrix for $D^2 f(\boldsymbol{x}_0)$ is

$$D^2 f(\boldsymbol{x}_0) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(\boldsymbol{x}_0) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1}(\boldsymbol{x}_0) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n}(\boldsymbol{x}_0) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(\boldsymbol{x}_0) \end{bmatrix}$$

which we call the *Hessian* of f at $\boldsymbol{x}_0$. There is a fundamental link between the local curvature of a function f and its Hessian, as the following exercise suggests.

Exercise 2.6. Let $f(\boldsymbol{x}) = \frac{1}{2} \boldsymbol{x}^\top A \boldsymbol{x}$ be a quadratic form for a symmetric matrix $A \in \mathbb{R}^{n \times n}$. Show that the Hessian of f is exactly A, i.e., $D^2 f(\boldsymbol{x}) = A$ for all $\boldsymbol{x}$. Further, show that f has a unique minimum at $\mathbf{0}$ if A is positive definite, and has a unique maximum if A is negative definite. Moreover, if A is indefinite (i.e., has both positive and negative eigenvalues), prove that there are directions $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ such that $f(\alpha \boldsymbol{u}) \to +\infty$ and $f(\alpha \boldsymbol{v}) \to -\infty$ as $\alpha \to \infty$.

2.6 Asymptotic Notation

Asymptotic notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is used in computer science to classify algorithms according to how their run time or space requirements grow as the input size grows. In analysis, it is used to provide growth bounds via easier-to-understand functions, such as polynomials, logarithms, or exponentials Wikipedia 2022.

2.6.1 Big-O notation

The *big-O notation* is used to provide *upper bounds* on the growth of a function.

Definition 2.7 (Big-O notation)**.** Let f and g be real valued functions defined on some unbounded subset of the positive real numbers. We write

$$f(x) = \mathcal{O}(g(x)) \quad \text{as} \quad x \to \infty$$

if there exists a positive scalar M and a real number $x_0 > 0$ such that

$$|f(x)| \leq M\,|g(x)| \quad \text{for all } x \geq x_0.$$

In this case, we say that $f(x)$ is of order $\mathcal{O}(g(x))$ asymptotically. Similarly, for a fixed number $a \in \mathbb{R}$ we write

$$f(x) = \mathcal{O}(g(x)) \quad \text{as} \quad x \to a,$$

if there exists some $\delta > 0$ and $M > 0$ such that $|f(x)| \leq M|g(x)|$ for all $0 < |x - a| < \delta$.

In the definition above, it is usually the case that g is an easy-to-understand function and the growth behavior of f is controlled by g. Moreover, one should always mention the limiting argument (if the limit is taken as $x \to \infty$ or $x \to a$).

Example 2.8 (Approximation error)**.** Let f be the second order Taylor expansion of the exponential function around the point 0:

$$f(x) = 1 + x + \frac{x^2}{2}.$$

Obviously, f is only an approximation and to get an idea of how well it approximates e^x around 0, we can use the big-O notation:

$$e^x - f(x) = \mathcal{O}(x^3) \quad \text{as} \quad x \to 0,$$

that is, the error is smaller than some constant times x^3 if x is close enough to 0. This result is due to the Taylor's theorem.

We now list two important rules for manipulating $\mathcal{O}$ terms:

- If $g(x) = \mathcal{O}(f(x))$ then $cg(x) = \mathcal{O}(f(x))$ for any constant c.
- If $g_1(x)$ and $g_2(x)$ are both $\mathcal{O}(f(x))$ then so is $g_1(x) + g_2(x)$.

2.6.2 *Little-o notation*

The *little-o* notation can be used to express that a function grows slower than some other function. For example, $f(x) = o(g(x))$ signifies that f grows much slower than g and is insignificant in comparison.

Definition 2.9 (Little-o notation)**.** Let f and g be two real valued functions defined on some unbounded subset of the positive real numbers, and let a be a fixed real number or infinity. Provided that g is nonzero in proximity of a, we write

$$f(x) = o(g(x)) \quad \text{as} \quad x \to a$$

if

$$\lim_{x \to a} \frac{f(x)}{g(x)} = 0.$$

Example 2.10. We have $x^n = o(e^x)$ as $x \to \infty$ for any $n > 0$.

While Big-O notation can be intuitively interpreted as "grows ultimately as fast as", the little-o notation can be understood as "ultimately grows slower than".

3
Probability Theory

In this chapter, we will recap the fundamentals of probability theory. It is a challenge to introduce probability theory while keeping a good level of rigor. We will take a middle ground and keep ourselves away from a fully rigorous treatment of the topic.

Roadmap

We start from scratch, by introducing what is a probability space in Section 3.1. In Section 3.2 we introduce one of the cornerstones of this chapter: random variables. We define what is the distribution of a random variable and how to characterize it in general via the CDF. We then focus on two classes of random variables: discrete and continuous. Discrete random variables can take a discrete set of values and can be described easily via the probability mass function. This is the content of Section 3.2.1, where we also introduce some famous distributions. In Section 3.2.2 and Section 3.2.3, we deal with continuous random variables, which are trickier that discrete ones. We only focus on those that have a so-called density, and introduce some famous continuous distributions at the end. Afterwards, in Section 3.2.4 we introduce another cornerstone of our exposition: the expected value. We will also recall variance of random variables in Section 3.2.5.

After understanding a single random variable, we move on to the situation of multiple random variables. This is the subject of Section 3.3. There, we introduce notions such as joint distribution, independence, marginals and conditional distribution. We finish by mentioning three important theorems about conditional distributions in Section 3.3.9. Section 3.4 is one of the most important parts of our exposition. There we recall how to find the expected value of a function of several random variables, and understand the important notion of conditional expectation. We finish our exposition with the important Gaussian (or normal) distribution in Section 3.5, and bring an exemplar of theorems that show why Gaussians are important.

Learning Objectives

After reading this chapter you should know

- what is a probability space.
- what is a random variable, its distribution, CDF, and what does it mean for a random variable to be discrete or continuous.
- what is PMF and PDF.
- what does expected value represent and how to compute it for a discrete or continuous random variable.

- what is the law of unconscious statistician.
- joint distributions, joint PMF and PDF.
- how to marginalize on a subset of random variables.
- how to verify if events are independent, notion of conditional probability for random variables, law of total probability, chain rule, and Bayes rule.
- how to verify if random variables are independent, notion of conditional distribution of random variables.
- how to compute the expected value of a function of several random variable, linearity of expectation.
- how to interpret $\mathbb{E}_X[f(X,Y)]$.
- the covariance matrix.
- normal distribution.
- the law of large numbers.

3.1 *Probability Spaces*

Let Ω be any set. We will call Ω the *sample space*, and it will be interpreted as the set of all possible outcomes of an experiment. Choosing a suitable Ω is a part of modelling the real world experiment and thus is not necessarily unique. For example, think about possible outcomes of throwing a dart to a dartboard.

- $\Omega = \mathbb{R}^2$ could model the exact place of landing,
- $\Omega = \{0, 1\}$ could model a hit or miss, and
- $\Omega = \{0, 10, 20, \ldots, 100\}$ could model the score.

Most of the time, especially when Ω is a complicated set (like in the first example), we do not care too much about *what* exactly happened. Rather than asking whether a certain element of Ω has happened or not, we want to ask harder questions. For example, we want to know whether the score is higher than 60 or not, or the point is at least 1 cm away from the center of the dartboard. Notice that these types of questions naturally correspond to specific subsets of Ω. For example, the former question corresponds to the subset $\{80, 100\}$. We call each of these subsets an *event*. We also say that "event A occurs" if the outcome of the experiment is in A.

Bonus Material

We call a family $\mathcal{F}$ of subsets of Ω that encode our questions of interest, the *family of events*. Notice how logical "and" is translated into intersection of events, "or" into union, and "not" into complement. Thus, we desire our family of events to be closed under these operations, that is, $\mathcal{F}$ needs to be a so-called *σ-algebra*:

- $\Omega \in \mathcal{F}$,
- if $A \in \mathcal{F}$, then also $A^c \in \mathcal{F}$,

- for $A_1, A_2, \dots \in \mathcal{F}$, their union must also be contained in $\mathcal{F}$.

Note that these properties also imply that $\mathcal{F}$ is closed under intersection. Examples of $\mathcal{F}$ satisfying these properties are $\mathcal{F} = \{\Omega, \emptyset\}$ or $\mathcal{F} = \mathcal{P}(\Omega)$, the power set of Ω.

After identifying the sample space (Ω) and the set of all events ($\mathcal{F}$), we can go ahead and assign probabilities to the events.[1] We describe this assignment via a function $\mathbb{P} : \mathcal{F} \to [0, 1]$, called the *probability function*. To have a consistent theory, this assignment has to obey certain axioms:

1. Total probability equals one: $\mathbb{P}(\Omega) = 1$, and
2. Probability is additive for disjoint events: for all at most countably many, pairwise disjoint events $A_i \in \mathcal{F}$ it should hold that

$$\mathbb{P}\left(\bigcup_i A_i\right) = \sum_i \mathbb{P}(A_i).$$

We call the triple $(\Omega, \mathcal{F}, \mathbb{P})$ a *probability space*.

[1] The question of *how* to assign probabilities to events is a philosophical one. There are several proposals, and we deal with *frequentist* and *Bayesian* approach in this course. In later sections, these conceptions will be introduced.

Example 3.1. Consider the experiment of throwing a die and looking at the top face. We model this problem by setting the sample space to be $\Omega = \{\boxdot, \boxdot, \dots, \boxdot\}$. As any subset of Ω can be written as a union of singletons (sets of size 1), we only have to define the probability function for singletons. For example, let $\mathbb{P}(\{\boxdot\}) = p_1, \dots, \mathbb{P}(\{\boxdot\}) = p_6$, where $p_1, \dots, p_6$ are some numbers between 0 and 1 adding up to 1. Observe that, e.g., $\mathbb{P}(\{\boxdot, \boxdot\}) = p_2 + p_5$.

The following two exercises remind you of two important properties of a probability function.

Exercise 3.2 (Monotonicity)**.** Let $A, B \in \mathcal{F}$ be such that $B \subseteq A$. Show that $\mathbb{P}(B) \leq \mathbb{P}(A)$.

Exercise 3.3 (Union Bound)**.** Suppose we have at most countably many not necessarily disjoint events $A_i \in \mathcal{F}$. Prove

$$\mathbb{P}\left(\bigcup_i A_i\right) \leq \sum_i \mathbb{P}(A_i).$$

This inequality comes in quite handy in many applications and proofs.

3.2 Random Variables

Random variables are a quantitative way of looking at an experiment. Their job is to take the outcome of an experiment and assign a number to it.

Definition 3.4 (Random Variable)**.** A random variable X is a function that assigns a real number to every outcome of an experiment, $X : \Omega \to \mathbb{R}$.[2] Notice that the function is not random, while the "randomness" is in the input to the function.

[2] As this course is an introductory course, we do not concern ourselves with measure theory.

We bring this quote from Williams 1991:

Once Tyche, Goddess of Chance, decides the outcome $\omega \in \Omega$, the values of all random variables lock into place.

We can naturally use random variables to create events. A notion that is useful in this regard is the *inverse image*. Given a subset of $\mathbb{R}$, we ask what elements of Ω are mapped into that subset. Formally, for a random variable X and a subset $A \subset \mathbb{R}$, we define

$$X^{-1}(A) = \{\omega \in \Omega : X(\omega) \in A\}.$$

We sometimes write this set as $\{X \in A\}$. For example, for an interval $[a, b]$, the event $\{a \leq X \leq b\}$ is the set of all outcomes whose X-value is between a and b. It turns out that all interesting events that one might consider are intervals and their (countable) unions and intersections. We denote by $\mathcal{B}(\mathbb{R})$ the family of all these subsets. The *distribution* of a random variable tells us the probabilities of all these events, and is sufficient for a full characterization of X.

Definition 3.5 (Distribution of a Random Variable)**.** For any $A \in \mathcal{B}(\mathbb{R})$, we define

$$\mathbb{P}_X(A) := \mathbb{P}\left(X^{-1}(A)\right) = \mathbb{P}\left(\{\omega \in \Omega : X(\omega) \in A\}\right),$$

We call $\mathbb{P}_X$ the distribution (or *law*) of X.

Notice that $\mathbb{P}_X$ acts on subsets of $\mathbb{R}$ and tells us the probability that X takes a value in that subset, while $\mathbb{P}$ acts on events and tells us how probable that event is. The nice thing about $\mathbb{P}_X$ is that it is defined over subsets of a fixed set ($\mathbb{R}$), and to characterize it, we do not need to think in terms of Ω, which can sometimes be a difficult set to reason about. Indeed, $(\mathbb{R}, \mathcal{B}(\mathbb{R}), \mathbb{P}_X)$ is a probability space! The proof of this claim is straightforward, and we leave it as an exercise.

It turns out that one can describe any probability function on $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ by its values on the intervals of the form $(-\infty, x]$ with $x \in \mathbb{R}$. Hence, to describe the law of a random variable, it suffices to identify its value on these subsets only.

Definition 3.6 (Cumulative Distribution Function)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space and $X : \Omega \to \mathbb{R}$ a random variable. Then, the cumulative distribution function (CDF) of X is defined as

$$\begin{aligned} F_X(x) &:= \mathbb{P}_X((-\infty, x]) \\ &= \mathbb{P}\left(X^{-1}((-\infty, x])\right) \\ &= \mathbb{P}\left(\{\omega \in \Omega \mid X(\omega) \leq x\}\right) \\ &=: \mathbb{P}\left(X \leq x\right). \end{aligned}$$

The CDF of a random variable X has the following properties:

- $\lim_{x\to\infty} F_X(x) = 1$ and $\lim_{x\to-\infty} F_X(x) = 0$,
- it is monotonically increasing,

[3] Meaning that for every $x \in \mathbb{R}$, the function F_X agrees with its right limit: $\lim_{y\to x^+} F_X(y) = F_X(x)$.

- it is right-continuous,[3]
- and $\mathbb{P}(a < X \leq b) = F_X(b) - F_X(a)$.

In general, F_X does not need to be continuous. Assume, for example, that for some $x_0 \in \mathbb{R}$, $\mathbb{P}(X = x_0) = \alpha > 0$. Then $F_X(x_0) - \lim_{x \to x_0^-} F_X(x) = \alpha > 0$ and F_X cannot be continuous, see Figure 3.1.

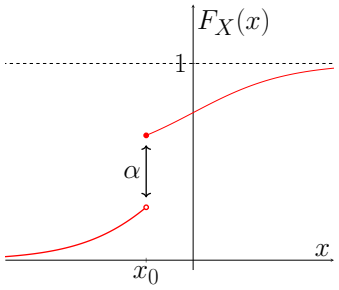


Figure 3.1: The CDF of a random variable with a discontinuity at x_0.

[4] In the die example, $\mathcal{X} = \{100, 200, \ldots, 600\}$.

In the following sections, we focus on *discrete* and *continuous* random variables. For each of these cases, one can often describe the law of the random variable in an easier way.

3.2.1 Discrete Random Variables

A random variable is *discrete* if the set of values it can output, denoted by $\mathcal{X}$, is a discrete set (a finite or countable set).[4]

Example 3.7 (Dice, continued)**.** In the experiment of throwing a die, suppose we will multiply the face number by 100 and donate that amount to charity. This naturally corresponds to creating a function that translates any outcome into a number (the amount of money we donate). We can then perform the experiment (i.e., throw the die) and apply the function to the outcome and donate the value.

Recall for each possible value $x \in \mathcal{X}$, the notion of the event $\{X = x\}$, which is a shorthand for $\{\omega \in \Omega : X(\omega) = x\}$. Knowing the probability of these events completely identifies $\mathbb{P}_X$, as all other events related to X can be written as a union of sets of the form $\{X = x\}$.

Definition 3.8 (Probability Mass Function)**.** Let X be a discrete random variable and $\mathcal{X}$ be the set of all of its values. We define for each $x \in \mathcal{X}$

$$p_X(x) = \mathbb{P}(X = x) = \mathbb{P}_X(\{x\}).$$

We call p_X the *probability mass function* (PMF) of X. If X is clear from the context, we just write $p(x)$ instead of $p_X(x)$.

Example 3.9 (Dice, continued)**.** Let X be the amount of money we donate. Then, the probability mass function of X is $p_X(100) = p_1, \ldots, p_X(600) = p_6$. Moreover, the probability that we donate less than 350 is

$$\mathbb{P}(X < 350) = \mathbb{P}(\{X = 100\} \cup \{X = 200\} \cup \{X = 300\}) = p_1 + p_2 + p_3.$$

Remark. In our running example, one observes that the actual mechanism that resulted in a donation value is not important (whether it was a die that we threw and we multiplied its face by 100, or we asked a random person on the street to choose a real number in $[0, 1]$, and we multiply it by 600 and round it up to a number in $\{100, \ldots, 600\}$). The only thing that matters is "what values can X take" and "with what probability each value is produced". This information is encoded in p_X. That is why we mostly forget about $\Omega, \mathcal{F}, \mathbb{P}$ and directly talk about a random variable X with a probability mass function p_X.

Below is a list of discrete random variables with their associated PMF. We also bring a model probability space on which these random variables are defined.

- **Bernoulli.** We write $X \sim \text{Ber}(q)$ if $\mathcal{X} = \{0,1\}$ and $p_X(1) = q$ (and hence, $p_X(0) = 1-q$). A model for this random variable is the throw of a (biased) coin, whose probability of landing Heads is q. Defining $X(\text{Heads}) = 1$ and $X(\text{Tails}) = 0$ will result in the desired random variable.
- **Categorical.** We write $X \sim \text{Cat}(p_1, \dots, p_k)$ if $\mathcal{X} = \{1, \dots, k\}$ and $p_X(i) = p_i$ for $i = 1, \dots, k$ (we require $\sum p_i = 1$ and $p_i > 0$). A model for this random variable is selecting among k different choices, each with some probability, and assigning $1, \dots, k$ to the choices.
- **Binomial.** We write $X \sim \text{Binom}(q, n)$, with $q > 0$ and $n \in \mathbb{N} \setminus \{0\}$, if $\mathcal{X} = \{0, 1, \dots, n\}$ and

$$p_X(k) = \binom{n}{k} q^k (1-q)^{n-k}, \quad k = 0, 1, \dots, n.$$

  A model for this random variable is the number of heads in a sequence of n independent tosses of the same coin, where q is the probability of a head.

Bonus Material

- **Poisson.** We write $X \sim \text{Poisson}(\lambda)$, with $\lambda > 0$, if $\mathcal{X} = \mathbb{N}$ and

$$p_X(k) = e^{-\lambda} \frac{\lambda^k}{k!}, \quad k = 0, 1, 2 \dots$$

  The Possion distribution with parameter λ is a good approximation of the Binomial distribution with parameters n and q if n is large, q is small and $\lambda = nq$. In fact, provided that $\lambda = nq$, then

$$\lim_{n \to \infty} p_{X,\text{Binom}}(k) = e^{-\lambda} \frac{\lambda^k}{k!}, \tag{3.1}$$

  where with $p_{X,\text{Binom}}(k)$ we denote the Binomial PMF with parameters n and q. As a result, we can use the Possion random variable to model Binomial random variables where n is large and q is small (e.g. the number of car accidents in a city on a given day, since the number of cars is large and the probability of accident for a single car is very small).

  Exercise 3.10. Prove Equation 3.1. Hint: if $\lambda = nq$, then the Binomial PMF with parameters n and q can be rewritten as

$$\begin{aligned} p_X(k) &= \frac{n!}{(n-k)!k!} q^k (1-q)^{n-k} \\ &= \frac{n(n-1)\cdots(n-k+1)}{n^k} \cdot \frac{\lambda^k}{k!} \cdot \left(1 - \frac{\lambda}{n}\right)^{n-k}. \end{aligned}$$

3.2.2 *Continuous Random Variables*

In this section, we introduce *continuous* random variables. Our goal is to give a somewhat rigorous, but simultaneously intuitive, short

intro for the sake of completeness. A truly rigorous exposition can be found in probability theory books (see, e.g., Durrett 2019).

A random variable is *continuous*, if, loosely speaking, the set of values it can produce is uncountably infinite and the probability of attaining a single value is zero, that is $\mathbb{P}(X = x) = 0$ for all $x \in \mathbb{R}$.[5]

[5] In these notes, we do not differentiate between continuous and absolutely continuous random variables.

Example 3.11. Take the experiment of choosing a "random" point on a disk. That is, let $\Omega = \{x \in \mathbb{R}^2 \mid \|x\| \leq 1\}$ be the two-dimensional disk, and $\mathbb{P}$ be the uniform measure, i.e., for $A \subseteq \Omega$ we have $\mathbb{P}(A) = \text{area}(A)/\text{area}(\Omega)$.[6] Let

$$X : \Omega \to \mathbb{R}, \quad X(\omega) = \|\omega\|$$

be the distance of the point ω to the center of the disk. Then X is an example of a continuous random variable. Notice that $\mathbb{P}(X = a) = 0$ for any $a \in [0, 1]$ (why? try to prove this fact).

[6] One has to be careful here, as there are subsets of the disk that one cannot assign any area to them. To resolve this issue, the Borel σ-algebra and measurability should be rigorously defined, which we do not discuss here.

Remark. It turns out that all random variables are either discrete, continuous, or "a mixture of the two".[7] This implies that we only have to treat two cases: discrete and continuous.

[7] This rather technical statement is called the Lebesgue decomposition theorem.

3.2.3 *Probability Density*

To understand a continuous random variable, knowing its CDF is sufficient. However, many random variables that we are considering in this course can be characterized even easier, in terms of a density function. Before we go into details, we first bring an intuitive explanation of what a density is.

Let M be a non-homogeneous physical object, for example, a rock. We define $\text{diam}(M)$ to be the diameter of M, $\text{vol}(M)$ to be its volume, and $m(M)$ to be its mass. Take a point $x \in M$ and consider small balls around x. For each of these neighborhoods, compute the mass and volume. By physical intuition, we know that if we make the neighborhood smaller and smaller, these two quantities tend to zero, i.e., if I is a neighborhood around x, we have

$$\lim_{\substack{x \in I \\ \text{diam}(I) \to 0}} \text{vol}(I) = 0, \qquad \lim_{\substack{x \in I \\ \text{diam}(I) \to 0}} m(I) = 0.$$

However, if we divide these two numbers, the ratio converges to a number which we call *density of M at x*,

$$\lim_{\substack{x \in I \\ \text{diam}(I) \to 0}} \frac{m(I)}{\text{vol}(I)} = \rho(x).$$

The relation between density and mass becomes clear with the following formula: for any subset A of M,

$$m(A) = \int_A \rho(x)\, dx.$$

That is, the density is there to be integrated!

Remember that if you ask "What is the mass of a single point $m(\{x\})$?" the answer would be 0, but one can assign a density to a single point, which could be nonzero.

We can do the same thing with continuous random variables by replacing "mass" with "probability". We can then introduce the density at the point $a \in \mathbb{R}$ (if it exists) as

$$p_X(a) := \lim_{\substack{a \in I \\ |I| \to 0}} \frac{\mathbb{P}(X \in I)}{|I|}.$$

Here, I is an interval containing x and $|I|$ is its length.

As in our physical example, for a subset $A \subseteq \mathbb{R}$ we have

$$\mathbb{P}_X(A) = \mathbb{P}(X \in A) = \int_A p_X(x)\, dx.$$

Definition 3.12 (Probability Density Function)**.** Let X be a random variable. If there exists a (measurable) function $p_X : \mathbb{R} \to [0, \infty)$, such that

$$\mathbb{P}_X(I) = \mathbb{P}(X \in I) = \int_I p_X(x)\, dx$$

for all intervals I in $\mathbb{R}$, we call it the *probability density function* (PDF) of X.

Notice that $\int_{\mathbb{R}} p_X(x)\, dx = 1$, since $\mathbb{P}_X(\mathbb{R}) = 1$. The relation between CDF and density is as you might guess:

$$F_X(x) = \mathbb{P}(X \leq x) = \int_{-\infty}^{x} p_X(t)\, dt.$$

Consequently, if F_X is differentiable, it holds that

$$\frac{dF_X}{dx}(x_0) = p_X(x_0). \tag{3.2}$$

Here, we bring some famous continuous random variables and their densities:

- **Uniform** We write $X \sim \text{Unif}([a, b])$, with $a, b \in \mathbb{R}$ and $a < b$, if $X \in [a, b]$ and

  $$p_X(x) = \begin{cases} c & \text{if } x \in [a, b] \\ 0 & \text{otherwise,} \end{cases}$$

  where c can be determined from a and b using

  $$\int_a^b c\, dx = c(b - a) = 1,$$

  hence

  $$c = \frac{1}{b - a}.$$

  The uniform random variable can be used to model events where intervals of the same length are equally likely.

- **Exponential** We write $X \sim \text{Exp}(\lambda)$, with $\lambda > 0$, if $X \in \mathbb{R}$ and

  $$p_X(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

An exponential random variable can be used, for example, to model the amount of time before a piece of equipment breaks down. Notice that the probability that X exceeds a certain value $x \geq 0$ decreases exponentially with growing values of x:

$$\Pr(X \geq a) = \int_a^\infty \lambda e^{-\lambda x} dx = -e^{-\lambda x}\Big|_a^\infty = e^{-\lambda a}.$$

- **Normal** We write $X \sim \mathcal{N}(\mu, \sigma)$, with $\sigma > 0$ and $\mu \in \mathbb{R}$, if $X \in \mathbb{R}$ and
$$p_X(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$
X is a *standard normal random variable* if $X \sim \mathcal{N}(0,1)$. The normal random variable plays a fundamental role in signal processing, since it generally models well the additive effect of many independent factors. Mathematically, this is captured by the *central limit theorem*, which states that the sum of a large number of independent random variables drawn from the same distribution tends to a normal distribution, even if the original variables were not normally distributed.

3.2.4 Expected Value

If we redo the same experiment many times and look at the average of the observed values of a random variable X, it starts to "converge" to a number, which we call the *expected value* of X. We will restate this fact in a rigorous way later. If we want to guess what this value would be for a discrete random variable, the following definition would make sense:

Definition 3.13 (Expected Value, Discrete)**.** Let X be a discrete random variable with values in $\mathcal{X}$. We define the *expected value of* X (or its *first moment*) as

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \cdot \mathbb{P}(X = x) = \sum_{x \in \mathcal{X}} x\, p_X(x).$$

If the above sum does not converge, we say X has no expected value.

For continuous random variables that have a PDF, the definition of expected value is as follows:

Definition 3.14 (Expected Value, Continuous)**.** Let X be a continuous random variable with density p_X. We define the expected value of X as[8]

$$\mathbb{E}[X] := \int_{\mathbb{R}} x\, p_X(x)\, dx.$$

Exercise 3.15 (St. Petersburg Problem)**.** Suppose you enter the following game: you flip a fair coin until it comes up heads. Let X denote the round that the coin turns up heads.

(a) What is the PMF of X and its expected value? This random variable follows the so-called *Geometric distribution*.

[8] The integral in Definition 3.14 can be infinite. An example is the random variable X with the PDF

$$p_X(x) = \frac{1}{\pi(1+x^2)}.$$

This distribution is called the Cauchy distribution, and looks surprisingly much like the normal distribution (see Figure 3.2). However, it has so-called "heavy tails", which means that the density does not become small fast enough as $|x| \to \infty$.

(b) Suppose you get a reward of 2^n Francs if the game stops at round n. What is the expected value of your reward?

The following theorem is sometimes called the "law of the unconscious statistician," as one thinks that it is trivial, while it is not. It is a nice exercise to prove it for the discrete case:

Theorem 3.16. *Let X be a random variable, and $g : \mathbb{R} \to \mathbb{R}$ be a function.*

- *If X is discrete,*
$$\mathbb{E}[g(X)] = \sum_{x \in \mathcal{X}} g(x)\, p_X(x).$$
- *If X is continuous having a PDF p_X, then*
$$\mathbb{E}[g(X)] = \int_{\mathbb{R}} g(x)\, p_X(x)\, dx.$$

3.2.5 Variance

The *variance* of a random variable, in contrast to its expectation, does not measure the location, but rather the spread.

Definition 3.17 (Variance)**.** Let X be a random variable. The variance of X is defined as
$$\operatorname{Var}(X) = \mathbb{E}\left[(X - \mathbb{E}[X])^2\right] = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2.$$

Note that the variance of X is finite, if and only if $\mathbb{E}\left[X^2\right]$ (often called *the second moment of X*) is finite. Also, note that $\operatorname{Var}(X) \geq 0$ by Jensen's inequality (Lemma 3.54).

3.3 Jointly Distributed Random Variables

In many occasions, there are several random variables defined over the same probability space, and we wish to study events that involve more than one of the variables. First notice that there can be "dependency" among the random variables. Take the following extreme example: in a die tossing experiment, let X be the top face and Y be the bottom. Clearly, if the die is fair, X and Y have the same distribution (both uniform on $\{1, \ldots, 6\}$). However, it is always the case that $X + Y = 7$, regardless of what happens. Hence, knowing X determines Y.

3.3.1 Joint Distributions

In the general case, to understand a set of random variables, there is no other way than asking for the probabilities of all possible events that concern all random variables.

When looking at two random variables X and Y at the same time, it is useful to pack them together and look at them as a function that maps every outcome of the experiment to a *vector* in $\mathbb{R}^2$,

that is, $\omega \in \Omega$ is mapped to $(X(\omega), Y(\omega)) \in \mathbb{R}^2$. The events (questions) that are related to both X and Y can be described as subsets of $\mathbb{R}^2$.

This viewpoint is the key to the definition of the joint distribution, which is very similar to Definition 3.5.

Definition 3.18 (Joint Distribution)**.** Let X, Y be two random variables defined on the same probability space $(\Omega, \mathcal{F}, \mathbb{P})$. The joint distribution of X, Y is the map $\mathbb{P}_{X,Y}$ that takes a subset A of $\mathbb{R}^2$ as input,[9] and gives the probability

$$\mathbb{P}_{X,Y}(A) = \mathbb{P}\left(\{\omega \in \Omega : (X(\omega), Y(\omega)) \in A\}\right)$$

[9] Just to be careful here, not all subsets of $\mathbb{R}^2$ are allowed. However, almost all of the subsets that are interesting for us are valid inputs to $\mathbb{P}_{X,Y}$ and we call these $\mathcal{B}(\mathbb{R}^2)$.

Example 3.19. Suppose we choose a person uniformly at random from a population Ω. We measure the person's weight in kilograms and call that X, and measure the person's height in meters and call it Y. The event "the person has a body-mass-index lower than 25" can be described using X and Y:

$$\left\{\omega \in \Omega : X(\omega)/Y(\omega)^2 < 25\right\} \subset \Omega.$$

The same event can be described as the set of all points (x, y) in $\mathbb{R}^2$ such that $\{x/y^2 < 25\}$. The probability function $\mathbb{P}$ gets the former set as input and gives a probability, while the probability distribution function $\mathbb{P}_{X,Y}$ gets the latter set as input and give out the same number, that is,

$$\mathbb{P}\left(\left\{X/Y^2 < 25\right\}\right) = \mathbb{P}_{X,Y}\left(\left\{(x, y) \in \mathbb{R}^2 : x/y^2 < 25\right\}\right).$$

It turns out that the *joint cumulative distribution function* is sufficient to identify the joint distribution.

Definition 3.20 (Joint CDF)**.** Given a collection of random variables $X_1, \ldots, X_n$ defined on the same probability space, their joint CDF is defined as

$$F_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = \mathbb{P}\left(X_1 \leq x_1, \ldots, X_n \leq x_n\right)$$

for all $x_1, \ldots, x_n \in \mathbb{R}$.[10]

[10] Notice the notation:

$$\mathbb{P}\left(X = x, Y = y\right) = \mathbb{P}\left(\{X = x\} \cap \{Y = y\}\right).$$

In what follows, we treat the discrete and continuous random variables separately, and show how the joint distribution can be described in different ways.

Let X, Y be two discrete random variables taking values in $\mathcal{X}, \mathcal{Y}$ respectively. It turns out that probabilities of the form $\mathbb{P}\left(X = x, Y = y\right)$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are sufficient to give a complete characterization of X and Y.

Definition 3.21 (Joint PMF)**.** Let $X_1, \ldots, X_n$ be discrete random variables, defined on the same probability space, and taking values in $\mathcal{X}_1, \ldots, \mathcal{X}_n$. The function

$$p_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = \mathbb{P}\left(X_1 = x_1, \ldots, X_n = x_n\right)$$

defined over $\mathcal{X}_1 \times \cdots \times \mathcal{X}_n$ is called the *joint probability mass function of* $X_1, \ldots, X_n$. We write $p(x_1, \ldots, x_n)$ if the random variables are clear from the context.

With the same ideas described in Section 3.2.3, we can define the joint PDF of a set of continuous random variables (if it exists):

Definition 3.22 (Joint PDF)**.** Let the random variables $X_1, \dots, X_n$ be defined on the same probability space. We say these random variables have a *joint probability density function* $p_{X_1,\dots,X_n}$ if for all subsets $A \in \mathcal{B}(\mathbb{R}^n)$ it holds

$$\mathbb{P}_{X_1,\dots,X_n}(A) = \mathbb{P}\left((X_1, \dots, X_n) \in A\right) = \int_A p_{X_1,\dots,X_n}(x_1, \dots, x_n)\, dx_1 \cdots dx_n.$$

This definition again emphasizes the role of the density: to compute the probability of events about a set of random variables, one can integrate the density.

The same relation with CDF holds in jointly distributed random variables:

Theorem 3.23. *If the random variables $X_1, \dots, X_n$ with joint CDF F have a joint density at $\boldsymbol{x} \in \mathbb{R}^n$, then*

$$p_{X_1,\dots,X_n}(\boldsymbol{x}) = \frac{\partial^n F}{\partial x_1 \cdots \partial x_n}(\boldsymbol{x}).$$

Not all collections of continuous random variables have a joint density:

Example 3.24. Let X be uniformly distributed on $[0, 1]$ and let $Y = X$. Then X, Y do *not* have a joint density.

Remark. Let X be a continuous and Y be a discrete random variable, both defined on the same probability space. We can still describe the joint distribution by the joint CDF. However, things might get complicated, as something as straightforward as PMF or PDF does not exist in this case. In this course, whenever this situation of mixed joint random variables occurs, it is better to think in terms of conditional distributions, see Section 3.3.5.

Sometimes we talk about a *random vector*. A random vector $\boldsymbol{X} = (X_1, ..., X_n)$ is just a vector made up of random variables X_i defined on the *same* probability space. X_i can have a joint distribution, density and so on. This notion brings forward the fact that every realization is a point in $\mathbb{R}^n$ and bears some geometric meaning.

3.3.2 *Marginals*

The joint distribution of $X_1, \dots, X_n$ encodes the distribution of every one of X_i's too.

Definition 3.25 (Marginal Distribution)**.** Suppose we are given the joint distribution $\mathbb{P}_{X,Y}$ of X and Y. Then, for any $A \in \mathcal{B}(\mathbb{R})$, one can compute the distribution $\mathbb{P}_X$ as

$$\mathbb{P}_X(A) = \mathbb{P}_{X,Y}(A \times \mathbb{R}) = \mathbb{P}_{X,Y}\left(\left\{(x, y) \in \mathbb{R}^2 : x \in A\right\}\right).$$

In this context, $\mathbb{P}_X$ is sometimes referred to as the marginal distribution of X. The act of computing $\mathbb{P}_X$ from $\mathbb{P}_{X,Y}$ is called *marginalization on* X. If the joint CDF of X, Y is given, then the CDF of X can be computed as

$$F_X(x) = F_{X,Y}(x, +\infty).$$

If instead of the distribution or CDF, we are given a joint PMF or PDF, we can marginalize as follows:

Let X, Y be discrete random variables. Fix $x \in \mathcal{X}$, and compute

$$\sum_{y \in \mathcal{Y}} p(x,y) = \sum_y \mathbb{P}\left(X = x, Y = y\right) = \mathbb{P}\left(\{X = x\} \cap \left(\bigcup_y \{Y = y\}\right)\right),$$

and notice that $\bigcup_y \{Y = y\} = \Omega$, and hence

$$\sum_{y \in \mathcal{Y}} p(x,y) = \mathbb{P}\left(\{X = x\} \cap \Omega\right) = \mathbb{P}\left(X = x\right) = p_X(x).$$

Thus, summing over all values of Y of the joint PMF gives the PMF of X.

With continuous random variables one has to "integrate out" the other variable. For the case of two continuous random variables X, Y with joint density $p_{X,Y}$, the (marginal) density of X is given by

$$p_X(x) = \int_{\mathbb{R}} p_{X,Y}(x,y)dy.$$

Marginalization can be easily generalized to a set of random variables. Let $X_1, \ldots, X_n$ be discrete random variables taking values in $\mathcal{X}_1, \ldots, \mathcal{X}_n$ and joint PMF p. If we want to know the joint distribution of a subset $I \subset \{1, \ldots, n\}$, we have to sum over all values of the random variables not in I. For example, if $I = \{1, \ldots, k\}$, then for any $(x_1, \ldots, x_k) \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_k$ we have

$$p_{X_1,\ldots,X_k}(x_1, \ldots, x_k) = \sum_{x_{k+1} \in \mathcal{X}_{k+1}} \cdots \sum_{x_n \in \mathcal{X}_n} p_{X_1,\ldots,X_n}(x_1, x_2, \ldots, x_n).$$

The same holds for continuous random variables $X_1, \ldots, X_n$ with joint density $p_{X_1,\ldots,X_n}$. The marginal PDF of $X_1, \ldots, X_k$ is

$$p_{X_1,\ldots,X_k}(x_1, \ldots, x_k) = \int \cdots \int p_{X_1,\ldots,X_n}(x_1, x_2, \ldots, x_n)\, dx_{k+1} \cdots dx_n.$$

3.3.3 *Independence and Conditional Probability of Events*

First, an intuitive primer on independence is in order. Consider infinitely many tosses of a fair coin. If you look at the ratio of heads in the first n tosses, this number converges to $\frac{1}{2}$ as n increases. Now what if you just look at the tosses in even rounds? What if you just look at the times which are prime, i.e., $2, 3, 5, 7, 11, \ldots$? Our intuition tells us that the ratio of heads is still $\frac{1}{2}$. But now what if we look at the times when the coin has come Heads? The ratio now is going to be 1.

More generally, let A and B be two events. We do the experiment multiple times and try to estimate $\mathbb{P}(A)$ by the ratio of the number of experiments that A happened to the total number of experiments. Now suppose we only look at those experiments that B has happened, and estimate the probability of $\mathbb{P}(A)$ on those. If the estimate changes, it means that A and B are dependent.

Definition 3.26 (Independence and Conditional Probability)**.** Let A, B be two events.[11]

(1) A and B are *independent* iff $\mathbb{P}(A, B) = \mathbb{P}(A)\mathbb{P}(B)$.

(2) If $\mathbb{P}(B) > 0$, the *conditional probability of A given B* is defined as

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A, B)}{\mathbb{P}(B)}.$$

(3) Let $A_1, \ldots, A_n$ be events. They are called independent, if for any $k \leq n$ and every k indices $1 \leq i_1 < \cdots < i_k \leq n$,

$$\mathbb{P}\left(A_{i_1}, \ldots, A_{i_k}\right) = \mathbb{P}\left(A_{i_1}\right) \cdots \mathbb{P}\left(A_{i_k}\right)$$

[11] We use comma (,) interchangeably with $\cap$ when writing down probabilities of events. So $\mathbb{P}(A, B)$ reads "probability of A and B" and is the same as $\mathbb{P}(A \cap B)$.

Example 3.27. In this example, we show that pairwise independence of events does not imply their independence. Throw a red and a blue fair coin. Let A, B, C be the following events: A happens when the red coin turns Heads, B happens when the blue coin turns Heads, and C happens when exactly one of the coins is Heads. Verify that A, B, C are pairwise independent, but $\mathbb{P}(A, B, C) = 0 \neq \mathbb{P}(A)\mathbb{P}(B)\mathbb{P}(C) = \frac{1}{8}$.

Exercise 3.28. Construct an experiment and n events such that each $(n-1)$ of them are independent, but they are not independent.

Here we bring a list of theorems that concern conditional probabilities and are useful in computations and proofs.

Let A, B be events and $A_1, ..., A_n$ be a partition of the sample space Ω into events, i.e., $\Omega = \bigcup_{i=1}^{n} A_i$ and $A_i \cap A_j = \emptyset$ for all $i \neq j$. Suppose that $\mathbb{P}(A_i) > 0$ for all i and $\mathbb{P}(B), \mathbb{P}(A) > 0$. Take any arbitrary events $C_1, ..., C_n$ with $\mathbb{P}(C_i) > 0$ for all i.

Theorem 3.29 (Law of Total Probability)**.** *One has*

$$\mathbb{P}(B) = \sum_{i=1}^{n} \mathbb{P}(B \mid A_i)\mathbb{P}(A_i).$$

Theorem 3.30 (Bayes Rule)**.** *One has*

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(B \mid A)\mathbb{P}(A)}{\mathbb{P}(B)}.$$

Theorem 3.31 (Chain Rule)**.** *It holds that*[12]

$$\mathbb{P}(C_1, \ldots, C_n) = \mathbb{P}(C_1)\mathbb{P}(C_2 \mid C_1) \cdots \mathbb{P}(C_n \mid C_1, \ldots, C_{n-1})$$

[12] Be careful of the notation again: the right hand side of this formula is indeed $\prod_{i=1}^{n} \mathbb{P}\left(C_i \middle| \bigcap_{j=1}^{i-1} C_j\right)$.

Exercise 3.32. We have two coins that look similar, one is unbiased (probability of Heads is $\frac{1}{2}$), and the other one is biased (probability of Heads is $\frac{1}{3}$). We pick one of them at random and throw it twice. The result was HT. What is the probability that we have picked the unbiased coin?

3.3.4 Independence of Random Variables

Next, we mention the concept of *independence* for random variables.

Definition 3.33 (Independence of Random Variables)**.** Two random variables X, Y are called independent, if their joint CDF factorizes. That is, for all $x, y \in \mathbb{R}$,

$$F_{X,Y}(x, y) = F_X(x)\, F_Y(y).$$

The following theorem summarizes equivalent conditions under which random variables are guaranteed to be independent:

Theorem 3.34. *Let X and Y be random variables. The following are equivalent:*

- *X and Y are independent.*
- *If X and Y are both discrete, their joint PMF factorizes: $p_{X,Y}(x, y) = p_X(x)p_Y(y)$ for all $x \in \mathcal{X}, y \in \mathcal{Y}$.*
- *If X and Y are both continuous and have a joint PDF, their joint PDF factorizes: $p_{X,Y}(x, y) = p_X(x)p_Y(y)$ for all $x, y \in \mathbb{R}$.*

Extension to more than two random variables is similar. For example, the continuous random variables $X_1, \ldots, X_n$ are independent iff

$$p_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = p_{X_1}(x_1) \cdots p_{X_n}(x_n)$$

for all $x_1, \ldots, x_n \in \mathbb{R}$.

3.3.5 Conditional Distribution

Given the joint distribution of two (or more) random variables, now we answer the question: "how much the knowledge about a random variable influence our belief/probabilities about others." To be more precise, we expect that if there is dependency among random variables, information about one "changes" the distribution of the others. This concept is formalized as *conditional distribution*. We avoid discussing this concept in full generality and bring three different cases that are of our interest.

3.3.6 Conditional Distribution: Discrete Case

Definition 3.35 (Conditional PMF)**.** Let X, Y be two discrete random variables on the same probability space taking values in $\mathcal{X}$ and $\mathcal{Y}$ respectively. For $y \in \mathcal{Y}$ such that $\mathbb{P}(Y = y) \neq 0$ we define the *conditional probability mass function of X given $Y = y$* as

$$p_{X|Y}(x \mid y) := \frac{p_{X,Y}(x, y)}{p_Y(y)}.$$

Clearly, if X and Y are independent, then $p(x \mid y) = p(x)$.

Notice that for any fixed $y \in \mathcal{Y}$ with $\mathbb{P}(Y = y) > 0$, the PMF $p_{X|Y}(\cdot \mid y)$ satisfies $p_{X|Y}(x \mid y) \geq 0$ for all $x \in \mathcal{X}$ and

$$\sum_{x \in \mathcal{X}} p_{X|Y}(x \mid y) = \sum_{x \in \mathcal{X}} \frac{p_{X,Y}(x,y)}{p_Y(y)} = \frac{1}{p_Y(y)} \sum_x p_{X,Y}(x,y) = 1$$

by definition. Thus, we can consider $X \mid Y = y$ as a random variable itself distributed with this PMF, and call it X *conditioned on* $Y = y$.

The notion of conditional distribution can be generalized to any number of random variables. Let $X_1, X_2, \ldots, X_n$ be discrete random variables taking values in $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_n$. For any $x_1 \in \mathcal{X}_1, x_2 \in \mathcal{X}_2, \ldots, x_k \in \mathcal{X}_k$ such that $\mathbb{P}(X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k) > 0$, the conditional distribution of $X_{k+1}, \ldots, X_n$ given $X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k$ is defined in terms of its PMF

$$p_{X_{k+1},\ldots,X_n|X_1,\ldots,X_k}(x_{k+1}, \ldots, x_n \mid x_1, \ldots, x_k) := \frac{p_{X_1,X_2,\ldots,X_n}(x_1, x_2, \ldots, x_n)}{p_{X_1,\ldots,X_k}(x_1, \ldots, x_k)}.$$

3.3.7 Conditional Distribution: Continuous Case

Here, we assume both X and Y are continuous. Recall that since Y is a continuous random variable, it holds that $\mathbb{P}(Y = y) = 0$ for all $y \in \mathbb{R}$. Therefore, the discrete definition that is based on probability mass would not be sensible. Instead, we define the *conditional density*.

Definition 3.36 (Conditional Density)**.** Let X, Y be two continuous random variables with joint density $p_{X,Y}$. The conditional density of X given $Y = y$ is defined as

$$p_{X|Y}(x \mid y) = \begin{cases} \frac{p_{X,Y}(x,y)}{p_Y(y)} & \text{if } p_Y(y) > 0 \\ 0 & \text{else.} \end{cases}$$

As in the discrete case, $X \mid Y = y$ is a new random variable with the PDF above,[13] and is called X conditioned on $Y = y$. If it is clear from the context, we sometimes drop the subscript of $p_{X|Y}(x \mid y)$ and just write $p(x \mid y)$.

[13] It is easy to verify that for a fixed $y \in \mathbb{R}$ such that $p_Y(y) > 0$, the function $p(\cdot \mid y)$ defined above is indeed a probability density.

Similar to the discrete case, the notion of conditional distribution can be extended to the case of more than two continuous random variables.

3.3.8 Conditional Distribution: Mixed Case

There are some cases where we study two (or more) random variables where some of them are continuous and some are discrete.

Suppose X is a continuous and Y is a discrete random variable, defined on the same probability space. For each $y \in \mathcal{Y}$, we have a conditional probability distribution $\mathbb{P}_{X|y}$ on $\mathbb{R}$, defined as follows: for each subset $A \in \mathcal{B}(\mathbb{R})$ we have

$$\mathbb{P}_{X|y}(A) = \frac{\mathbb{P}_{X,Y}(\{(x,y) \in \mathbb{R}^2 : x \in A\})}{\mathbb{P}_Y(\{Y = y\})} = \frac{\mathbb{P}(X \in A, Y = y)}{p_Y(y)}$$

When the conditional distribution above has a density, we call it *the conditional probability density of X given $Y = y$* and denote it as $p_{X|Y}(x \mid y)$. In this course, we can always assume that such a density exists for all $y \in \mathcal{Y}$.

What if we want to condition on $X = x$? In this case, the conditional distribution will be discrete, and Bayes theorem (Theorem 3.30) comes to the rescue:

$$p_{Y|X}(y \mid x) = \frac{p_{X|Y}(x \mid y)p_Y(y)}{p_X(x)}, \quad \forall y \in \mathcal{Y}.$$

Example 3.37 (Three radioactive materials)**.** Suppose we have three different radioactive materials y_1, y_2, y_3, each having a different rate of particle emission, and we mixed different quantities of these materials in a batch. Let X be the (random) time that a particle is emitted from the batch and let Y be the type of the material that caused the emission. Clearly, X is a continuous random variable, while Y is a discrete one. Moreover, there is a clear dependency between X and Y: knowing Y will determine the material, which as a result, changes the distribution of X.

Assume we observed a particle emitted from the batch at time x. A question one may ask is to determine how our perception of the distribution of Y changes after obtaining this new knowledge about X, i.e., we would like to know how likely it is that the materials y_1, y_2, y_3 were the cause of the emission knowing that the emission time was x. This can be translated in mathematical terms to finding the conditional distribution in terms of its PMF $p_{Y|X}(y \mid x)$ for the three possible values of Y.

At first, this may look difficult: we do not have an explicit description of $p_{Y|X}(y \mid x)$. However, we can exploit the knowledge of another conditional distribution which we know well: the conditional PDF $p_{X|Y}(x \mid y)$, i.e., the PDF of the emission time for each of the radioactive materials.

The two conditionals are related via Bayes theorem:

$$p_{Y|X}(y \mid x) = \frac{p_{X|Y}(x \mid y)p_Y(y)}{p_X(x)}. \tag{3.3}$$

In Equation 3.3 the PMF $p_Y(y)$ represents our prior beliefs about the distribution of Y, i.e., what we believed the chances of y_1, y_2, y_3 to emit a particle were before observing any particle emission. Using Equation 3.3 we would like to update our prior belief by leveraging the observed emission time x and obtain a posterior distribution in terms of its PMF $p_{Y|X}(y \mid x)$. The only term that remains unknown in Equation 3.3 is the PDF $p_X(x)$, which can be obtained using the law of total probability (Theorem 3.38):

$$p_X(x) = \sum_{y \in \mathcal{Y}} p_{X|Y}(x \mid y)p_Y(y).$$

3.3.9 *Theorems on Conditional Distribution*

Here we bring the random variable version of the results in Section 3.3.3. In what follows, X and Y are random variables, and p can be a probability density, or a probability mass function, depending on the context. If nothing is stated about p, it can be either of them.

Theorem 3.38 (Law of Total Probability). *If Y is continuous with a density, then*

$$p_X(x) = \int p_{X|Y}(x \mid y)\, p_Y(y)\, dy.$$

If Y is discrete, then

$$p_X(x) = \sum_{y \in \mathcal{Y}} p_{X|Y}(x \mid y)\, p_Y(y).$$

Theorem 3.39 (Bayes Rule). *If X is continuous with a density,*

$$p_{X|Y}(x \mid y) = \frac{p_{Y|X}(y \mid x)\, p_X(x)}{\int p_{Y|X}(y \mid x')\, p_X(x')\, dx'}$$

If X is discrete, the integral is replaced by a sum over $x' \in \mathcal{X}$.

Theorem 3.40 (Chain Rule). *Let $X_1, \ldots, X_n$ be random variables. Then*

$$p(x_1, \ldots, x_n) = p(x_1)\, p(x_2 \mid x_1) \cdots p(x_n \mid x_1, \ldots, x_n).$$

3.3.10 Conditional Independence

Let X, Y, Z be random variables. The random variables X and Y are said to be *conditionally independent given* Z if given any value of Z, the probability distribution of X is the same for all values of Y and the probability distribution of Y is the same for all values of X. That is,

$$p_{X,Y|Z}(x, y \mid z) = p_{X|Z}(x \mid z) p_{Y|Z}(y \mid z)$$

for all x, y, z.

3.4 Properties of Expectation

3.4.1 Expectation of Functions of Several Random Variables

Suppose X, Y are two discrete random variables and let $f : \mathbb{R}^2 \to \mathbb{R}$ be a function, whose job is to take the values of X and Y and produce a number. By the same argument as in Theorem 3.16, we have

$$\mathbb{E}\left[f(X, Y)\right] = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p_{X,Y}(x, y) f(x, y).$$

The same holds for continuous random variables. If $X_1, \ldots, X_n$ are continuous random variables and $f : \mathbb{R}^n \to \mathbb{R}$ is a function, then

$$\mathbb{E}\left[f(X_1, \ldots, X_n)\right] = \int_{\mathbb{R}^n} f(\boldsymbol{x})\, p(\boldsymbol{x})\, d\boldsymbol{x},$$

where $p(\boldsymbol{x})$ is the joint density of $X_1, \ldots, X_n$ evaluated at $\boldsymbol{x} = (x_1, \ldots, x_n)$.

A special case is when $f(\boldsymbol{x}) = x_1 + \cdots + x_n$. Then, regardless of X_i being independent or not, we have

$$\mathbb{E}\left[X_1 + \cdots + X_n\right] = \mathbb{E}\left[X_1\right] + \cdots + \mathbb{E}\left[X_n\right].$$

This property, together with the fact that for $\alpha \in \mathbb{R}$, $\mathbb{E}[\alpha X] = \alpha \mathbb{E}[X]$, constitute the *linearity of expectation*.

In the special case where X and Y are independent, the expectation also respects products, as the following theorem states.

Theorem 3.41. *Let X, Y be independent random variables. Then*

$$\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y].$$

3.4.2 Conditional Expectation

Conditional expectation in its simplest form is the expected value of the conditional distribution. To be more precise, let X, Y be random variables. Suppose we want to condition on $Y = y$ and see what is the expected value of X given this information. Using the conditional distribution, this is an easy task: If X is continuous and a conditional density exists, then

$$\mathbb{E}[X \mid Y = y] := \int x\, p_{X|Y}(x \mid y)\, dx.$$

The same holds if X is discrete; in that case, we replace the integral by a sum over $x \in \mathcal{X}$.

Notice that for each value of Y, we can compute the conditional expectation. This naturally introduces a new random variable:

$$\omega \in \Omega \mapsto Y(\omega) \mapsto \mathbb{E}[X \mid Y = Y(\omega)].$$

We define $\mathbb{E}[X \mid Y]$ to denote this new random variable and call it *the conditional expectation of X given Y*. Note that if Y is discrete, then this random variable is discrete, and if it is continuous, it can be discrete or continuous (depending on what values $\mathbb{E}[X \mid Y = y]$ can produce for different values of $y \in \mathbb{R}$). In any case, the conditional expectation is a *function* of the random variable Y whose value at $Y = y$ is $\mathbb{E}[X \mid Y = y]$. This viewpoint of treating the conditional expectation as a random variable is very important and is ubiquitous in the course.

A very important property of conditional expectation is the following theorem, that is incredibly useful in calculations:

Theorem 3.42 (Tower Property)**.** *Let X, Y be two random variables. Then*

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]].$$

The proof follows from the definition of conditional expectation and Theorem 3.16.

If Y is discrete, the theorem reads:

$$\mathbb{E}[X] = \sum_{y \in \mathcal{Y}} \mathbb{E}[X \mid Y = y]\; p_Y(y),$$

and if Y is continuous,

$$\mathbb{E}[X] = \int \mathbb{E}[X \mid Y = y]\; p_Y(y)\, dy.$$

The following exercise is from Ross 2014.

Exercise 3.43. Suppose that the number of people entering a store on a given day is a random variable with mean 50. Suppose further that the amounts of money spent by these customers are independent random variables having a common mean of \$8. Finally, suppose that the amount of money spent by a customer is also independent of the total number of customers who enter the store. What is the expected amount of money spent in the store on a given day?

Solution. Let $X_1, X_2, \dots$ denote the money spent by each customer, and let N be the number of customers entering the shop. What we want is to compute the expected value of $\sum_{i=1}^{N} X_i$. For this computation, we first compute the conditional expectation given on $N = n$, then use the tower property.

$$\begin{aligned}
\mathbb{E}\left[\sum_{i=1}^{N} X_i \mid N = n\right] &= \mathbb{E}\left[\sum_{i=1}^{n} X_i \mid N = n\right] \\
&= \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] && X_i \text{ and } N \text{ are independent} \\
&= \sum_{i=1}^{n} \mathbb{E}[X_i] && \text{linearity of expectation} \\
&= 8n.
\end{aligned}$$

Hence,

$$\begin{aligned}
\mathbb{E}\left[\sum_{i=1}^{N} X_i\right] &= \mathbb{E}\left[\mathbb{E}\left[\sum_{i=1}^{N} X_i \mid N\right]\right] && \text{tower property} \\
&= \mathbb{E}[8N] \\
&= 400.
\end{aligned}$$

What we have shown is that $\mathbb{E}\left[\sum_{i=1}^{N} X_i\right] = \mathbb{E}[N]\,\mathbb{E}[X_1]$. □

3.4.3 *Conditional Expectation and Variance Notations*

In this small section, we bring a new notation that is not standard in the literature, but is intuitive and useful in our study.

Oftentimes in the lecture, we deal with expectations or variances of functions of several variables (random or deterministic). In some cases, we use a subscript to indicate which variable is being averaged over (as opposed to the default case where we average over *all* variables; see Subsection 3.4.1). For instance,

$$\mathbb{E}_X[f(X, Y)]$$

denotes the average of the function f with respect to the (marginal) distribution of X. This notation propagates further to the variance, where

$$\mathrm{Var}_X(f(X, Y)) = \mathbb{E}_X\left[(f(X, Y) - \mathbb{E}_X[f(X, Y)])^2\right]$$

similarly denotes the variance of the function f with respect to the (marginal) distribution of X. Notice that $\mathbb{E}_X[f(X, Y)]$ or $\mathrm{Var}_X(f(X, Y))$ are functions of Y. Moreover, Y can be a random variable, or a parameter. If Y is a random variable and we want to

average over conditional distribution given $Y = y$, we sometimes write $\mathbb{E}_{X|y}[f(X,Y)]$ or $\mathrm{Var}_{X|y}(f(X,Y))$.

In the following list, we bring these notational examples and their meaning for the expectation, which similarly translate to the variance:

1. $\mathbb{E}_X[f(X,Y)] := \int f(x,Y)\, p_X(x)\, dx$. If Y is a deterministic variable, then this is a function of Y, otherwise, this becomes a random variable which is a function of Y.
2. $\mathbb{E}_{X|y}[f(X,Y)] := \int f(x,y)\, p_{X|Y}(x \mid y)\, dx$.

3.4.4 Covariance

When studying two or more random variables, we sometimes want a quantitative measure of how dependent they are. The covariance is an example of such a measure, that quantizes *linear* dependency between random variables.

Definition 3.44 (Covariance)**.** Let X and Y be random variables. The covariance of X and Y is defined as

$$\mathrm{Cov}(X,Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\,\mathbb{E}[Y].$$

Note that $\mathrm{Cov}(X,X) = \mathrm{Var}(X)$. Moreover, covariance is linear in both of its arguments. That is, $\mathrm{Cov}(aX + bY, Z) = a\,\mathrm{Cov}(X,Z) + b\,\mathrm{Cov}(Y,Z)$ for $a, b \in \mathbb{R}$.

For random vectors $\boldsymbol{X} = (X_1, ..., X_p)^\top$ we define the expectation

$$\mathbb{E}[\boldsymbol{X}] = (\mathbb{E}[X_1], ..., \mathbb{E}[X_p])^\top \in \mathbb{R}^p$$

and the covariance matrix

$$\mathrm{Cov}(\boldsymbol{X}) = \mathbb{E}\left[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])^\top\right] \in \mathbb{R}^{p\times p}.$$

Note that the covariance matrix is of the form

$$\mathrm{Cov}(\boldsymbol{X}) = \begin{pmatrix} \mathrm{Var}(X_1) & \dots & \mathrm{Cov}(X_1, X_p) \\ \vdots & \ddots & \vdots \\ \mathrm{Cov}(X_p, X_1) & \dots & \mathrm{Var}(X_p) \end{pmatrix} \in \mathbb{R}^{p\times p}.$$

Exercise 3.45. Prove that for any random vector $\boldsymbol{X}$ with finite second moments, $\mathrm{Cov}(\boldsymbol{X})$ is symmetric and positive semi-definite.

3.5 Normal Distributions

One of the most important distributions a random variable or random vector can have is the *normal distribution*. In particular, it is the subject of the central limit theorem (Theorem 3.52), which we will discuss later.

Definition 3.46 (Normal Distribution)**.** Define $p_{\mu,\Sigma} : \mathbb{R}^p \to [0,\infty)$ as

$$p_{\mu,\Sigma}(x) = \frac{1}{\sqrt{(2\pi)^p \det \Sigma}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right)$$

for a vector $\mu \in \mathbb{R}^p$ and a symmetric, positive definite matrix $\Sigma \in \mathbb{R}^{p\times p}$. If a random vector $\boldsymbol{X}$ has the density $p_{\mu,\Sigma}$, we say that it follows a normal distribution with mean μ and covariance matrix Σ, and we write $\boldsymbol{X} \sim \mathcal{N}(\mu,\Sigma)$. Conveniently, it holds that if $\boldsymbol{X} \sim \mathcal{N}(\mu,\Sigma)$, $\mathbb{E}[\boldsymbol{X}] = \mu$ and $\mathrm{Cov}(\boldsymbol{X}) = \Sigma$.

Note that if $p = 1, \mu = 0$ and $\Sigma = \sigma^2 = 1$, this translates to

$$\phi(x) := p_{0,1}(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right).$$

We call this the standard normal distribution $\mathcal{N}(0,1)$ and denote its c.d.f. by Φ. Figure 3.2 visualizes the standard normal distribution and the Cauchy distribution.

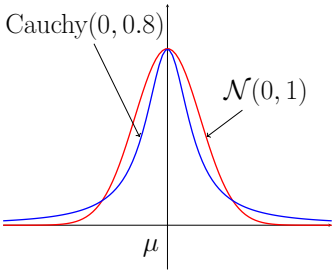


Figure 3.2: Densities of the normal distribution (red) and Cauchy distribution (blue) in comparison. Note the heavy tails of the Cauchy distribution.

Theorem 3.47 (Normal Distribution under Affine Transformations)**.** *Let $\boldsymbol{X} \sim \mathcal{N}(\mu,\Sigma)$. Then for any $A \in \mathbb{R}^{p\times p}$ and any $b \in \mathbb{R}^p$ it holds that*

$$AX + b \sim \mathcal{N}\left(A\mu + b, A\Sigma A^\top\right).$$

Let $1 \le k \le p$. By taking the matrix A as $A_{i,j} = 1$ if $i = j = k$ and $A_{i,j} = 0$ otherwise, we see the following corollary.

Bonus Material

Corollary 3.48 (Marginals of Normal Distributions)**.** *Let $\boldsymbol{X} \sim \mathcal{N}(\mu,\Sigma)$, with $\mu \in \mathbb{R}^p$ and $\Sigma \in \mathbb{R}^{p\times p}$. Then for any $1 \le k \le p$, it holds that*

$$X_k \sim \mathcal{N}(\mu_k, \Sigma_{kk}).$$

That is, the marginals of a normal distribution are also normal distributions.

Remark. The converse is *not* true. If you have two random variables X_1 and X_2 following normal distributions, (X_1, X_2) does not have to follow a two-dimensional normal distribution! For a collection of counterexamples, see Kowalski 1973.

3.6 Convergence of Empirical Averages to Expectation

In this section we deal with sequences of random variables and the behaviour of their average.

Definition 3.49 (i.i.d.)**.** The random variables $X_1, X_2, \ldots$ defined on the same probability space are called *independent with identical distribution (i.i.d.)* if they are independent and each X_i has the same distribution.

Let $X_1, X_2, \ldots$ be i.i.d. random variables with finite first moment. Recall the definition of the sample mean:

$$\bar{X}_n := \frac{1}{n}\sum_{i=1}^n X_i.$$

The laws of large numbers are central to many statistical procedures. Roughly speaking, they say that as the sample size n grows, the sample mean $\bar{X}_n$ converges to the expectation $\mathbb{E}[X_1]$. This is why the sample mean is often used as an *estimator* of the expectation.

Bonus Material

First, we need to define what convergence means for random variables.

Definition 3.50 (Convergence of Random Variables)**.** Let $\{X_n\}_{n\in\mathbb{N}}$ be a sequence of random variables and X another random variable. We say that

1. X_n converges to X *almost surely*, if

$$\mathbb{P}\left(\left\{\omega \in \Omega \mid \lim_{n\to\infty} X_n(\omega) = X(\omega)\right\}\right) = 1,$$

   and we write $X_n \xrightarrow{a.s.} X$ as $n \to \infty$.

2. X_n converges to X *in probability*, if for any $\varepsilon > 0$ we have

$$\lim_{n\to\infty} \mathbb{P}(|X_n - X| > \varepsilon) = 0$$

   and we write $X_n \xrightarrow{\mathbb{P}} X$ as $n \to \infty$.

3. X_n converges to X *in distribution* (a.k.a. in law), if for all continuity points x of F_X we have

$$\lim_{n\to\infty} F_{X_n}(x) = F_X(x)$$

   and we write $X_n \xrightarrow{law} X$ as $n \to \infty$.

   Without proving it here, it is worth noting that as $n \to \infty$,

$$X_n \xrightarrow{a.s.} X \implies X_n \xrightarrow{\mathbb{P}} X \implies X_n \xrightarrow{law} X.$$

   We can now proceed to derive the laws of the large numbers.

Theorem 3.51 (Laws of Large Numbers)**.** *Let $X_1, ..., X_n$ be i.i.d. random variables with finite first moment $\mu := \mathbb{E}[X_1]$. Then the weak law of large numbers (WLLN) states that as $n \to \infty$*

$$\bar{X}_n \xrightarrow{\mathbb{P}} \mu$$

and the strong law of large numbers (SLLN) states that as $n \to \infty$

$$\bar{X}_n \xrightarrow{a.s.} \mu.$$

From a statistical perspective, Theorem 3.51 means that the estimator $\bar{X}_n$ is *consistent*. It is noteworthy that $\bar{X}_n$ is also *unbiased*, because $\mathbb{E}[\bar{X}_n] = \mathbb{E}[X_1]$.

3.6.1 Central Limit Theorem

We will now come to a slightly more advanced result, called the *Central Limit Theorem* (CLT). Consider the setting of Theorem 3.51.

The law of large numbers states that the sum $S_n := \sum_{i=1}^n X_i$ with the scaling factor of $1/n$ *converges* to a fixed number (the expected value). This means that $1/n$ is small enough to make sure that S_n/n does not blow up, but simultaneously is not too small, which would make it converge to zero. The main idea of the CLT is that "interchanging $1/n$ with a slightly larger factor of $1/\sqrt{n}$ will make $S_n/\sqrt{n}$ also converge, but this time in distribution to a *normally distributed random variable*". What is so astonishing about this fact, is, that we make only very weak assumptions on the distribution of $X_1, ..., X_n$!

Bonus Material

Theorem 3.52 (Central Limit Theorem by Lindeberg-Lévy)**.** *Let $X_1, ..., X_n$ be i.i.d. random variables with finite first and second moments, i.e., their expectation μ and variance σ^2 exists. Then it holds that*

$$\frac{1}{\sqrt{n}} \sum_{i=1}^{n} (X_i - \mu) = \sqrt{n}(\bar{X}_n - \mu) \xrightarrow{\mathcal{D}} Z$$

where Z is a random variable with normal distribution $\mathcal{N}(0, \sigma^2)$.

Example 3.53 (Central Limit Theorem for Binomial Distribution)**.** Consider the case of $X_1, ..., X_n$ being i.i.d. $\mathrm{Ber}(p)$ random variables, i.e., $\mathbb{P}(X_i = 1) = 1 - \mathbb{P}(X_i = 0) = p$, $\mathbb{E}[X_i] = p$ and $\mathrm{Var}(X_i) = p(1-p)$. Then $S_n = \sum_{i=1}^n X_i$ is a random variable that follows the binomial distribution with parameters n and p. Therefore, S_n has expectation np and variance $np(1-p)$. By Theorem 3.52, for large enough n, this distribution should roughly be a normal distribution. Specifically,

$$\frac{1}{\sqrt{n}} S_n - \sqrt{n}p$$

should roughly be $\mathcal{N}(0, p(1-p))$ distributed. By rearranging, we see that S_n should roughly be $\mathcal{N}(np, np(1-p))$ distributed. Figure 3.3 visualises this phenomenon.

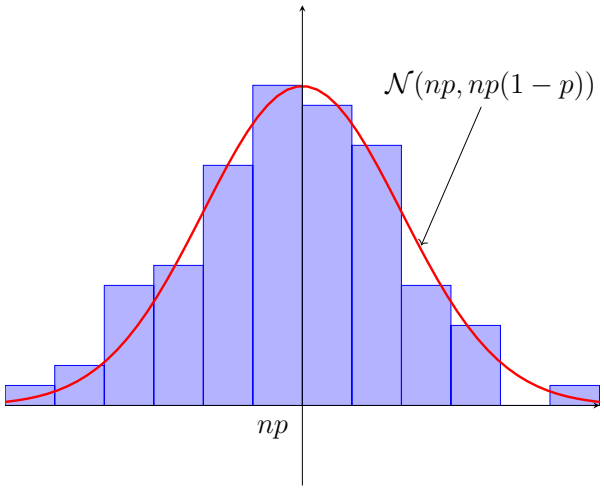


Figure 3.3: The central limit Theorem for the binomial distribution with parameters n and p. The figure shows the histogram (blue) of 100 realisations with $n = 100$ and $p = 0.5$. The respective normal distribution in red.

3.7 Useful Inequalities and Lemmas

Lemma 3.54 (Jensen's Inequality)**.** *Let X be a random variable with finite first moment and $g : \mathbb{R} \to \mathbb{R}$ a convex function. Then*

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

Lemma 3.55 (Chebychev-Markov Inequality)**.** *Let X be a random variable and $g : [0, \infty) \to [0, \infty)$ an increasing function with $g(x) > 0$ for all $x > 0$. Then it holds for all $\varepsilon > 0$ that*

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

Proof. It holds that

$$\begin{aligned} g(\varepsilon)\mathbb{P}(|X| \geq \varepsilon) &= \int g(\varepsilon)\mathbb{I}_{\{|X| \geq \varepsilon\}} d\mathbb{P} \\ &\leq \int g(|X|)\mathbb{I}_{\{|X| \geq \varepsilon\}} d\mathbb{P} \\ &\leq \int g(|X|) d\mathbb{P} \\ &= \mathbb{E}[g(|X|)] \end{aligned}$$

where we used that g is increasing in the second inequality and $g \geq 0$ in the last inequality. □

Part II

Supervised Learning

4
Linear Regression

This chapter introduces basic terminology for supervised learning and covers the main concepts related to linear regression and some of its variants.

Roadmap

In Section 4.1 we introduce supervised learning and basic terminology related to supervised learning and revisit the three main components of a machine learning method. We then introduce multiple linear regression as our first learning algorithm in Section 4.2. We derive its closed-form solution in both the under- and over-parameterized setting and introduce the minimum-norm solution. We end this section by deriving a universal expression for the multiple linear regression estimator that is valid in both settings. In the last section, we discuss a simple method that adds non-linearity to regression problems via feature maps.

Learning Objectives

After reading this chapter you should know:

- Basic machine learning terminology related to supervised learning.
- The fundamental components of a machine learning pipeline.
- How to derive the closed-form solution for linear regression in the under- and over-parameterized case.
- Geometric properties of multiple linear regression.
- Feature maps, how they introduce non-linearity, and how to solve the corresponding regression problem.

4.1 Supervised Learning Terminology and the ML Pipeline

Supervised learning is the task of learning a functional relationship between the *input* and the *output* based on example inputs and their corresponding outputs. For instance, one might be interested in predicting brain age from MRI images, or estimating house prices based on some given attributes such as area, number of bedrooms, location, etc. In the above instances, the input is the MRI image or the house properties, and the output is the brain age

or the price of the house, respectively. In both problems, we need a set of examples (age and MRI images of a group of people, or some houses with known attributes and price), and the goal is to learn the relationship between input and output from these sets of examples.

We now introduce general terminology that is commonly used in supervised learning and that we use throughout the notes.

(Input) attributes/covariates The (raw) attributes $\boldsymbol{x} \in \mathbb{R}^d$ (where d is the number of inputs) of a single example, often represented by a real-valued vector, that are used as input by the machine learning method.

Features Variables belonging to a single example, usually represented by a real-valued vector $\phi(x) \in \mathbb{R}^p$ (where p is the number of the features) that is a (nonlinear) transformation of the original input attributes.[1]

[1] In linear regression, the raw inputs $x \in \mathbb{R}^d$ fed into the model without additional (nonlinear) transformations. Thus, in linear regression, we use the terms "inputs" and "features" interchangeably. This will not be the case when we study nonlinear regression or, later, neural networks.

Outputs/Targets/Labels The output value $y \in \mathbb{R}$ assigned to each example in the problem.

Training set A set of (input, output) pairs from which we learn the functional relationship between input and output.

Test set A set of (input, output) pairs disjoint from the training set which we use to evaluate the quality of the learned functional relationship.

Predictor/Model A function $f : \mathbb{R}^p \to \mathbb{R}$ that maps input features into output. Note that in the statistics literature, a predictor usually refers to the features/covariates.

Loss function A function that quantifies the difference/distance between the predicted value and the true output value of an example. This loss helps us assess how good/bad the predicted values of the learned model are. We will denote by $\ell(f(\boldsymbol{x}), y)$ the (pointwise) loss function between the predicted value $f(\boldsymbol{x})$ and the output value y.

The ML pipeline. On a high level, a machine learning method can often be characterized by specific choices for the following components:

Function class F A (parameterized) set of functions within which we look for a suitable one that adequately "fits" our training data and achieves good "generalization ability" on unseen data.

Training loss $L(f)$ The average loss of a predictor $f \in F$ over the training set,

$$L(f) := \frac{1}{n} \sum_{i=1}^{n} \ell(f(\boldsymbol{x}_i), y_i).$$

Optimization method A method to (approximately) minimize the training loss L and learn the final predictor.

In this chapter, we start with the simple example of linear function classes and introduce the concept of loss minimization as a way to obtain one of the simplest machine learning model.

4.2 Multiple Linear Regression

Multiple linear regression uses multivariate inputs $\boldsymbol{x} \in \mathbb{R}^d$ that can be thought of as a d-dimensional representation of a sample. Moreover, multiple linear regression employs the parameterized class of all affine functions that can be expressed as $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + w_0$ for some *parameter* or *weight* vector $\boldsymbol{w} \in \mathbb{R}^d$ (the slope in 1-d) and intercept $w_0 \in \mathbb{R}$.

Remark. We have stated above that multiple linear regression fits the data using an estimator f from the class of all *affine functions*. However, for the rest of the course, we will assume that $w_0 = 0$ to simplify the calculations. We can assume this *without loss of generality* since any affine estimator $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + w_0$ can be rewritten as $f((\boldsymbol{x}, 1)) = (\boldsymbol{w}, w_0)^\top (\boldsymbol{x}, 1)$. Thus, searching in the function space $F_{\text{affine}} = \{f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + w_0 : \boldsymbol{w} \in \mathbb{R}^d, w_0 \in \mathbb{R}\}$ is equivalent to augmenting our inputs by a constant component ($\boldsymbol{x} \to (\boldsymbol{x}, 1)$) and searching in the function space $F_{\text{linear}} = \{f(\boldsymbol{x}) = \widehat{\boldsymbol{w}}^\top \boldsymbol{x} : \widehat{\boldsymbol{w}} \in \mathbb{R}^{d+1}\}$. For brevity, slightly different than in class, in these notes we will always assume that the inputs have been already transformed to $(\boldsymbol{x}, 1)$ to remove the intercept.

4.2.1 Loss function for multiple linear regression

The loss function most commonly used in multiple linear regression is the squared loss, which is defined as

$$\ell(f(\boldsymbol{x}), y) = (y - f(\boldsymbol{x}))^2.$$

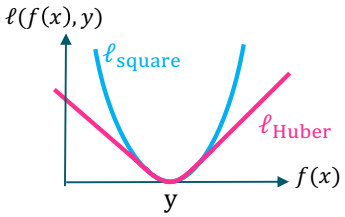
However, squared loss is rather sensitive to outliers. For instance, if most of the training datapoints lie close to the line $y = \boldsymbol{w}^\top \boldsymbol{x}$, and just one point (x_n, y_n) is very far away, the term $(y_n - f_w(\boldsymbol{x}_n))^2$ is going to dominate the training loss and thus produce a solution that is a bad fit for the rest of the data.

As an obvious solution, one might want to replace the squared loss with its square root, the *absolute loss*

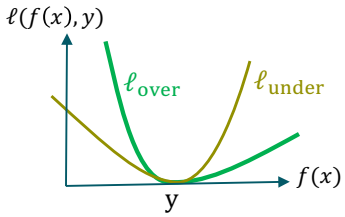
$$\ell_{abs}(f(\boldsymbol{x}), y) = |f(\boldsymbol{x}) - y|.$$

Although it does not amplify the loss of the outliers, it is not differentiable, which makes it difficult to find the optimal solution via common optimization methods. Instead, the so-called *Huber loss* is commonly utilized in robust regression (see Figure 4.1 (a) and Figure 4.1 (c). It is a piecewise function defined by

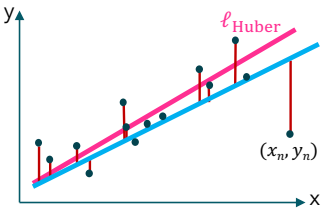
$$\ell_{huber}(f(\boldsymbol{x}), y) = \begin{cases} \frac{1}{2}(f(\boldsymbol{x}) - y)^2, \text{ if } |f(\boldsymbol{x}) - y| \leq \delta, \\ \delta(|f(\boldsymbol{x}) - y| - \frac{1}{2}\delta), \text{ if } |f(\boldsymbol{x}) - y| > \delta. \end{cases}$$



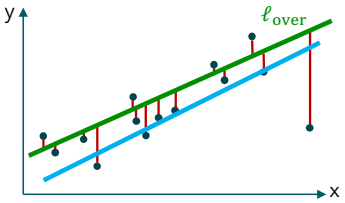
(a) Huber loss vs. square loss



(b) Two examples of asymmetric losses



(c) The estimator trained on Huber loss disregards the outlier (x_n, y_n).



(d) The estimator trained on ℓ_{over} tends to overestimate the values.

Figure 4.1: Different applications, such as data with strong outliers, may require other losses than the squared loss.

The parameter δ can be used to adjust the sensitivity to outliers.

Apart from tweaking sensitivity to outliers, it might be desirable for our specific task to weigh over- and underestimation of y differently. For this purpose, one could employ asymmetric losses, i.e. positive functions not symmetric around zero (Figure 4.1 (b)). In Figure 4.1 (d), ℓ_{over} penalizes underestimation more than overestimation, whereas ℓ_{under} overpenalizes overestimation. One example of an asymmetric loss is the so-called *quantile loss* (also called *pinball loss*), defined by

$$\ell_\tau(f(\boldsymbol{x}), y) = \tau \max\{y - f(\boldsymbol{x}), 0\} + (1 - \tau) \max\{f(\boldsymbol{x}) - y, 0\}.$$

Intuitively, if τ is small, a minimizer of ℓ_τ will tend to underestimate, since overestimation is penalized more strongly. If τ is close to one, any minimizer of ℓ_τ will overestimate.

Bonus Material

This bonus section relates to the probabilistic viewpoint that is only introduced in later chapters, so it can be skipped upon first reading. In particular, we discuss how the minimizer of the expected quantile loss is equivalent to the conditional quantile function.

First let's formally recap what a quantile is: For a real-valued random variable Y, the τ-quantile $q_P(\tau)$ is the smallest value that satisfies

$$P(Y \leq q_P(\tau)) = \tau.$$

For instance, the median is the $1/2$-quantile. Analogous to the definition of a quantile for a random variable, the conditional quantile function $f_\tau(x) = q_{P(Y|X=x)}(\tau)$ for given τ, x is defined as the smallest value q s.t. $P(Y \leq q|X) = \tau$. Interestingly, for any joint distribution, the minimizer of the expected quantile loss

$$\arg\min_f \mathbb{E}\left[\ell_\tau(f(\boldsymbol{X}), Y)\right]$$

is given by $f_\tau(x)$. This fact can be proven by treating the problem for each x separately and setting the derivative to zero with respect to the variable $q = f(x)$. The precise proof is left to the reader as an exercise. In this case, instead of the conditional mean, the minimizer of the quantile loss recovers the τ-quantile of the conditional distribution. In the special case $\tau = 1/2$, quantile loss becomes proportional to the absolute loss and yields the median of the conditional distribution.

We now illustrate the above in a simple example. Let's assume a linear relationship between X and Y, i.e. $Y = \beta^\top X + \epsilon$ with Gaussian noise ϵ that is independent of X. Then, the conditional distribution is given by $y|x \sim \mathcal{N}(x^\top \beta, \sigma^2)$. Minimizing the squared loss yields the regression function $\mathbb{E}[Y|X = x] = \beta^\top x$, which is the conditional mean of Y given $X = x$. Instead, minimizing the quantile loss with parameter τ outputs the conditional quantile function

$$f_\tau(x) = \beta^\top x + \sigma F^{-1}(\tau),$$

where F is the cumulative distribution function of the standard normal distribution.

4.2.2 *Finding the linear regression estimator*

Given the function class and the loss function, we can now find a function in our function class that "fits the data" in the sense that it minimizes the average loss in the training set:

$$\widehat{f} := \operatorname*{arg\,min}_{f \in F_{\text{linear}}} \quad L(f).$$

Note that any linear function in F_{linear} can be represented by a vector $\boldsymbol{w}$. This implies that one can associate any parameter vector $\boldsymbol{w} \in \mathbb{R}^d$ to a corresponding linear function $f_{\boldsymbol{w}}$, and vice versa. Therefore, we can rewrite our optimization problem over functions to be an optimization over parameter vectors instead

$$\widehat{\boldsymbol{w}} := \operatorname*{arg\,min}_{\boldsymbol{w} \in \mathbb{R}^d} L(f_{\boldsymbol{w}})$$

so that $\widehat{f} := f_{\widehat{\boldsymbol{w}}}$[2]. Concretely, if we denote our training set by $\{(\boldsymbol{x}_i, y_i)\}_{i=1\ldots n}$, the optimization problem above amounts to finding

$$\begin{aligned} \widehat{\boldsymbol{w}} &:= \operatorname*{arg\,min}_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - f_{\boldsymbol{w}}(\boldsymbol{x}_i))^2 \\ &= \operatorname*{arg\,min}_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \boldsymbol{w}^\top \boldsymbol{x}_i\right)^2. \end{aligned} \tag{4.1}$$

[2] In the lecture, we use a slight abuse of notation and write $L(\boldsymbol{w}) := L(f_{\boldsymbol{w}})$.

In what follows, we focus on solving the optimization problem in Equation 4.1.

First, let us rewrite the optimization problem in a more compact way. Let $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{y} \in \mathbb{R}^n$ be defined as

$$\boldsymbol{X} = \begin{pmatrix} x_1[1] & x_1[2] & \cdots & x_1[d] \\ x_2[1] & x_2[2] & \cdots & x_2[d] \\ \vdots & \vdots & \cdots & \vdots \\ x_n[1] & x_n[2] & \cdots & x_n[d] \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1^\top \\ \boldsymbol{x}_2^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{pmatrix}, \qquad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix},$$

where $x_i[j]$ is the j-th component of the vector $\boldsymbol{x}_i$. It is a simple linear algebra exercise to show that

$$\sum_{i=1}^{n} \left(y_i - \boldsymbol{w}^\top \boldsymbol{x}_i\right)^2 = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 .$$

Thus, Equation 4.1 can be written compactly in matrix vector notation as

$$\widehat{\boldsymbol{w}} = \operatorname*{arg\,min}_{\boldsymbol{w}\in\mathbb{R}^d} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 . \tag{4.2}$$

Notice that we removed the factor $\frac{1}{n}$ in front of Equation 4.1, as it is a constant and does not depend on $\boldsymbol{w}$, and thus is irrelevant for optimization.

We will now show that for this training loss, one can find a closed-form solution for Equation 4.2. Hence, we can study linear regression without having to worry about choosing an optimization procedure (for now).

By Theorem 2.4, any minimizer $\widehat{\boldsymbol{w}}$ of Equation 4.2 is a stationary point and thus satisfies

$$\nabla_w \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 = 2\boldsymbol{X}^\top \left(\boldsymbol{X}\widehat{\boldsymbol{w}} - \boldsymbol{y}\right) = \boldsymbol{0}, \tag{4.3}$$

which implies

$$\boldsymbol{X}^\top \boldsymbol{X}\widehat{\boldsymbol{w}} = \boldsymbol{X}^\top \boldsymbol{y}. \tag{4.4}$$

The next theorem shows how this condition, also called the *normal equation*, can be interpreted and derived geometrically.

Theorem 4.1. *Each solution $\widehat{\boldsymbol{w}}$ to the normal equation in Equation 4.4 has a corresponding vector of predictions $\widehat{\boldsymbol{y}} = \boldsymbol{X}\widehat{\boldsymbol{w}}$ that is equal to the orthogonal projection of $\boldsymbol{y}$ onto* span$(\boldsymbol{X})$.[3]

[3] Recall that the *image* or *range* of a matrix A, denoted as range(A) or span(A), is the span of its columns.

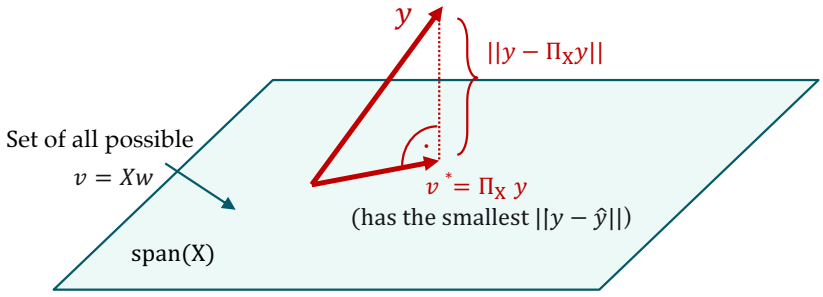


Figure 4.2: Linear regression estimate as a projection onto span$(\boldsymbol{X})$. The closest point to $\boldsymbol{y}$ on the plane is such that the difference $\boldsymbol{y} - \Pi_X(\boldsymbol{y})$ is perpendicular to the plane.

Proof. Let $\boldsymbol{x}_{[i]}$ be the i-th column of $\boldsymbol{X}$. First, notice that for any $\boldsymbol{w}$ we have $\boldsymbol{X}\boldsymbol{w} = \sum_{i=1}^{d} w_i \boldsymbol{x}_{[i]} \in$ span$(\boldsymbol{X})$. Thus, the optimal point

$$\widehat{\boldsymbol{w}} = \operatorname*{arg\,min}_{\boldsymbol{w}\in\mathbb{R}^d} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$$

is basically the closest point to $\boldsymbol{y}$ in span$(\boldsymbol{X})$ (see Figure 4.2). In Exercise 1.27, we show how $X\widehat{\boldsymbol{w}}$ has to be the orthogonal projection of y onto span$(\boldsymbol{X})$ and hence

$$X\widehat{\boldsymbol{w}} = \Pi_X(\boldsymbol{y}) = \underset{v \in \text{span}(\boldsymbol{X})}{\arg\min} \|\boldsymbol{y} - \boldsymbol{v}\|_2^2.$$

Now if $X\widehat{\boldsymbol{w}}$ is the orthogonal projection of y onto span$(\boldsymbol{X})$, it has to satisfy

$$\begin{aligned}
&y - X\widehat{\boldsymbol{w}} \perp X\boldsymbol{w} \text{ for all } \boldsymbol{w} \iff \\
&(y - X\widehat{\boldsymbol{w}})^\top X\boldsymbol{w} = 0 \text{ for all } \boldsymbol{w} \iff \\
&X^\top(X\widehat{\boldsymbol{w}} - y) = 0 \iff \\
&X^\top X\widehat{\boldsymbol{w}} = X^\top y,
\end{aligned}$$

i.e. $\widehat{\boldsymbol{w}}$ satisfies the normal equation.

□

Remark. We would like to note that the projection $\Pi_X(\boldsymbol{y})$ actually exists independent of whether X is full-rank or not. However, when X is not full-rank, there are multiple $\widehat{\boldsymbol{w}}$ that lead to the same prediction vector $X\widehat{\boldsymbol{w}} = \Pi_X(\boldsymbol{y})$.

The solution of the normal equations in Equation 4.4 depends on the rank of X and whether $d \leq n$ or $d > n$, as discussed in the next section. Further, if not otherwise noted, we henceforth assume $\boldsymbol{X}$ is full-rank in both cases for simplicity.

4.2.3 *Closed-form solution for linear regression*

Assuming that $d \leq n$ and $\boldsymbol{X}$ is full-rank, we have that $\boldsymbol{X}^\top\boldsymbol{X}$ is invertible (see the sidenote in Exercise 1.26), and obtain a unique solution for the normal equation 4.4 that is

$$\widehat{\boldsymbol{w}} = \left(\boldsymbol{X}^\top\boldsymbol{X}\right)^{-1}\boldsymbol{X}^\top\boldsymbol{y}. \tag{4.5}$$

As discussed in Section 2.3, just because $\widehat{\boldsymbol{w}}$ is a stationary point, it does not necessarily have to be a local or global minimizer (for example, it may be a maximizer or a saddle point). However, by the projection argument in Theorem 4.1, we can argue that $\widehat{\boldsymbol{w}}$ is a minimizer of the loss since

1. $X\widehat{\boldsymbol{w}}$ is the orthogonal projection of y on span(X) and
2. by Exercise 1.27, the orthogonal projection is the unique solution of the problem $\min_{v \in \text{span}(X)} \|\boldsymbol{y} - \boldsymbol{v}\|_2^2$, hence, for any $\boldsymbol{w}$ other than $\widehat{\boldsymbol{w}}$, $X\boldsymbol{w}$ will have a larger distance to y than $X\widehat{\boldsymbol{w}}$ and thus a greater loss value.

In Chapter 5, we will use a general convexity argument and second order conditions to argue that $\widehat{\boldsymbol{w}}$ is indeed a unique minimzer.

When there are more parameters than the number of examples (a.k.a. the over-parameterized case), that is when $d > n$, or more generally when the rank of X is smaller than d, the matrix $X^\top X$ is not invertible anymore. In this case, an optimum still has to satisfy the stationary point condition Equation 4.3, but there will be multiple possible solutions since the kernel of $\boldsymbol{X}^\top \boldsymbol{X}$ is now non-trivial (see Theorem 1.7). In particular, for any solution $\boldsymbol{w}$ that satisfies Equation 4.4, adding a vector $\boldsymbol{u} \in \ker(\boldsymbol{X})$ to $\boldsymbol{w}$ still results in a vector that satisfies Equation 4.4, since $\boldsymbol{X}^\top \boldsymbol{X}(\boldsymbol{w} + \boldsymbol{u}) = \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w}$.

In our prediction pipeline, however, outputting an entire subspace of the domain is not helpful for a practitioner who is trying to use the method. Ideally, we want to find a one-for-all closed-form solution (as the output of our method called linear regression) that is well-defined for all n and d and independent of the rank of $\boldsymbol{X}$. A sensible candidate is to replace the inverse by the pseudo-inverse and output

$$\widehat{\boldsymbol{w}} = \left(\boldsymbol{X}^\top \boldsymbol{X}\right)^\dagger \boldsymbol{X}^\top \boldsymbol{y}. \tag{4.6}$$

Note that this expression can be computed for any $\boldsymbol{X}$. First of all, when $\boldsymbol{X}$ has rank d, this expression is equivalent to the unique optimum in Equation 4.5. Next, we discuss how the pseudo-inverse solution Equation 4.6 naturally arises for the particular case when X has rank exactly $n < d$.

When $\boldsymbol{X}$ is full-rank If X is full-rank and $n < d$, then the equation $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}$ and the normal equation has infinitely many solutions. One can see this by starting from a solution $\boldsymbol{w}$ (such that $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}$) and adding to it a vector in the null-space $\ker(\boldsymbol{X})$. That is, for any $\boldsymbol{w}_{\text{ker}} \in \ker(\boldsymbol{X})$, we have that $\tilde{\boldsymbol{w}} = \boldsymbol{w} + \boldsymbol{w}_{\text{ker}}$ also fulfills $\boldsymbol{y} = \boldsymbol{X}\tilde{\boldsymbol{w}}$. All the solutions constructed this way are valid answers to the minimization problem induced by linear regression, because for such $\boldsymbol{w}$,

$$\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 = \|\boldsymbol{y} - \boldsymbol{y}\|_2^2 = 0.$$

In the next theorem, we show that the specific solution in Equation 4.6 corresponds to the solution of $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}$ with the smallest Euclidean norm

$$\underset{\boldsymbol{w} \in \mathbb{R}^d}{\text{minimize}} \, \|\boldsymbol{w}\|_2 \text{ s.t. } \boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}. \tag{4.7}$$

The so-called *minimum-norm solution* arises from running a simple optimization procedure on the original loss. This is discussed in more detail in **Exercise sheet 1, Exercise 3** (Gradient Descent for Overparameterized Linear Regression).

Theorem 4.2 (Min-Norm Solution)**.** *For a full-rank matrix $\boldsymbol{X}$ with $n < d$, the solution to the minimum norm problem defined in Equation 4.7 is*

$$\widehat{\boldsymbol{w}} = \boldsymbol{X}^\top \left(\boldsymbol{X}\boldsymbol{X}^\top\right)^{-1} \boldsymbol{y} = (\boldsymbol{X}^\top \boldsymbol{X})^\dagger \boldsymbol{X}^\top \boldsymbol{y}.$$

Bonus Material

Proof. Since $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, we can decompose $\mathbb{R}^d$ as

$$\mathbb{R}^d = \operatorname{span}(\boldsymbol{X}^\top) \oplus \operatorname{span}(\boldsymbol{X}^\top)^\perp.$$

As a result, every vector $\boldsymbol{w} \in \mathbb{R}^d$ can be written uniquely as $\boldsymbol{w} = \boldsymbol{w}_S + \boldsymbol{w}_P$, where $\boldsymbol{w}_S \in \operatorname{span}(\boldsymbol{X}^\top)$ and $\boldsymbol{w}_P \in \operatorname{span}(\boldsymbol{X}^\top)^\perp$. Because of orthogonality of $\boldsymbol{w}_S$ and $\boldsymbol{w}_P$, we have

$$\|\boldsymbol{w}\|_2^2 = \|\boldsymbol{w}_S\|_2^2 + \|\boldsymbol{w}_P\|_2^2.$$

Moreover, since $\boldsymbol{w}_P \in \operatorname{span}(\boldsymbol{X}^\top)^\perp$, it is perpendicular to all columns of $\boldsymbol{X}^\top$. Equivalently,

$$(\boldsymbol{X}^\top)^\top \boldsymbol{w}_P = \boldsymbol{X}\boldsymbol{w}_P = 0,$$

which implies

$$\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}\left(\boldsymbol{w}_S + \boldsymbol{w}_P\right) = \boldsymbol{X}\boldsymbol{w}_S.$$

This means that we can set $\boldsymbol{w}_P$ to $\mathbf{0}$, as $\boldsymbol{X}\boldsymbol{w}$ does not depend on it, while setting it to zero decreases the norm of $\boldsymbol{w}$. Thus, it suffices to only consider the vectors $\boldsymbol{w} \in \operatorname{span}(\boldsymbol{X}^\top)$. Consequently, any optimal solution to Equation 4.7 can be written as $\widehat{\boldsymbol{w}} = \boldsymbol{X}^\top \boldsymbol{v}$ for some $\boldsymbol{v} \in \mathbb{R}^n$. Moreover, since $\boldsymbol{X}$ is full rank, the vector $\boldsymbol{v}$ is uniquely defined for any given $\widehat{\boldsymbol{w}}$. In particular, it guarantees that the minimum norm is attained. Thus,

$$\boldsymbol{y} = \boldsymbol{X}\widehat{\boldsymbol{w}} = \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{v} \implies \boldsymbol{v} = \left(\boldsymbol{X}\boldsymbol{X}^\top\right)^{-1} \boldsymbol{y},$$

and

$$\widehat{\boldsymbol{w}} = \boldsymbol{X}^\top \boldsymbol{v} = \boldsymbol{X}^\top \left(\boldsymbol{X}\boldsymbol{X}^\top\right)^{-1} \boldsymbol{y}.$$

Finally, one can show using properties of pseudo-inverse that

$$\boldsymbol{X}^\top \left(\boldsymbol{X}\boldsymbol{X}^\top\right)^\dagger = \left(\boldsymbol{X}^\top \boldsymbol{X}\right)^\dagger \boldsymbol{X}^\top. \qquad \square$$

Exercise 4.3. Show that when X is not full-rank, $\widehat{\boldsymbol{w}}$ from Theorem 4.2 can be interpreted as the minimum norm solution of

$$\underset{\boldsymbol{w} \in \mathbb{R}^d}{\text{minimize}} \|\boldsymbol{w}\|_2 \text{ s.t. } \boldsymbol{X}^\top \boldsymbol{y} = \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w}. \tag{4.8}$$

4.3 Non-linear Least Squares

We can generalize the linear model discussed in the previous sections by introducing a feature mapping $\phi : \mathbb{R}^d \to \mathbb{R}^p$. The idea is that instead of performing linear regression on our data inputs, we can first apply a (possibly non-linear) transformation to them, before the transformed data points are then used in the linear regression pipeline (see Figure 4.3). More precisely, our model will be

$$f_w(\boldsymbol{x}) = \sum_{j=1}^{p} w_j \phi_j(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle$$

instead of $f_w(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$ as in standard linear regression. We often call the transformed vector $\phi(x)$ the *features* corresponding to the input represented by x and the regression problem with this function

class the *non-linear least squares problem*.

It is worthwhile to mention that in this case, the class of functions $\{f_w \mid w \in \mathbb{R}^p\}$ can be very different from F_{linear}. The example below shows this in a simple case.

Example 4.4. Consider the case where $d = 1$, that is, the inputs are scalars. One possible feature mapping is $\phi(x) = (1, x, x^2, x^3, \ldots, x^p)$. Observe that for $w = (w_0, \ldots, w_p)$ we have

$$f_w(x) = \langle w, \phi(x) \rangle = w_0 + w_1 x + w_2 x^2 + \cdots + w_p x^p,$$

which is a polynomial of degree at most p. Thus, the function class $\{f_w \mid w \in \mathbb{R}^{p+1}\}$ is exactly the set of all polynomials of degree at most p (which definitely contains non-linear functions if $p > 1$). The linear regression problem for this function class is often called *polynomial regression*.

Any fixed feature mapping ϕ induces a function class F_ϕ which includes non-linear relationships between the inputs and the target. This function class is described as

$$F_\phi = \{f : \mathbb{R}^d \to \mathbb{R} \mid f(x) = \sum_{j=1}^{p} w_j \phi_j(x) \text{ for } w \in \mathbb{R}^p\}$$

where $\phi_1(x), ..., \phi_p(x)$ are components of ϕ, often known as *basis functions*. Note, however, that the feature transformation is fixed and thus requires us to come up with appropriate mappings on our own. As we will see later, neural networks enable us to learn the feature map ϕ "automatically" to fit the data.

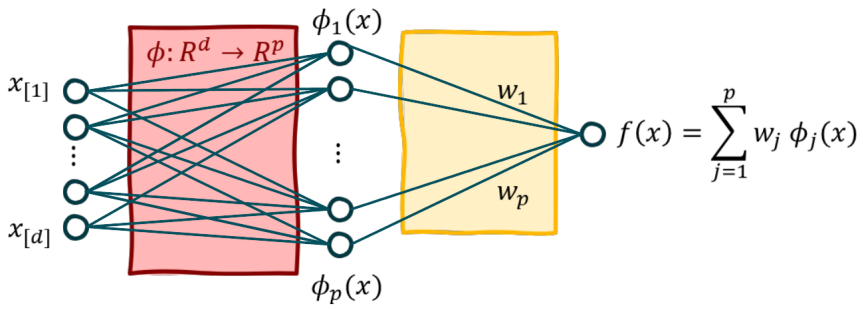


Figure 4.3: The depicted feature map ϕ transforms the d-dimensional input $(x_1, ..., x_d)$ into p features $(\phi_1(x), ..., \phi_p(x))$ through a non-linear map $\phi : \mathbb{R}^d \to \mathbb{R}^p$ (red). On the transformed features, linear regression is performed (yellow). The first part of the diagram can be replaced by a neural network which, in addition, also learns the "best possible" map ϕ from the data. The yellow part (linear regression) would then correspond to the last layer of a neural network.

Remark. Observe that non-linear least squares is a generalization of multiple linear regression, since one can easily recover the multiple linear regression model by using the feature map $\phi_j(x) = x_j$ for $j = 1, ..., d$.

The process of fitting non-linear least squares to the training data is analogous to fitting multiple linear regression, since the model is still linear in the parameters w. The only difference is that we now seek to minimize the squared loss in the function space F_ϕ instead of F_{linear}. Since the feature map ϕ is fixed, we can simply replace X with a matrix $\Phi \in \mathbb{R}^{n \times p}$ containing the transformed inputs as rows

and adjust Equation 4.2 accordingly. This leads us to the following characterization of the non-linear least squares solution

$$\widehat{\boldsymbol{w}} = \underset{\boldsymbol{w} \in \mathbb{R}^p}{\arg\min} \, \|\boldsymbol{y} - \Phi \boldsymbol{w}\|_2^2 \, .$$

Now, we can simply apply the same reasoning as in Section 4.2 and find the solution to the problem above.

5
Optimization

In this chapter, we introduce a special class of optimization algorithms, known as iterative optimization algorithms. Specifically, we focus on the gradient descent algorithm and some of its variants. We also discuss the notion of convexity, which enables us to give guarantees about the output of the optimization algorithms.

Roadmap

We begin this chapter by introducing iterative optimization in Section 5.1 and highlighting its advantages over analytical methods. In Section 5.2, we introduce gradient descent and derive its most important characteristics. Then, in Section 5.3, we derive conditions under which we can guarantee convergence of gradient descent for linear regression. Before moving on to stochastic gradient descent (Section 5.4.4), we shortly mention popular alternatives and variations of gradient descent in Section 5.4. In Section 5.5, we introduce the notion of convexity which is helpful to theoretically analyze the solution found by gradient descent.

Learning Objectives

After reading this chapter you should know:

- Why iterative methods are often preferred over analytical ones.
- How and why gradient descent works for minimzing a loss function.
- The role of the learning rate for convergence of gradient descent.
- Variants of gradient descent such as stochastic gradient descent and minibatch gradient descent.
- How to identify a convex function.
- Why the notion of convexity is important for gradient descent.

5.1 Closed-form Solution vs. Iterative Optimization

Recall from Section 4.1 that the goal in supervised learning is to find a predictor $\hat{f}$ in some parametric function class F that (approximately) minimizes the training loss L over the training set.

Assuming that the functions in F are parameterized by vectors $\boldsymbol{w} \in \mathbb{R}^d$, this minimization can be written as[1]

$$\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}\in\mathbb{R}^d} L(\boldsymbol{w}) := \arg\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \ell(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i)$$

and setting $\widehat{f} = f_{\widehat{\boldsymbol{w}}}$. Notice the new notation $L(\boldsymbol{w}) := L(f_{\boldsymbol{w}})$.

[1] Recall that $\{(\boldsymbol{x}_i, y_i)\}$ is the training set, and by $f_{\boldsymbol{w}}$ we mean the function in the function class F corresponding to the parameters $\boldsymbol{w}$.

In Chapter 4, we explicitly computed $\widehat{\boldsymbol{w}}$ in closed form for linear regression. However, for more complex function classes, or more complex loss functions ℓ, it is not always possible to find a closed form solution to the above optimization problem. Furthermore, even when there exists a closed-form solution, the computational cost to calculate it might be too large. For instance, the closed-form solution of linear regression requires $\mathcal{O}(d^3 + nd^2)$ operations.[2] First, one needs to compute the matrix product $\boldsymbol{X}^\top\boldsymbol{X}$, which takes $\mathcal{O}(nd^2)$ operations. Then, inversion of the corresponding $d \times d$ matrix takes $\mathcal{O}(d^3)$ operations. For high dimensions, this inversion operation might become computationally infeasible.

[2] Recall that from Definition 2.7, the $\mathcal{O}$ notation is used to provide *upper bounds* on the growth of a function and

$$f(x) = \mathcal{O}(g(x)) \quad \text{as} \quad x \to \infty$$

signifies that the growth behavior of f is controlled by g.

Instead of looking for a closed-form solution we could potentially use iterative methods. These methods can be applied to problems for which a closed-form solution does not exist or is computationally expensive to calculate. In the latter case, it might provide a reasonably good approximate solution with relatively lower computational cost. The downside of these methods is that we may not be able to recover the exact solution of the minimization problem. However, we will later provide guarantees under which the iterative algorithm will give solutions arbitrarily close to the optimal solution.

These iterative methods usually adhere to the following scheme: one starts with an initial guess of the solution, and then iteratively updates this estimate until a stopping criterion is satisfied. This is shown in the following algorithm.

Algorithm 1: Template for iterative optimization.

1: $t \leftarrow 0$
2: $\boldsymbol{w}^0 \leftarrow \boldsymbol{w}_{\text{initial guess}}$
3: **repeat**
4: $\quad \boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \text{update}$
5: $\quad t \leftarrow t + 1$
6: **until** stopping criterion is satisfied
7: Return $\boldsymbol{w}^t$

In this course, we focus on those iterative optimization algorithms whose "update" (in line 4 of the algorithm) is of the form $\tilde{\eta}_t \boldsymbol{v}^t$, where $\boldsymbol{v}^t$ is the *update direction*, and $\tilde{\eta}_t$ is the *step size* (signifying the magnitude of update) at iteration t. Different choices for the update direction, step size, as well as stopping criterion leads to different optimization algorithms. In the following sections, we discuss some common choices and analyze their behavior.

5.2 Gradient Descent

Since our goal is to minimize the loss function L, a natural choice for the update direction is the direction in which L decreases the most, locally. Assuming that L is differentiable, this direction is given by the negative gradient of L (see property 1 of the gradient in Section 2.1). Below, we will re-derive this fact via Taylor approximation.

Let the update direction and step size in the current iteration of our iterative optimization be the unit vector $\boldsymbol{v}$ and η respectively. Here we assume that the update direction is a unit vector, so that it represents only the direction, and not the amount we move in that direction. Writing the Taylor expansion of L around the current point $\boldsymbol{w}^t$ (Theorem 2.5), we can get an estimate of the loss after one update

$$\begin{aligned} L(\boldsymbol{w}^{t+1}) &= L(\boldsymbol{w}^t + \eta \boldsymbol{v}) \\ &= L(\boldsymbol{w}^t) + \eta \left\langle \nabla L(\boldsymbol{w}^t), \boldsymbol{v} \right\rangle + o(\eta). \end{aligned}$$

For small values of η, we can neglect the $o(\eta)$ term,[3] and say that approximately

$$L(\boldsymbol{w}^{t+1}) \approx L(\boldsymbol{w}^t) + \eta \left\langle \nabla L(\boldsymbol{w}^t), \boldsymbol{v} \right\rangle.$$

[3] Recall that from Definition 2.9 $f(x) = o(g(x))$ signifies that f grows much slower than g and is insignificant in comparison.

Now, to find a $\boldsymbol{v}$ that decreases L the most, we wish to minimize the right-hand side with respect to $\boldsymbol{v}$. That is,

$$\begin{aligned} \boldsymbol{v}^* &= \arg\min_{\|\boldsymbol{v}\|_2=1} L(\boldsymbol{w}^t) + \eta \left\langle \nabla L(\boldsymbol{w}^t), \boldsymbol{v} \right\rangle \\ &= \arg\min_{\|\boldsymbol{v}\|_2=1} \left\langle \nabla L(\boldsymbol{w}^t), \boldsymbol{v} \right\rangle. \end{aligned}$$

We dropped $L(\boldsymbol{w}^t)$ and η, as they do not depend on $\boldsymbol{v}$. By the Cauchy-Schwarz inequality (Theorem 1.12), we have

$$\left\langle \nabla L(\boldsymbol{w}^t), \boldsymbol{v} \right\rangle \geq - \left\| \nabla L(\boldsymbol{w}^t) \right\|_2,$$

and the equality holds if and only if $\boldsymbol{v}$ and $-\nabla L(\boldsymbol{w}^t)$ are in the same direction. Thus, the optimal $\boldsymbol{v}$ is

$$\boldsymbol{v}^* = -\frac{\nabla L(\boldsymbol{w}^t)}{\|\nabla L(\boldsymbol{w}^t)\|_2}.$$

The general take-away from this discussion is summarized in the following lemma.

Lemma 5.1. *Let L be a differentiable function. The negative gradient of L is the direction of locally steepest descent of L.*

Hence, it makes sense to set our update rule (line 4 of Algorithm 8) to be

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \tilde{\eta}_t \frac{\nabla L(\boldsymbol{w}^t)}{\|\nabla L(\boldsymbol{w}^t)\|_2}.$$

In this formulation, $\tilde{\eta}_t > 0$ is the step size at iteration t (which we can choose) and $-\nabla L(\boldsymbol{w}^t) / \left\|\nabla L(\boldsymbol{w}^t)\right\|_2$ the *normalized* update direction. If we fix some $\eta > 0$ and choose $\tilde{\eta}_t = \eta \left\|\nabla L(\boldsymbol{w}^t)\right\|_2$, we get the update rule

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta \nabla L(\boldsymbol{w}^t).$$

It is common to still call η the step size, however, since $\nabla L(\boldsymbol{w}^t)$ is *unnormalized* the real amount which we "move" at iteration t is

$$\left\|\boldsymbol{w}^{t+1} - \boldsymbol{w}^t\right\|_2 = \left\|-\eta \nabla L(\boldsymbol{w}^t)\right\|_2 = \eta \left\|\nabla L(\boldsymbol{w}^t)\right\|_2.$$

The resulting update rule is convenient because we do not have to normalize the update direction in every step. Moreover, we can make use of the behaviour around stationary points where $\nabla L(\boldsymbol{w}^t) \approx \mathbf{0}$. If the algorithm is close to a stationary point (e.g., a minimum), the amount that the algorithm moves is small since $\eta \left\|\nabla L(\boldsymbol{w}^t)\right\|_2 \approx 0$. Therefore, the algorithm takes large steps when it is far from a stationary point, and little steps when it is close. This could, for instance, prevent it from 'overshooting' a stationary point.

A common stopping criterion for this update rule is to see whether we are close to a stationary point, which by the previous discussion can be evaluated by how much progress we make, i.e., the distance $\left\|\boldsymbol{w}^{t+1} - \boldsymbol{w}^t\right\|_2$. If this distance is smaller than some threshold $\epsilon > 0$, we stop. Note that we could also rewrite it as $\left\|\nabla L(\boldsymbol{w}^t)\right\|_2 \leq \epsilon / \eta$.

This choice of stopping criterion, as well as the update rule above, gives rise to the *gradient descent* algorithm.

Algorithm 2: Gradient Descent.

$t \leftarrow 0$
$\boldsymbol{w}^0 \leftarrow \boldsymbol{w}_{\text{initial guess}}$
repeat
    $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t - \eta \nabla L(\boldsymbol{w}^t)$
    $t \leftarrow t + 1$
until $\left\|\boldsymbol{w}^t - \boldsymbol{w}^{t-1}\right\|_2 \leq \epsilon$
Return $\boldsymbol{w}^t$

A useful property of using the negative gradient direction is that for small enough η, we can guarantee that our function value actually decreases at each iteration. We call such a direction a *descent direction*.

Theorem 5.2 (Descent Direction)**.** *The gradient descent update is a descent direction for small enough η.*

Remark. The correct choice of η is crucial for getting a fast algorithm, as it heavily influences whether, and how fast, gradient descent converges to a stationary point of our loss function. If the step size is

too large, we might diverge, as the steps simply "overshoot" the stationary points. This is in contrast to Theorem 5.2 which guarantees decrease in function value only for small enough η. On the other hand, for a too small step size, convergence might take more steps than necessary.

Bonus Material

Lets state and prove a slightly more formal version of Theorem 5.2:

For any $\boldsymbol{w}^0 \in \mathbb{R}^d$ and any number of iterations T, there exists a step size $\eta > 0$ such that the gradient descent updates satisfy $L(\boldsymbol{w}^{t+1}) < L(\boldsymbol{w}^t)$ as long as $\nabla L(\boldsymbol{w}^t) \neq 0$ and $t \leq T$.

Proof. Fix any $t \leq T$ for now. Without loss of generality, assume that $\nabla L(\boldsymbol{w}^t)$ is nonzero (otherwise the gradient descent makes no move, and we are already at a stationary point). Using the multivariate Taylor expansion (Theorem 2.5), we have

$$\begin{aligned} L(\boldsymbol{w}^{t+1}) &= L(\boldsymbol{w}^t - \eta \nabla L(\boldsymbol{w}^t)) \\ &= L(\boldsymbol{w}^t) - \eta \left\langle \nabla L(\boldsymbol{w}^t), \nabla L(\boldsymbol{w}^t) \right\rangle + \xi\left(-\eta \nabla L(\boldsymbol{w}^t)\right) \\ &= L(\boldsymbol{w}^t) - \eta \left\| \nabla L(\boldsymbol{w}^t) \right\|_2^2 + \xi\left(-\eta \nabla L(\boldsymbol{w}^t)\right) \end{aligned} \tag{5.1}$$

where $\xi : \mathbb{R}^d \to \mathbb{R}$ is the remainder term. This term is of $o(\eta \left\| \nabla L(\boldsymbol{w}^t) \right\|_2)$, i.e.,

$$\lim_{\eta \to 0} \frac{\xi\left(-\eta \nabla L(\boldsymbol{w}^t)\right)}{\eta \left\| \nabla L(\boldsymbol{w}^t) \right\|_2} = 0.$$

Then, by definition of the limit, for all $\epsilon_t > 0$ there exists $\eta_t > 0$ small enough such that

$$\left| \frac{\xi\left(-\eta_t \nabla L(\boldsymbol{w}^t)\right)}{\eta_t \left\| \nabla L(\boldsymbol{w}^t) \right\|_2} \right| \leq \epsilon_t,$$

or

$$\left| \xi\left(-\eta_t \nabla L(\boldsymbol{w}^t)\right) \right| \leq \epsilon_t \eta_t \left\| \nabla L(\boldsymbol{w}^t) \right\|_2.$$

Applying this to Equation 5.1 and picking $\epsilon_t < \left\| \nabla L(\boldsymbol{w}^t) \right\|_2$ we get

$$\begin{aligned} L(\boldsymbol{w}^{t+1}) &= L(\boldsymbol{w}^t) - \eta_t \left\| \nabla L(\boldsymbol{w}^t) \right\|_2^2 + \xi\left(-\eta_t \nabla L(\boldsymbol{w}^t)\right) \\ &\leq L(\boldsymbol{w}^t) - \eta_t \left\| \nabla L(\boldsymbol{w}^t) \right\|_2^2 + \epsilon_t \eta_t \left\| \nabla L(\boldsymbol{w}^t) \right\|_2 \\ &= L(\boldsymbol{w}^t) + \eta_t \left\| \nabla L(\boldsymbol{w}^t) \right\|_2 \left(\epsilon_t - \left\| \nabla L(\boldsymbol{w}^t) \right\|_2\right) \\ &< L(\boldsymbol{w}^t). \end{aligned}$$

Since this argument holds for all $t \leq T$ if $\nabla L(\boldsymbol{w}^t) \neq 0$, we can take $\eta = \min_{t \leq T} \eta_t$ so that it holds for all $t \leq T$ simultaneously as long as $\nabla L(\boldsymbol{w}^t) \neq 0$. This finishes the proof. □

We established that in every iteration, gradient descent finds a point where the function value is lower than in the previous iteration. Does this mean it will converge to a stationary point? Or even a minimum? Unfortunately, this is not true in general.[4] Instead, the guarantees we can give have to be more nuanced, as you will see in Section 5.3 and Section 5.5.

[4] As an exercise, you can try to come up with examples, where this is not true.

Finally, the Lemma below connects the steepest descent intuition

using level sets to choosing the negative gradient direction. Recall that the level set of L at a level $c \in \mathbb{R}$ is defined as

$$\left\{ \boldsymbol{w} \in \mathbb{R}^d : L(\boldsymbol{w}) = c \right\}.$$

For $d = 2$ level sets can be thought of as lines in the two-dimensional plane, like the contour lines in a two-dimensional map for mountaineering. The following fact has already been stated without proof in Section 2.1.

Lemma 5.3. *At any point $\boldsymbol{w}_0$, the gradient of L is orthogonal to the level set of L at level $L(\boldsymbol{w}_0)$.*[5]

[5] see Figure 2.1 for an illustration.

Bonus Material

Lemma 5.3 might be intuitive, but formalizing it takes some effort. The main question we need to answer is: What does it even mean to be orthogonal to a level set?

For that, we need to define something called the tangent space of the level set. Let $c = L(\boldsymbol{w}_0)$ and denote $L^{-1}(c) = \left\{ \boldsymbol{w} \in \mathbb{R}^d : L(\boldsymbol{w}) = c \right\}$ as a short-hand notation for the level set at level c. Consider the set $\Gamma(\boldsymbol{w}_0)$ of all differentiable curves $\gamma : (-1,1) \to \mathbb{R}^d$ that satisfy $\gamma((-1,1)) \subseteq L^{-1}(c)$ and $\gamma(0) = \boldsymbol{w}_0$. The derivatives of such curves are d-dimensional vectors, $\frac{d\gamma}{dt}(t_0) \in \mathbb{R}^d$, and the tangent space of $L^{-1}(c)$ at $\boldsymbol{w}_0$ is defined as the set of vectors

$$\left\{ \frac{d\gamma}{dt}(0) : \gamma \in \Gamma(\boldsymbol{w}_0) \right\}.$$

This set forms a linear subspace of $\mathbb{R}^d$, which can be thought of as the best linear approximation of $L^{-1}(c)$ at $\boldsymbol{w}_0$.

When we say that the gradient is orthogonal to the level set, what we actually mean is that it is orthogonal to all vectors in the tangent space. So let's prove that.

For any $\gamma \in \Gamma(\boldsymbol{w}_0)$ we have $c = L(\gamma(t))$ for all $t \in (-1,1)$. Using the chain rule (Theorem 2.1) we get

$$0 = \frac{dc}{dt}(0) = \frac{d(L \circ \gamma)}{dt}(0) = \left\langle \nabla L(\gamma(0)), \frac{d\gamma}{dt}(0) \right\rangle = \left\langle \nabla L(\boldsymbol{w}_0), \frac{d\gamma}{dt}(0) \right\rangle.$$

This concludes the proof.

With a fundamental understanding of gradient descent, we can now look at the behavior of gradient descent applied to linear regression.

5.3 Convergence of Gradient Descent for Linear Regression

In this section, we will analyze when and how fast gradient descent converges to the optimal solution for a linear regression problem. We start with a definition that will be useful in our analysis.

Definition 5.4 (Operator Norm)**.** The operator norm of a matrix A is

defined as

$$\|A\|_{\text{op}} := \sup_{\boldsymbol{x}\neq\boldsymbol{0}} \frac{\|A\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} = \sup_{\|\boldsymbol{x}\|_2=1} \|A\boldsymbol{x}\|_2 .$$

Intuitively, $\|A\|_{\text{op}}$ is the largest value by which A stretches a vector. This is indeed the same as the largest singular value of A.

Exercise 5.5. Show that for a real and symmetric matrix A the operator norm in terms of the eigenvalues of A is given by

$$\|A\|_{\text{op}} = \max\{|\lambda_{\max}(A)|, |\lambda_{\min}(A)|\} = \sqrt{\lambda_{\max}(A^\top A)}, \tag{5.2}$$

where $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ are the largest and smallest eigenvalues of A.

The operator norm is very useful if one wants to bound $\|A\boldsymbol{x}\|_2$ for some vector $\boldsymbol{x}$, as the following Lemma shows.

Lemma 5.6. *For any matrix A and vector $\boldsymbol{x}$, we have*

$$\|A\boldsymbol{x}\|_2 \leq \|A\|_{\text{op}} \|\boldsymbol{x}\|_2 .$$

The following Theorem establishes the convergence of gradient descent applied to linear regression. More precisely, under some assumptions, the solution found by gradient descent will converge to the minimizer of the squared loss exponentially fast in the number of iterations.

Theorem 5.7. *Consider the (modified) linear regression problem given by*

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} L(\boldsymbol{w}) = \min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 .$$

If $\boldsymbol{X}^\top\boldsymbol{X}$ is full-rank and the learning rate satisfies $\eta < 2/\lambda_{\max}(\boldsymbol{X}^\top\boldsymbol{X})$, then gradient descent converges linearly to the optimal solution $\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}\in\mathbb{R}^d} L(\boldsymbol{w})$.

Remark. A few remarks are in order, before we prove the Theorem.

- The original linear regression problem has a factor of $\frac{1}{n}$ in front of it. For notational simplicity, we replaced it with $\frac{1}{2}$. This simplifies our analysis but does not affect the result.
- The condition that $\boldsymbol{X}^\top\boldsymbol{X}$ is full-rank is equivalent to $\lambda_{\min}(X^\top\boldsymbol{X}) > 0$. This is due to the fact that $\boldsymbol{X}^\top\boldsymbol{X}$ is p.s.d. (hence all eigenvalues are non-negative), and the rank of a matrix is the number of its nonzero singular values.
- By linear convergence, a rather unfortunate wording used in the optimization literature, we mean that the distance of our estimate $\boldsymbol{w}^t$ (after t iterations) to the optimal solution $\widehat{\boldsymbol{w}}$ decreases like $C\rho^t$ for some $\rho < 1$ and some constant $C > 0$. Its called linear, because we multiply by a fixed constant ρ in every iteration. However, note that the convergence is *exponential* in the number of iterations.

Proof. Let us first derive the explicit update rule of gradient descent for this problem instance. The gradient of L is given by

$$\nabla L(\boldsymbol{w}) = \boldsymbol{X}^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) = -\boldsymbol{X}^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}),$$

and the gradient descent update rule will be

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta \nabla L(\boldsymbol{w}^t) = \boldsymbol{w}^t + \eta \boldsymbol{X}^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}^t).$$

Therefore, the distance of $\boldsymbol{w}^{t+1}$ from the optimal solution $\widehat{\boldsymbol{w}}$ is given by

$$\begin{aligned}
\boldsymbol{w}^{t+1} - \widehat{\boldsymbol{w}} &= \boldsymbol{w}^t + \eta \boldsymbol{X}^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}^t) - \widehat{\boldsymbol{w}} \\
&= \boldsymbol{w}^t + \eta (\boldsymbol{X}^\top \boldsymbol{y} - \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w}^t) - \widehat{\boldsymbol{w}} \\
&= \boldsymbol{w}^t + \eta (\boldsymbol{X}^\top \boldsymbol{X}\widehat{\boldsymbol{w}} - \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w}^t) - \widehat{\boldsymbol{w}}.
\end{aligned}$$

For the last equality, we used the fact that $\widehat{\boldsymbol{w}}$ satisfies the normal equations Equation 4.4. Simplifying further yields

$$\begin{aligned}
\boldsymbol{w}^{t+1} - \widehat{\boldsymbol{w}} &= \boldsymbol{w}^t - \widehat{\boldsymbol{w}} + \eta \boldsymbol{X}^\top \boldsymbol{X}(\widehat{\boldsymbol{w}} - \boldsymbol{w}^t) \\
&= (\boldsymbol{I} - \eta \boldsymbol{X}^\top \boldsymbol{X})(\boldsymbol{w}^t - \widehat{\boldsymbol{w}}).
\end{aligned}$$

Then by Lemma 5.6 and letting $\rho := \left\| \boldsymbol{I} - \eta \boldsymbol{X}^\top \boldsymbol{X} \right\|_{\text{op}}$ we obtain

$$\begin{aligned}
\left\| \boldsymbol{w}^{t+1} - \widehat{\boldsymbol{w}} \right\|_2 &\leq \left\| \boldsymbol{I} - \eta \boldsymbol{X}^\top \boldsymbol{X} \right\|_{\text{op}} \left\| \boldsymbol{w}^t - \widehat{\boldsymbol{w}} \right\|_2 \\
&= \rho \left\| \boldsymbol{w}^t - \widehat{\boldsymbol{w}} \right\|_2 \\
&\leq \rho^2 \left\| \boldsymbol{w}^{t-1} - \widehat{\boldsymbol{w}} \right\|_2 \\
&\vdots \\
&\leq \rho^{t+1} \left\| \boldsymbol{w}^0 - \widehat{\boldsymbol{w}} \right\|_2.
\end{aligned} \tag{5.3}$$

The last inequality follows from recursively applying the second inequality. Observe that if $\rho < 1$, the upper bound on $\left\| \boldsymbol{w}^{t+1} - \widehat{\boldsymbol{w}} \right\|_2$ converges linearly to zero as $t \to \infty$. Thus, all that is left to do is to investigate under what circumstances $\rho < 1$ holds. Using the characterization in Equation 5.2, we can write[6]

$$\begin{aligned}
\rho &= \max \left\{ |\lambda_{\max}(\boldsymbol{I} - \eta \boldsymbol{X}^\top \boldsymbol{X})|, |\lambda_{\min}(\boldsymbol{I} - \eta \boldsymbol{X}^\top \boldsymbol{X})| \right\} \\
&= \max \left\{ |1 - \eta \lambda_{\min}(\boldsymbol{X}^\top \boldsymbol{X})|, |1 - \eta \lambda_{\max}(\boldsymbol{X}^\top \boldsymbol{X})| \right\}.
\end{aligned}$$

[6] Here, we made use of the fact that if $\lambda_1, \ldots, \lambda_d$ are the eignvalues of the matrix $\boldsymbol{X}^\top \boldsymbol{X}$, then the eigenvalues of $I - \eta \boldsymbol{X}^\top \boldsymbol{X}$ are exactly $1 - \eta\lambda_1, \ldots, 1 - \eta\lambda_d$. Notice that the order of the eigenvalues is reversed.

If $\eta \leq \frac{1}{\lambda_{\max}(\boldsymbol{X}^\top \boldsymbol{X})}$, both of the terms in the expression are non-negative and hence we can drop the absolute value function

$$\begin{aligned}
\rho &= \max \left\{ 1 - \eta \lambda_{\min}(\boldsymbol{X}^\top \boldsymbol{X}), 1 - \eta \lambda_{\max}(\boldsymbol{X}^\top \boldsymbol{X}) \right\} \\
&= 1 - \eta \lambda_{\min}(\boldsymbol{X}^\top \boldsymbol{X}).
\end{aligned}$$

Therefore, $\rho < 1$, as we have assumed $\lambda_{\min}(\boldsymbol{X}^\top \boldsymbol{X}) > 0$ (recall that this is the same as $\boldsymbol{X}^\top \boldsymbol{X}$ being full-rank).

On the other hand, if $\eta > \frac{1}{\lambda_{\max}(\boldsymbol{X}^\top \boldsymbol{X})}$, we have

$$\begin{aligned}\rho &= \max\left\{|1-\eta\lambda_{\min}(\boldsymbol{X}^\top\boldsymbol{X})|, |1-\eta\lambda_{\max}(\boldsymbol{X}^\top\boldsymbol{X})|\right\} \\ &= \max\left\{1-\eta\lambda_{\min}(\boldsymbol{X}^\top\boldsymbol{X}), \eta\lambda_{\min}(\boldsymbol{X}^\top\boldsymbol{X})-1, \eta\lambda_{\max}(\boldsymbol{X}^\top\boldsymbol{X})-1\right\} \\ &= \max\left\{1-\eta\lambda_{\min}(\boldsymbol{X}^\top\boldsymbol{X}), \eta\lambda_{\max}(\boldsymbol{X}^\top\boldsymbol{X})-1\right\}.\end{aligned} \tag{5.4}$$

If we want $\rho < 1$, two conditions shall be satisfied:

(i) $1-\eta\lambda_{\min}(\boldsymbol{X}^\top\boldsymbol{X}) < 1$, which is already fulfilled, as we assumed $\lambda_{\min}(\boldsymbol{X}^\top\boldsymbol{X}) > 0$, and

(ii) $\eta\lambda_{\max}(\boldsymbol{X}^\top\boldsymbol{X}) - 1 < 1$, which holds if $\eta < \frac{2}{\lambda_{\max}(\boldsymbol{X}^\top\boldsymbol{X})}$.

Hence, it is clear that both conditions are satisfied under our assumptions. This concludes the proof. □

We can see from Equation 5.3 that the smaller ρ is, the fewer the steps we need in order to guarantee that $\left\|\boldsymbol{w}^{t+1} - \widehat{\boldsymbol{w}}\right\|_2 \leq \epsilon$ for a given ϵ. In the next section, we derive a theoretically optimal step size to minimize ρ such that convergence speed is optimized.

5.3.1 Convergence Speed

As mentioned before, the right choice of η is not only crucial for convergence, but also for its speed. Let us recap what we derived in the proof of Theorem 5.7. For brevity, let us write $\lambda_{\max} := \lambda_{\max}(\boldsymbol{X}^\top\boldsymbol{X})$ and $\lambda_{\min} := \lambda_{\min}(\boldsymbol{X}^\top\boldsymbol{X})$.

- If $0 < \eta \leq \frac{1}{\lambda_{\max}}$, we get $\rho = 1 - \eta\lambda_{\min}$.
- If $\frac{1}{\lambda_{\max}} < \eta < \frac{2}{\lambda_{\max}}$, we get $\rho = \max\{1-\eta\lambda_{\min}, \eta\lambda_{\max}-1\}$.
- If $\eta > \frac{2}{\lambda_{\max}}$, it will be too large, and we have no more convergence guarantee, as ρ can be well above one. In that case, gradient descent might not converge and can even diverge.

In the second case, we have either $\rho = 1-\eta\lambda_{\min}$ (which happens if $1-\eta\lambda_{\min} \geq \eta\lambda_{\max} - 1$, or equivalently, $\eta < \frac{2}{\lambda_{\min}+\lambda_{\max}}$), or $\rho = \eta\lambda_{\max} - 1$ (which happens if $\eta\lambda_{\max} - 1 \geq 1 - \eta\lambda_{\min}$, or equivalently, $\eta \geq \frac{2}{\lambda_{\max}+\lambda_{\min}}$). In summary,

$$\rho = \begin{cases} 1-\eta\lambda_{\min} & \text{if } \eta < \frac{1}{\lambda_{\max}}, \\ 1-\eta\lambda_{\min} & \text{if } \eta \in \left[\frac{1}{\lambda_{\max}}, \frac{2}{\lambda_{\min}+\lambda_{\max}}\right], \\ \eta\lambda_{\max}-1 & \text{if } \eta \in \left[\frac{2}{\lambda_{\min}+\lambda_{\max}}, \frac{2}{\lambda_{\max}}\right), \\ \geq 1 & \text{if } \eta \geq \frac{2}{\lambda_{\max}}. \end{cases}$$

From the shape of this function (which is a piecewise-linear function of η) we get that the optimal η, resulting in minimal ρ, is

$$\eta_{\text{opt}} = \frac{2}{\lambda_{\max} + \lambda_{\min}}.$$

The minimal ρ one can achieve is

$$\rho_{\min} = 1 - \eta_{\text{opt}}\lambda_{\min} = 1 - \frac{2\lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}}.$$

Defining $\kappa := \frac{\lambda_{\max}}{\lambda_{\min}}$ (read "kappa"), we can rewrite the optimal ρ as

$$\rho_{\min} = \frac{\kappa - 1}{\kappa + 1}.$$

Hence one notices that the speed at which gradient descent converges depends on the number κ, which in turn depends only on the eigenvalues of $\boldsymbol{X}^\top \boldsymbol{X}$! This number κ is called the *condition number* of $\boldsymbol{X}^\top \boldsymbol{X}$, and bears a very important geometric meaning. Look at the contour lines of the function L (which is a quadratic form, see Section 1.6), which are concentric ellipsoids as depicted in Figure 5.1. The condition number tells us how "spherical" these ellipsoids are. If $\kappa \approx 1$, these ellipsoids are much like spheres, while for $\kappa \gg 1$ it means the ellipsoids are very "stretched and distorted" (the difference between length of the longest and the shortest semi-axes is large).

It is evident that for a very large condition number, $\rho_{\min}$ is close to one, making convergence rather slow even with the optimal step size. In this situation, we usually say that the optimization problem is *ill-conditioned* in the sense that magnitude of loss change varies drastically for different directions due to the "stretched and distorted" shape of the ellipsoidal contour lines (see Figure 5.1). In contrast, in a well-conditioned optimization problem, the condition number κ is close to one, meaning that the magnitude of loss change in different directions is similar. This corresponds to having eigenvalues of similar magnitude and contours that are close to spherical in shape.

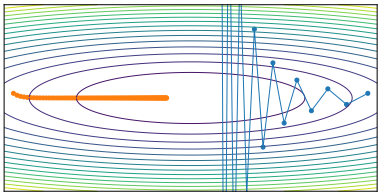


Figure 5.1: First iterates of gradient descent on an ill-conditioned loss. The left trajectory is GD with low learning rate (0.01) and 100 iterations, and the right trajectory is with higher learning rate (0.06) and only 15 iterations. The iterations overshoot the optimal point (located at the middle).

As one can see in Figure 5.1, the success of gradient descent largely depends on how the loss landscape is conditioned. For well-conditioned landscapes, gradient descent finds the optimum with only a few number of steps. If, however, the loss landscape is ill-conditioned and we have a low learning rate, gradient descent makes almost no progress in *flat regions* and becomes impractical. Unfortunately, increasing the learning rate is futile, as it causes gradient descent to oscillate over the ravine. Even worse, gradient descent often diverges in ill-conditioned landscapes with a too high learning rate.

5.4 Other Gradient-Based Methods

As discussed above, gradient descent might not be suitable in all cases due to the issues caused by ill-conditioned landscapes. Thus, numerous alternatives have been developed, attempting to alleviate the problems that come with gradient descent.

5.4.1 Momentum-based Methods

A class of methods, inspired by physics, proposes adding a momentum-based quantity to the update term, meaning that we add the gradients from previous updates to the current update as

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t + \alpha(\boldsymbol{w}^t - \boldsymbol{w}^{t-1}) - \eta \nabla L(\boldsymbol{w}^t).$$

Here, β is the momentum coefficient. Usually, we let $\alpha = \beta \cdot \eta$, with $\beta \in [0, 1)$. By expanding the recurrence relation above, one observes that the influence of previous gradients decays exponentially, and for $\beta = 0$ we recover plain gradient descent. The effect of momentum is that the oscillations of gradient descent in ill-conditioned landscapes (cf. Figure 5.1) are dampened and consequently it yields faster convergence.

5.4.2 Adaptive Methods

Another class of optimization procedures uses parameter-specific learning rates to improve the convergence rate. One such method uses the idea of calculating the learning rate based on previous changes in each coordinate. Intuitively, the coordinates which have already changed a lot, will have smaller stepsize. For w_i denoting the i^{th}-coordinate of $\boldsymbol{w}$, the update is as follows:

$$w_i^{t+1} = w_i^t - \frac{\eta}{\sqrt{\delta_i^t + \gamma}} \frac{\partial L}{\partial w_i}(\boldsymbol{w}^t), \quad i = 1, \ldots, d$$

with $\delta_i^t = (w_i^t - w_i^{t-1})^2$, and $\gamma > 0$ to prevent division by zero.

Other popular adaptive methods one can mention are Adam by Kingma and Ba 2017, AdaGrad by Duchi, Hazan, and Singer 2011, and RMSProp by G. Hinton 2011.

5.4.3 Second Order Methods

A third class of optimization methods makes use of second order information of L through its Hessian. The Hessian provides a richer description of the geometry of the function around a point, and this helps optimization. However, computing the Hessian can be substantially more expensive than computing the gradient. While there are methods to mitigate this, we do not explore second-order optimization methods in this class.

5.4.4 Stochastic Gradient Method

This section covers a popular variation of gradient descent that is particularly useful for training machine learning models on large-scale datasets. Its popularity stems from its simplicity and low computational complexity.

Recall that the training loss for a parametrized function class F on a training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ is given as

$$L(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} \ell(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i).$$

In every iteration of the gradient descent algorithm, the gradient must be calculated at every point in the training set

$$\nabla L(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{w}} \ell(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i),$$

which makes the gradient descent algorithm costly. Concretely, at each iteration $\mathcal{O}(nd)$ variables must be stored in the memory (for the dataset) and $\mathcal{O}(n \times$ cost of computing $\nabla_{\boldsymbol{w}}\ell)$ computations are done. To reduce space and computational complexity, it is common to restrict each update to randomly selected data points in the training set. For instance, in *minibatch stochastic gradient descent* only a random subset $\mathcal{S} \subset \{1, 2, \ldots, n\}$ of points is used to calculate the update direction at each iteration, i.e.,

$$\nabla L_{\mathcal{S}}(\boldsymbol{w}) = \frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} \nabla_{\boldsymbol{w}} \ell(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i).$$

When $\mathcal{S}$ is just one random data point, the algorithm is simply called *stochastic gradient descent* (SGD).[7]

[7] Strictly speaking, the naming is misleading (see next paragraph) and it is more precise to think of SGD as a stochastic gradient method.

Algorithm 3: Minibatch Stochastic Gradient Descent with batch size $|\mathcal{S}|$.

$t \leftarrow 0$
$\boldsymbol{w}^0 \leftarrow \boldsymbol{w}_{\text{initial guess}}$
repeat
  Choose a random subset $\mathcal{S} \subseteq \{1, \ldots, n\}$
  $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t - \eta \nabla L_{\mathcal{S}}(\boldsymbol{w}^t)$
  $t \leftarrow t + 1$
until $\left\|\boldsymbol{w}^t - \boldsymbol{w}^{t-1}\right\|_2 \leq \epsilon$
Return $\boldsymbol{w}^t$

Since only a subset of training points is used to calculate the update direction, the argument in Theorem 5.2 (where we show GD is descending) does not hold anymore; that is, the update direction is not necessarily a direction of descent for small enough stepsize. In particular, the training loss might increase at some iterations and the path taken by the iterates becomes much more erratic (see Figure 5.2). However, when data points in set $\mathcal{S}$ are sampled uniformly with replacement, the SGD update direction is a descent direction *in expectation*, as the following lemma suggests.

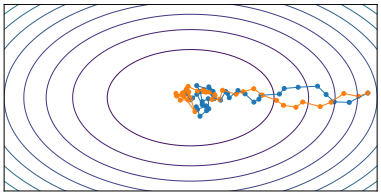


Figure 5.2: First iterates of stochastic gradient descent. The blue and orange curve show two runs of SGD with the same initial point.

Lemma 5.8. *Suppose we select k data points independently and uniformly at random from $\{1, \ldots, n\}$, and put them inside the set $\mathcal{S}$. Then, given the data is fixed, the expected value of the stochastic gradient $\nabla L_{\mathcal{S}}$ is identical to ∇L, that is, $\mathbb{E}_{\mathcal{S}}\left[\nabla L_{\mathcal{S}}(\boldsymbol{w})\right] = \nabla L(\boldsymbol{w})$.*

Remark. Before we start the proof, we stress the fact that the set $\mathcal{S}$ can have repeated elements (it is actually a so-called multi-set).

Proof. Let us denote the randomly selected data points by $I_1, \ldots, I_k$ with k being the size of set $\mathcal{S}$ and note that $I_l = I_m$ might be true

for some $l \neq m$ and $l, m \in \{1, 2, \dots, k\}$. We can now compute the expected value of $\nabla L_{\mathcal{S}}(\boldsymbol{w})$ when the data is fixed:

$$\begin{aligned}
\mathbb{E}_{\mathcal{S}}\left[\nabla L_{\mathcal{S}}(\boldsymbol{w})\right] &= \mathbb{E}_{\mathcal{S}}\left[\frac{1}{k}\sum_{j=1}^{k}\nabla_{\boldsymbol{w}}\ell(f_{\boldsymbol{w}}(\boldsymbol{x}_{I_j}), y_{I_j})\right] \\
&= \frac{1}{k}\sum_{j=1}^{k}\mathbb{E}_{I_j}\left[\nabla_{\boldsymbol{w}}\ell(f_{\boldsymbol{w}}(\boldsymbol{x}_{I_j}), y_{I_j})\right] \\
&= \frac{1}{k}\sum_{j=1}^{k}\left(\sum_{i=1}^{n}\frac{1}{n}\nabla_{\boldsymbol{w}}\ell(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i)\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\nabla_{\boldsymbol{w}}\ell(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i) \\
&= \nabla L(\boldsymbol{w})
\end{aligned}$$

where the second inequality is due to linearity of expectation, and the third equality is because $\mathbb{P}\left(I_j = i\right) = \frac{1}{n}$ for all $i = 1, \dots, n$, and the fourth equality is due to the fact that all the terms in the outer sum are the same. □

As a consequence, one can show that Theorem 5.2 still holds if we weaken the statement to hold in expectation.

In practice, instead of sampling with replacement, we usually cycle through the data set by partitioning the training set into minibatches $\mathcal{S}_1, \dots, \mathcal{S}_b$. After we went through the entire data set, we say that one *epoch* has passed and we start over again. One can also shuffle the samples before each epoch to avoid some convergence issues.

The size of the minibatch has a big impact on the *variance* of the updates as well as the computational complexity. As the batch size increases, the variance decreases but the computational complexity increases: The variance decreases, since we estimate the gradient based on more data points, making our estimate more reliable (compare Figure 5.3). In the extreme case of a single partition, we recover gradient descent which has zero variance but high computational complexity compared to minibatch gradient descent.

Remark. Having more variance in the stochastic gradient is not always a bad thing! This might come handy when L has bad stationary points (this happens, e.g., when L is *non-convex*). While gradient descent might get stuck in saddle points or other bad local minima, minibatch gradient descent can escape those saddle points thanks to the randomness in the update direction.

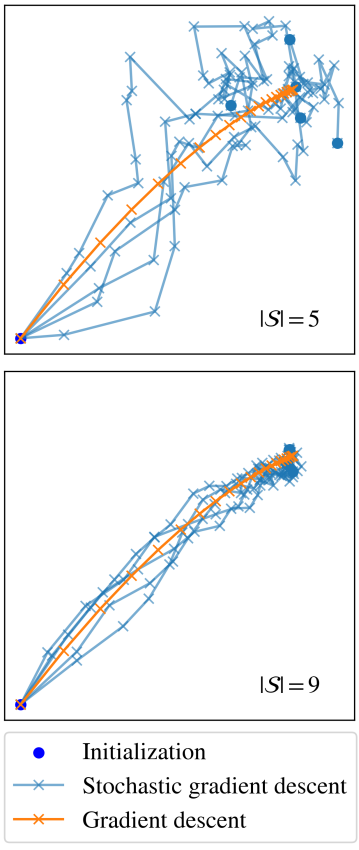


Figure 5.3: Varying the batch size $|\mathcal{S}|$ influences the variance of SGD. We plot five paths of SGD on a linear regression problem with 10 samples. For higher batch sizes, the paths are closer to gradient descent.

5.5 Convexity

For an appropriately chosen step size, we have shown in Theorem 5.2 that gradient descent decreases the function value in every iteration, and we saw that in the linear regression setting, gradient descent converges to a minimizer of the loss. To make a statement

about what gradient descent converges to for general functions f, we need the notion of *convexity*. As it will turn out, a stationary point of a convex function is guaranteed to be a global minimum, and under some assumptions gradient descent converges to this minimum. To show this, we first need to introduce the notion of convexity.

Definition 5.9 (Convexity)**.** A set $\mathcal{W} \subseteq \mathbb{R}^d$ is convex if it contains the line segment between any two of its points. That is, for all $\boldsymbol{w}, \boldsymbol{v} \in \mathcal{W}$ and all $\lambda \in [0,1]$, $\lambda \boldsymbol{w} + (1-\lambda)\boldsymbol{v} \in \mathcal{W}$. Let $f : \mathcal{W} \to \mathbb{R}$ be a function defined over a convex set $\mathcal{W}$. We say f is convex if and only if for all $\boldsymbol{w}, \boldsymbol{v} \in \mathcal{W}$ and all $\lambda \in [0,1]$,

$$f(\lambda \boldsymbol{w} + (1-\lambda)\boldsymbol{v}) \leq \lambda f(\boldsymbol{w}) + (1-\lambda) f(\boldsymbol{v}).$$

Geometrically speaking, a function is convex if for any $\boldsymbol{w}, \boldsymbol{v} \in \mathcal{W}$, every point on the line segment connecting $(\boldsymbol{w}, f(\boldsymbol{w}))$ and $(\boldsymbol{v}, f(\boldsymbol{v}))$ lies *above or on* the graph of the function between $\boldsymbol{w}$ and $\boldsymbol{v}$ (compare Figure 5.4 and Figure 5.5).

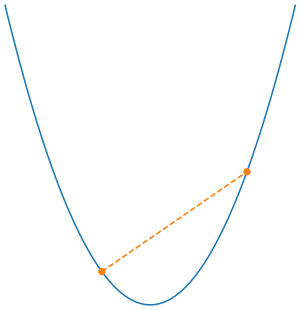


Figure 5.4: All line segments connecting points on a graph of a convex function lie above the graph of the function.

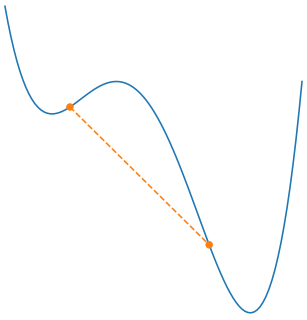


Figure 5.5: Some line segments connecting points on a graph of a non-convex function lie below the graph of the function.

The characterization of convexity in Definition 5.9 is known as *zeroth-order condition*. Alternatively, if the function is once or twice differentiable, we can characterize convexity with the following two equivalent conditions, respectively:

First-order condition: Let f be continuously differentiable. f is convex if and only if for all $\boldsymbol{w}, \boldsymbol{v} \in \mathcal{W}$, the value of the linear approximation of f around $\boldsymbol{w}$ evaluated at $\boldsymbol{v}$ must be smaller than $f(\boldsymbol{v})$, that is,

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

Said differently, the graph of the linear approximation of f at any point $\boldsymbol{w}$ lies below the graph of f, cf. Figure 5.6.

Second order condition: Let f be twice continuously differentiable. f is convex if and only if its Hessian $D^2 f(\boldsymbol{w})$ is positive semi-definite for all $\boldsymbol{w} \in \mathcal{W}$. This is equivalent to having non-negative curvature at every $\boldsymbol{w}$. See also Section 2.5.

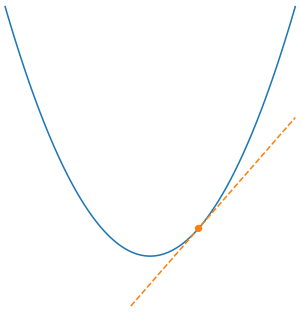


Figure 5.6: Visualization of the underestimation of the linear approximation that is given by the 1st order condition for a convex function

These three characterizations can be used to establish convexity of functions. Moreover, certain operations are known to preserve convexity. Using these operations, we can show convexity by starting with functions that are known to be convex, and then apply one of the following operations repeatedly:

Lemma 5.10. *1. The weighted sum $\alpha f + \beta g$ of two convex functions f and g is convex if $\alpha, \beta \geq 0$.*

2. The composition $f \circ g$ is convex if f is convex and g is affine, or if f is convex and non-decreasing and g is convex.

3. The pointwise maximum of two convex functions is convex.

Exercise 5.11. Prove Lemma 5.10.

Exercise 5.12. Are the following functions convex?

(i) $h(\boldsymbol{x}) = \|A\boldsymbol{x} + b\|^2$
(ii) $h(x) = (f \circ g)(x)$ for $f(x) = x^2$ and $g(x) = x^2 - 1$?

Exercise 5.13 (Convexity of Linear Least Squares)**.** Prove that the function $L(\boldsymbol{w}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$ is convex in w.

The main benefit of convexity for our discussion is that any stationary point is guaranteed to be a global minimum if our objective function is known to be convex.

Theorem 5.14 (Optimality of Stationary Points)**.** *Let $f : \mathcal{W} \to \mathbb{R}$ be a* convex *and differentiable function, defined over the convex (open) set $\mathcal{W} \subseteq \mathbb{R}^d$. Further, suppose that $\boldsymbol{w}^* \in \mathcal{W}$ is a stationary point of f, that is $\nabla f(\boldsymbol{w}^*) = \boldsymbol{0}$. Then, $\boldsymbol{w}^*$ is a global minimum of f.*

Proof. Since $\boldsymbol{w}^*$ is a stationary point, we have $\nabla f(\boldsymbol{w}^*) = \boldsymbol{0}$. By applying the first-order condition of convexity at $\boldsymbol{w} = \boldsymbol{w}^*$, we get

$$\forall \boldsymbol{v} \in \mathcal{W}, \quad f(\boldsymbol{v}) \geq f(\boldsymbol{w}^*) + \langle \nabla f(\boldsymbol{w}^*), \boldsymbol{v} - \boldsymbol{w}^* \rangle = f(\boldsymbol{w}^*)$$

which establishes $f(\boldsymbol{v}) \geq f(\boldsymbol{w}^*)$ for all $\boldsymbol{v} \in \mathcal{W}$. □

Unfortunately, convexity alone does not imply that this global minimum is unique. To ensure uniqueness, we define *strong convexity* below.

Definition 5.15 (Strong Convexity)**.** Let $f : \mathcal{W} \to \mathbb{R}$ be a function and $\mathcal{W} \subseteq \mathbb{R}^d$ a convex set. f is strongly convex with parameter $m > 0$ if and only if for all $\boldsymbol{w}, \boldsymbol{v} \in \mathcal{W}$ and $\lambda \in [0, 1]$

$$f(\lambda \boldsymbol{w} + (1-\lambda)\boldsymbol{v}) \leq \lambda f(\boldsymbol{w}) + (1-\lambda) f(\boldsymbol{v}) - \frac{m}{2}\lambda(1-\lambda)\|\boldsymbol{w} - \boldsymbol{v}\|_2^2$$

holds.

Compared to Definition 5.9, strong convexity implies that the line segment connecting any two points of the graph of the function is *strictly above* $f(\lambda \boldsymbol{w} + (1-\lambda)\boldsymbol{v})$, cf. Figure 5.8. This property is called strict convexity. *Importantly*, strong convexity should not be confused with strict convexity, because strong convexity implies strict convexity, but the same does not hold vice versa.

Note that we can relate strong convexity to convexity in the following way: A function f is strongly convex with parameter $m > 0$ if and only if $f - \frac{m}{2}\|\cdot\|_2^2$ is convex. Intuitively, this means that f is at least as convex as a quadratic function.

As before, Definition 5.15 is known as the zeroth order condition and we can define the first- and second-order conditions for strong convexity almost analogously to the convex case.

First order condition: Let f be continuously differentiable. f is m-strongl convex, if and only if for all $\boldsymbol{w}, \boldsymbol{v} \in \mathcal{W}$,

$$f(\boldsymbol{v}) \geq f(\boldsymbol{w}) + \nabla f(\boldsymbol{w})^\top (\boldsymbol{v} - \boldsymbol{w}) + \frac{m}{2}\|\boldsymbol{w} - \boldsymbol{v}\|^2 .$$

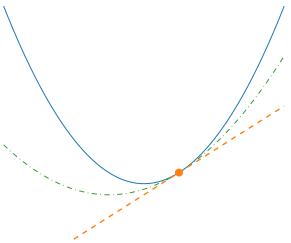


Figure 5.7: If f is strongly convex, at every point there is always a quadratic function (green) that fits between f and the tangent of f.

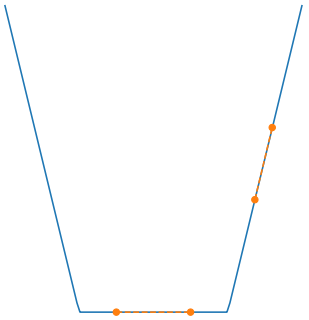


Figure 5.8: A convex, but not strongly convex function. Informally speaking, the notion of strong convexity excludes functions with "linear parts".

Second order condition: Let f be twice continuously differentiable. f is m-strongly convex, if and only if $D^2 f(\boldsymbol{w}) - mI$ is positive semi-definite for all $\boldsymbol{w} \in \mathcal{W}$, i.e.,

$$D^2 f(\boldsymbol{w}) \succcurlyeq mI.$$

This condition implies that all of the eigenvalues of the Hessian are larger than $m > 0$.

Having introduced strong convexity, we can use it to prove the uniqueness of the global minimum.

Theorem 5.16 (Uniqueness of Global Minimum)**.** *Let $f : \mathcal{W} \to \mathbb{R}$ be a strongly convex and differentiable function, defined over the convex (open) set $\mathcal{W} \subseteq \mathbb{R}^d$, and suppose that $\boldsymbol{w}^* \in \mathcal{W}$ is a stationary point of f, that is, $\nabla f(\boldsymbol{w}^*) = \mathbf{0}$. Then, $\boldsymbol{w}^*$ is the unique global minimizer of f.*

Proof. Observe that strong convexity implies convexity. Thus, we can make use of Theorem 5.14 to see that $\boldsymbol{w}^*$ is a global minimum. It remains to prove that $\boldsymbol{w}^*$ is unique. We argue by contradiction. Suppose that there is another global minimum $\boldsymbol{v}^* \neq \boldsymbol{w}^*$ which attains the same value $f(\boldsymbol{w}^*) = f(\boldsymbol{v}^*)$. Using the definition of strong convexity, we get

$$\begin{aligned} f(\tfrac{1}{2}\boldsymbol{w}^* + \tfrac{1}{2}\boldsymbol{v}^*) &\leq \frac{1}{2} f(\boldsymbol{w}^*) + \frac{1}{2} f(\boldsymbol{v}^*) - \frac{m}{8} \|\boldsymbol{w}^* - \boldsymbol{v}^*\|^2 \\ &< \frac{1}{2} f(\boldsymbol{w}^*) + \frac{1}{2} f(\boldsymbol{v}^*) \\ &= f(\boldsymbol{w}^*) \end{aligned}$$

which is a contradiction, as $\boldsymbol{w}^*$ was a global minimum. □

We now come back to our analysis of gradient descent for general functions $f : \mathbb{R}^d \to \mathbb{R}$. It turns out that gradient descent converges to the unique global minimizer of a strongly convex function f.[8]

[8] Technically, we need an additional assumption which deals with problems arising for too large step-size η. This is discussed in the bonus material.

Theorem 5.17 (Informal)**.** *Suppose that f is differentiable and strongly convex. Then there exists a unique minimizer $\boldsymbol{w}^*$ of f and gradient descent with sufficiently small step size and arbitrary initialization $\boldsymbol{w}^0$ satisfies*

$$\lim_{t \to \infty} \boldsymbol{w}^t = \boldsymbol{w}^*.$$

Remark. The assumption that we make on the function f is *global*, meaning that f has to be strongly convex on all of $\mathbb{R}^d$. In many modern machine learning procedures, such as learning with neural networks, we encounter optimization problems where this assumption is not true and the objective function f is not even convex. However, the argument from Theorem 5.18 carries over to these problems when the function is *locally* strongly convex around a *local* minimum, and we initialize gradient descent within this local neighbourhood.

Bonus Material

We provide a formal version of the Theorem by introducing another assumption on the function f. Assume that f is differentiable and there exists a $L > 0$ such that for all $\boldsymbol{w}, \boldsymbol{v} \in \mathbb{R}^d$

$$f(\boldsymbol{w} + \boldsymbol{v}) \leq f(\boldsymbol{w}) + \langle \nabla f(\boldsymbol{w}), \boldsymbol{v} \rangle + \frac{L}{2} \|\boldsymbol{v}\|_2^2 .$$

If f satisfies this, we say that f is L-smooth, not to be confused with the notion of smoothness from differential calculus. This condition implies that f increases at most quadratically around any point, which allows us to bound the norm of the gradient in each iteration of gradient descent. We can make use of this to prove the following Theorem, which is the culmination of our discussion on convexity.

Theorem 5.18 (Convergence of Gradient Descent)**.** *Suppose that f is differentiable, m-strongly convex and L-smooth. Then there exists a unique minimizer $\boldsymbol{w}^*$ of f and gradient descent with step size $\eta = 1/L$ and arbitrary initialization $\boldsymbol{w}^0$ satisfies*

$$\lim_{t \to \infty} \boldsymbol{w}^t = \boldsymbol{w}^* .$$

Proof. The existence of $\boldsymbol{w}^*$ is due to the strong convexity (think about why!) and the uniqueness of $\boldsymbol{w}^*$ is proved in Theorem 5.16.

We start by making use of the m-strong convexity, which implies

$$\left\langle \nabla f(\boldsymbol{w}^t), \boldsymbol{w}^t - \boldsymbol{w}^* \right\rangle \geq f(\boldsymbol{w}^t) - f(\boldsymbol{w}^*) + \frac{m}{2} \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 .$$

Since f is smooth with some $L > 0$, we have that

$$\begin{aligned} f(\boldsymbol{w}^*) &\leq f(\boldsymbol{w}^{t+1}) \\ &\leq f(\boldsymbol{w}^t) - \eta \left\|\nabla f(\boldsymbol{w}^t)\right\|_2^2 + \frac{L\eta^2}{2} \left\|\nabla f(\boldsymbol{w}^t)\right\|_2^2 \\ &= f(\boldsymbol{w}^t) - \frac{1}{2L} \left\|\nabla f(\boldsymbol{w}^t)\right\|_2^2 \end{aligned}$$

where the equality follows from our choice of η. Using these two inequalities, we get the bound

$$\begin{aligned} \left\|\boldsymbol{w}^{t+1} - \boldsymbol{w}^*\right\|_2^2 &= \left\|\boldsymbol{w}^t - \eta \nabla f(\boldsymbol{w}^t) - \boldsymbol{w}^*\right\|_2^2 \\ &= \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 - 2\eta \left\langle \nabla f(\boldsymbol{w}^t), \boldsymbol{w}^t - \boldsymbol{w}^* \right\rangle + \eta^2 \left\|\nabla f(\boldsymbol{w}^t)\right\|_2^2 \\ &\leq \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 - 2\eta \left(f(\boldsymbol{w}^t) - f(\boldsymbol{w}^*) + \frac{m}{2} \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 \right) + \eta^2 \left\|\nabla f(\boldsymbol{w}^t)\right\|_2^2 \\ &\leq \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 - 2\eta \left(f(\boldsymbol{w}^t) - f(\boldsymbol{w}^*) + \frac{m}{2} \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 \right) + 2L\eta^2 \left(f(\boldsymbol{w}^t) - f(\boldsymbol{w}^*) \right) \\ &= (1 - \eta m) \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 + 2\eta(L\eta - 1) \left(f(\boldsymbol{w}^t) - f(\boldsymbol{w}^*) \right) \\ &= \left(1 - \frac{m}{L}\right) \left\|\boldsymbol{w}^t - \boldsymbol{w}^*\right\|_2^2 \end{aligned}$$

where the last equality again follows from our choice $\eta = 1/L$. Applying this bound inductively and taking the limit as $t \to \infty$ yields the result. □

6

Model evaluation and selection

In this chapter, we introduce techniques to quantitatively evaluate the performance of a trained machine learning model. We define in mathematical terms what it means for a model to be *good* and discuss the bias-variance trade-off underlying the *generalization error*, which can be controlled using *regularization techniques*. Finally, we introduce cross-validation - a standard procedure to use data to choose the best among different methods, e.g. obtained by varying the regularization coefficient, function class or optimization hyperparameters.

Roadmap

In Section 6.1 we define the generalization and the estimation error and discuss their similarities and limitations. In Section 6.2 we explain how the generalization error can be estimated by splitting the training dataset, and in Section 6.3 we study the technique of cross-validation.

Learning Objectives

After reading this chapter you should

- understand the definitions of estimation and generalization error
- understand why it is necessary to split the training dataset in order to estimate the generalization error efficiently
- understand the difference between training, validation, and test error
- be able to explain the algorithm of cross-validation and the effect of the parameter K of this algorithm

In Chapter 4, and more specifically in Section 4.1, we took a first look into the general pipeline of machine learning. Figure 6.1 is a concise illustration of this framework.

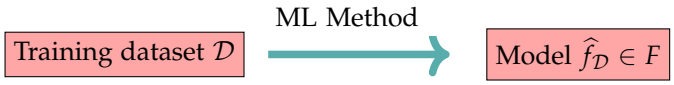


Figure 6.1: General machine learning pipeline. The practitioner chooses an ML method that returns a model $\widehat{f}_{\mathcal{D}}$ given a training data set $\mathcal{D}$.

Choosing the ML method so far consists of choosing the objective function (in particular the loss function) to optimize, the function class over which to minimize, and the algorithm that is used to solve the optimization problem. [1]

[1] There are also other ML methods that are not based on loss minimization, such as nearest neighbor classifiers, decision trees etc.

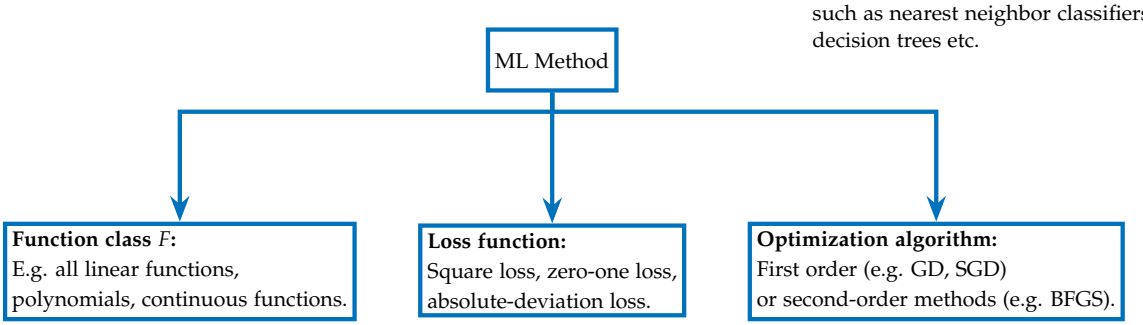


Figure 6.2: The trident of an ML method. The ML scientist should choose the function class, the loss function, and the optimization algorithm.

In Chapter 5 we discussed an answer to the question:

If we choose a loss function ℓ and a function class F that is parametrized by a vector $\boldsymbol{w} \in \mathbb{R}^d$, how can we find the parameter vector $\hat{\boldsymbol{w}}$ that minimizes the average loss over the training samples?

However, when the scientist has a specific training dataset $\mathcal{D}$ at their disposal, it is usually unclear which function class F, which loss function ℓ, and which optimization algorithm to choose. Hence, in this chapter, we target the following question instead:

If $M_1, \ldots, M_r$ are some of the potential methods (i.e. choices for the function class F, the loss function, and the optimization algorithm), which of them should we choose?

Intuitively, we should choose a method that "performs the best" when applied to data points outside the training dataset. More precisely, we explore how to compare the so-called *generalization* properties of the corresponding final models that different ML methods output. To do this, we use a quantity called the *generalization error*, which we define and analyze theoretically in Section 6.1. In Section 6.2, we then discuss how to estimate it empirically.

Before we discuss the generalization error, we would like to clarify how a few key terms are used in this chapter. Given training data $\mathcal{D}$, each *method* M outputs a *model* $M(\mathcal{D}) = \widehat{f}_{\mathcal{D}}$ (where the subscript D may sometimes be omitted), that models the relation between $\boldsymbol{x}$ and y. In linear regression for example, this function is $\widehat{f}(\boldsymbol{x}) = \widehat{\boldsymbol{w}}^T \boldsymbol{x}$, where $\widehat{\boldsymbol{w}}$ is obtained as in Equation 4.4. Oftentimes in the ML literature, the term "model" is used to refer to both the final trained model $\widehat{f}_{\mathcal{D}}$, and the method M (in particular in the term "model selection", which really is a procedure for "method selection"). We will also use both terms interchangeably in the chapters to follow, as their meaning should always be clear from the context. In particular, we will use the following three terms:

- *model selection* refers to the choice of the best among candidate methods $M_1, \ldots, M_r$.
- *model training* refers to the optimization of the weights of a chosen method M.
- *model evaluation* refers to the estimation of how good a model $\widehat{f}$ is. It is the estimation of the generalization error of the model, where the term *generalization error* will be formally defined later, in Section 6.1.

6.1 Estimation, Prediction and Generalization errors

We now move to the first fundamental question of this chapter: how can we formally define the "goodness" of a model during test time?

Assume for example that there exists an *ideal function* $f^\star : \mathcal{X} \to \mathcal{Y}$ that the data scientist would like to learn. In this case, if we learn $\widehat{f}$ using training data, we would like it to be as close to the ground truth function $f^\star$ as possible.

In general, a good predictive model $\widehat{f}$ during test time is one that is close to $f^\star$ on some unseen input x sampled independently of the training dataset, for instance, a model that has a small squared loss

$$\ell\left(\widehat{f}(x), f^\star(x)\right) = \left(\widehat{f}(x) - f^\star(x)\right)^2$$

on a randomly sampled test point x. This error is called the estimation error. If our function class can be parameterized as $\{f_w,\ w \in \mathbb{R}^d\}$, then the estimation error is closely related to the quantity $\|\hat{\boldsymbol{w}} - \boldsymbol{w}^\star\|^2$. [2]

[2] Note that so far we have only used the square loss for evaluation - this does not necessarily have to be the case, see for example the next chapter on classification. The concepts of estimation, prediction and generalization error apply to arbitrary losses, and are instantiated for regression in this chapter purely for concreteness.

However, we usually do not have access to $f^\star(X)$ for any point X, not even in our training data. Instead, the labels Y are "noisy measurements" of $f^\star(X)$. This noise could be inherent, for instance, due to unobserved variables: for a fixed $\boldsymbol{x}$, differences in these unobserved variables will lead to different targets y. Other sources of noise can include measurement errors or/and quantization by the measurement device.

Hence, an alternative error to consider is the *prediction error*: Instead of infeasible evaluation of $f^\star$, we compute the prediction error as

$$\ell\left(\widehat{f}(x), y\right) = \left(\widehat{f}(x) - y\right)^2$$

for a function $\widehat{f} \in F$.

Above, we have defined the prediction error for just one datapoint (x, y). However, during test time we normally observe many inputs and thus would like not only the individual, but the average estimation/prediction error over these inputs to be small.

In machine learning, the most relevant metric is the *generalization error*, that is, the "average" prediction error over possible test inputs. In order to make the term "average" more precise, we now discuss how we can view both training and test data as samples from a probability distribution.

6.1.1 Probabilistic model of the data

In general, we can think of test features x as having different probability of occurrence. For example, for a randomly sampled house, its size is likely to be within some realistic range, and very unlikely to be less than $10\,m^2$ or more than $10000\,m^2$. Statistically speaking, we can assume that the inputs during test time are independent and identically distributed samples from a *probability distribution* $\mathbb{P}_X$. The *expected estimation error* over an unseen test input is then defined as

$$\mathbb{E}_X \ell\left(f(X), f^\star(X)\right).$$

The expectation can be computed as in Subsection 3.4.1. The subscript X denotes that X is the random quantity over which we average and that the expectation is taken with respect to the probability distribution of X in $\mathcal{X}$.

As shown in Figure 6.3, if the inputs and labels are real scalars, then the expected estimation error corresponds to the area between the graphs of the functions $f, f^\star$ weighted by the distribution of the inputs. Hence, if we could compute or at least approximate the estimation error for each model $\widehat{f}^i_{\mathcal{D}} = M_i(\mathcal{D})$, we could simply pick the method M_i that yields the model with the smallest estimation error.

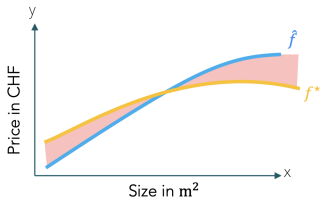


Figure 6.3: The area of the shaded region is the expected estimation error of the estimator $\widehat{f}$. In this particular example, x is the area of an apartment, and y is its price.

As discussed before, we usually **do not have direct access to** $f^\star$ for *any* input x, so the (expected) estimation error is impossible to compute nor approximate. Instead, for any input x, the label y that we can observe is usually a noisy version of $f^\star(x)$

$$y = f^\star(x) + \varepsilon, \tag{6.1}$$

where often the noise random variable ε has zero mean, i.e. $\mathbb{E}[\varepsilon] = 0$, and we denotes its variance by $\sigma^2 = \mathrm{Var}(\varepsilon) = \mathbb{E}\left[\varepsilon^2\right] - \left(\mathbb{E}[\varepsilon]\right)^2 = \mathbb{E}\left[\varepsilon^2\right]$.

Formally, the noise distribution defines a conditional distribution of Y given X. In particular, if $\mathbb{P}_\varepsilon$ is the noise distribution,

$$\mathbb{P}_{Y|X}(Y \leq a) = \mathbb{P}_\varepsilon(\varepsilon \leq a - f^\star(x)). \tag{6.2}$$

Note that we have now defined a joint distribution over X, Y that we denote as $\mathbb{P}_{X,Y}$. A test data point (x, y) can now be viewed as a random sample from $\mathbb{P}_{X,Y}$ (with the target y being unobserved).

An illustration of $f^\star$, the distribution, and sampled data is shown in Figure 6.4 (in this example, x denotes the size of an apartment and y its price). Here, $f^\star$ would correspond to the average market price of an apartment for a given size, whereas the observed

samples do not exactly follow $f^\star$, which can be for example due to unobserved variables such as a different features of the house (such as, e.g., the number of bathrooms) that might differ for a house of the same size, or the timing/incentives of the sale.

We now formally introduce the ultimate performance metric known as the *generalization error*[3] (also called the *population risk*), which is the expected prediction error:

Definition 6.1 (Generalization error)**.** Let $f : \mathcal{X} \to \mathbb{R}$ be a function, and assume that the data (X, Y) during test time comes from the distribution $\mathbb{P}_{X,Y}$ on $\mathcal{X} \times \mathcal{Y}$. We define the generalization error of f as

$$L\left(f; \mathbb{P}_{X,Y}\right) = \mathbb{E}_{X,Y} \ell\left(f(X), Y\right).$$

Remark. Notice that, unlike the (expected) estimation error, the generalization error can be used even without assuming the existence of a ground-truth function. Furthermore, due to the noise, even the ground truth function has a strictly positive generalization error: indeed,

$$\mathbb{E}_{X,Y} \ell\left(f^\star(x_i), y_i\right) = \mathbb{E}_{X,Y}\left(f^\star(x_i) - y_i\right)^2 = \varepsilon_i^2 > 0.$$

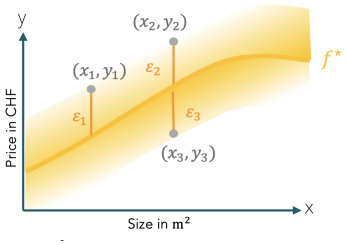


Figure 6.4: Ground truth function and noisy training data. The noise ε has finite variance σ^2, so it is more likely that the data will concentrate around the graph of $f^\star$. The darker yellow color denotes a larger probability of data points being located there.

[3] In Lemma 6.2, we show that the generalization error is a good proxy for the expected estimation error.

6.1.2 Expected estimation error vs. generalization error

Recall that our goal is the accurate estimation of $f^\star$, so the quantity that we would ideally like to evaluate is the expected estimation error. In this section, we prove why evaluating a model via the generalization error is almost equivalent to evaluating its estimation error.

The following lemma shows that if a model $\widehat{f}$ has a low generalization error, it implies that $\widehat{f}$ is a good estimation of $f^\star$, i.e. that the expected estimation error is also small.

Lemma 6.2. *Under the statistical model Equation 6.1, and under the square loss, the generalization error and the expected estimation error of a model $\widehat{f}$ satisfy the relation*

$$L\left(\widehat{f}; \mathbb{P}_{X,Y}\right) = \mathbb{E}_X\left(\widehat{f}(X) - f^\star(X)\right)^2 + \sigma^2. \tag{6.3}$$

Proof. Observe that, for any point x, since $y = f^\star(x) + \varepsilon$, it follows that the prediction error is decomposed as

$$\begin{aligned}
\ell\left(\widehat{f}(x), y\right) &= \left(\widehat{f}(x) - y\right)^2 \\
&= \left(\widehat{f}(x) - f^\star(x) - \varepsilon\right)^2 \\
&= \left(\widehat{f}(x) - f^\star(x)\right)^2 + \varepsilon^2 - 2\varepsilon\left(\widehat{f}(x) - f^\star(x)\right).
\end{aligned}$$

Thus, the generalization error is equal to

$$\begin{aligned} L\left(\widehat{f}; \mathbb{P}_{X,Y}\right) &= \mathbb{E}_{X,\varepsilon}\ell\left(\widehat{f}(X), Y\right) \\ &= \mathbb{E}_{X,\varepsilon}\left[\widehat{f}(X) - f^\star(X)\right]^2 + \mathbb{E}_{X,\varepsilon}\left[\varepsilon^2\right] \\ &- 2\mathbb{E}_{X,\varepsilon}\left[\varepsilon\left(\widehat{f}(X) - f^\star(X)\right)\right] \\ &= \mathbb{E}_X\left[\widehat{f}(X) - f^\star(X)\right]^2 + \mathbb{E}_\varepsilon\left[\varepsilon^2\right] \\ &- 2\mathbb{E}_\varepsilon[\varepsilon] \cdot \mathbb{E}_X\left[\widehat{f}(X) - f^\star(X)\right] \\ &= \mathbb{E}_X\left[\widehat{f}(X) - f^\star(X)\right]^2 + \sigma^2, \end{aligned}$$

where we used the independence of X, ε and the fact that $\mathbb{E}_\varepsilon[\varepsilon] = 0$ and $\mathbb{E}_\varepsilon[\varepsilon^2] = \sigma^2$. □

The last expression consists of a term that is equal to the expected estimation error, and another term that only depends on the noise, and not on the model $\widehat{f}$.[4]

From this derivation, it follows that the generalization error is a valid proxy for the expected estimation error under the statistical model $Y = f^\star(X) + \varepsilon$. In other words, choosing the model with the lowest generalization error guarantees that the expected estimation error is also minimized. For this reason, the generalization error is going to be the central criterion for model selection.

[4] It is also necessary to explain why the subscript was X, ε instead of X, Y, as in the definition of the generalization error. This happens because in the particular statistical model, the only sources of randomness are X, ε. If these are given, then Y is deterministically defined from the equality $Y = f^\star(X) + \varepsilon$.

6.2 *Estimation of the generalization error from data*

From Definition 6.1 of the generalization error, it is clear that if we had access to the joint distribution $\mathbb{P}$ of a test datapoint (x, y), we could evaluate the generalization error directly and thus get information about the "goodness" of our model. However, in reality we don't have access to the joint distribution $\mathbb{P}$, but only to the training dataset $\mathcal{D}$.

Hence, the question becomes: how can we hope to learn a model $\widehat{f}_\mathcal{D}$ that has a low generalization error from just the training samples $\mathcal{D}$? In the standard setting, we assume that the training data consists of independent and identically distributed (*i.i.d.*) samples from the same distribution $\mathbb{P}_{X,Y}$ as during test time [5]. i.e.

$$y_i = f^\star(x_i) + \varepsilon_i,\ i = 1, \ldots, n.$$

[5] we later discuss distributional shifts, where test and training distribution might differ

The methods we considered so far return as $\widehat{f}_\mathcal{D} \in F$ the minimizer of the training error that encourages the predictions $\widehat{f}(x_i)$ to be close to y_i on average over $i = 1, \ldots, n$, i.e.

$$L\left(f; \mathcal{D}\right) = \frac{1}{n}\sum_{i=1}^{n} \ell\left(f(x_i), y_i\right),$$

also called the *empirical risk*[6], i.e. $\widehat{f}_\mathcal{D} = \arg\min_{f \in F} L\left(f; \mathcal{D}\right)$. Note

[6] Beware of the slight abuse of notation in the definitions of the training and generalization error: $L(f; \mathcal{D})$ denotes the training error and $\mathcal{D}$ is a set, while $L(f; \mathbb{P}_{X,Y})$ denotes the generalization error and $\mathbb{P}_{X,Y}$ is a probability distribution.

that contrary to Section 5.1, the notation in this chapter explicitly includes the dependence of the training error on the training data set $\mathcal{D}$.

One first thought would be the following: the practitioner uses the training dataset $\mathcal{D}$ to train a model $\widehat{f}_{\mathcal{D}} \in F$ which we for simplicity assume to minimize the training error, i.e

$$\widehat{f} = \arg\min_{f \in F} L\left(f; \mathcal{D}\right) = \arg\min_{f \in F} \left[\frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)\right].$$

To estimate the generalization error $L\left(\widehat{f}_D; \mathbb{P}_{X,Y}\right)$, the practitioner could use the training loss

$$L\left(\widehat{f}_{\mathcal{D}}; \mathcal{D}\right) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(\widehat{f}_{\mathcal{D}}(x_i), y_i\right)$$

of the given model. Would that be a good estimator of the generalization error?

In fact, $\widehat{f}_{\mathcal{D}}$ is the minimizer of the training error, so it has been designed to *fit the particular set $\mathcal{D}$ of data as best as possible*. Thus, the estimation $L\left(\widehat{f}_{\mathcal{D}}; \mathcal{D}\right)$ is biased, and it is probably a too optimistic estimation of the generalization error. In other words, if we draw another set of data $\mathcal{D}' = \{(x_i', y_i')\}_{i=1}^{n}$ independently from the distribution $\mathbb{P}_{X,Y}$, chances are that the error

$$L\left(\widehat{f}_{\mathcal{D}}; \mathcal{D}'\right) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(\widehat{f}_{\mathcal{D}}(x_i'), y_i'\right)$$

of $\widehat{f}_{\mathcal{D}}$ on this new dataset would be larger than $L\left(\widehat{f}_{\mathcal{D}}; \mathcal{D}\right)$.

An example of the insufficiency of training error to estimate generalization error is illustrated in Figure 6.5.

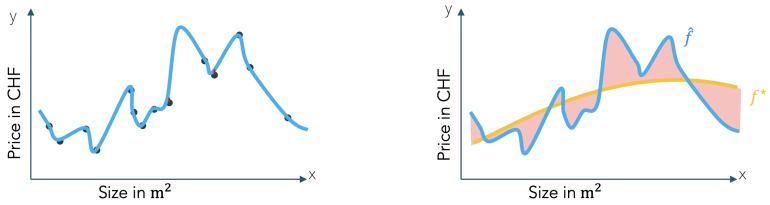


Figure 6.5: The training error is zero, but the expected estimation error, which is the shaded area, is quite significant. From Equation 6.3 the generalization error is also going to be large.

Hence, to estimate the generalization error of a model $\widehat{f}_{\mathcal{D}}$ accurately, we must use data points that are independent of $\mathcal{D}$. This way, we will eliminate any bias induced by the fact that $\widehat{f}_{\mathcal{D}}$ is the training error minimizer on $\mathcal{D}$.

Where could we find additional independent samples? One standard approach is to split the entire available dataset $\mathcal{D}_{\text{full}}$ into two subsets: one training set that we again call $\mathcal{D}$, and one hold-out test set $\mathcal{D}_{\text{test}}$, that is not used during training. This is illustrated in Figure 6.6.

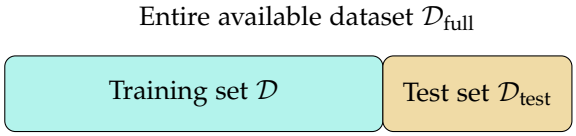


Figure 6.6: The entire dataset is separated in two parts $\mathcal{D}$ and $\mathcal{D}_{\text{test}}$. These parts are disjoint, so the elements of the first are independent of the elements of the second.

To train a model we only use the subset $\mathcal{D}$, while $\mathcal{D}_{\text{test}}$ is left untouched. In other words, during training, we solve the optimization problem

$$\arg\min_{f\in F} L(f;\mathcal{D}) = \arg\min_{f\in F}\left[\frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}} \ell\left(f(x),y\right)\right].$$

After we find the optimal model $\widehat{f}_{\mathcal{D}}$, we utilize the test set $\mathcal{D}_{\text{test}}$ to estimate the generalization error $L\left(\widehat{f}_{\mathcal{D}};\mathbb{P}_{X,Y}\right) = \mathbb{E}_{X,Y}\ell\left(\widehat{f}_{\mathcal{D}}(X),Y\right)$. The estimation for that is the average *test error* over the test dataset:

$$L\left(\widehat{f}_{\mathcal{D}};\mathcal{D}_{\text{test}}\right) = \frac{1}{|\mathcal{D}_{\text{test}}|}\sum_{(x,y)\in\mathcal{D}_{\text{test}}} \ell\left(\widehat{f}_{\mathcal{D}}(x),y\right).$$

According to Theorem 3.51 (Law of large numbers), this estimation is going to be close to the target $\mathbb{E}_{X,Y}\ell\left(\widehat{f}_{\mathcal{D}}(X),Y\right)$ if $|\mathcal{D}_{\text{test}}|$ is large enough. Note that the assumptions of that theorem are satisfied because $\widehat{f}_{\mathcal{D}}$ is independent of the elements of the test set.

Bonus Material

Although we saw in Figure 6.5 that the training error is not a valid guideline for evaluating a model, this phenomenon might appear as a paradox from a theoretical point of view.

More specifically, one might think that the random variables $\ell\left(\widehat{f}_{\mathcal{D}}(x_1),y_1\right),\ldots,\ell\left(\widehat{f}_{\mathcal{D}}(x_n),y_n\right)$ are independent, since the training data points $(x_1,y_1),\ldots,(x_n,y_n)$ are chosen independently from $\mathbb{P}_{X,Y}$. Hence, for a sufficiently large sample size n, the training loss

$$L\left(\widehat{f}_{\mathcal{D}};\mathcal{D}\right) = \frac{1}{n}\sum_{i=1}^{n}\ell\left(\widehat{f}_{\mathcal{D}}(x_i),y_i\right)$$

must be close to the generalization error $\mathbb{E}_{x,y}\ell\left(\widehat{f}_{\mathcal{D}}(x),y\right) = L\left(\widehat{f}_{\mathcal{D}};\mathbb{P}_{X,Y}\right)$ due to the law of large numbers. This conclusion however would contradict our previous claim that the training loss is an overly optimistic estimator of the generalization loss.

The mistake of the previous argument is that the random variables

$$\ell\left(\widehat{f}_{\mathcal{D}}(x_1),y_1\right),\ldots,\ell\left(\widehat{f}_{\mathcal{D}}(x_n),y_n\right)$$

were assumed to be independent, which is not correct. The reason is the dependence of these random variables on the model $\widehat{f}_{\mathcal{D}}$ which was chosen as the minimizer of the training loss $L(f;\mathcal{D})$. Hence, $\widehat{f}_{\mathcal{D}}$ inherently depends on the training dataset $\mathcal{D} = \{(\boldsymbol{x}_1,y_1),\ldots,(\boldsymbol{x}_n,y_n)\}$, which consists of the independently from $\mathbb{P}_{X,Y}$ sampled data points by our assumption above. That's why we have to consider $\widehat{f}_{\mathcal{D}}$ as a random instead of fixed function with the randomness coming from $\mathcal{D}$. Because the random variables above all additionally depend on $\widehat{f}_{\mathcal{D}}$, which itself depends on the random dataset

$\mathcal{D}$, these random variables are not independent. The lecture "Guarantees for Machine Learning" analyzes the difference between the generalization error and training loss systematically.

We now know all error metrics of a machine learning pipeline, which we summarize in Figure 6.7.

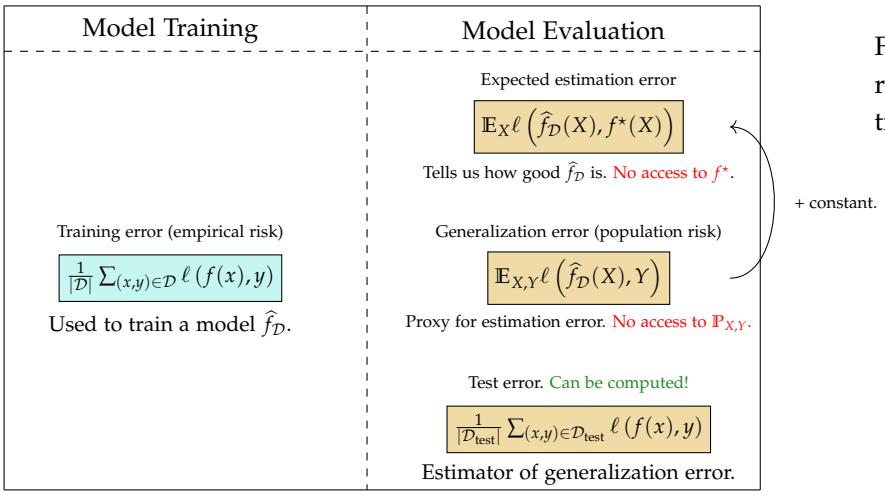


Figure 6.7: The types of error that are used for model training and evaluation.

One natural question that arises now is: how should we split $\mathcal{D}_{\text{full}}$ into $\mathcal{D}$ and $\mathcal{D}_{\text{test}}$? More specifically, what percentage of the data should each of these sets contain? Two possible splits are shown in Figure 6.8.

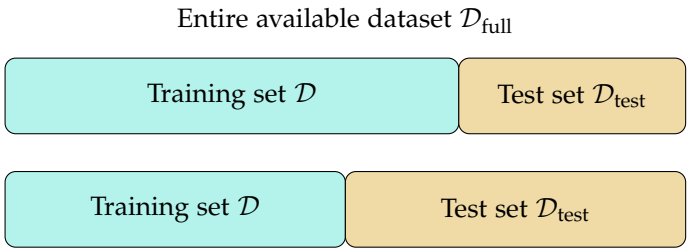


Figure 6.8: Two possible ways to split the entire dataset into training and test parts.

Each of these choices has its pros and cons. The first one is better for model training. Since we are using a larger training set $\mathcal{D}$, it is likely that this set is more representative of the true distribution of (X, Y), so optimizing the weights is more likely to result in a model $\widehat{f}_{\mathcal{D}}$ that is closer to the ground truth $f^\star$. At the same time $|\mathcal{D}_{\text{test}}|$ is smaller, so the test error might not be a very accurate estimator of the generalization error.

In contrast, the second choice with a larger test set might not lead to a model that performs well, but the estimation of its generalization error by the test error will be more accurate.

6.3 Model selection using cross-validation

6.3.1 Naive approach

So far, we have seen how we can evaluate a single model by estimating its generalization error. On the other side, if are given multiple methods (combinations of function class F, loss function ℓ, and optimization algorithm) which one should we choose for a specific prediction task?

Following the discussion in the previous section, a naive approach would be to just use the pipeline presented in Figure 6.9.

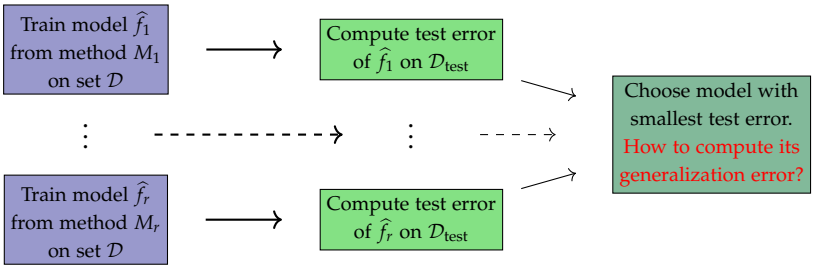


Figure 6.9: Model selection pipeline.

One remaining question is: after choosing the best model, how do we estimate the generalization error of *the chosen model*? At the start, the answer might seem obvious: with the test error of that model. Quite surprisingly, this would not be a valid choice.

Indeed, let $\widehat{f}$ be the chosen model. This model was trained on $\mathcal{D}$ and it was chosen as a result of its small average error on the set $\mathcal{D}_{\text{test}}$ (test error). Hence, this model depends both on $\mathcal{D}$ and $\mathcal{D}_{\text{test}}$. Consequently, the test error

$$\frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(x,y)\in\mathcal{D}_{\text{test}}} \ell\left(\widehat{f}(x), y\right)$$

is not a sum of independent quantities, since $\widehat{f}$ depends on all $(x, y) \in \mathcal{D}_{\text{test}}$. As a result, the assumptions of Theorem 3.51 (Law of large numbers) are not satisfied, so the test error does not necessarily converge to the generalization error $\mathbb{E}_{X,Y}\ell\left(f(X), Y\right)$. In other words, no matter how large $|\mathcal{D}_{\text{test}}|$ is, the test error might not get close to the generalization error of $\widehat{f}$.

6.3.2 Model selection with a validation set

What solution can be found to this standoff? One possible trick would be to split the entire dataset into three parts instead of two, as shown in Figure 6.10.

Just like before, we would train the models on $\mathcal{D}$, but afterward, we would compute their average errors on the validation test $\mathcal{D}_{\text{valid}}$ instead of the test set $\mathcal{D}_{\text{test}}$. After choosing the model with the lowest average error on the validation set, we would estimate its

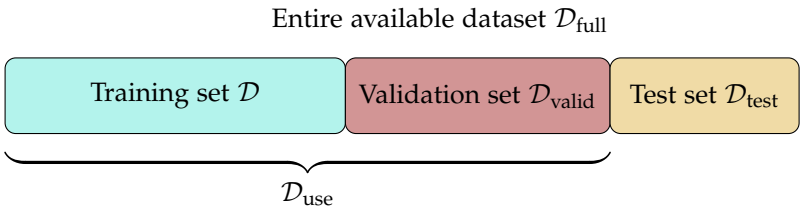


Figure 6.10: The training set is used to train the model. The validation set is used for model selection. The test set is used to estimate the generalization error of the chosen model.

generalization error using its average error on $\mathcal{D}_{\text{test}}$. Since $\mathcal{D}, \mathcal{D}_{\text{valid}}$, and $\mathcal{D}_{\text{test}}$ are independent, this new estimate of the generalization error would be valid.

Regarding the size of each subset, common splits are 80% - 10% - 10% and 50% - 25% - 25%. Just like before, each of them has its pros and cons when it comes to model training and model evaluation, and it is up to the practitioner to choose the one that is the most appropriate for each particular case[7].

[7] For example, if $|\mathcal{D}_{\text{full}}|$ is very large, then the second split might make more sense since there would be enough data points for training, selection, and evaluation. However, the size of the dataset is not the only factor that determines the split. The nature of the data, and whether we want to emphasize model training, model selection, or model evaluation also play a role.

Although splitting the entire dataset into three parts and using one part for training, one for model selection, and one for model evaluation is a valid technique for the selection of the best model and the estimation of its generalization error, it has an obvious downside: it forces us to exclude a large amount of data from the training phase. As a result, the trained models might not be as efficient as we would like.

6.3.3 *K-fold cross-validation*

After setting aside the test set $\mathcal{D}_{\text{test}}$ to obtain an unbiased estimate of the generalization error, would it be possible to use the entirety of the remaining data $\mathcal{D}_{\text{use}}$ for training **and** selecting the best model? It turns out that there exists a procedure to achieve this, known as *K-fold cross-validation*, or simply *cross-validation* if the number of folds is obvious from the context. Assume that we have a method M, i.e. a choice of a function class, a loss function, and an optimization algorithm. The main idea is to partition the set

$$\mathcal{D}_{\text{use}} := \mathcal{D}_{\text{full}} \backslash \mathcal{D}_{\text{test}} \tag{6.4}$$

in K disjoint subsets (folds), as shown in Figure 6.11.

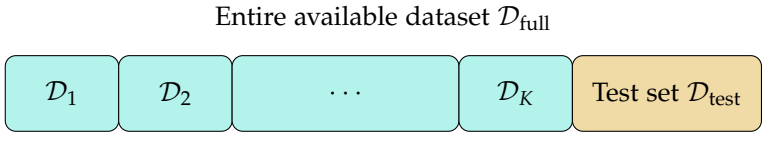


Figure 6.11: In cross-validation, we split the dataset $\mathcal{D}_{\text{use}}$ in K different folds.

In the first step (Split 1) we use $\mathcal{D}_2 \cup \ldots \cup \mathcal{D}_K$ as the training dataset and we train a model $\widehat{f}_{M,1}$. We estimate the generalization error of this model using its average error

$$L_1^{(M)} := L\left(\widehat{f}_{M,1}; \mathcal{D}_1\right) = \frac{1}{|\mathcal{D}_1|} \sum_{(x,y)\in\mathcal{D}_1} \ell\left(\widehat{f}_{M,1}(x), y\right)$$

on $\mathcal{D}_1$, which is now used as a validation dataset.

Analogously, in the second step (Split 2) we repeat this process using $\mathcal{D}_1 \cup \mathcal{D}_3 \cup \ldots \cup \mathcal{D}_k$ as a training dataset and $\mathcal{D}_2$ as a validation set for the model $\widehat{f}_{M,2}$ that is produced. We iterate K times until all K folds have been used as validation sets exactly once.

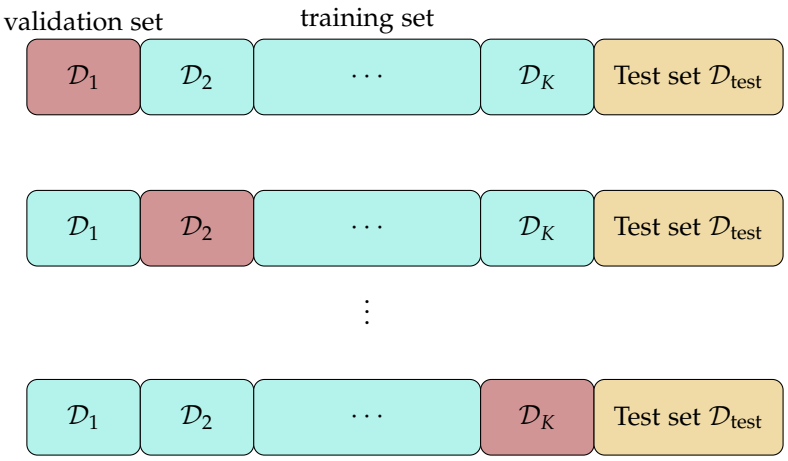


Figure 6.12: The cross-validation algorithm. The brown fold denotes the validation set of each split.

As for model selection, the pivot quantity is the average validation score

$$\text{CV}_K(M) := \frac{1}{K}\sum_{k=1}^{K} L_k^{(M)} = \frac{1}{K}\sum_{k=1}^{K} L\left(\widehat{f}_{M,k}; \mathcal{D}_k\right),$$

known as the *cross-validation error* of the method M. More specifically, if we have several methods $M_1, \ldots, M_r$ to compare, we execute the above iterative procedure for each of them, and we compute their cross-validation errors. In the end, we select the method with the lowest cross-validation error.

To make sure that we make the most of the selected method M^* and that we do not waste any of the training data points like before, we finally train it on the whole dataset $\mathcal{D}_{\text{use}}$ and obtain the *full model* $\widehat{f} := \widehat{f}_{M^*,\mathcal{D}_{\text{use}}}$. To estimate the generalization error of this model, we compute its test error on $\mathcal{D}_{\text{test}}$:

$$L\left(\widehat{f}; \mathcal{D}_{\text{test}}\right) = \frac{1}{|\mathcal{D}_{\text{test}}|}\sum_{(x,y)\in\mathcal{D}_{\text{test}}} \ell\left(\widehat{f}(x), y\right).$$

The whole algorithm can be written in a compact form as follows:

The main advantage of cross-validation is that it allows us to take advantage of the whole set $\mathcal{D}_{\text{use}}$ to train our model, instead of splitting it into two parts, $\mathcal{D}$ and $\mathcal{D}_{\text{valid}}$. As a result, the final model has *higher probability* of being more accurate[8].

Another advantage of cross-validation is that it is valid for any design choices for method M, i.e. any function class, loss function, and optimization algorithm. On the other hand, one of its downsides is that it is computationally intense. Indeed, the training of a single model can often be a very time-consuming task, and even

[8] We emphasize that, while as a rule of thumb training on a larger dataset gives better models, there are usually no theoretical guarantees proving that. This is why we used the phrase *higher probability*. The design of training methods that are *monotone*, i.e. they constantly improve as sample size increases, is an active area of research. See Bousquet et al. 2022 for more information.

Algorithm 4: Cross-Validation.

$i, k \leftarrow 1$
repeat
  repeat
    Train model $\widehat{f}_{M_i,k}$ on $\mathcal{D}_{\text{use}} \backslash \mathcal{D}_k$.
    Compute validation error $L_k^{(M_i)}$.
    $k \leftarrow k + 1$
  until $k = K$
  Compute and store $\text{CV}_K(M_i)$.
  $i \leftarrow i + 1$
until $i = r$
Find $i^* = \arg\min_{1 \leq i \leq r} \text{CV}_K(M_i)$.
Model selection: pick method $M^* = M_{i^*}$.
Model training: train model $\widehat{f} = \widehat{f}_{M^*,\mathcal{D}_{\text{use}}}$ on $\mathcal{D}_{\text{use}}$.
Model evaluation: estimate the generalization error using $\mathcal{D}_{\text{test}}$.

one run of cross-validation already involves the training of multiple models.

6.3.4 Choice of K

One interesting aspect of cross-validation is the hyperparameter K denoting the number of folds. This parameter is not pre-specified, so its choice is up to the machine learning practitioner. What is however the effect of this parameter on the performance of the algorithm?

The efficiency of cross-validation depends on two factors:

i. How close the sub-models $\widehat{f}_{M_k}$, $k = 1, \ldots, K$ are to the *full model* $\widehat{f}_{M_k,\mathcal{D}_{\text{use}}}$: indeed, the algorithm is used to select the best among

$$\widehat{f}_{M_1,\mathcal{D}_{\text{use}}}, \ldots, \widehat{f}_{M_r,\mathcal{D}_{\text{use}}},$$

but the selection stage only involves the sub-models $\widehat{f}_{M_i,k}$ that are trained on parts of $\mathcal{D}_{\text{use}}$. Since these models are used as proxies for $\widehat{f}_{M_i,\mathcal{D}_{\text{use}}}$, we would like them to be as similar as possible to them.

ii. How close the cross-validation score $\text{CV}_K(M_i)$ is to the actual generalization error of the model $\widehat{f}_{M_i,\mathcal{D}_{\text{use}}}$: the algorithm chooses the model with the smallest cross-validation error, but the actual measure of performance of a model is the generalization error. Hence, we would like the cross-validation error to provide an accurate estimation of the generalization error.

The choice of K works as a tradeoff between these two factors. The extreme scenario is when each fold contains exactly one element. In this case, K takes its largest possible value, namely $K = |\mathcal{D}_{\text{use}}|$. This is known as *Leave-One-Out Cross-Validation*, or *LOOCV*, because the training set of each sub-model is all $\mathcal{D}_{\text{use}}$ except from one element.

In this setting, each sub-model $\widehat{f}_{M,k}$ is indeed very close to the full model $\widehat{f}_{M,\mathcal{D}_{\text{use}}}$, because their training sets differ by only one element. On the other hand, though, the validation error $L_k^{(M)}$ is the average error over only one (!) element. Hence, the cross-validation score might not be a very accurate estimator of the generalization error[9].

To summarize, increasing K brings each sub-model closer to the full model, but at the same time decreases the precision of $L_k^{(M)}$ as an estimator of the generalization error $L\left(\widehat{f}_{M,k}; \mathbb{P}_{X,Y}\right)$. Figure 6.13 shows an example of this behavior for one fixed method (whereby we omit the subscript M).

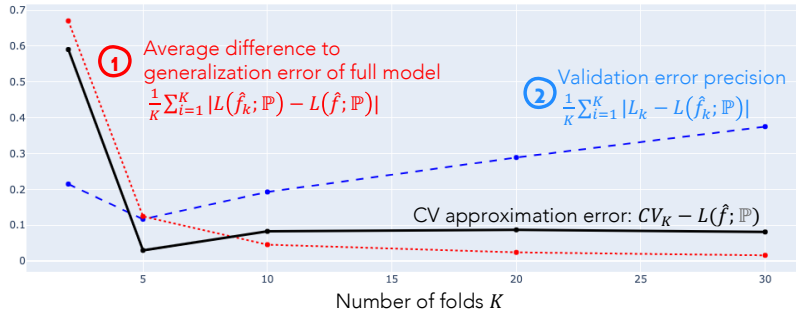


Figure 6.13: Increasing the number of folds decreases the distance between $\widehat{f}_k := \widehat{f}_{M,k}$ and $\widehat{f} := \widehat{f}_{M,\mathcal{D}_{\text{use}}}$ (red curve), but increases the precision of the validation error (blue curve). The cross-validation error is minimized for $K = 5$, but it does not increase significantly when K increases more.

Another problem with LOOCV is that it is very intensive computationally since it trains $K = |\mathcal{D}_{\text{use}}|$ models per method M. Hence, we often tend to choose a relatively smaller value of K to achieve better computational performance. The most popular values are $K = 5$ and $K = 10$.

[9] However, it is worth noting that, although each summand of the cross-validation error is not a good estimate of the generalization error, the cross-validation itself might still be a good estimate of this error. The reason is that, in LOOCV, the K different sub-models are very similar to each other because the training datasets of each pair of them differ by only one point. They are also very similar to the full model. Hence the cross-validation score is the average validation error of K sub-models that are almost identical to the full model. Thus, it is still likely that the cross-validation score approximates the generalization error of the full model.

7
Bias-Variance Tradeoff and Regularization

Roadmap

In this chapter we think about the effect of model complexity on the performance of a machine learning algorithm, and how this effect can be formalized. Specifically, in Section 7.1 we look at the effect of the model complexity on training and generalization error, and in Section 7.2 we explain this effect mathematically via the bias-variance tradeoff. Finally, in Section 7.3, we present some ways to control the model complexity with the use of regularization.

Learning Objectives

By the end of this chapter, you should

- understand intuitively what the term model complexity refers to,
- understand the effect of model complexity on training error,
- understand the effect of model complexity on generalization error,
- know the definitions of bias and variance of a model,
- be able to explain the bias-variance tradeoff,
- be able to state the bias-variance decomposition,
- know how ridge regression and LASSO help control model complexity,
- know what the differences between ridge and LASSO are.

7.1 Effect of Model Complexity

In the previous chapters, we have mentioned multiple times that one of the most important ingredients of a machine learning method is the so-called function class F used by the algorithm. When trying to find a function that approximates some ground

truth $f^\star$, all possible outputs of the method will be functions from F, but F does not necessarily contain $f^\star$. For example, one of the function classes that we studied extensively in Chapter 4 is the class of linear functions in $\mathbb{R}^d$:

$$F_{\text{linear}} = \left\{ f : \mathbb{R}^d \to \mathbb{R} \mid f(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle \,,\ \boldsymbol{w} \in \mathbb{R}^d \right\}.$$

But more complex functions such as polynomials can also be used as we saw in Example 4.4.

The question that we address in this section is:

Given a number of different possible function classes to search in, should we choose a simple *function class F or a more* complex *one?*[1] *How does the choice of F impact the training and generalization errors?*

We will not define strictly what a *simple* and a *complex* function class is, but we are going to use these terms in a relative sense. For example, the class of constant functions $F = \{f : \mathbb{R} \to \mathbb{R} \mid f(x) = c, c \in \mathbb{R}\}$ is simpler than the class of linear functions, which in turn is simpler than the class of polynomials of degree less or equal to four.[2]

7.1.1 Effect of Model Complexity on the Training Error

Let us first focus on the training error. Recall that we have a training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \mid i = 1, \dots, n\}$ that consists of n i.i.d. draws from a distribution $\mathbb{P}_{X,Y}$. The data points satisfy the relation from Equation 6.1

$$y_i = f^\star(\boldsymbol{x}_i) + \varepsilon_i,\ i = 1, \dots, n,$$

with the random noise ε_i being independent of x_i and having mean zero and variance σ^2.

Suppose that we are trying to fit this set of points using the function class of constant functions. In other words, we look for the best constant that describes the data-generating process, as shown in Figure 7.1.

A constant function does not have enough flexibility to fit the training data points. As a result, the training error

$$L(\widehat{f}; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \ell(\widehat{f}(x), y)$$

is large. For that reason, we decide to increase the model complexity by using richer function classes, as shown in Figure 7.2.

Although the function in Figure 7.2 seems to be quite adequate for the description of the data-generating process, we could still try to see what happens if we increase the model complexity even further. This could be done by allowing for a higher degree of polynomials or non-polynomial functions such as exponential, trigonometric, or hyperbolic.

[1] It will soon become clear what *simple* and *complex* mean.

[2] However, it is worth noting that there also exist absolute measures of complexity of a function class, like the VC dimension, the Gaussian complexity, and the Rademacher complexity. These concepts are further analyzed in courses such as *Guarantees for Machine Learning*, *Empirical Process Theory and Applications*, and *Statistical Learning Theory*.

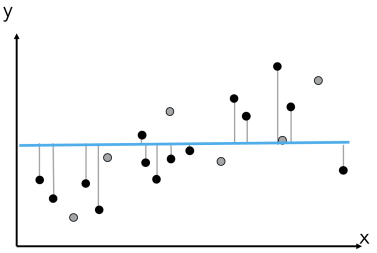


Figure 7.1: We fit the training dataset with a constant function. The gray points are the ones in the test set, so they are not used during training. The training error is very large and the model complexity low.

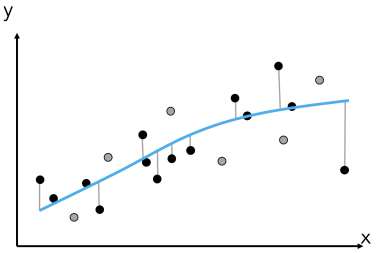


Figure 7.2: The training error decreases if we allow for moderately complex functions.

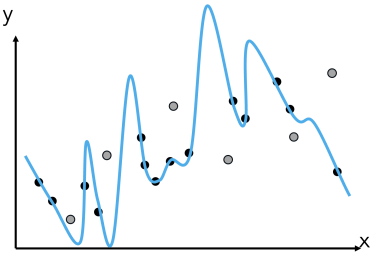


Figure 7.3: The fitted model fits the training data almost perfectly.

As shown in Figure 7.3, a further increase in model complexity would eventually lead to a perfect fit of the training data and a zero training loss. This situation is commonly known as data *interpolation*.

The above examples reveal that increasing the model's complexity (almost) always improves the fit to the training data points and reduces the training error of the resulting model. This might suggest that increasing the model complexity as much as possible is always a good tactic.

However, recall from the previous sections that, when designing a model, we focus on minimizing the *generalization* error, not the training error. Indeed, the whole purpose of machine learning is to build models that *generalize well*, which means performing well on new unseen data points. Hence, decreasing the training error does not necessarily imply that the model became better.

7.1.2 *Effect of Model Complexity on the Generalization Error*

For that reason, it is important to look at the effect of model complexity on the generalization error. Under the model from Equation 6.1, the generalization error is, up to constants, equal to the estimation error, as we have seen in Equation 6.3. Recall that the estimation error of a model $\widehat{f}$ is the area between the graphs of the functions $\widehat{f}$ and $f^\star$.

We are now looking at the behavior of the generalization error as model complexity increases. When using the class of constant functions, the estimation error (and so the generalization error too) is quite large, as shown in Figure 7.4.

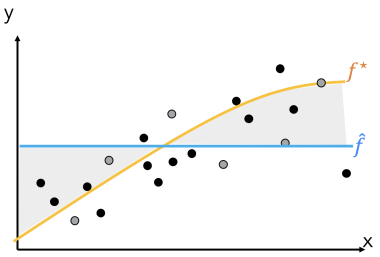


Figure 7.4: The estimation error when allowing only for constant functions is very large.

When we switch to a bit more complex functions (e.g. quadratic), the fit improves, and the generalization error decreases as shown in Figure 7.5. We obtain a very good estimation of the ground-truth function.

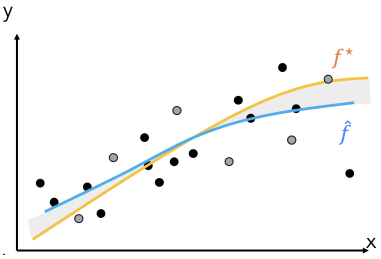


Figure 7.5: Allowing for moderately complex functions can bring us to better estimations of $f^\star$.

So far, the behavior of the generalization error matches the behavior of the training error that was observed earlier - they both decrease as model complexity increases. What happens when we increase model complexity even more though?

As shown in Figure 7.6, the effect of such an increase is exactly the opposite of what was observed for the training error. Although a more complex function has enough flexibility to adapt to the training data, it does not get closer to the ground truth but rather starts deviating more from it. In parallel, the generalization error starts soaring.

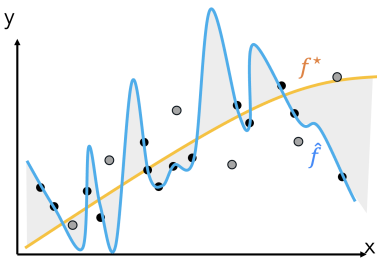


Figure 7.6: Increasing model complexity even more has a negative effect: The generalization error starts to increase.

This phenomenon might initially look strange. One would expect that allowing for more complex functions would give more flexibility to our method, so we would be capable of constructing models that are always closer to the ground truth. Why is this not the case?

The answer lies in the statistical model that lies behind the training data points. If we look at Equation 6.1, we realize that these data points are not faithful reflections of $f^\star$. The existence of the noise ε distorts the data, so what we observe is not the result of the act of $f^\star$, but rather a distorted version of it.

When we give more flexibility to the method by allowing for more complex function classes F, it eventually reaches a point where it cannot extract more information about $f^\star$ from the training data. Hence, beyond that point, the model can only use its extra available capacity to learn information about the noise. Since it cannot distinguish between the signal from $f^\star$ and the noise, it starts getting distracted by the noise and moving away from $f^\star$. This phenomenon is known as *overfitting*.[3]

To conclude, increasing model complexity as much as possible is usually not a good idea. On the other hand, using a very simple model is also not ideal because very elementary function classes (e.g. constant functions) are utterly unable to extract a lot of information for $f^\star$ from the training data; This opposite phenomenon is known as *underfitting*. Hence, a crucial task of a machine-learning scientist is determining the right model complexity so that they can learn $f^\star$ as best as possible, without overfitting. This tradeoff is illustrated in Figure 7.7.

[3] It is worth mentioning that, lately, a lot of machine-learning scientists have observed that even though some models overfit, they still generalize well to unseen data. This phenomenon is called *benign overfitting*. It is not yet well understood mathematically why and when it happens and is a very active research area. See Bartlett et al. 2020 and Tsigler and Bartlett 2020 for more information.

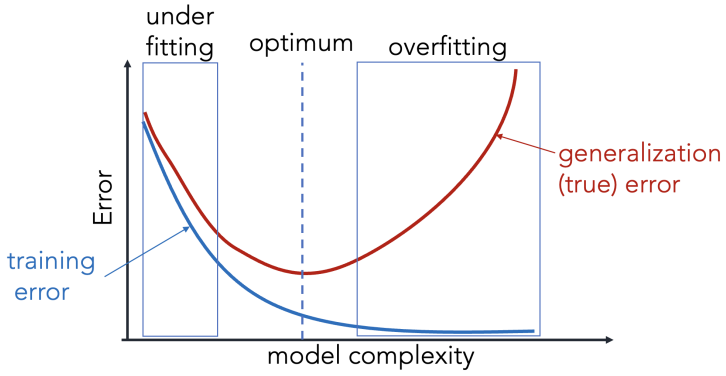


Figure 7.7: The model complexity should neither be too small (underfitting) nor too large (overfitting).

One plausible question that comes out of this discussion is the following: Having only the training set at their disposal, how can the practitioner be sure that they are choosing the right model complexity? There are many protective measures that we can take to avoid overfitting. These measures are often known as *regularization* techniques, and they will be studied in the next sections.

7.2 Bias-Variance Tradeoff

The biggest surprise of the previous section was perhaps the behavior of the generalization error as model complexity varies. As we saw, the generalization error follows a U-shaped curve, so it is large for very simple, as well as for very complex models. This is in contrast to the behavior of the training error, which strictly decreases as

we increase model complexity.

We intuitively attributed this phenomenon to the fact that, after extracting as much information as possible about $f^\star$ from the training data, the model starts fitting the noise. We now discuss how this phenomenon can be explained mathematically in terms of two key statistical quantities: the bias and the variance of a model.

7.2.1 Bias and Variance of a Model

Suppose that we train a method M in several datasets $\mathcal{D}_1, \dots, \mathcal{D}_J$, and obtain the models $\widehat{f}_1 = M(\mathcal{D}_1), \dots, \widehat{f}_J = M(\mathcal{D}_J)$.[4] Although we might not hope that each individual model will be close to the ground truth $f^\star$, it would still be interesting to see how the *average model*

$$\overline{f} = \frac{1}{J} \sum_{j=1}^{J} \widehat{f}_j$$

behaves.

[4] We have seen such an example in cross-validation, where we train the same method in different parts of the training dataset. In that case, the datasets were not independent since their intersections were non-empty.

In the following, we will assume that $\mathcal{D}_1, \dots, \mathcal{D}_J$ are independent. This implies that for all $\boldsymbol{x}$, $\overline{f}(\boldsymbol{x})$ converges almost surely to $\mathbb{E}_{\mathcal{D}} \widehat{f}_{\mathcal{D}}(\boldsymbol{x})$ when $J \to \infty$ due to Theorem 3.51 (Law of large numbers). Therefore, we can think of $\overline{f}$ as a (pointwise) proxy for $\mathbb{E}_{\mathcal{D}} \widehat{f}_{\mathcal{D}}$. As denoted by the subscript, the randomness is taken over the dataset $\mathcal{D} = \{(x_i, y_i) \mid i = 1, \dots, n\}$, which consists of i.i.d. draws from the distribution $\mathbb{P}_{X,Y}$. The squared difference between $f^\star$ and the average model gives us the so-called *squared model bias*, which can also be seen as the squared bias caused by the training method M.

Definition 7.1 (Model bias)**.** Let M be a machine learning method and $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \mid i = 1, \dots, n\}$ a random training dataset. If $\widehat{f}_{\mathcal{D}} = M(\mathcal{D})$ denotes the output of training M on $\mathcal{D}$, then the *squared bias at* $\boldsymbol{x} \in \mathcal{X}$ of the random model $\widehat{f}_{\mathcal{D}}$ is defined as[5]

$$\text{Bias}^2_{\mathcal{D}} \left(\widehat{f}_{\mathcal{D}}, \boldsymbol{x} \right) := \left(\mathbb{E}_{\mathcal{D}} \left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x}) \right] - f^\star(\boldsymbol{x}) \right)^2, \tag{7.1}$$

[5] The subscript $\mathcal{D}$ in $\text{Bias}^2_{\mathcal{D}}(\cdot)$ indicates that we refer to the bias over the "random variable" $\mathcal{D}$.

where $f^\star$ is the ground truth function, related to the data points as in Equation 6.1. The *squared model bias* is then defined as

$$\text{Bias}^2_{\mathcal{D}} \left(\widehat{f}_{\mathcal{D}} \right) := \mathbb{E}_X \left[\text{Bias}^2_{\mathcal{D}} \left(\widehat{f}_{\mathcal{D}}, X \right) \right] \tag{7.2}$$

by additionally taking the expectation over a random $X \in \mathcal{X}$.

Remark. Remember that when we talk about model bias, we take the expectation over the randomness in two quantities, the dataset $\mathcal{D}$ and the test point $X \in \mathcal{X}$. By first considering the randomness in $\mathcal{D}$, we obtain a random model or random function $\widehat{f}_{\mathcal{D}} = M(D)$ through the method M. Intuitively, one can think of a random function as an infinite dimensional random vector or as infinitely many random variables $\left\{ \widehat{f}_{\mathcal{D}}(\boldsymbol{x}) \right\}_{\boldsymbol{x} \in \mathcal{X}}$. By evaluating this random function at some fixed test point $\boldsymbol{x} \in \mathcal{X}$, we obtain a single random variable $\widehat{f}_{\mathcal{D}}(\boldsymbol{x})$ with the randomness contained only in $\mathcal{D}$. The expectation

$\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]$ then corresponds to the function value at the point $\boldsymbol{x}$ averaged over different realizations of the random function $\widehat{f}_{\mathcal{D}}$. By further considering the test point $\boldsymbol{x}$ as a random variable X, we can compute the expectation $\mathbb{E}_{X,\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(X)\right] = \mathbb{E}_X\left[\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(X)\right]\right]$ which corresponds to the average over some random test point X and over different realizations of the random function $\widehat{f}_{\mathcal{D}}$. If we only take expectation with respect to the randomness coming from $\mathcal{D}$ we can define the *function* $\mathbb{E}_{\mathcal{D}}\widehat{f}_{\mathcal{D}} : \mathcal{X} \to \mathbb{R}$ that maps $\boldsymbol{x} \mapsto \mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]$. Note that Theorem 3.51 only guarantees pointwise convergence of $\overline{f}$ to $\mathbb{E}_{\mathcal{D}}\widehat{f}_{\mathcal{D}}$.

At least intuitively, it sounds reasonable that in $\overline{f}$ the weaknesses of the individual models would be averaged out, and so the average model might achieve higher performance. But how close can this model get to the ground truth? That is, how small can the bias get using our function class?

If the function class is too simple (e.g. constant functions), then even the average model cannot approximate the ground truth sufficiently as shown in Figure 7.8. In mathematical terms, this means that a very simple model has a large bias. This explains the observation that very simple models have a large generalization error.

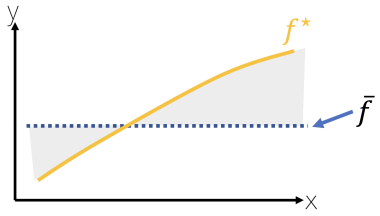


Figure 7.8: When the complexity is too low, we cannot get close to the ground truth no matter how many models we average over.

On the other hand, very complex functions might individually be very wiggly, but they can model the ground truth very well on average, as shown in Figure 7.9. In mathematical terms, this means that a very complex model has a very low bias.

This raises a fundamental question:

> *Since the increase in the generalization error is not owed to a large bias, where does it come from?*

The solution is to look at the *model variance*. Intuitively, this variance is defined as the average distance of each individual model $\widehat{f}_{\mathcal{D}}$ to the average model $\mathbb{E}_{\mathcal{D}}\widehat{f}_{\mathcal{D}}$, which can also be seen as the variance caused by the randomness in $\mathcal{D}$.

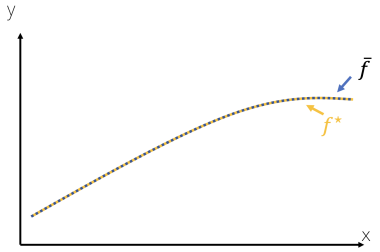


Figure 7.9: Although individual models are not good approximations of $f^\star$, the average model approximates it very accurately. As the number J of models grows to infinity, the average model is able to determine $f^\star$ almost precisely.

Definition 7.2 (Model variance)**.** Let M be a machine learning method and $\mathcal{D} = \{(x_i, y_i) \mid i = 1, \ldots, n\}$ a random training dataset. If $\widehat{f}_{\mathcal{D}} = M(\mathcal{D})$ denotes the output of training M on $\mathcal{D}$, then the *variance at* $\boldsymbol{x} \in \mathcal{X}$ of the random model $\widehat{f}_{\mathcal{D}}$ is defined as

$$\operatorname{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right) := \mathbb{E}_{\mathcal{D}}\left[\left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x}) - \mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]\right)^2\right]. \tag{7.3}$$

The *model variance* is then defined as

$$\operatorname{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) := \mathbb{E}_X\left[\operatorname{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}(X)\right)\right] \tag{7.4}$$

by additionally taking the expectation over a random $X \in \mathcal{X}$.

One estimate for the variance at $\boldsymbol{x} \in \mathcal{X}$ is

$$\frac{1}{J}\sum_{j=1}^{J}\left(\widehat{f}_j(\boldsymbol{x}) - \overline{f}(\boldsymbol{x})\right)^2.$$

Notice that we have simply replaced the expectation over $\mathcal{D}$ with the sample average of the J independent models, and the *true average model* $\mathbb{E}_{\mathcal{D}}\widehat{f}_{\mathcal{D}}$ with the *sample average model* $\overline{f}$.[6]

As we have seen, a very complex model leads to very wiggly functions that diverge a lot from the average function. According to Definition 7.2, this implies that the model has a large variance. This is illustrated in Figure 7.10.

On the contrary, a simple model (e.g. constant functions) has a low variance as shown in Figure 7.11.

Our final conclusion is that simple models have a large generalization error because they have a large bias: No matter how much we try, we can never get close to the ground truth. On the contrary, extremely complex models have a large generalization error because they have a large variance: when we average over many of them we can get close to the ground truth, but each individual model deviates significantly from the average.

Hence, to obtain a low generalization error we need to find the right tradeoff between very simple and very complex models. This is known as the *bias-variance tradeoff*, and it is illustrated in Figure 7.12.

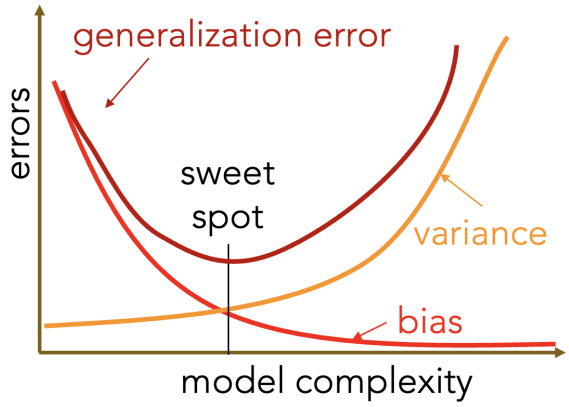


Figure 7.12: Achieving low generalization error boils down to finding the right tradeoff between bias and variance, which is controlled by the complexity of the function class.

7.2.2 Bias-Variance Decomposition

Although we claimed that bias and variance both have an impact on the generalization error, we did not explain from a mathematical viewpoint why this happens. In particular, we can show that the expected generalization error can be decomposed into the model variance, squared model bias, and the irreducible observation noise

[6] This is completely analogous to the estimator $\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$ of the variance $\mathbb{E}\left(X - \mathbb{E}\left[X\right]\right)^2$ of a random variable X.

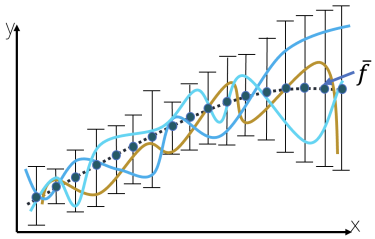


Figure 7.10: The individual functions vary strongly around the average one. In other words, the variance is also large when the model complexity is large.

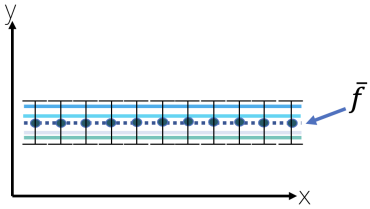


Figure 7.11: When we use a simple function class, all the individual functions are close to the average one. Hence, the variance is low.

variance. Precisely,

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}\left[L\left(\widehat{f}_{\mathcal{D}};\mathbb{P}_{X,Y}\right)\right] &= \mathbb{E}_{X,Y,\mathcal{D}}\left[\left(\widehat{f}_{\mathcal{D}}(X)-Y\right)^2\right] \\ &= \mathbb{E}_{X,\mathcal{D}}\left[\left(\widehat{f}_{\mathcal{D}}(X)-\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(X)\right]\right)^2\right] \\ &\quad + \mathbb{E}_X\left[\left(\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(X)\right]-f^\star(X)\right)^2\right] \\ &\quad + \mathbb{E}_{X,Y}\left[\left(f^\star(X)-Y\right)^2\right] \\ &= \mathrm{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right)+\mathrm{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right)+\sigma^2\end{aligned} \tag{7.5}$$

with $\sigma^2 = \mathrm{Var}(\varepsilon)$. This is known as the *bias-variance decomposition*.

Observe in the given equation how this decomposition boils down to splitting the squared error between the actual model output $\widehat{f}_{\mathcal{D}}(\boldsymbol{x})$ and the observed label y into the error between the actual and average model output, the average and true model output, and the true model output and the observed label.

Bonus Material

In the following, we provide a proof for the bias-variance decomposition. Based on Lemma 6.2, we can write the generalization error under the square loss in terms of the expected estimation error

$$\begin{aligned}L\left(\widehat{f}_{\mathcal{D}};\mathbb{P}_{X,Y}\right) &= \mathbb{E}_{X,Y}\left[\left(Y-\widehat{f}_{\mathcal{D}}(X)\right)^2\right] \\ &= \mathbb{E}_X\left[\left(\widehat{f}_{\mathcal{D}}(X)-f^\star(X)\right)^2\right]+\sigma^2.\end{aligned} \tag{7.6}$$

We can further decompose the estimation error for a fixed sample $\boldsymbol{x}$ into

$$\widehat{f}_{\mathcal{D}}(\boldsymbol{x})-f^\star(\boldsymbol{x}) = \left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x})-\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]\right)+\left(\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]-f^\star(\boldsymbol{x})\right). \tag{7.7}$$

In the first error term, one can already see the notion of model variance represented by the deviation of the actual model $\widehat{f}_{\mathcal{D}}(\boldsymbol{x})$ from the mean model $\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]$, which in fact is independent of the true model $f^\star$. Similarly, the second term already reveals model bias as the deviation between the mean model and true model, which is independent of the actual dataset $\mathcal{D}$.

We continue by applying the binomial formula to the squared estimation error in expectation over $\mathcal{D}$, which results in

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}\left[\left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x})-f^\star(\boldsymbol{x})\right)^2\right] &= \mathbb{E}_{\mathcal{D}}\left[\left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x})-\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]\right)^2\right]+\mathbb{E}_{\mathcal{D}}\left[\left(\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(x)\right]-f^\star(\boldsymbol{x})\right)^2\right] \\ &= \mathbb{E}_{\mathcal{D}}\left[\left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x})-\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]\right)^2\right]+\left(\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]-f^\star(\boldsymbol{x})\right)^2 \\ &= \mathrm{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right)+\mathrm{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}},\boldsymbol{x}\right)\end{aligned} \tag{7.8}$$

where the mixed term is zero because

$$\begin{aligned} &2\mathbb{E}_{\mathcal{D}}\left[\left(\widehat{f}_{\mathcal{D}}(\boldsymbol{x}) - \mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]\right)\left(\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right] - f^{\star}(\boldsymbol{x})\right)\right] \\ &\quad = 2\left(\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right] - \mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right]\right)\left(\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}(\boldsymbol{x})\right] - f^{\star}(\boldsymbol{x})\right) \\ &\quad = 0 \end{aligned}$$

by the independence of the two terms in Equation 7.7. By combining Equation 7.6 and Equation 7.8, we arrive at the bias-variance decomposition

$$\begin{aligned} \mathbb{E}_{\mathcal{D}}\left[L\left(\widehat{f}_{\mathcal{D}}; \mathbb{P}_{X,Y}\right)\right] &= \mathbb{E}_{\mathcal{D},X}\left[\left(\widehat{f}_{\mathcal{D}}(X) - f^{\star}(X)\right)^2\right] + \sigma^2 \\ &= \mathbb{E}_X\left[\mathrm{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}(X)\right)\right] + \mathbb{E}_X\left[\mathrm{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}, X\right)\right] + \sigma^2 \\ &= \mathrm{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) + \mathrm{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) + \sigma^2. \end{aligned}$$

This decomposition provides intuitive insights about the machine learning method M:

- The variance of the model measures the deviation of some actual model $\widehat{f}_{\mathcal{D}}$ to the average model $\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}\right]$. This quantity captures the variability of M and indicates how much the models produced by M for different datasets $\mathcal{D}_j$ tend to vary, or in other words how much they overfit to the dataset.
- The squared bias of the model measures the squared deviation of the average model $\mathbb{E}_{\mathcal{D}}\left[\widehat{f}_{\mathcal{D}}\right]$ to the true model $f^{\star}$. This quantity captures the expressivity of M and indicates how closely a model produced by M fits the true model $f^{\star}$ on average, or in other words how much it underfits the dataset.
- The variance of the irreducible observation noise ε measures the deviation of the true model $f^{\star}$ to the observed labels y. This quantity captures the error that is even incurred by the true model $f^{\star}$ when trying to predict y from a given $\boldsymbol{x}$.

7.3 Ridge and LASSO Regularization

In this section, we are looking at ways to control model complexity and avoid overfitting. As mentioned earlier, this procedure is called *regularization*. We start with the following example.

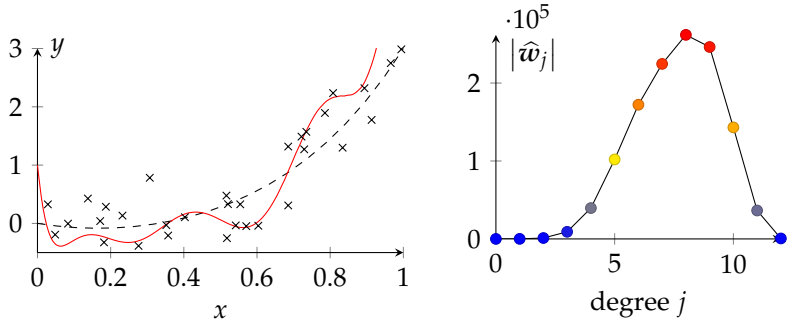
Example 7.3. Assume that the ground truth function $f^{\star} : \mathbb{R} \to \mathbb{R}$ is given by the polynomial $f^{\star}(x) = x^5 + 3x^2 - x$ and we have $n = 35$ noisy samples $(x_1, y_1), \ldots, (x_n, y_n)$ from the model $y = f^{\star}(x) + \varepsilon$, where the noise ε satisfies the usual assumptions.

Without having access to the ground truth, we decide to perform linear regression using the feature map $\phi(x) = (1, x, x^2, \ldots, x^m)$, with $m = 12$. In other words, we are looking for the vector $\boldsymbol{w} \in \mathbb{R}^{13}$ that minimizes the training error

$$\|\boldsymbol{y} - \Phi\boldsymbol{w}\|_2^2 = \sum_{i=1}^{n} \left(y_i - \langle \boldsymbol{w}, \phi(x_i)\rangle\right)^2.$$

The matrix Φ has columns $\Phi_i = (1, x_i, \ldots, x_i^{12})$, $i = 1, \ldots, 12$. If we do not use a feature transformation, then instead of Φ we would have the design matrix $\boldsymbol{X}$ of the plain features $x_1, \ldots, x_n$. For more information about feature maps and non-linear least squares regression, see Section 4.3.

The complexity of the model we are using is rather large, given that the ground-truth function is a polynomial of degree five. This is confirmed when we plot the fitted function $\widehat{f}(x) = \langle \widehat{w}, \phi(x) \rangle$. We observe in Figure 7.13 that the fitted function is indeed a polynomial of degree larger than five and that the largest absolute coefficients are those corresponding to powers of higher degree.



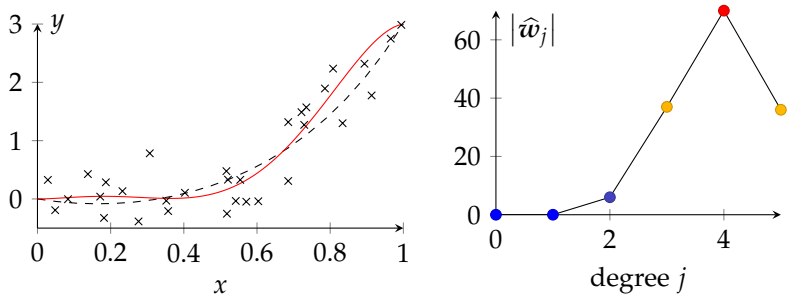
(a) The fitted polynomial.

(b) The coefficients of the fitted polynomial.

Figure 7.13: (a) The fitted polynomial appears to overfit the training data. (b) The plot of its coefficients reveals that the degree of the fitted polynomial is larger than five.

7.3.1 The LASSO

One of the solutions to avoid overfitting in the previous example is to control the degree of the feature map $\phi(x)$ and allow only for up to fifth-degree powers. This seems to be a decently effective solution, as it outputs a regression function that is much closer to the ground truth. As shown in Figure 7.14, that function is a fifth-degree polynomial, exactly like the ground truth.



(a) The fitted polynomial.

(b) The coefficients of the fitted polynomial

Figure 7.14: When we use $\phi(x) = (1, x, \ldots, x^5)$, the model is much closer to the ground truth and there are no signs of overfitting.

However effective this method of controlling ϕ might look, it has a fundamental problem: The scientist has no access to the ground truth, so they have no way of determining which degree to choose, such as $m = 5$ in our example. The choice $m = 5$ works well, but in practice, the scientist does not know that the data points are from a fifth-degree polynomial. Hence, it becomes obvious that we cannot rely on such methods, since they require knowledge or random guessing about the true nature of the data.

One general idea that addresses this issue is the enforcement of constraints on the vector $\widehat{w}$, which will hopefully direct $\widehat{w}$ to a good

solution without the scientist having to make random guesses. In other words, we are looking for a set $\mathcal{R} \subset \mathbb{R}^d$ of suitable restrictions that will force the optimization problem

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 \quad \text{such that} \quad \boldsymbol{w} \in \mathcal{R}$$

to have a solution of degree closer to five, i.e., the degree of the ground truth.[7] Here $\boldsymbol{X}$ would be replaced by Φ if we had used a feature transformation.

[7] In the case of Example 7.3, the dimension d equals 13.

One first attempt could be to control the number of nonzero entries of the vector $\widehat{\boldsymbol{w}}$. In other words, we could try to solve the minimization problem

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 \quad \text{such that} \quad \|\boldsymbol{w}\|_0 \leq k,$$

where $\|\boldsymbol{w}\|_0$ is the number of nonzero elements of $\boldsymbol{w}$. This does not require us to guess the *right* degree of the ground truth function, but it hopefully forces $\boldsymbol{w}$ to have zero high-degree coefficients. However, to do that, we would have to search for the *best subset of cardinality at most k*, a problem that is infeasible for large d due to the large number of such subsets.[8]

[8] There are $\binom{d}{k}$ subsets with cardinality exactly k. For $d \gg k$, this number's order of magnitude is d^k. Further, how to choose k in a smart way is also an open question of this method.

One of the most promising regularization methods is a modification of the above technique, with the ℓ_1-norm[9] replacing $\|\cdot\|_0$. In other words, the method consists of solving the constrained optimization problem

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 \quad \text{such that} \quad \|\boldsymbol{w}\|_1 \leq R, \tag{7.9}$$

where R is a pre-defined positive constant. Just like before, $\boldsymbol{X}$ is replaced by the transformed matrix Φ if we use a feature transformation. A solution $\widehat{\boldsymbol{w}}$ of the optimization problem in Equation 7.9 is called the *LASSO*, short for *Least Absolute Shrinkage and Selection Operator*. An in-depth discussion of the LASSO is provided in the course *High-Dimensional Statistics*.

[9] Recall that the ℓ_1-norm is given by $\|\boldsymbol{w}\|_1 = \sum_{i=1}^d |w_i|$.

In geometric terms, the LASSO searches for the optimal $\boldsymbol{w}$, not over all $\mathbb{R}^d$, but only over the set $\mathcal{B}_1^d(R) := \left\{\boldsymbol{w} \in \mathbb{R}^d : |\boldsymbol{w}|_1 \leq R\right\}$. This set is called the *ℓ_1-ball of radius $R > 0$* and, if $d = 2$, it looks as in Figure 7.15.

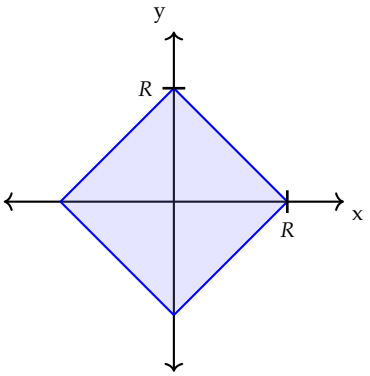


Figure 7.15: The ℓ_1-ball of radius R in $\mathbb{R}^2$. Notice the difference from the ℓ_2-ball which is a disk of radius R.

Due to the convexity of the ℓ_1-ball, the optimization problem from Equation 7.9 is equivalent to the following one:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \left(\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_1\right), \tag{7.10}$$

where $\lambda > 0$ depends on $\boldsymbol{X}, \boldsymbol{y}$ and R. This form is often called the *Lagrangian form* of the optimization problem.[10]

[10] More information on the equivalence of the two problems in Equation 7.9 and Equation 7.10 can be found in the course *Convex Optimization*.

Both forms of the problem require the a priori choice of a hyperparameter: R for the first form and λ for the second. Later we will talk about how these hyperparameters are tuned, but for now, we

can postpone this part of the problem. However, we should point out that the smaller choices for R as well as larger choices for λ force us towards vectors $\boldsymbol{w}$ with smaller ℓ_1-norm.

Exercise 7.4. Prove that if $\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I}_d$, the solution $\widehat{\boldsymbol{w}}_\lambda$ to Equation 7.10 satisfies

$$[\widehat{\boldsymbol{w}}_\lambda]_j = \operatorname{sign}(w_j^{\mathrm{LS}}) \max\left\{0, |w_j^{\mathrm{LS}}| - \lambda\right\}$$

where $\boldsymbol{w}^{\mathrm{LS}} = \boldsymbol{X}^\top \boldsymbol{y}$ is the ordinary least-square solution, i.e., the minimizer for $\lambda = 0$.

Solution. The objective of the optimization problem from Equation 7.10, using $\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I}_d$, can be expanded as

$$\frac{1}{2}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_1 = \frac{1}{2}\boldsymbol{y}^\top \boldsymbol{y} - \boldsymbol{y}^\top \boldsymbol{X}\boldsymbol{w} + \frac{1}{2}\boldsymbol{w}^\top \boldsymbol{w} + \lambda \|\boldsymbol{w}\|_1 ,$$

which is coordinate-wise separable, so that we just need to solve the following optimization problem for all j,

$$\min_{w_j}\left(-w_j^{\mathrm{LS}} w_j + \frac{1}{2}w_j^2 + \lambda |w_j|\right).$$

Importantly, a solution must satisfy that if $w_j^{\mathrm{LS}} > 0$, then $w_j \geq 0$, since otherwise $-w_j$ would attain a smaller value. Similarly, if $w_j^{\mathrm{LS}} < 0$, then $w_j \leq 0$. We can, therefore, make a case distinction.

Case 1: $w_j^{\mathrm{LS}} > 0$. By the first-order stationary condition $\frac{d}{dw_j}(-w_j^{\mathrm{LS}} w_j + \frac{1}{2}w_j^2 + \lambda w_j) = 0$, we obtain that

$$w_j = w_j^{\mathrm{LS}} - \lambda = \operatorname{sign}(w_j^{\mathrm{LS}}) \max\left\{0, |w_j^{\mathrm{LS}}| - \lambda\right\}$$

where the second equality is due to $w_j \geq 0$.

Case 2: $w_j^{\mathrm{LS}} < 0$. Analogous to the previous case, the first-order condition $\frac{d}{dw_j}(-w_j^{\mathrm{LS}} w_j + \frac{1}{2}w_j^2 - \lambda w_j) = 0$ yields

$$w_j = w_j^{\mathrm{LS}} + \lambda = \operatorname{sign}(w_j^{\mathrm{LS}}) \max\left\{0, |w_j^{\mathrm{LS}}| - \lambda\right\}$$

where the second equality is due to $w_j \leq 0$.

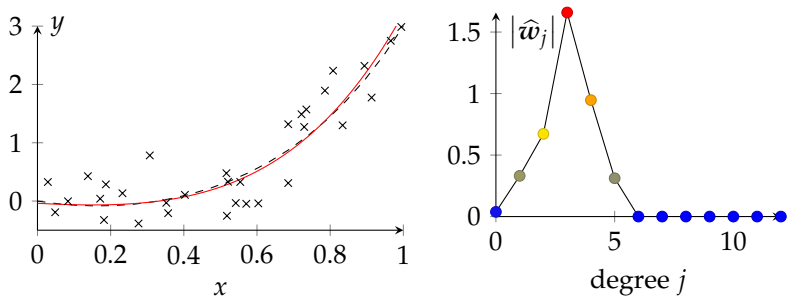
Case 3: $w_j^{\mathrm{LS}} = 0$. If $w_j^{\mathrm{LS}} = 0$, the objective is minimized by $w_j = 0$, so that $w_j = \operatorname{sign}(w_j^{\mathrm{LS}}) \max\left\{0, \left|w_j^{\mathrm{LS}}\right| - \lambda\right\}$ holds. This concludes the proof. □

Let us now go back to the problem discussed in Example 7.3, and attempt to solve it with the initial feature map

$$\phi(x) = (1, x, \ldots, x^{12})$$

and the ℓ_1-norm constraint. We will use the Lagrangian form with $\lambda = 0.005$. The resulting model is shown in Figure 7.16.

Even though we still use the polynomial feature map ϕ with degree $m = 12$, we obtain a much better model than before (Figure 7.13). Hence, it looks like the ℓ_1-norm constraint indeed prevents overfitting. The method seems to automatically discard all terms of a large degree, and it outputs a polynomial of degree five, equal to the degree of the ground truth. This might seem surprising, given that the method does not know that the data actually comes from a fifth-degree polynomial, but demonstrates the effectiveness of the LASSO.



(a) The fitted polynomial.

(b) The coefficients of the fitted polynomial.

Figure 7.16: (a) ℓ_1-penalization leads to a new fitted function that does not show signs of overfitting. (b) This new function is a polynomial of degree five, so ℓ_1-norm penalization forces all terms of higher degree to zero.

7.3.2 *Ridge Regression*

Instead of an ℓ_1-norm constraint, we could also try to impose an ℓ_2-norm constraint as a regularization factor. The optimization problem in this case would be

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \|\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}\|_2^2 \quad \text{such that} \quad \|\boldsymbol{w}\|_2 \leq R, \tag{7.11}$$

and due to the convexity of the ℓ_2-ball, it would be equivalent to its Lagrangian form,

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \left(\|\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2\right). \tag{7.12}$$

This regularization approach is known as *ridge regression*.

It can be noticed that, in the Lagrangian form, the penalty term uses $\|\cdot\|_2^2$ instead of $\|\cdot\|_2$. Although it is a common convention to phrase ridge regression in this way, the alternative problem without the square has also been investigated.

Some advantages of adding a square are that this problem has a closed-form solution

$$\widehat{\boldsymbol{w}} = \left(\boldsymbol{X}^\top\boldsymbol{X} + \lambda \boldsymbol{I}_d\right)^{-1}\boldsymbol{X}^\top\boldsymbol{y}, \tag{7.13}$$

and that the objective function $\|\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2$ is differentiable, so the gradient methods that were studied in Chapter 5 can be used directly. These two characteristics were not true in general for the ℓ_1-norm-constraint problem from Equation 7.10.

Exercise 7.5. Prove that the solution of the ridge regression problem Equation 7.12 is given by Equation 7.13.

Solution. Notice that $\|\boldsymbol{x}\|_2^2 = \boldsymbol{x}^\top\boldsymbol{x}$ for any $\boldsymbol{x}\in\mathbb{R}^d$, so the objective function of that minimization problem can be written as

$$\begin{aligned}\|\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2 &= (\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})^\top(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}) + \lambda\boldsymbol{w}^\top\boldsymbol{w}\\ &= \boldsymbol{y}^\top\boldsymbol{y} + \boldsymbol{w}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}^\top\boldsymbol{X}\boldsymbol{w} - \boldsymbol{w}^\top\boldsymbol{X}^\top\boldsymbol{y} + \lambda\boldsymbol{w}^\top\boldsymbol{w}.\end{aligned}$$

Given that $\boldsymbol{y}^\top\boldsymbol{X}\boldsymbol{w}\in\mathbb{R}$, it is equal to its transpose; $\boldsymbol{y}^\top\boldsymbol{X}\boldsymbol{w} = \boldsymbol{w}^\top\boldsymbol{X}^\top\boldsymbol{y}$. Also, the term $\boldsymbol{y}^\top\boldsymbol{y}$ does not depend on $\boldsymbol{w}$. Hence, the minimization problem boils down to

$$\min_{\boldsymbol{w}\in\mathbb{R}^d}\left(\boldsymbol{w}^\top\left(\boldsymbol{X}^\top\boldsymbol{X}+\lambda\boldsymbol{I}_d\right)\boldsymbol{w} - 2\boldsymbol{y}^\top\boldsymbol{X}\boldsymbol{w}\right).$$

We differentiate and set the gradient equal to zero. Since $\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_d$ is a symmetric matrix, we get

$$\frac{\partial}{\partial \boldsymbol{w}} \boldsymbol{w}^\top \left(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_d \right) \boldsymbol{w} = 2 \left(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_d \right) \boldsymbol{w},$$
$$\frac{\partial}{\partial \boldsymbol{w}} \boldsymbol{y}^\top \boldsymbol{X} \boldsymbol{w} = \boldsymbol{X}^\top \boldsymbol{y}.$$

Hence, we end up with the equation $(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_d)\boldsymbol{w} = \boldsymbol{X}^\top \boldsymbol{y}$. From Theorem 1.38 it follows that the matrix $\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_d$ is strictly positive definite, so it is invertible. Thus, we obtain the desired formula for $\widehat{\boldsymbol{w}}$. Finally, the Hessian matrix of the objective function is $\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_d$, which is strictly positive definite, so the stationary point $\widehat{\boldsymbol{w}}$ that we detected is really a global minimum. □

As we discussed earlier, when the number of features d exceeds the number of data samples n, that is, $d > n$, unregularized methods often overfit because the method wrongly starts fitting the noise instead of the actual signal of the data points. In this regime, most of the features are unnecessary and the data can be explained by only a strict subset of them. When this happens, an increase in the norms $\|\widehat{\boldsymbol{w}}\|_1$, $\|\widehat{\boldsymbol{w}}\|_2$ is also observed when increasing the level of the noise (i.e. the variance σ^2 of the noise variables). Ideally, this should not happen because the ground truth $\boldsymbol{w}^*$ is independent of the noise.

Adding constraints seems to fix this problem. As shown in Figure 7.17, adding a ridge penalty term $\lambda \|\boldsymbol{w}\|_2^2$ to the least-squares objective function drastically controls the norm of $\widehat{\boldsymbol{w}}$ with respect to the level of noise. Furthermore, it illustrates how increasing the value of λ forces $\|\widehat{\boldsymbol{w}}\|_2$ to be smaller.

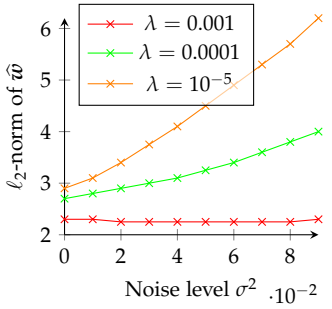


Figure 7.17: Raising the noise level leads to an increase of the ℓ_2-norm of $\widehat{\boldsymbol{w}}$. The larger λ, the less $\|\widehat{\boldsymbol{w}}\|_2$ is affected by increasing the noise. The case $\lambda = 0$ is not in the plot, because then $\|\widehat{\boldsymbol{w}}\|_2$ is *many orders of magnitude* larger.

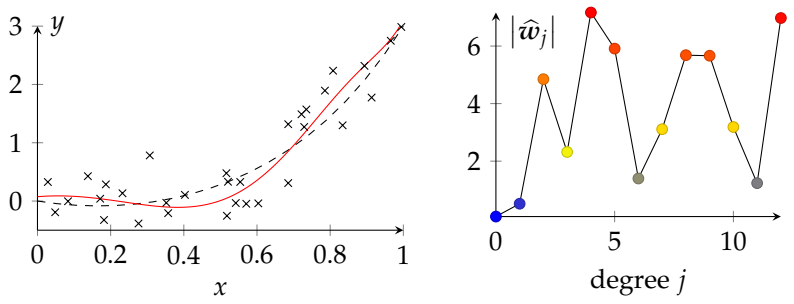
It is worth noting that using the LASSO controls the ℓ_1-norm of $\widehat{\boldsymbol{w}}$ in an analogous way, avoiding a strong increase of $\|\widehat{\boldsymbol{w}}\|_1$ as the noise level increases.

7.3.3 *Comparison of LASSO and Ridge Regularization*

The two kinds of constraints that we introduced, namely ℓ_1 and ℓ_2-penalization have one big difference in how they force $\|\widehat{\boldsymbol{w}}\|_2$ (or $\|\widehat{\boldsymbol{w}}\|_1$) to be small. We already saw in Figure 7.16 that LASSO regression forces a lot of coefficients of $\widehat{\boldsymbol{w}}$ to be zero and allows just a few of them to be nonzero.

The effect of a ridge penalty is significantly different. Ridge regression applied in the setting of Example 7.3 appears to prevent overfitting, and it outputs a model very close to the ground truth. However, as shown in Figure 7.18, the resulting model is actually a polynomial of degree twelve. This comes in contrast to the model produced by LASSO, which was a polynomial of degree five.

Thus, both regularization techniques control the magnitude of $\|\widehat{\boldsymbol{w}}\|_2$ and prevent overfitting, but they do it in different ways. In short, we say that the LASSO induces *sparsity* (many or most of the

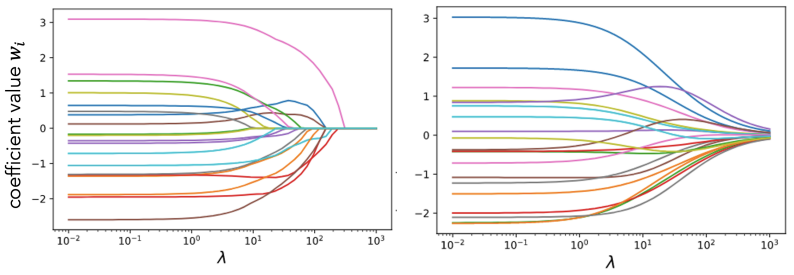


(a) The fitted polynomial.

(b) The coefficients of the fitted polynomial.

Figure 7.18: Ridge regression does not lead to many coefficients being equal to zero. Instead, it controls the ℓ_2-norm of $\widehat{w}$ by shrinking its entries *only close* to zero.

coefficients set to zero), while ridge does not. This is also illustrated in Figure 7.19 which shows the progress of the entries of $\widehat{w}$ as the value of λ varies.



(a) LASSO

(b) Ridge

Figure 7.19: (a) Under LASSO constraints, the coefficients are forced to be equal to zero as λ increases. (b) On the contrary, under ridge constraints, the coefficients shrink but do not get exactly equal to zero.

What is the deeper reason behind the difference in the action of these two kinds of constraints? This proves to be a very interesting question, and the explanation that we are going to give, although not completely rigorous, is motivated both by algebraic and geometrical insights.

Interpretation via norm definitions. Consider the two vectors $w_{\text{dense}}, w_{\text{sparse}} \in \mathbb{R}^d$ with

$$w_{\text{dense}} = \frac{1}{\sqrt{d}}(1, 1, \ldots, 1) \quad \text{and} \quad w_{\text{sparse}} = (1, 0, 0, \ldots, 0).$$

These two vectors have the same ℓ_2-norm, but when it comes to the ℓ_1-norm,

$$\sqrt{d} = \|w_{\text{dense}}\|_1 \gg \|w_{\text{sparse}}\|_1 = 1.$$

Thus, in most settings, it would be more likely for the LASSO to choose w_{sparse} instead of w_{dense}, whereas ridge would not be able to differentiate between the two in terms of the ℓ_2-norm. This is visualized in Figure 7.20. Another way to view it is to consider the largest ℓ_2-ball inscribed in an ℓ_1-ball of radius 1, which has radius $1/\sqrt{d}$ (see Figure 7.21). At the same time, the volume of both remain of the same order $(C/d)^d$ by stirlings formula.

For some fixed ℓ_2-norm, the same argument can be made for all dense vectors, meaning vectors that have only non-zero components, as well as all sparse vectors, meaning vectors that have at

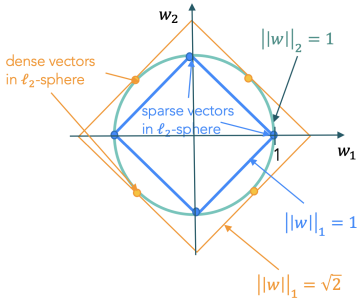


Figure 7.20: There is no difference between dense and sparse vectors in ℓ_2-norm, but the ℓ_1-norm distinguishes between them.

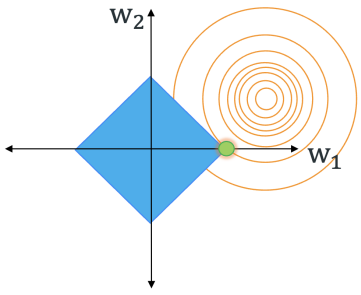
least one component equal exactly zero. Since, for a fixed ℓ_2-norm, there are *many more*[11] of such dense vectors than sparse vectors, in most settings, it is much more likely for ridge regression to pick a dense one. On the other hand, since the ℓ_1-norm of sparse vectors is smaller, LASSO is much more likely to pick a sparse vector.

Geometric interpretation. Let us look at the constrained form of the LASSO optimization problem Equation 7.9. From a geometric point of view, this problem has two components:

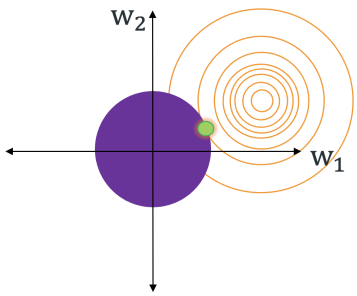
1. We try to find the minimizer $\boldsymbol{w}^{\mathrm{LS}}$ of $\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$.
2. We must remain within the ℓ_1-ball of radius R.

As shown in Figure 7.22(a), these two goals might contradict each other: the minimizer $\boldsymbol{w}^{\mathrm{LS}}$ might not lie within the ℓ_1-ball. In this case, we try to get *as close as possible* to $\boldsymbol{w}^{\mathrm{LS}}$. As we have mentioned, the *level sets* of the function $L(\boldsymbol{w}) := \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$ are defined as $L^{-1}(c) := \{\boldsymbol{w} \in \mathbb{R}^d : L(\boldsymbol{w}) = c\}$ for the different values of $c \in \mathbb{R}$. It can be proved that the level sets of $\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$ are ellipsoids, and the value of c increases as we move away from $\boldsymbol{w}^{\mathrm{LS}}$. Hence, our goal is to find the ellipsoid closest to $\boldsymbol{w}^{\mathrm{LS}}$ that also intersects the ℓ_1-ball of radius R. As shown in Figure 7.22(a), when R is small enough, that point lies on one of the coordinate axes. This means that one of its coordinates is zero and so the corresponding solution is sparse.

Of course, there are more (infinitely many) intersection points of the ℓ_1-ball with the level sets of the objective function, but these lie on ellipsoids that are further away from $\boldsymbol{w}^{\mathrm{LS}}$, so they correspond to larger values of the objective function.



(a) LASSO



(b) Ridge

Figure 7.22: (a) Due to the geometry of the ℓ_1-ball, the contour line that is closest to $\boldsymbol{w}^{\mathrm{LS}}$ meets it precisely at a sparse point. This explains the preference of LASSO for sparse estimators. (b) The ridge estimator tends to be a dense point due to the geometry of the ℓ_2-norm.

On the contrary, the geometry of the ℓ_2-ball does not have these characteristics. It could still meet the contour lines at sparse points, but this would be a coincidence rather than a systematic behavior. This is illustrated in Figure 7.22(b). There, the level sets meet the ℓ_2-ball at a sparse point only when their center $\boldsymbol{w}^{\mathrm{LS}}$ is on one of the axes, which is generally not the case.

Remark. Note that the above geometric argument primarily serves to illustrate the difference between the LASSO and ridge estimator: The geometry of the ℓ_1-ball tends to induce more sparse solutions

[11] "Many more" is a term we did not define, but intuitively it should be clear what we mean. This notion can be defined mathematically though; *measure theory* provides a toolbox to make statements like this rigorous.

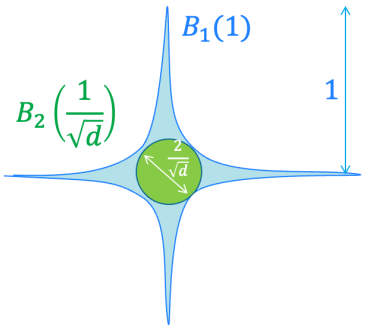


Figure 7.21: The largest ℓ_2-ball inscribed in an ℓ_1-ball of radius 1 has radius $1/\sqrt{d}$. Hence, in high dimensions, it shrinks a lot. This figure depicts the *intuition* in high dimensions—but of course the ℓ_1-ball remains convex.

than the ℓ_2-ball, in particular for large enough λ in the penalized or small enough R in the constrained version.

Let us for example consider the two-dimensional spherical case $\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I}_2$. It's easy to see geometrically that the ridge estimator only induces sparse solutions (for some $R > 0$) when the least square minimizer $\boldsymbol{w}^\star$ lies on one of the axes. Whether or not the LASSO induces a sparse solution for small enough R (or large enough λ) also depends on $\boldsymbol{w}^{\text{LS}}$ and $\boldsymbol{X}$. In the example where $\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I}_2$, whenever $|w_1^{\text{LS}}| = |w_2^{\text{LS}}|$, the solution will not be sparse for any R except $R = 0$ which results in the trivial solution $\widehat{\boldsymbol{w}} = 0$. However, in all other directions, there exists a small enough R or large enough λ such that the LASSO solution is sparse. This stands in contrast to the ridge solution.

Implementation

The methods discussed so far, i.e., choices of a function class, loss function, and optimization algorithm are documented and can be readily implemented in several programming languages. In `Python`, the main library containing many of those methods is `scikit-learn`. Some useful commands can be found in the following table:

Function Class (`fct_class`)	Objective Function (`loss`)	Optimization (`option:solver`)
`linear_model`	`LinearRegression`	`solver='cholesky'`
`kernel_ridge`	`Ridge`	`solver='svd'`
`neural_network`	`Lasso`	`solver='sag'`
`tree`	`KernelRidge`	`solver='lbfgs'`
	`LogisticRegression`	

Some of the methods these functions correspond to have not been discussed yet in these notes, but they will be studied later on. The general definition of a method in `Python` is the following:

```
method = sklearn.<fct_class>.<{loss}>([hyperparams, solver])
```

The option `hyperparams` includes `alpha` (which refers to the regularization parameter λ), `tol` (which is the precision that the optimization algorithm should reach) and `max_iter` (which is the maximum number of iterations that the optimization algorithm should perform). For example,

```
method = sklearn.linear_model.Ridge([alpha=0.1])
```

defines a ridge regression model with regularization parameter $\lambda = 0.1$. The `solver` is not specified here, so `scikit-learn` uses the default one `'lbfgs'`. To train the model on some training data with design matrix `X` and response vector `y`, we use the command

```
method.fit(X,y)
```

If we later want to test the fitted model on `X_test, y_test`, we write

```
y_pred = method.predict(X_test)
test_error = sklear.metrics.mean_squared_error(y_test,y_pred)}
```

that gives us a vector of predictions `y_pred`, and then we can use `sklear.metrics.mean_squared_error` to compute the error on the test dataset.

7.4 Regularization and Bias-Variance Tradeoff

Let us now look at the effect of regularization on bias and variance.[12] We talked in Section 7.2 about the tradeoff between them, and analyzed how they are connected to underfitting and overfitting. As we saw, regularization prevents overfitting, so it must have some effect on the bias and the variance of the model.

[12] See Definition 7.1, Definition 7.2, and related discussion.

Imagine a ridge regression estimator

$$\widehat{\boldsymbol{w}}_\lambda = \underset{w\in\mathbb{R}^d}{\arg\min} \left(\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2 \right), \tag{7.14}$$

or a LASSO estimator

$$\widehat{\boldsymbol{w}}_\lambda = \underset{w\in\mathbb{R}^d}{\arg\min} \left(\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_1 \right),$$

where $\boldsymbol{X}, \boldsymbol{y}$ is a noisy training dataset. If λ increases, what is going to happen to the bias and the variance of the model?

The answer is that, as λ increases, the estimator $\widehat{\boldsymbol{w}}_\lambda$ becomes simpler, either in the sense that it has fewer nonzero components (LASSO) or in the sense that it has a smaller ℓ_2-norm (ridge). From the previous section, we know that simpler models have a larger bias and cannot get close enough to the true value $\boldsymbol{w}^*$, but at the same time they have a smaller variance. Thus, increasing the value of λ leads to an increase in the bias and a decrease in the variance of the resulting model.

To make this statement more formal, we decompose the generalization error of the ridge regression estimator in its squared bias and variance term as in Equation 7.5. We get that, for $\widehat{f}_{\mathcal{D}}(x) = \langle \widehat{\boldsymbol{w}}_\lambda, x \rangle$ with $\widehat{\boldsymbol{w}}_\lambda$ from Equation 7.14, it holds

$$\begin{aligned} \operatorname{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) &= \left\| V \operatorname{diag}\left(\frac{\lambda}{\sigma_i^2 + \lambda}\right) V^\top \boldsymbol{w}^* \right\|_2^2, \\ \operatorname{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) &= \sigma^2 \sum_{i=1}^{d} \frac{\sigma_i^2}{(\sigma_i^2 + \lambda)^2}, \end{aligned} \tag{7.15}$$

with variance σ^2 of the noise ε and singular values σ_i of $\boldsymbol{X}$, obtained from the SVD decomposition $\boldsymbol{X} = U\Sigma V^\top$, cf. Theorem 1.20. Here $\operatorname{diag}\left(\lambda/(\sigma_i^2 + \lambda)\right)$ denotes the $d \times d$-matrix having $\lambda/(\sigma_i^2 + \lambda), i = 1, \ldots, d$ on its diagonal. The proof of Equation 7.15 is provided in Exercise 7.6 at the end of this section.

Regarding the expected squared bias, we can observe based on the term $\frac{\lambda}{\sigma_i^2+\lambda}$ that it is zero for plain linear regression with $\lambda = 0$. Intuitively, the linear regression estimator does not exhibit any bias, since it solely depends on the dataset $\mathcal{D}$. However, for $\lambda \to \infty$ the expected squared bias approaches $\|\boldsymbol{w}^*\|_2^2$, which makes sense as we essentially force the ridge estimator to be the zero vector for any dataset.

Regarding the expected variance, we can observe that it corresponds to $\sigma^2 \sum_{i=1}^{n} \frac{1}{\sigma_i^2}$ for plain linear regression with $\lambda = 0$. This means the variance of the estimator depends on the observation noise variance σ^2 and, in particular, how well-conditioned the given data matrix X is. While small singular values σ_i of X heavily increase the variance, large singular values do not.

This is where regularization comes into play, which adds a constant value to the denominator. By taking $\lambda \to \infty$ the expected variance converges to zero, which makes sense as we essentially force the ridge estimator to be the zero vector.

From this analysis, we can conclude that ridge regression generally increases the bias and reduces the variance. The choice of λ majorly depends on how well-conditioned the given dataset is and must be made carefully to tradeoff bias and variance of the model. Practical selection strategies for λ are investigated next.

Exercise 7.6. Assume that the test distribution $\mathbb{P}_{X,Y}$ satisfies $\mathbb{E}[X] = 0$ and $\mathrm{Cov}(X) = \boldsymbol{I}_d$, as well as the training data matrix $\boldsymbol{X}$ being fixed (deterministic). Show that the bias and variance of the ridge regression estimator is given by Equation 7.15.

Solution. Recall that the optimal function and ridge regression estimator are given by $f^\star(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}^*$ and $\widehat{f}_{\mathcal{D}}(\boldsymbol{x}) = \boldsymbol{x}^\top \widehat{\boldsymbol{w}}_\lambda$. We first show an alternative expression for the ridge regression estimator starting with Equation 7.13:

$$\begin{aligned}
\widehat{\boldsymbol{w}}_\lambda &= \left(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}_d\right)^{-1} \boldsymbol{X}^\top \boldsymbol{y} \\
&\overset{(1)}{=} \left(V \Sigma^\top \Sigma V^\top + \lambda V V^\top\right)^{-1} V \Sigma^\top U^\top \boldsymbol{y} \\
&= V \left(\Sigma^\top \Sigma + \lambda \boldsymbol{I}_d\right)^{-1} \Sigma^\top U^\top \boldsymbol{y} \\
&\overset{(2)}{=} V \operatorname{diag}\left(\frac{\sigma_i}{\sigma_i^2 + \lambda}\right) U^\top (\boldsymbol{X}\boldsymbol{w}^* + \varepsilon) \\
&\overset{(1)}{=} V \operatorname{diag}\left(\frac{\sigma_i}{\sigma_i^2 + \lambda}\right) U^\top (U \Sigma V^\top \boldsymbol{w}^* + \varepsilon) \\
&= V \operatorname{diag}\left(\frac{\sigma_i^2}{\sigma_i^2 + \lambda}\right) V^\top \boldsymbol{w}^* + M_\lambda \varepsilon
\end{aligned}$$

with $M_\lambda := V \operatorname{diag}\left(\frac{\sigma_i}{\sigma_i^2+\lambda}\right) U^\top$. We applied the SVD decomposition $\boldsymbol{X} = U \Sigma V^\top$ with singular values in $\Sigma \in \mathbb{R}^{n \times d}$ and orthogonal matrices U and V in the steps (1) used the relation $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}^* + \varepsilon$ from Equation 6.1 in (2).

We assume that the training input samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ in the dataset $\mathcal{D}$, or equivalently the data matrix $\boldsymbol{X}$, are fixed. Hence, taking the expectation over $\mathcal{D}$ corresponds to the expectation over the observation noise $\varepsilon \sim \mathcal{N}(0, \sigma^2 \boldsymbol{I}_n)$. This gives us

$$\mathbb{E}_{\mathcal{D}}[\widehat{\boldsymbol{w}}_\lambda] = \mathbb{E}_{\varepsilon}[\widehat{\boldsymbol{w}}_\lambda] = V \operatorname{diag}\left(\frac{\sigma_i^2}{\sigma_i^2 + \lambda}\right) V^\top \boldsymbol{w}^*.$$

Plugging this into the definition of the squared bias of $\widehat{f}_{\mathcal{D}}$ yields

$$\begin{aligned}\text{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) &= \mathbb{E}_X\left[\left(X^\top \mathbb{E}_{\mathcal{D}}\left[\widehat{\boldsymbol{w}}_\lambda\right] - X^\top \boldsymbol{w}^*\right)^2\right] \\ &= \mathbb{E}_X\left[\left(X^\top V \operatorname{diag}\left(\frac{\sigma_i^2}{\sigma_i^2+\lambda}\right) V^\top \boldsymbol{w}^* - X^\top \boldsymbol{w}^*\right)^2\right] \\ &= \mathbb{E}_X\left[\left(X^\top V \left(\operatorname{diag}\left(\frac{\sigma_i^2}{\sigma_i^2+\lambda}\right) - I_d\right) V^\top \boldsymbol{w}^*\right)^2\right].\end{aligned}$$

By realizing that $\frac{\sigma_i^2}{\sigma_i^2+\lambda} - 1 = \frac{-\lambda}{\sigma_i^2+\lambda}$, we obtain the following expression

$$\begin{aligned}&\text{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) \\ &= \mathbb{E}_X\left[\left(X^\top V \operatorname{diag}\left(\frac{\lambda}{\sigma_i^2+\lambda}\right) V^\top \boldsymbol{w}^*\right)^2\right] \\ &= \left(V \operatorname{diag}\left(\frac{\lambda}{\sigma_i^2+\lambda}\right) V^\top \boldsymbol{w}^*\right)^\top \mathbb{E}_X\left[XX^\top\right] \left(V \operatorname{diag}\left(\frac{\lambda}{\sigma_i^2+\lambda}\right) V^\top \boldsymbol{w}^*\right).\end{aligned}$$

Since we further assume that the test input sample $X \in \mathbb{R}^d$ is generated from a distribution with mean $\mathbf{0}$ and covariance $\boldsymbol{I}_d$, implying $\mathbb{E}\left[XX^\top\right] = \boldsymbol{I}_d$, we finally have derived the squared bias of $\widehat{f}_{\mathcal{D}}$ as

$$\text{Bias}^2_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) = \left\| V \operatorname{diag}\left(\frac{\lambda}{\sigma_i^2+\lambda}\right) V^\top \boldsymbol{w}^* \right\|_2^2.$$

We proceed similarly with the model variance. We can write

$$\begin{aligned}\text{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) &= \mathbb{E}_{\mathcal{D},X}\left[\left(X^\top\left(\widehat{\boldsymbol{w}}_\lambda - \mathbb{E}_{\mathcal{D}}\left[\widehat{\boldsymbol{w}}_\lambda\right]\right)\right)^2\right] \\ &= \mathbb{E}_{\mathcal{D},X}\left[\left(X^\top M_\lambda \varepsilon\right)^2\right] \\ &= \mathbb{E}_{\mathcal{D}}\left[\varepsilon^\top M_\lambda^\top \mathbb{E}_X\left[XX^\top\right] M_\lambda \varepsilon\right].\end{aligned}$$

Applying $\mathbb{E}_X\left[XX^\top\right] = \boldsymbol{I}_d$ and making use of the cyclic property of the trace (it holds $\operatorname{tr}(ABC) = \operatorname{tr}(CAB)$ for any three matrices A, B, C with matching dimensions) this expression reduces to

$$\begin{aligned}\mathbb{E}_\varepsilon\left[\varepsilon^\top M_\lambda^\top M_\lambda \varepsilon\right] &= \mathbb{E}_\varepsilon\left[\operatorname{tr}\left(\varepsilon^\top M_\lambda^\top M_\lambda \varepsilon\right)\right] \\ &= \mathbb{E}_\varepsilon\left[\operatorname{tr}\left(\varepsilon\varepsilon^\top M_\lambda^\top M_\lambda\right)\right].\end{aligned}$$

Using the linearity of the trace, we see that the variance is equal to

$$\begin{aligned}\text{Var}_{\mathcal{D}}\left(\widehat{f}_{\mathcal{D}}\right) &= \operatorname{tr}\left(\mathbb{E}_\varepsilon\left[\varepsilon\varepsilon^\top\right] M_\lambda^\top M_\lambda\right) \\ &= \sigma^2 \operatorname{tr}\left(M_\lambda^\top M_\lambda\right) \\ &= \sigma^2 \sum_{i=1}^n \frac{\sigma_i^2}{(\sigma_i^2+\lambda)^2}\end{aligned}$$

which concludes the proof. □

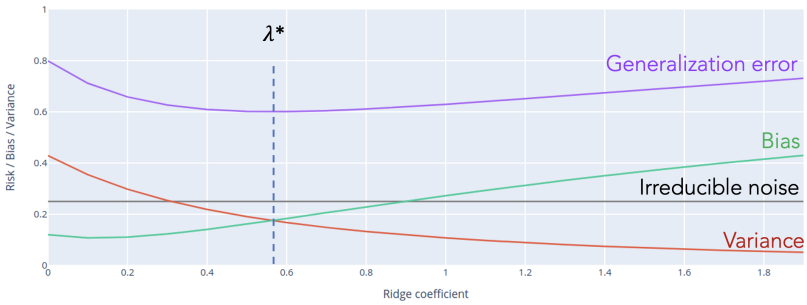


Figure 7.23: As λ increases, the bias also increases, whereas the variance drops. The noise term stays constant.

7.5 *Selection of Regularization Parameter*

As a result of the bias-variance decomposition, the generalization error does not exhibit a monotonic behavior, but it is minimized for a specific value of λ, say λ^*, as shown in Figure 7.23.

Remember that our goal is the minimization of the generalization error, so we would like to find this optimal value of λ. How is it possible to do that?

We can find the optimal value of λ with cross-validation, following the steps of Algorithm 5. We use the notation from Figure 6.12.

To execute this algorithm, we always create a finite grid for λ and search over that grid for the optimal value. It is common practice to compare values of λ on a logarithmic scale, e.g., we might search for the best value in the set $\{10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\}$.

Implementation

In `Python`, we perform this algorithm using the following syntax:

```
    for lambda_value in lambda_grid:

        ridge_reg = sklearn.linear_model.Ridge(alpha=lambda_value)
        scores = cross_val_score(ridge_reg, X, Y, scoring = "neg_mean_squared_error", cv = K)
```

Algorithm 5: cross-validation to tune the hyperparameter λ. Although it is presented here for ridge regression, it works also for LASSO regression after replacing $\|\cdot\|_2^2$ with $\|\cdot\|_1$.
Given a method M, a grid Λ of values λ_i, and the number of folds K:

1: **for** $\lambda \in \Lambda$ **do**
2: $k \leftarrow 1$
3: **repeat**
4: Compute $\widehat{f}_k^\lambda = \langle \widehat{\boldsymbol{w}}_k^\lambda, \cdot \rangle = M_\lambda(\mathcal{D}_{\text{use}} \backslash \mathcal{D}_k)$ with

$$\widehat{\boldsymbol{w}}_k^\lambda = \underset{\boldsymbol{w} \in \mathbb{R}^d}{\arg\min} \left(L(\boldsymbol{w}; \mathcal{D}_{\text{use}} \backslash \mathcal{D}_k) + \lambda \|\boldsymbol{w}\|_2^2 \right)$$

5: Compute validation loss on fold k:

$$L_k(\lambda) = \frac{1}{|\mathcal{D}_k|} \sum_{(\boldsymbol{x},y) \in \mathcal{D}_k} \ell\left(\widehat{f}_k^\lambda(\boldsymbol{x}), y\right).$$

6: $k \leftarrow k + 1$
7: **until** $k = K$
8: Compute cross-validation error

$$\text{CV}_K(\lambda) = \frac{1}{K} \sum_{k=1}^{K} L_k(\lambda).$$

9: **end for**
10: Model selection: pick λ with the lowest error $\text{CV}_K(\lambda)$.
11: Model training: compute final model $\widehat{f} = \widehat{f}^\lambda = M_\lambda(\mathcal{D}_{\text{use}})$.
12: Model evaluation: estimate generalization error using $\mathcal{D}_{\text{test}}$.

8

Classification

Roadmap

In Section 8.1, we introduce the concept of classification and distinguish it from regression. In Section 8.2, we study linear binary classification and show how feature maps can help increase the power of linear classifiers. In Section 8.3, we introduce the concept of surrogate loss and explain why it is essential in binary classification tasks. In Section 8.5, we move to multiclass classification and adapt the surrogate loss functions to match the new framework. In Section 8.7, we look at hypothesis testing and study situations where some types of errors are more important to avoid than others. Finally, in Section 8.7.2, we analyze ROC curves, a measure of the performance of a classification model.

Learning Objectives

After reading this chapter you should

- be able to give examples of a binary classification task,
- know the loss function used in classification as well as the surrogate loss functions that are used to train a classification model,
- be able to describe logistic regression,
- know what a feature map is and give examples where the use of nonlinear feature maps is necessary,
- know what the max-margin solution is, and be able to describe it both as the limit of logistic regression and as the solution of the support vector machine (SVM),
- be able to give examples of multiclass classification tasks,
- know the cross-entropy loss and explain why it is a generalization of the logistic loss,
- be able to explain what true and false positives/negatives and what type I and type II errors are,
- know the most common asymmetric loss measures such as the precision, the power, the FPR and FDR, and the recall,
- know what an ROC curve is and what the ROC curve of a good classifier looks like.

8.1 What is classification?

So far, we have considered a supervised machine learning pipeline, as shown in Figure 8.1, where a method takes a training set and outputs a mapping $\widehat{f}_{\mathcal{D}} : \mathcal{X} \to \mathcal{Y}$ that we then use for prediction. The model $\widehat{f}_{\mathcal{D}}$ is chosen from a set of models, e.g., by cross-validation. In the previous chapters, we focused on regression, where the domain $\mathcal{X}$ was the d-dimensional Euclidean space $\mathbb{R}^d$, and the output space $\mathcal{Y}$ was $\mathbb{R}$, the real line. Further, in our study of regression, we have mainly focused on the squared loss both as a training and evaluation metric.

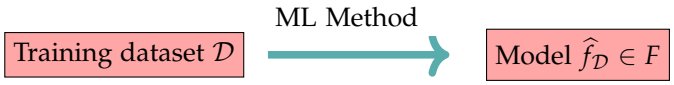
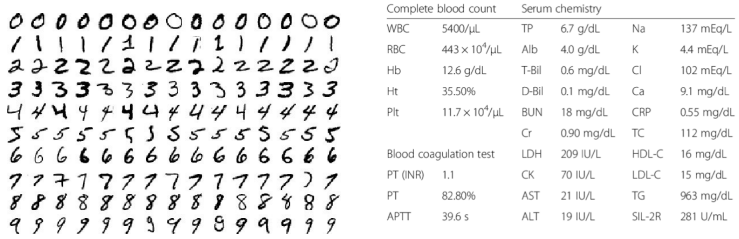


Figure 8.1: General machine learning pipeline. The scientist must choose the function class F, the training loss, and the optimization algorithm.

However, these choices might not be suitable for certain other prediction tasks, for example for the recognition of handwritten digits or risk identification using medical health records, shown in Figure 8.2.



Complete blood count		Serum chemistry			
WBC	5400/μL	TP	6.7 g/dL	Na	137 mEq/L
RBC	443 × 10^4/μL	Alb	4.0 g/dL	K	4.4 mEq/L
Hb	12.6 g/dL	T-Bil	0.6 mg/dL	Cl	102 mEq/L
Ht	35.50%	D-Bil	0.1 mg/dL	Ca	9.1 mg/dL
Plt	11.7 × 10^4/μL	BUN	18 mg/dL	CRP	0.55 mg/dL
		Cr	0.90 mg/dL	TC	112 mg/dL
Blood coagulation test		LDH	209 IU/L	HDL-C	16 mg/dL
PT (INR)	1.1	CK	70 IU/L	LDL-C	15 mg/dL
PT	82.80%	AST	21 IU/L	TG	963 mg/dL
APTT	39.6 s	ALT	19 IU/L	SIL-2R	281 U/mL

Figure 8.2: On the left, the task is to predict a digit from a picture containing a handwritten digit (MNIST). On the right, the task could be to predict whether a patient is at risk or not at risk based on blood measurements.

For these tasks, the output y is not continuous but lies in a discrete set. For the MNIST dataset, this set is $\{0, 1, 2, \ldots, 9\}$, while for the medical example it is $\{\text{at risk, not at risk}\}$. Hence, we would like our predictions $\widehat{y}$ to take values in certain *discrete subsets* of the real line. The task of predicting the label from a discrete set is known as *classification*.[1] If there are only two possible outcomes/labels, the classification task is referred to as *binary*; otherwise, it is called *multiclass*. The different possible outcomes are called *classes*. In Figure 8.3, we can an example of binary classification (left) and one of multiclass classification (right). In the following, we first focus on the binary classification case.

To solve a classification task, we face the following questions:

- How can we predict a discrete value using a function that usually maps to the real line?
- What would be a good loss to evaluate the prediction performance of a trained model?
- How can we obtain a model that has a small prediction error?

[1] When the label is from a continuous set, the task is called *regression*.

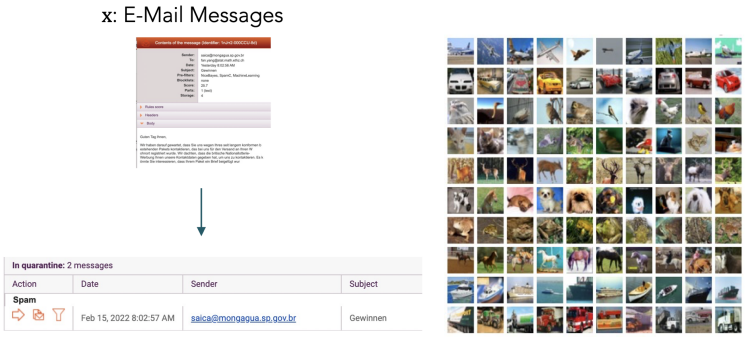


Figure 8.3: On the left, the model predicts whether an email is spam or not. On the right, it classifies an image as a *cat*, a *dog*, a *ship* etc. This dataset is known as *CIFAR-10* and, just like MNIST, it is used as a benchmark for model evaluation.

Similar to regression, before we can formulate the classification task as a mathematical problem, we need to convert the attributes of an instance and the corresponding label into mathematical values. While in the case of the digits, the mapping from digit to number is immediate, in the case of patients the choice is more arbitrary – for example, one can assign the value $+1$ to "at risk" and -1 or 0 to "not at risk" or vice versa. For multiclass classification, the choice could be even more delicate as shown in Section 8.5.

In regression, we trained a model $\widehat{f}_D : \mathbb{R}^d \to \mathbb{R}$ that outputs values anywhere on the real line $\mathbb{R}$. A simple and standard way to map these continuous values to label predictions in $\{-1, +1\}$ for binary classification is to use the sign function, i.e.

$$\widehat{y} = \operatorname{sign} \widehat{f}(\boldsymbol{x}).$$

The predicted label $\widehat{y}$ is either $+1$ or -1, which corresponds to the possible classes. An illustration of this system is shown in Figure 8.4. The dataset consists of points that are labeled either as class $+1$ (represented by green dots) or class -1 (represented by red dots). We then fit a function $\widehat{f}$ (the blue curve) to these points, e.g., via nonlinear regression. Our prediction rule for a new input point x is based on the sign of $\widehat{f}(x)$. If it is positive, we predict the class of x to be $+1$, and if it is negative, we predict it to be -1.

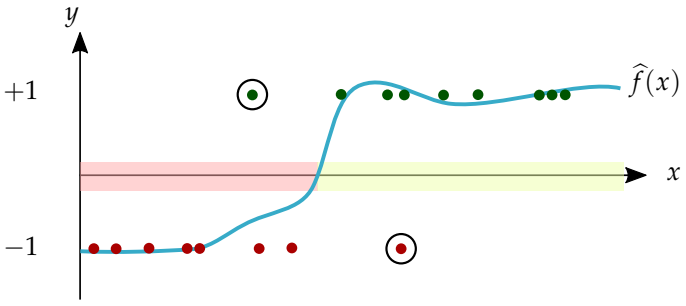


Figure 8.4: The red and green shaded regions on the x axis show which points will be predicted to belong to class -1 and $+1$ respectively. In this particular example, the function $\widehat{f}$ makes two prediction errors on the training data, marked with a circle in the figure.

A model is correct on an input point $\boldsymbol{x}$ if $\widehat{y}$ matches the true label y, that is, $\widehat{y} = y$. Hence, a natural pointwise metric to evaluate the predictions of a model $\widehat{f}_D$ is the zero-one loss $\ell_{0-1} : \mathcal{Y} \times \mathcal{Y} \to \{0, 1\}$

defined as

$$\ell_{0-1}(\widehat{y}, y) = \mathbb{I}_{\widehat{y} \neq y} = \begin{cases} 1, & \text{if } \widehat{y} \neq y; \\ 0, & \text{otherwise.} \end{cases}$$

Notice that, since $\widehat{y}, y \in \{-1, 1\}$, the zero-one loss can also be expressed as $\mathbb{I}_{\widehat{y} \cdot y < 0}$.

Remark. At this point, let us make three remarks.

1. The zero-one loss is closely related to the notion of *accuracy*. For instance, when we say that a classifier has 78% accuracy on a given dataset, we mean that $\ell_{0-1}(\widehat{y}, y) = 0$ for 78% of the datapoints, i.e. on these datapoints, the classifier "guesses" correctly.
2. For some applications, it is favorable to utilize alternative losses instead, for example, asymmetric losses. We will discuss this in Section 8.7.
3. For a classifier $\widehat{f}$, the set $\{\boldsymbol{x} : \widehat{f}(\boldsymbol{x}) = 0\}$ is a set of inputs $\boldsymbol{x}$ that separate the sets of points predicted as $+1$ and -1 respectively. This set is called the *decision boundary*. For linear classifiers in d dimensions, this set is a hyperplane (see Section 8.2 for details). However, decision boundaries for more general classifiers can be much more complex and consist of disconnected components.

8.2 The power of (featurized) linear classification

In this section, we discuss the specific class of linear classifiers, i.e. where $\widehat{f}(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$.[2] For any vector $\boldsymbol{w} \in \mathbb{R}^d$, we can define a linear classifier as a function of the form

$$\boldsymbol{x} \mapsto \begin{cases} 1, & \text{if } \boldsymbol{w}^\top \boldsymbol{x} > 0 \\ -1, & \text{if } \boldsymbol{w}^\top \boldsymbol{x} < 0. \end{cases}$$

What about points x for which $\boldsymbol{w}^\top \boldsymbol{x} = 0$? Recall that $\boldsymbol{w}^\top \boldsymbol{x} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle = \|\boldsymbol{w}\|_2 \|\boldsymbol{x}\|_2 \cos\theta$, where θ is the angle between the vectors $\boldsymbol{w}$ and $\boldsymbol{x}$. Thus, $\widehat{f}(\boldsymbol{x})$ is positive when the angle between $\boldsymbol{w}$ and $\boldsymbol{x}$ is acute and negative when it is obtuse. As a result, the points $\boldsymbol{x}$ that are classified as $+1$ lie on one side of the decision boundary - a hyperplane described by the equation $\boldsymbol{w}^\top \boldsymbol{x} = 0$ - while the points classified as -1 lie on the other side. In two dimensions, these two sides are half-planes (called half-spaces in more dimensions), as shown in Figure 8.5. The decision boundary is the border between the two half-planes.

How powerful are linear classifiers on the raw input vectors? On the one hand, the function class seems overly restrictive (even when the bias is included). For example, consider the scenarios in Figure 8.6: in Figure 8.6(a) and Figure 8.6(b), linear classifiers perform well, but in Figure 8.6(c) and Figure 8.6(d) they are completely unable to determine the classes of points. We hence conclude that

[2] As we have discussed in Chapter 4, we can incorporate a bias term into $\boldsymbol{w}$ by adding an entry equal to 1 to the feature vector $\boldsymbol{x}$, hence we omit the bias term w_0 for simplicity from here on.

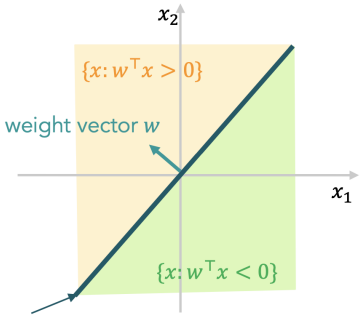
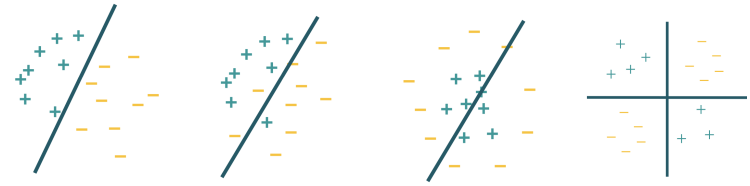


Figure 8.5: Points lying in the orange half-plane form an acute angle with $\boldsymbol{w}$ so they are classified as $+1$. Similarly, points in the green half-plane are classified as -1.

although linear classifiers can be effective when the classes are linearly separable, they can easily fail when the class distribution is more complex.



(a) Full linear separation

(b) Noisy linear separation

(c) Circularly arranged data are not linearly separable.

(d) The classes located in two opposite quadrants each are not linearly separable.

Figure 8.6: Examples of linearly separable data and data which are linearly separable after applying an appropriate feature map.

On the other hand, the picture changes drastically when we consider linear classification on featurized representations instead of the raw attributes themselves: When the input vector $\boldsymbol{x} \in \mathbb{R}^d$ is mapped to a different (often higher dimensional) vector $\phi(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), \ldots, \phi_p(\boldsymbol{x})) \in \mathbb{R}^p$ by a transformation $\phi : \mathbb{R}^d \to \mathbb{R}^p$ (as in Section 4.3), the effective function space becomes

$$F_\phi = \left\{ f : \mathbb{R}^d \to \mathbb{R} \mid f(\boldsymbol{x}) = \boldsymbol{w}^\top \phi(\boldsymbol{x}),\ \boldsymbol{w} \in \mathbb{R}^p \right\}.$$

Although the feature maps are nonlinear, the functions in F_ϕ are still linear in $\phi(\boldsymbol{x})$.

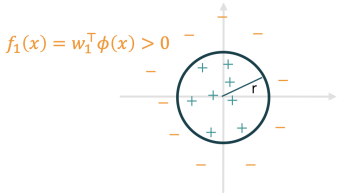
For instance, with certain pre-processing steps and a good choice of feature maps, linear classifiers can achieve up to 92% prediction accuracy in the MNIST dataset shown in Figure 8.2. On CIFAR-10 (Figure 8.3), a linear model can achieve up to 90% accuracy.[3]

[3] However, the latest neural networks can achieve more than 98% accuracy on CIFAR-10.

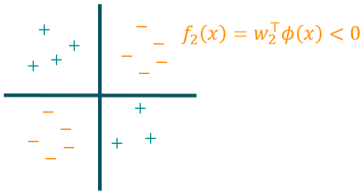
In Figure 8.6(c) and Figure 8.6(d), the positive and negative classes cannot be separated by a linear decision boundary. We now illustrate how a simple choice of a feature map ϕ would enable a linear classifier to separate these classes. Let's start by inspecting Figure 8.6(c). In this example, the points with the label "+" concentrate within a circle with a radius $r > 0$, while the points that are further than r from the origin have the label "-". The distance of a point from the origin corresponds to its norm and can be calculated using the squared components of the input vector $\boldsymbol{x}$. Indeed, using the feature map $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ with $\phi(x_1, x_2) = (1, x_1^2, x_2^2)$, the function $\widehat{f}(\boldsymbol{x}) = \boldsymbol{w}^\top \phi(\boldsymbol{x}) = r^2 - x_1^2 - x_2^2 \in F_\phi$ where $\boldsymbol{w} = (r^2, -1, -1)$ can perfectly classify (after applying the sign function) points that have a distance larger or smaller than r from the origin. The corresponding decision boundary is shown in Figure 8.7(a).

Similarly, Figure 8.6(d), the data points with the label "-" are those whose coordinates have the same sign, and the points with

the label "+" have coordinates with different signs. Therefore in this case, the feature map $\phi : \mathbb{R}^2 \to \mathbb{R}$ with $\phi(x_1, x_2) = x_1 x_2$ renders the data linearly separable. In particular, the linear function on the features $\widehat{f}(\boldsymbol{x}) = \boldsymbol{w}^\top \phi(\boldsymbol{x}) = w_1 x_1 x_2$ with $\boldsymbol{w} = -1$ can separate the two classes perfectly. The corresponding decision boundary $\{\boldsymbol{x} : x_1 x_2 = 0\}$ is shown in Figure 8.7(b).



(a) The decision boundary is a circle with radius r that contains the points labeled with "+". These points are the ones for which the function $\widehat{f}(\boldsymbol{x}) = r^2 - x_1^2 - x_2^2$ is positive. For the orange points, the function is negative.



(b) Points in the first and third quadrants are labeled with "-", and the ones in the second and fourth quadrants are labeled as "+". The function $\widehat{f}(x_1, x_2) = -x_1 x_2$ is negative in the first and third quadrants and positive in the other two.

Figure 8.7: Two examples where linear classifiers can perfectly separate complex datasets after applying feature maps.

In this section, we saw how powerful linear classifiers can be. In particular, when the input points are not linearly separable, feature maps may allow us to transform them into a space where linear separation is possible. Well-chosen features can make linear classifiers applicable to broad real-world scenarios where data may be quite complex. In later chapters, we will see that neural network classifiers are essentially linear classifiers based on features learned from a specific dataset!

8.3 *Surrogate loss functions for training*

In Section 8.2, we discussed how classifiers based on linear functions can predict binary labels. In this section, we discuss how one could train models $\widehat{f}_D$ (for general and linear function classes in particular) that have good classification accuracy using the loss minimization framework that we already used for regression.

A naive approach would be to replicate what we did in regression and minimize the evaluation metric directly on the training set. For classification, we could try to minimize the zero-one loss, as opposed to the squared loss in regression. In other words, we would need to minimize the classification error over the training dataset:

$$\sum_{(\boldsymbol{x},y)\in\mathcal{D}} \ell_{0-1}(\operatorname{sign} f_{\boldsymbol{w}}(\boldsymbol{x}), y) = \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \mathbb{I}_{\operatorname{sign} f_{\boldsymbol{w}}(\boldsymbol{x})\neq y} = \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \mathbb{I}_{f_{\boldsymbol{w}}(\boldsymbol{x})\cdot y<0}. \tag{8.1}$$

Even though we could use trial and error to find a function that achieves a small training error, in most cases, this endeavor would be computationally infeasible. In particular, the zero-one loss is a non-continuous (and hence not differentiable) and non-convex objective function.[4] This lack of convexity and differentiability hinders

[4] To see that the indicator function $\mathbb{I}_{t<0}$ is not convex, draw its graph and connect one point with a positive t-coordinate to one point with a negative t-coordinate. Recall that, in convex functions, the segments from one such point to another lie above the function's graph.

the application of standard optimization algorithms like (stochastic) gradient descent, which rely on these properties for efficiency and reliability. To overcome this issue, we introduce *surrogate loss functions* as a substitute for the zero-one loss function during training. We would like to emphasize that surrogate losses are usually only used for training, but not for evaluation purposes. For evaluation, it is most common to report the accuracy or error as defined in Equation 8.1.

Recall that the zero-one loss can be expressed as $\ell_{0-1}(\operatorname{sign} f_{\boldsymbol{w}}(\boldsymbol{x}), y) = \mathbb{I}_{yf_{\boldsymbol{w}}(\boldsymbol{x})<0}$, so it effectively only depends on its two arguments $\boldsymbol{x}$ and y via the variable $z := yf_{\boldsymbol{w}}(\boldsymbol{x})$. We would now like to discuss a few candidate losses (depicted in Figure 8.8) that only depend on z, and are both convex and differentiable (hence well-suited for standard optimization methods). Which of these functions would make a good surrogate loss?

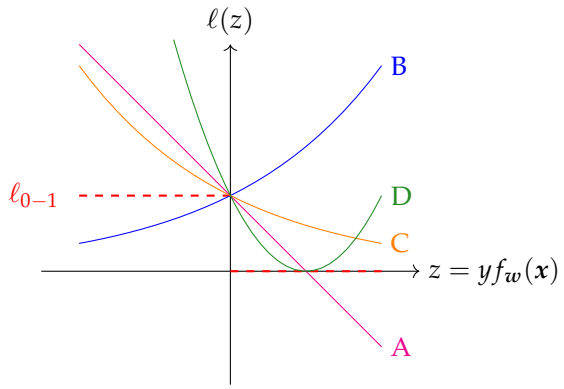
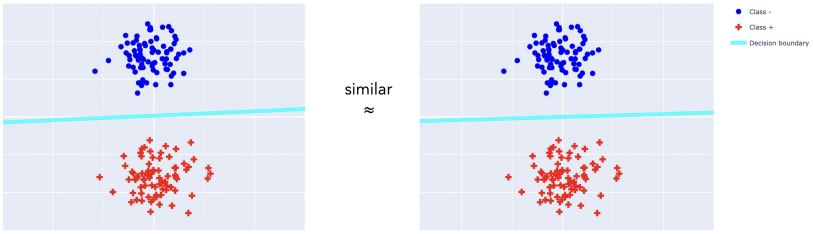


Figure 8.8: Some possible surrogate loss functions, plotted as functions of $z = yf_{\boldsymbol{w}}(\boldsymbol{x})$. We want the surrogate loss to be convex, so we only depict convex functions. However, only options (A) and (C) would be reasonable here, since they are the only ones that are decreasing in z.

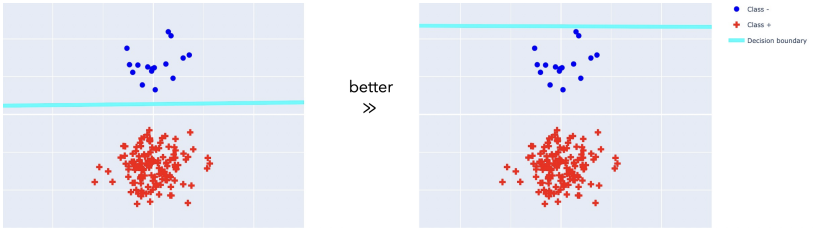
To select a good candidate, we first need to discuss which properties the function should satisfy. Note that the zero-one loss is a decreasing function of this quantity: It is equal to 1 for $z < 0$ and equal to 0 otherwise. The surrogate loss should follow a similar trend: the loss should be higher for wrong predictions (where $z < 0$ or equivalently $\operatorname{sign} f_{\boldsymbol{w}}(\boldsymbol{x}) \neq y \Leftrightarrow yf_{\boldsymbol{w}}(\boldsymbol{x}) < 0$) than for correct predictions (where $z > 0$). This requirement rules out candidates B and D.

It remains to decide between candidates A, a linear decreasing function, and C, a decreasing function bounded from below by zero. In a dataset with balanced classes, the two loss functions may yield quite similar results (see Figure 8.9(a)). However, since the linear loss becomes more negative as z grows, i.e., it rewards large values of $yf_{\boldsymbol{w}}(\boldsymbol{x})$, the linear loss can result in a suboptimal classifier in datasets with imbalanced classes (see Figure 8.9(b)). In contrast, the exponential function *C* does not exhibit this behavior.

Given the aforementioned issues with the linear loss, what remains are losses that look like curve C. We discuss two candidates that fall into this category: the exponential $\ell_{\exp}(z) = \mathrm{e}^{-z}$ and the logistic loss function $\ell_{\log}(z) = \log(1 + \mathrm{e}^{-z})$. Neither of them has



(a) Exponential and linear loss yield similar results if the classes are balanced.



(b) If the dataset is unbalanced, the linear loss function rewards pulling the decision boundary away from the majority class more than it penalizes misclassifying minority-class points. As a result, the model may prioritize correctly classifying the majority class at the expense of the minority class. The model with the exponential loss performs better in this case.

Figure 8.9: Exponential vs. linear loss for balanced and unbalanced classes.

the issues that arose for the other candidates above. For example, they only penalize incorrect predictions. Further, they are both decreasing, convex, and differentiable.

Which of the two should we then use? One of the obvious differences arises during optimization. In particular, the exponential loss function has unbounded values and derivatives, which could make optimization unstable and hypersensitive to noisy outliers. Indeed, $\frac{\mathrm{d}}{\mathrm{d}z}\ell_{\exp}(z) = -\mathrm{e}^{-z}$ explodes as $z \to -\infty$. The logistic loss mitigates this behavior. More precisely, the derivative of the logistic loss function

$$\frac{\mathrm{d}}{\mathrm{d}z}\ell_{\log}(z) = -\frac{1}{1+\mathrm{e}^{z}}$$

is bounded between $-1/2$ and -1 for misclassified points i.e. for $z < 0$, which ensures the stability of optimization and prevents *exploding* gradients. [5] This is only one practical reason why logistic loss is often preferred in practice. In Chapter 13, we will see that another natural motivation comes from its equivalence to the log-likelihood for a popular probabilistic model of classification data.

[5] A gradient is called *exploding* when its norm is extremely large, and *vanishing* when its norm is very close to zero.

The classification method that outputs a linear function $f_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$ and aims to minimize the average logistic loss is commonly called *logistic regression*. The objective function minimized in logistic regression is

$$L(w) = \frac{1}{n}\sum_{i=1}^{n} \log\left(1+\mathrm{e}^{-y_i w^\top x_i}\right). \tag{8.2}$$

The use of the term *regression* might initially seem odd, but we will later explain this denomination. Note that as in Section 8.2, logistic regression can also be performed on transformed features, usually

leading to better performance on complex nonlinear datasets.

8.4 Max-margin solution and logistic regression

For this section, we assume that the bias term w_0 is zero.

Suppose we have a linearly separable dataset $(x_1, y_1), \ldots, (x_n, y_n)$, such that there exists a linear decision boundary that perfectly separates the two classes: for some weight vector $w \in \mathbb{R}^d$ it holds $y_i \langle w, x_i \rangle > 0$ for all i. In such a setting, multiple decision boundaries can perfectly interpolate the training data, as visualized in Figure 8.10, and it becomes essential to select one of the corresponding classifiers in a principled manner.

In this section, we discuss the following questions:

Which classifier that interpolates the training data should we prefer?
And which classifier will gradient descent with logistic loss (8.2) choose?

Among the decision boundaries in Figure 8.10, B can be singled out to be *maximally far* from the data points in both classes. Intuitively, this is a good choice because the training data would need to be perturbed strongly to incur an error. If we assume the test data to follow a similar distribution to the training data, we can expect good generalization properties of this classifier. Thus, it makes sense to try and find a vector w such that the corresponding linear classifier maximizes the distance between the boundary and the data points that are closest to it. How can this corresponding vector be expressed mathematically, in terms of the training points?

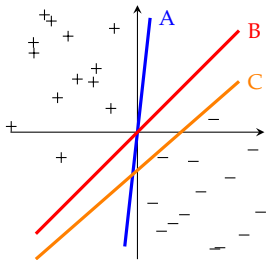


Figure 8.10: All decision boundaries classify all training data points correctly.

For that, we first need to find an expression for the distance between a datapoint and the decision boundary in terms of the weights w. Denote the angle between some $w, x_0 \in \mathbb{R}^d$ as θ. Using basic trigonometry, it is easy to see that the distance of x_0 to the decision boundary, i.e., the set $\left\{x \in \mathbb{R}^d : \langle w, x \rangle = 0\right\}$, is equal to $\|x_0\|_2 |\cos(\theta)|$. For the case $d = 2$, this is visualized in Figure 8.11. This expression can be rewritten as

$$\|x_0\|_2 |\cos(\theta)| = \|x_0\|_2 \frac{|\langle w, x_0 \rangle|}{\|w\|_2 \|x_0\|_2} = \frac{|\langle w, x_0 \rangle|}{\|w\|_2}.$$

If w is a unit-norm vector, the denominator is one, so that the distance can be written as $|\langle w, x_0 \rangle|$. Thus, we will restrict our search to vectors satisfying $\|w\|_2 = 1$. Note that if w classifies (x, y) correctly, we have that $|\langle w, x \rangle| = y \langle w, x \rangle$.

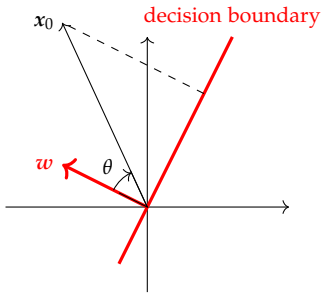


Figure 8.11: The distance between x_0 and the boundary $\left\{x \in \mathbb{R}^2 : \langle w, x \rangle = 0\right\}$ is given by $\|x\|_2 |\cos(\theta)|$.

We aim to maximize the (signed) distance of the decision boundary to the closest datapoints x_i, which for unit norm w is called the *margin* of w and is denoted as

$$\text{margin}(w) = \min_{1 \le i \le n} y_i \langle w, x_i \rangle .$$

The margin of a classifier w is visualized in Figure 8.12. Therefore,

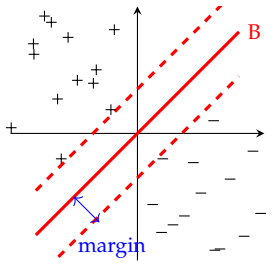


Figure 8.12: Margin of the classifier.

our objective is to solve

$$\max_{\|\boldsymbol{w}\|_2=1} \min_{1\le i\le n} y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle . \tag{8.3}$$

We denote $\boldsymbol{w}_{\text{MM}}$ as the solution of the optimization problem in Equation 8.3, where "MM" is short for "maximum margin".

The form of the optimization problem in Equation 8.3 is not straightforward to solve. To address this, we will show that a solution to the optimization problem from Equation 8.3 aligns in direction with the following optimization problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \|\boldsymbol{w}\|_2 \quad \text{such that} \quad y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \ge 1, \ \forall\, i = 1, \dots, n. \tag{8.4}$$

This form of the problem is commonly referred to as *support vector machine (SVM)*, and we denote a solution to Equation 8.4 as $\boldsymbol{w}_{\text{SVM}}$.

Lemma 8.1. *For linearly separable data, the solutions $\boldsymbol{w}_{\text{MM}}$ to Equation 8.3 and $\boldsymbol{w}_{\text{SVM}}$ to Equation 8.4 are equal up to scaling, that is,*

$$\frac{\boldsymbol{w}_{\text{SVM}}}{\|\boldsymbol{w}_{\text{SVM}}\|_2} = \boldsymbol{w}_{\text{MM}}.$$

Proof. Since both $\boldsymbol{w}_{\text{MM}}$ and $\boldsymbol{w}_{\text{SVM}}$ interpolate the data, and $\boldsymbol{w}_{\text{MM}}$ is the maximum margin classifier from Equation 8.3, we know that

$$\min_i y_i \langle \boldsymbol{w}_{\text{MM}}, \boldsymbol{x}_i \rangle \ge \min_i y_i \left\langle \frac{\boldsymbol{w}_{\text{SVM}}}{\|\boldsymbol{w}_{\text{SVM}}\|_2}, \boldsymbol{x}_i \right\rangle \ge \|\boldsymbol{w}_{\text{SVM}}\|_2^{-1}$$

where the second inequality follows from Equation 8.4. Consequently, $\|\boldsymbol{w}_{\text{SVM}}\|_2 \boldsymbol{w}_{\text{MM}}$ is a feasible solution to Equation 8.4 with norm

$$\|\|\boldsymbol{w}_{\text{SVM}}\|_2 \boldsymbol{w}_{\text{MM}}\|_2 = \|\boldsymbol{w}_{\text{SVM}}\|_2 \|\boldsymbol{w}_{\text{MM}}\|_2 = \|\boldsymbol{w}_{\text{SVM}}\|_2 .$$

Therefore, $\|\boldsymbol{w}_{\text{SVM}}\|_2 \boldsymbol{w}_{\text{MM}}$ is not only a feasible solution but also achieves the minimum $\|\boldsymbol{w}_{\text{SVM}}\|_2$. Since the minimum of Equation 8.4 is unique and given by $\boldsymbol{w}_{\text{SVM}}$, we know that $\|\boldsymbol{w}_{\text{SVM}}\|_2 \boldsymbol{w}_{\text{MM}} = \boldsymbol{w}_{\text{SVM}}$. Rearranging yields the claim. □

The naming of SVMs has its root in the fact that the points of the two classes that are closest to the decision boundary are called *support vectors*.

Remark. Note that when we add a bias term w_0 to the linear model, the definition of the margin changes to

$$\text{margin}(\boldsymbol{w}, w_0) = \min_{1\le i\le n} y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + w_0),$$

The max-margin solution

$$\boldsymbol{w}_{\text{MM}} := \operatorname*{arg\,max}_{\|\boldsymbol{w}\|_2=1, w_0} \min_{1\le i\le n} y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + w_0)$$

still searches for a vector where $\boldsymbol{w}$ has a unit norm (instead of searching for unit-norm vectors $(\boldsymbol{w}, w_0)$). Similarly, the SVM problem only minimizes the norm of the $d-$dimensional vector $\boldsymbol{w}$

$$\min_{(\boldsymbol{w},w_0)\in\mathbb{R}^{d+1}} \|\boldsymbol{w}\|_2 \text{ such that } y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + w_0) \geq 1, \text{ for all } i = 1, \ldots, n.$$

You can then show with the same arguments as above that Lemma 8.1 still holds.

8.4.1 Implicit bias of gradient descent

In practice, instead of solving Equation 8.3 or Equation 8.4 explicitly, we may prefer to run gradient descent (GD) on a differentiable and convex surrogate loss, such as the logistic loss from Equation 8.2.

If we continue to assume that the dataset $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$ is linearly separable, then the objective in Equation 8.2 is convex, but no global optimum exists. Consequently, when running GD with an appropriate step size, the training loss from Equation 8.2 approaches zero asymptotically,[6] but the iterates $\{\boldsymbol{w}^t : t \in \mathbb{N}\}$ do not converge to any fixed point, and can be proven to diverge in norm. Nonetheless, they may still converge *in direction*, that is, $\lim_{t\to\infty} \boldsymbol{w}^t / \|\boldsymbol{w}^t\|_2$ may exist. Recall that for linear classification, the direction of the weights fully determine the decision boundary.

[6] Note that the logistic loss from Equation 8.2 can never attain the value 0 exactly, but it can be artibrarily small.

It is possible to show that, even though the objective (8.2) does not explicitly incorporate the margin or the norm of the weights, gradient descent on the logistic loss with separable data will *implicitly* converge to the maximum margin classifier from Equation 8.3 *in direction*. This inherent preference of gradient descent with logistic loss is called the *implicit bias* of gradient descent.

Theorem 8.2. *Assume the data is linearly separable with positive margin, that is, there exists a $\boldsymbol{w} \in \mathbb{R}^d$ so that $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i\rangle > 0$ for all i. Then the logistic regression problem $\min_{\boldsymbol{w}} L(\boldsymbol{w})$ with L from Equation 8.2 is convex and gradient descent iterates $\boldsymbol{w}^t$ with step-size $\eta = 1$ satisfy*

$$\lim_{t\to\infty} \frac{\boldsymbol{w}^t}{\|\boldsymbol{w}^t\|_2} = \boldsymbol{w}_{\mathrm{MM}}.$$

The proof can be found in Ji and Telgarsky 2018 and Soudry, Hoffer, and Srebro 2017.

It is now natural to ask:

What happens if the data is not *linearly separable?*

To address this question in full generality is hard (cf. Ji and Telgarsky 2018), but we can treat one special case in which we can show that the loss from Equation 8.2 has a unique global minimizer to which gradient descent converges.

Theorem 8.3. *Assume that the data is* strictly *not linearly separable, meaning that for any $\boldsymbol{w} \neq \mathbf{0}$ there exists a $i \in \{1, \dots, n\}$ such that $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle < 0$. Then there exists a global minimizer $\widehat{\boldsymbol{w}}$ of the problem $\min_{\boldsymbol{w}} L(\boldsymbol{w})$ with L from Equation 8.2 and the gradient descent iterates $\boldsymbol{w}^t$ with step-size $\eta = 4/\sigma_{\max}^2(\boldsymbol{X})$ satisfy $\lim_{t\to\infty} \boldsymbol{w}^t = \widehat{\boldsymbol{w}}$.*

Bonus Material

For the proof, we will make use of the result in the following exercise.

Exercise 8.4. Show that for any twice-differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ that has a positive definite Hessian everywhere (called strictly convex) and that satisfies $\lim_{\|\boldsymbol{x}\|_2 \to \infty} f(\boldsymbol{x}) = \infty$ there exists a unique global minimizer $\boldsymbol{x}^* \in \mathbb{R}^d$ of f.

Proof of Theorem 8.3. First, recall the notation of the data matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ that has the i-th data point $\boldsymbol{x}_i$ as its i-th row. Notice how the assumption that for any $\boldsymbol{w} \neq \mathbf{0}$ there exists a $i \in \{1, \dots, n\}$ such that $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle < 0$ implies that the kernel of $\boldsymbol{X}$ is trivial, that is, $\ker(\boldsymbol{X}) = \{\mathbf{0}\}$. This is easy to see by contradition, because otherwise for any $\boldsymbol{w} \in \ker(\boldsymbol{X}) \setminus \{\mathbf{0}\}$ it would hold for all i that $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle = \mathbf{0}$. In turn, this implies that $n \geq d$.

We now derive the Hessian of L. Recall that L is defined as

$$L(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle)\right).$$

By defining the helper function $H(\boldsymbol{v}) = \frac{1}{n} \sum_{i=1}^{n} \log(1 + \exp(v_i))$ and

$$\boldsymbol{Z} = \begin{pmatrix} -y_1 \boldsymbol{x}_1^\top \\ \vdots \\ -y_n \boldsymbol{x}_n^\top \end{pmatrix} \in \mathbb{R}^{n\times d},$$

it is easy to see that $L(\boldsymbol{w}) = H(\boldsymbol{Z}\boldsymbol{w})$. Consequently, we have that

$$\nabla^2 L(\boldsymbol{w}) = \boldsymbol{Z}^\top \nabla^2 H(\boldsymbol{Z}\boldsymbol{w}) \boldsymbol{Z} = \boldsymbol{X}^\top \operatorname{diag}\left(g(\boldsymbol{w}, \boldsymbol{x}_i, y_i)\right) \boldsymbol{X}$$

with $g(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = \frac{\exp(-y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle)}{(1+\exp(-y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle))^2} > 0$.

Therefore, because $\ker\{\boldsymbol{X}\} = \{\mathbf{0}\}$, which holds if an only if the smallest singular value is larger than zero, $\sigma_{\min}(\boldsymbol{X}) = \sqrt{\lambda_{\min}\left(\boldsymbol{X}^\top \boldsymbol{X}\right)} > 0$, we can show that $\nabla^2 L(\boldsymbol{w})$ is positive definite on any ball $B_2(R) = \left\{\boldsymbol{w} \in \mathbb{R}^d : \|\boldsymbol{w}\|_2 \leq R\right\}, R > 0$. To see this, notice that

$$\min_{\|\boldsymbol{x}\|_2 = 1, \boldsymbol{w} \in B_2(R)} \boldsymbol{x}^\top \nabla^2 L(\boldsymbol{w}) \boldsymbol{x} \geq \min_{i, \boldsymbol{w} \in B_2(R)} g(\boldsymbol{w}, \boldsymbol{x}_i, y_i) \sigma_{\min}^2(\boldsymbol{X}) > 0$$

where the last inequality holds because $\min_{i, \boldsymbol{w} \in B_2(R)} g(\boldsymbol{w}, \boldsymbol{x}_i, y_i)$ is strictly positive for any R. It follows that L is strongly convex on any $B_2(R), R > 0$ and strictly convex on $\mathbb{R}^d$. Analogously, we can show that L is smooth everywhere because

$$\max_{\|\boldsymbol{x}\|=1, \boldsymbol{w}} \boldsymbol{x}^\top \nabla^2 L(\boldsymbol{w}) \boldsymbol{x} \leq \frac{1}{4} \sigma_{\max}^2(\boldsymbol{X}) =: \mu$$

where we use that $g \leq 1/4$ and $\sigma_{\max}(\boldsymbol{X}) = \sqrt{\lambda_{\max}\left(\boldsymbol{X}^\top \boldsymbol{X}\right)}$ denotes the largest singular value.

The derivations above hold in general, but they do not guarantee convergence of gradient descent yet. Since for any $\boldsymbol{w} \neq \mathbf{0}$ there is at least one data point $(\boldsymbol{x}_i, y_i)$ such that $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle < 0$, we know that

$$\max_{\|\boldsymbol{w}\|_2=1} \min_i y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \leq -\delta$$

for some $\delta > 0$. Therefore, for any $\boldsymbol{w}$, it holds

$$\begin{aligned} L(\boldsymbol{w}) &\geq \min_i \log(1 + \exp(-y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle)) \\ &= \log\left(1 + \exp\left(-\min_i y_i \left\langle \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|_2}, \boldsymbol{x}_i \right\rangle \|\boldsymbol{w}\|_2\right)\right) \\ &\geq \log(1 + \exp(\delta \|\boldsymbol{w}\|_2)) \\ &\to \infty \end{aligned}$$

for $\|\boldsymbol{w}\|_2 \to \infty$. From the continuity and strict convexity of L, it follows that L must have a unique global minimum $\widehat{\boldsymbol{w}}$ (Exercise 8.4) and there must exist a $R_1 > 0$ such that $\widehat{\boldsymbol{w}} \in B_2(R_1)$.

Further, if we initialize gradient descent at $\mathbf{0}$, gradient descent with step-size $\eta = 1/\mu$ decreases the function value in each iteration (known as sufficient decrease):

$$L(\boldsymbol{w}^{t+1}) \leq L(\boldsymbol{w}^t) - \frac{1}{2\mu} \left\|\nabla L(\boldsymbol{w}^t)\right\|_2^2.$$

Therefore, $\lim_{\|\boldsymbol{w}\|_2 \to \infty} L(\boldsymbol{w}) = \infty$ also implies that gradient descent stays inside $B_2(R_2)$ for some $R_2 > 0$.

Define $R = \max\{R_1, R_2\}$. We know that L is strongly convex on $B_2(R)$ and smooth everywhere, and gradient descent stays within $B_2(R)$ which also contains the unique global minimum $\widehat{\boldsymbol{w}}$. Since we could replace L outside of $B_2(R)$ with a strongly convex continuation of L, we can invoke Theorem 5.18 to get the desired result. This finishes the proof. □

8.4.2 *Soft-margin solution*

A natural question is whether we can generalize the maximum margin/SVM classifier to the setting where the data is not linearly separable. One approach to do that is discussed next.

The SVM objective in Equation 8.4 can be extended to cover cases when the two classes are not linearly separable. In that case, the constraint $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \geq 1$ cannot be satisfied for all $i = 1, \ldots, n$ so we need to relax it. This is achieved by the introduction of n additional nonnegative variables $\xi = (\xi_1, \ldots, \xi_n)$ that enter the problem in the following form: Instead of requiring $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \geq 1$, we now ask that

$$y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \geq 1 - \xi_i$$

and $\xi_i \geq 0$ for $i = 1, \ldots, n$. Hence, the optimization problem is transformed into

$$\min_{\boldsymbol{w} \in \mathbb{R}^d, \xi \in \mathbb{R}^n} \left(\|\boldsymbol{w}\|_2^2 + \lambda \sum_{i=1}^n \xi_i \right)$$

$$\text{such that} \begin{cases} y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \geq 1 - \xi_i \\ \xi_i \geq 0 \end{cases} \qquad \text{for all } i = 1, \ldots, n.$$

This form of the SVM is known as the *soft-margin SVM*, in contrast to the solution of Equation 8.4, which is also often referred to as the *hard-margin SVM* under the assumption of linear separability.

Here, the hyperparameter $\lambda > 0$ decides how much a violation of the constraints $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \geq 1$ should be penalized: If $\lambda \to \infty$, then we fit the model with the smallest cumulative violation possible, and if the data is linearly separable, we recover Equation 8.4. If $\lambda \to 0$ we lose the margin constraint altogether, and the solution will converge to $\boldsymbol{w} = 0$. In practice, $\lambda > 0$ can be tuned via cross-validation, as in Algorithm 5.

There is also another perspective of the soft-margin SVM: For a given $\boldsymbol{w} \in \mathbb{R}^d$, the optimal ξ is given by

$$\xi_i = \begin{cases} 1 - y_i \boldsymbol{w}^\top \boldsymbol{x}_i & \text{if } y_i \boldsymbol{w}^\top \boldsymbol{x}_i \leq 1 \\ 0 & \text{else.} \end{cases}$$

Thus, the optimization problem can also be expressed in the unconstrained form

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \left[\|\boldsymbol{w}\|_2^2 + \lambda \sum_{i=1}^{n} \max(0, 1 - y_i \boldsymbol{w}^\top \boldsymbol{x}_i) \right].$$

The quantity $\sum_{i=1}^{n} \max(0, 1 - y_i \boldsymbol{w}^\top \boldsymbol{x}_i)$ is known as the *hinge loss*. Hence, the soft-margin SVM can be interpreted as an ℓ_2-penalized hinge-loss optimization problem.

8.5 Multiclass classification

So far we have focused only on binary classification, that is, problems where there are only two classes. However, as we discussed in Section 8.1, in practice the number of classes is usually larger. Similar to the binary classification case we can assign integers to labels. For example, if we want to classify an image as *cat*, *dog*, or *ship*, then we could do the assignment

$$(\text{cat} \to 1), (\text{dog} \to 2), (\text{ship} \to 3).$$

Hence, an image showing a cat would be given the label 1, one that shows a dog would be given the label 2, and one that shows a ship would be labeled as 3.

In contrast to binary classification, one model $\widehat{f}$ is usually not enough to solve a multiclass classification task. In particular, prediction based on sign $\widehat{f}(\boldsymbol{x})$ as in binary classification can only output two different class labels. How about a more refined discretization of the function values? Even though that would allow us to return a discrete set as well, unless the classes are naturally *monotonically* related (for example the severity of an illness), such a discretization would not correctly reflect the structure of the data. For example, assigning "1" to cats, "2" to ships, and "3" to trees and trying to distinguish all classes with one function would suggest that cats are

closer to ships than trees which might be nonsensical. The same issue would arise for any other assignment.

Instead, we usually train K different models $\widehat{f}_1, \ldots, \widehat{f}_K \in F$, one for each class. During prediction time, for each point $\boldsymbol{x}$ we can then predict the class for which the function value is largest, i.e.

$$\widehat{y}(\boldsymbol{x}) = \arg\max_{1 \leq k \leq K} \widehat{f}_k(\boldsymbol{x}). \tag{8.5}$$

Let us now consider a multiclass classification task in $\mathbb{R}^2$ with $K = 3$ classes. If we are given $K = 3$ different linear classifiers $\widehat{f}_1, \widehat{f}_2, \widehat{f}_3$ on a training dataset $\mathcal{D}$, each of them will split the plane into two half-spaces: one set where $\widehat{f}_i(x) > 0$ and one where $\widehat{f}_i(x) < 0$, just like in binary classification. In the end, we'd like to predict three labels - what are the sets that will be predicted as $1, 2, 3$ respectively? By Equation 8.5, a point $\boldsymbol{x}$ is predicted to

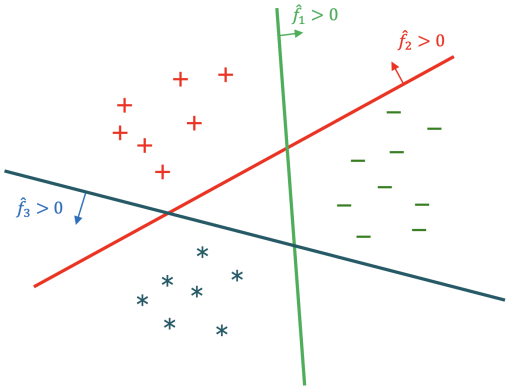


Figure 8.13: Each model $\widehat{f}_j$ splits the space into two parts. When we consider all of them together, the plane is split into seven parts.

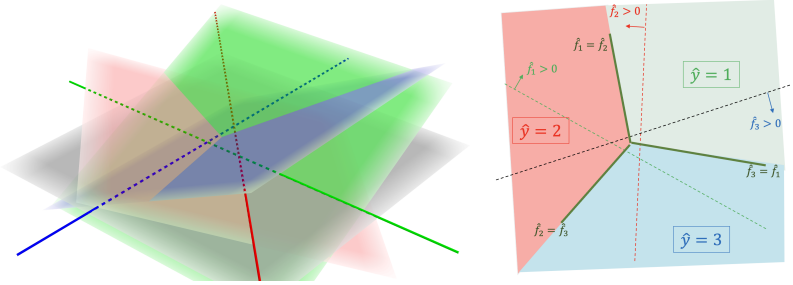
be from class k if $\widehat{f}_k(\boldsymbol{x}) \geq \max_j \widehat{f}_j(\boldsymbol{x})$. Hence, the areas that are classified with the same label are

$$A_1 := \left\{ \boldsymbol{x} \in \mathbb{R}^2 : \widehat{f}_1(\boldsymbol{x}) > \widehat{f}_2(\boldsymbol{x}) \text{ and } \widehat{f}_1(\boldsymbol{x}) > \widehat{f}_3(\boldsymbol{x}) \right\},$$
$$A_2 := \left\{ \boldsymbol{x} \in \mathbb{R}^2 : \widehat{f}_2(\boldsymbol{x}) > \widehat{f}_1(\boldsymbol{x}) \text{ and } \widehat{f}_2(\boldsymbol{x}) > \widehat{f}_3(\boldsymbol{x}) \right\},$$
$$A_3 := \left\{ \boldsymbol{x} \in \mathbb{R}^2 : \widehat{f}_3(\boldsymbol{x}) > \widehat{f}_1(\boldsymbol{x}) \text{ and } \widehat{f}_3(\boldsymbol{x}) > \widehat{f}_2(\boldsymbol{x}) \right\}.$$

If a test point $\boldsymbol{x} \in A_j$, then $\widehat{y}(\boldsymbol{x}) = j$. An example of when $d = 2$ is visualized in Figure 8.14.

In Figure 8.14(b), we show an example of a partition induced by three classifiers in Figure 8.13. Notice that there are now three decision boundaries, and they are the straight lines on which $\widehat{f}_1(\boldsymbol{x}) = \widehat{f}_2(\boldsymbol{x}), \widehat{f}_2(\boldsymbol{x}) = \widehat{f}_3(\boldsymbol{x})$, and $\widehat{f}_3(\boldsymbol{x}) = \widehat{f}_1(\boldsymbol{x})$ respectively. If the classifiers are nonlinear, the prediction using multiple models is identical, however, the decision boundaries will, in general, be curved.

Now that we know how to predict from multiple classes using an ensemble of K models, we discuss how we can train such an ensemble to achieve good accuracy.



(a) Three hyperplanes defined by the linear functions $\widehat{f}_i(\boldsymbol{x}) = \langle \boldsymbol{w}_i, \boldsymbol{x} \rangle, i = 1, 2, 3$ yield the partition of the feature space.

(b) Each point $\boldsymbol{x}$ is now classified according to which of the three areas it belongs to.

Figure 8.14: Three linear functions partition the feature space into three sections by assigning the label to the function with the highest value.

Training one-vs-rest binary classifiers. One first approach is to train each model separately in a one-vs-rest way, where we repeat the following process for each class $k \in \{1, \dots, K\}$:

- Assign the new label $\tilde{y} = -1$ to all points whose label is $y = k$,
- Assign the new label $\tilde{y} = +1$ to all points whose label is $y \neq k$,
- Perform binary classification on the relabeled dataset $\mathcal{D}_k$ to obtain model $\widehat{f}_k$.

For $k = 1$, this method is presented in Figure 8.15.

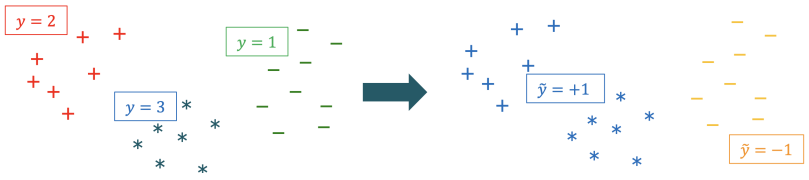


Figure 8.15: We relabel all data points so that the points of the class $y = 1$ are separated from the rest. Then, we find the decision boundary using binary classification techniques. We repeat this process for $k = 2, \dots, k$ to separate all classes.

Implementation

In Python, the code to perform multiclass classification with this method is the following:

```
method = sklearn.linear_model.Logistic_Regression(multiclass='ovr')
method.fit(X,y)
```

The problem with this technique is that we have to solve K different classification tasks, one for each class, which might be substantially time-consuming. Therefore, it would be very useful to be able to train all models $\widehat{f}_1, \dots, \widehat{f}_K$ simultaneously.

Cross-entropy To do this, we neither need to change the function class nor the optimization algorithm. Linear functions and the standard gradient optimization algorithms work just as well as before. The key change lies in the loss function. The logistic loss is crafted for binary classification so it cannot support the existence of more than two classes. However, an extension of the logistic loss is the

so-called *cross-entropy loss*. This loss function is defined as

$$\ell_{\text{ce}}\left(\widehat{f}_1(\boldsymbol{x}), \widehat{f}_2(\boldsymbol{x}), \dots, \widehat{f}_K(\boldsymbol{x}), y\right) = -\log\left(\frac{e^{\widehat{f}_y(\boldsymbol{x})}}{\sum_{k=1}^{K} e^{\widehat{f}_k(\boldsymbol{x})}}\right),$$

where $y \in \{1, \dots, K\}$ and $\widehat{f}_1(\boldsymbol{x}), \dots, \widehat{f}_K(\boldsymbol{x}) \in \mathbb{R}$. Often we will denote the vector $(\widehat{f}_1(\boldsymbol{x}), \dots, \widehat{f}_K(\boldsymbol{x}))$ by $\widehat{f}(\boldsymbol{x})$. Note that this loss function encourages the value $\widehat{f}_{y_i}(\boldsymbol{x}_i)$ of the model corresponding to the *true class* y_i to be *larger* than all other function values $\widehat{f}_k(\boldsymbol{x}_i)$, $k \neq y_i$.

For $K = 2$, one can show that the cross-entropy loss coincides with the logistic loss if we use the labels $\{-1, +1\}$.

Exercise 8.5. Show that the logistic loss of a model $\widehat{f}$ and is equal to the cross-entropy loss of the models $\widehat{f}_1 = \widehat{f}/2, \widehat{f}_{-1} = -\widehat{f}/2$.

Solution. Plugging into the definition of the cross-entropy loss yields

$$\begin{aligned}
-\log\left(\frac{e^{\widehat{f}_y(\boldsymbol{x})}}{\sum_{k\in\{-1,+1\}} e^{\widehat{f}_k(\boldsymbol{x})}}\right) &= \log\left(\frac{\exp(\widehat{f}_1(\boldsymbol{x})) + \exp(\widehat{f}_{-1}(\boldsymbol{x}))}{\exp(\widehat{f}_y(\boldsymbol{x}))}\right) \\
&= \log\left(\exp(\widehat{f}_1(\boldsymbol{x}) - \widehat{f}_y(\boldsymbol{x})) + \exp(\widehat{f}_{-1}(\boldsymbol{x}) - \widehat{f}_y(\boldsymbol{x}))\right) \\
&= \begin{cases} \log(1 + \exp(-\widehat{f}(\boldsymbol{x}))) & \text{if } y = 1 \\ \log(1 + \exp(\widehat{f}(\boldsymbol{x}))) & \text{if } y = -1 \end{cases} \\
&= \log(1 + \exp(-y\widehat{f}(\boldsymbol{x})))
\end{aligned}$$

which is exactly the logistic loss of the model $\widehat{f}$. □

The minimization problem that we are now trying to solve is

$$\min_{f_1,\dots,f_K \in F} \left[\sum_{i=1}^{n} \ell_{\text{ce}}\left(f(\boldsymbol{x}_i), y_i\right)\right],$$

where $(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)$ are the training data points. Just like in binary classification, we usually consider functions $f_1(\boldsymbol{x}), \dots, f_K(\boldsymbol{x})$ that are parametrized by finite-dimensional parameters $\boldsymbol{w}_1, \dots, \boldsymbol{w}_K \in \mathbb{R}^d$, so that the optimization problem can be restated as

$$\min_{\boldsymbol{w}_1,\dots,\boldsymbol{w}_K \in \mathbb{R}^d} \left[\sum_{i=1}^{n} \ell_{\text{ce}}\left(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i\right)\right],$$

where $\boldsymbol{w}$ denotes the matrix $(\boldsymbol{w}_1, \dots, \boldsymbol{w}_K)$. Running gradient descent on this loss function yields K weight vectors $(\widehat{\boldsymbol{w}}_1, \dots, \widehat{\boldsymbol{w}}_K)$ that result in K models with $\widehat{f}_k = f_{\widehat{\boldsymbol{w}}_k}$ for $k = 1, \dots, K$.

Implementation

Multiclass classification with the use of the cross-entropy loss is implemented in `Python` in almost the same way as the previous one-vs-rest approach was. The code snippet is the following:

```
method = sklearn.linear_model.Logistic_Regression(multiclass='multinomial')
method.fit(X,y)
```

Before closing this section, we would like to illustrate in Figure 8.16 how the two above multiclass classification approaches, one-vs-rest and cross-entropy, generally can lead to vastly different decision boundaries even for linear models.

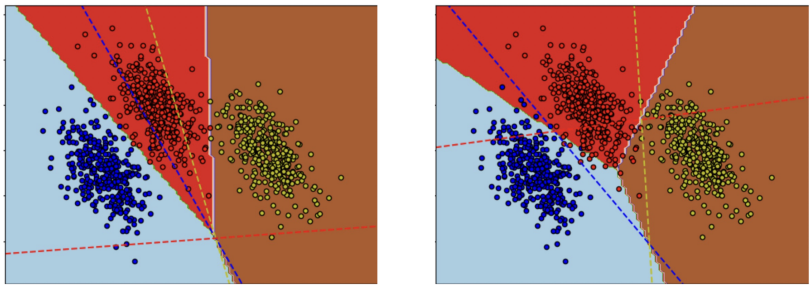


Figure 8.16: The decision boundaries in the left picture have been drawn with the use of the cross-entropy loss. On the right, we have used the one-vs-rest approach.

8.6 *Generalization of classifiers*

In this section, we discuss how to evaluate the performance of a classifier on unseen data. We will assume that there exists a "true" or "best" labeling function y^* which maps every input $\boldsymbol{x}$ to its true label. A "good" classifier" $\widehat{y}$ should be close to y^*, i.e. it should have a low labeling error

$$\ell(\widehat{y}(\boldsymbol{x}), y^*(\boldsymbol{x})) = \mathbb{I}_{\widehat{y}(\boldsymbol{x}) \neq y^*(\boldsymbol{x})}.$$

Ideally, we would like to know the average labeling error across all samples in the test set in which the inputs $X \in \mathcal{X}$ come from a probability distribution $\mathbb{P}_X$. Since we use the 0-1 loss to evaluate the accuracy of a classifier, the expected classification (0-1) error for a point $x \in \mathcal{X}$ is defined as

$$\mathbb{E}_X \ell\left(\widehat{y}(\boldsymbol{x}), y^\star(\boldsymbol{x})\right) = \mathbb{E}_X \mathbb{I}_{\widehat{y}(\boldsymbol{x}) \neq y^\star(\boldsymbol{x})} = \mathbb{P}_X\left(\widehat{y}(\boldsymbol{x}) \neq y^\star(\boldsymbol{x})\right),$$

where we assume that the inputs $X \in \mathcal{X}$ comes from a probability distribution $\mathbb{P}_X$. A *good* classification model has a low expected 0-1 error.

Similar to regression, we can neither compute nor estimate the expected 0-1 error directly using data since we do not have access to the true label $y^\star(\boldsymbol{x})$ for any x. Instead, we would again like to use an estimate of the generalization error $\mathbb{E}_{X,Y} \ell\left(\widehat{y}(X), Y\right)$, where Y is the label we can observe.

To understand the relation between the generalization error and the expected classification error, we again need to specify a model for the joint distribution of the observed samples (X, Y).

We assume that the inputs are distributed according to some probability distribution $\mathbb{P}_x$ and that there exists a ground-truth labeling function y^* which maps $\boldsymbol{x}$ to its true label $y^*(\boldsymbol{x})$ (for simplicity, we focus on binary classification, i.e. $y^*(\boldsymbol{x}) \in \{-1, +1\}$). However, the labels that we measure are noisy (not precisely equal

to $y^*(\boldsymbol{x})$). We model this by defining the conditional label y given $\boldsymbol{x}$ as

$$(y|\boldsymbol{x}) = y^*(\boldsymbol{x})\varepsilon, \tag{8.6}$$

where the noise $\varepsilon \in \{-1, +1\}$ is distributed according to some distribution $\mathbb{P}_\varepsilon$ and effectively causes label flip. Thus, we can describe the joint distribution of the data by

$$\mathbb{P}(\boldsymbol{x}, y) = \mathbb{P}_x(\boldsymbol{x})\mathbb{P}(y|\boldsymbol{x}),$$

with the conditional distribution $\mathbb{P}(y|\boldsymbol{x})$ defined as above. Note that Equation 8.6 is a *multiplicative noise model*, as opposed to the additive noise model $Y = f^\star(X) + \varepsilon$ we used for regression (compare with Section 6.1). Furthermore, in classification, we do not require the noise ε to be independent of $\boldsymbol{x}$, since in many settings the number of errors (label flips) grows as $\boldsymbol{x}$ gets closer to the decision boundary.

An example where there is such a dependence is shown in Figure 8.17. There, the points that are further away from the decision boundary have a lower probability of having a flipped label.

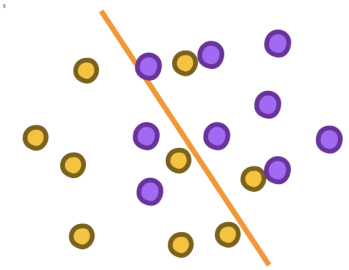


Figure 8.17: In this example, points that lie closer to the decision boundary have a larger probability of having a flipped label. Thus, the noise ε depends on the position $\boldsymbol{x}$ of a point.

A good classifier has a low generalization error on the joint distribution defined above, which for the zero-one loss can be written as

$$\begin{aligned} L\left(\widehat{f}; \mathbb{P}_{X,Y}\right) &= \mathbb{E}_{X,Y}\ell\left(\widehat{f}(X), Y\right) \\ &= \mathbb{E}_{X,Y}\ell_{0-1}\left(\widehat{y}, y\right) \\ &= \mathbb{E}_{X,Y}\left(\mathbb{I}_{\widehat{y}\cdot y<0}\right) \\ &= \mathbb{P}_{X,Y}\left(\widehat{y} \neq y\right), \end{aligned} \tag{8.7}$$

where we used the property $\mathbb{E}\left[\mathbb{I}_A\right] = \mathbb{P}(A)$. This shows that the generalization error in classification can be interpreted as the theoretical probability of misclassification of a randomly sampled test datapoint.

One question remains: which joint distribution $\mathbb{P}$ do we compute the generalization error on? So far (e.g. in Chapter 6), we have focused on the so-called *i.i.d. assumption*, according to which the training and the test data are i.i.d., i.e. they are independently sampled from the same distribution. However, in many settings, the test distribution can deviate from the training distribution, thus, one wants the model to perform well not only on $\mathbb{P}_{\text{train}}$ but also on a set of possible test distributions. *Out-of-distribution generalization* is an active area of research and is discussed in more detail in Section 8.7.3. Another active area of research, *fairness*, considers the performance of a classifier on various subpopulations of the test data. In addition to maximizing the accuracy of a classifier on unseen data, one aims to make the prediction fair with respect to these subgroups. Fairness is introduced and discussed in more detail in Section 8.7.4.

For the remainder of this section, we stick to the i.i.d. assumption and now discuss empirical approximation of the generalization error in the i.i.d. setting. As discussed in Chapter 6, the practitioner typically has no access to $\mathbb{P}_{X,Y}$, so the generalization error is not computable. Thus, to assess the performance of a given model $\widehat{y}$, we evaluate the empirical classification error on a test set $\mathcal{D}_{\text{test}}$ that is held out during training time by computing

$$\frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(x,y)\in\mathcal{D}_{\text{test}}} \mathbb{I}_{\widehat{y}(x)\neq y}.$$

For model selection, we can use cross-validation like in Section 6.3.

8.7 Other evaluation metrics for classifiers

The average classification error is not the only possible or even desirable way to evaluate a classifier, even for finite samples. In the following subsections, we discuss alternative metrics that might be of interest to the practitioner in many scenarios.

8.7.1 Asymmetric errors and hypothesis testing

So far in classification, we have treated all wrong predictions equally. No matter whether an object in class "1" was mistakenly classified as "2" or an object in class "3" was classified as "1", all mistakes were counted equally. However, what happens when one of the classes is more important to get right? Or when e.g. mistaking a cat for a ship is worse than mistaking a cat for a dog?

Example 8.6. Consider two tests designed by a medical lab, whose purpose is to classify the examined individuals as *healthy* or *infected*. The lab wants to choose one of these tests and promote it for mass production to be used broadly (e.g., during the Covid-19 pandemic). We evaluate the tests in 20 people, 10 of which have the virus. The results are shown in the following tables:

Test A

Prediction \ Reality	infected	healthy
infected	9	5
healthy	1	5

Test B

Prediction \ Reality	infected	healthy
infected	5	1
healthy	5	9

Table 8.1: The results of the two tests. The green numbers represent the correct predictions and the red numbers represent the wrong ones.

We observe that both tests have the same number of correct predictions, namely 14 out of 20. Consequently, when only considering the average classification error on the test set (i.e. the number of mistakes), one would arrive at the conclusion that the two tests perform equally well.

However, the two tests behave very differently. Test A is very good at detecting people that have the virus. Of the 10 people that are truly sick, only one was wrongly classified as *healthy* by test A. On the other hand, it makes many mistakes for people that do not have the virus. Of the 10 healthy people, it falsely labeled five of them as *infected*.

The second test has the complete opposite behavior: it is very accurate for people that do not have the virus, but it misclassifies many *infected* people as *healthy*.

Thus, which test we will choose largely depends on our objectives. If our main goal is to prevent the spread of the virus and protect public health, then we would choose test A which effectively detects infected people. On the other hand, if the virus is not so dangerous and our main goal is not to restrict the individual freedom of the people by putting them unnecessarily into quarantine, then we would choose test B.

In situations like the previous example, where there are two classes but one of them is more important to get right, we often resort to ideas from hypothesis testing, which is a classical topic in statistics. For simplicity of exposition, we assign the important class to the *negative* label -1. The other class will be the *positive* class and is assigned the label $+1$. The hypothesis that an object belongs to the negative class is called the *null hypothesis* and the hypothesis that it belongs to the positive class is called the *alternative hypothesis*.

In Example 8.6, if we focus on the protection of public health, then the important class will be one of the infected people because these are the most important to detect [7]. We consider this to be the negative class. Given a person $\boldsymbol{x}$, the hypothesis that $y = -1$, that is, the hypothesis that $\boldsymbol{x}$ is infected, is the null hypothesis, whereas the hypothesis that $\boldsymbol{x}$ is healthy is the alternative. Typically, $\widehat{y}$ denotes our prediction for that person, which is either -1 or $+1$.

[7] The statistical terminology contradicts the one used in real life in this particular case. In reality, when a person is infected we always use the term *positive*, and if they are healthy we call them *negative*. Hence, in what follows we should be careful not to get confused by the use of the opposite notation.

Based on this terminology, we define the following types of predicted samples:

		Reality	
		$y = -1$	$y = +1$
Prediction	$\widehat{y} = -1$	**TN**	**FN (Type II error)**
	$\widehat{y} = +1$	**FP (Type I error)**	**TP**

Table 8.2: The four new metrics expressed in terms of the true label y and the predicted label $\widehat{y} = -1$.

- **True negative** (TN) denotes an input point $\boldsymbol{x}$ whose real label is $y = -1$ and whose predicted label is $\widehat{y} = -1$.
- **True positive** (TP) denotes an observation $\boldsymbol{x}$ whose real label is $y = 1$ and whose predicted label is $\widehat{y} = 1$.
- **False negative** (FN) denotes an observation $\boldsymbol{x}$ whose real label is $y = 1$ but its predicted label is $\widehat{y} = -1$.
- **False positive** (FP) denotes an observation $\boldsymbol{x}$ whose real label is $y = -1$ but its predicted label is $\widehat{y} = 1$.

In terms of Example 8.6, a true positive is a person $\boldsymbol{x}$ who is healthy and is also classified as healthy by the medical test. A false negative is a person that is in fact healthy (positive) but who is classified as infected (negative) by the medical test.

A simple rule that helps us distinguish the four new metrics is the following: the word *negative* or *positive* that each of them contains refers to the *predicted* label (not the real one!). The word *true* or *false* reflects whether the prediction was correct.

For the two types of error, namely false positive and false negative, there are two extra terms used in statistical parlance. A false positive is called a *type I error* and a false negative is called a *type II error*. Since it is more important to predict the negative class correctly, our top priority is to control the type I error.

To work with the above types of error, we will recall the notion of the *empirical measure*:

Definition 8.7 (Empirical measure)**.** Let $\mathcal{D} = \{(x_i, y_i) \mid i = 1, \ldots, n\} \subset \mathcal{X} \times \{-1, 1\}$ denote the training dataset. The *empirical measure* $\mathbb{P}_n$ is a probability measure on the same space as $\mathbb{P}_{X,Y}$, which is defined as

$$\mathbb{P}_n(A) := \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{(x_i, y_i) \in A}$$

for any event $A \subset \mathcal{X} \times \{-1, 1\}$.

In other words, $\mathbb{P}_n(A)$ is equal to the percentage of training data points that belong to the set A. $\mathbb{P}_n$ is an *estimate* of $\mathbb{P}_{X,Y}$: due to the law of large numbers,

$$\mathbb{P}_n(A) \to \mathbb{P}_{X,Y}(A) \text{ as } n \to \infty$$

for all events A in the domain of $\mathbb{P}_n, \mathbb{P}$.

In previous sections where the mistakes were all equally important, we used the average zero-one error

$$\frac{1}{n} \sum_{(x,y) \in \mathcal{D}_{\text{test}}} \mathbb{I}_{\widehat{y}(x) \neq y}$$

to evaluate a model $\widehat{y}(x)$. Now that type I and type II errors have different severity, it would be more appropriate to use a weighted sum of the two errors such as

$$\frac{c_{\text{FN}}}{\#\{y = +1\}} \sum_{(x,y):y=1} \mathbb{I}_{\widehat{y}(x)=-1} + \frac{c_{\text{FP}}}{\#\{y = -1\}} \sum_{(x,y):y=-1} \mathbb{I}_{\widehat{y}(x)=+1}$$

$$= c_{\text{FN}} \frac{\#\text{FN}}{\#\{y = +1\}} + c_{\text{FP}} \frac{\#\text{FP}}{\#\{y = -1\}},$$

where the constants $c_{\text{FN}}, c_{\text{FP}}$ are the weights assigned to a false negative and a false positive respectively. The symbol # stands for the number of points that belong to a set (cardinality).

Bonus Material

It is easy to see that the minimization of the error function

$$\frac{c_{\text{FN}}}{\#\{y=+1\}} \sum_{(\boldsymbol{x},y):y=1} \mathbb{I}_{\widehat{y}(\boldsymbol{x})=-1} + \frac{c_{\text{FP}}}{\#\{y=-1\}} \sum_{(\boldsymbol{x},y):y=-1} \mathbb{I}_{\widehat{y}(\boldsymbol{x})=+1}$$

depends only on the ration $\delta = c_{\text{FN}}/c_{\text{FP}}$ and not on the exact values of $c_{\text{FN}}, c_{\text{FP}}$. More specifically, this minimization is equivalent to the minimization of the function

$$\frac{\delta}{\#\{y=+1\}} \sum_{(\boldsymbol{x},y):y=1} \mathbb{I}_{\widehat{y}(\boldsymbol{x})=-1} + \frac{1}{\#\{y=-1\}} \sum_{(\boldsymbol{x},y):y=-1} \mathbb{I}_{\widehat{y}(\boldsymbol{x})=+1}.$$

When the two classes are linearly separable, we can use a modified, *cost-sensitive* SVM to solve this problem: instead of

$$\left(\arg\min_{\widetilde{\boldsymbol{w}}\in\mathbb{R}^d} \|\widetilde{\boldsymbol{w}}\|_2\right) \text{ such that } y_i \widetilde{\boldsymbol{w}}^\top \boldsymbol{x}_i \geq 1 \text{ for all } i = 1, \ldots, n,$$

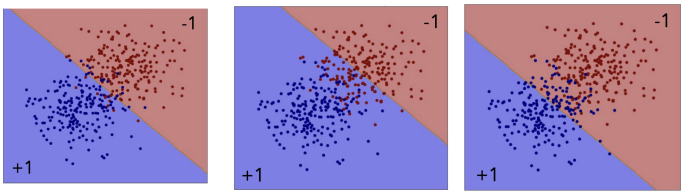
we should now solve

$$\left(\arg\min_{\widetilde{\boldsymbol{w}}\in\mathbb{R}^d} \|\widetilde{\boldsymbol{w}}\|_2\right) \text{ such that } \begin{cases} y_i \widetilde{\boldsymbol{w}}^\top \boldsymbol{x}_i \geq 1 & \text{if } y_i = -1 \\ y_i \widetilde{\boldsymbol{w}}^\top \boldsymbol{x}_i \geq \delta & \text{if } y_i = +1 \end{cases}$$

This will give us a decision boundary with different margins for each class.

In what follows we will look at an alternative of this technique, where, instead of changing the optimization problem, we change the decision rule.

As we discussed, false positives (type I errors) are usually the type of error that we prioritize to avoid, so c_{FP} is generally larger than c_{FN}. The effect of varying c_{FP} and c_{FN} is presented in Figure 8.18. When $c_{\text{FP}} < c_{\text{FN}}$, it is more important to control false negatives. In other words, we do not want to classify points as -1 when their true label is $+1$. This is why the boundary moves closer to the negative class. In the converse case, we observe an analogous shift of the boundary towards the positive class.



$c_{\text{FP}} = c_{\text{FN}}$ $c_{\text{FP}} < c_{\text{FN}}$ $c_{\text{FP}} > c_{\text{FN}}$

Figure 8.18: When $c_{\text{FP}} < c_{\text{FN}}$, the boundary moves closer to the negative class. In the converse case, the boundary shifts towards the positive class.

Figure 8.18 reveals a tradeoff between type I and type II errors. If we want to reduce type I errors (false positives), we can increase c_{FP}, which will force the decision boundary to move closer to the positive class. Despite its efficiency in eliminating type I errors, this strategy inevitably increases the number of type II errors. Conversely, if we try to reduce type II errors, the type I errors will likely be increased.

In commonly used asymmetric errors, the following ratios often appear:

$$\text{FNR} := \frac{\#\text{FN}}{\#\{y=+1\}} \quad \text{and} \quad \text{FPR} := \frac{\#\text{FP}}{\#\{y=-1\}}$$

These ratios are known as the *false negative rate (FNR)* and *false positive rate (FPR)* and take values in $[0,1]$. Similarly, we define

$$\text{TPR} = \frac{\#\text{TP}}{\#\{y=+1\}}, \ \text{TNR} = \frac{\#\text{TN}}{\#\{y=-1\}}.$$

In statistical parlance, the TPR is also known as the *power* or the *recall*.

Bonus Material

Finally, the *precision* is defined as the percentage of points labeled as $+1$ whose true label is also $+1$. The *false discovery rate (FDR)* is the percentage of points labeled as $+1$ whose true label is -1. In other words,

$$\text{precision} = \frac{\#\text{TP}}{\#\{\widehat{y}=+1\}} \quad \text{and} \quad \text{FDR} = \frac{\#\text{FP}}{\#\{\widehat{y}=+1\}}.$$

Precision and FDR add up to one.

Remark. FNR and FPR denote the percentage of false negatives and false positives respectively. By the definition of the empirical measure,

$$\text{FNR} = \frac{1}{\#\{y=1\}} \sum_{(\boldsymbol{x},y)\in\mathcal{D} \text{ s.t. } y=1} \mathbb{I}_{\widehat{y}(\boldsymbol{x})=-1} = \mathbb{P}_n\left(\widehat{y}(\boldsymbol{x}) = -1 \mid y(\boldsymbol{x}) = 1\right)$$

and

$$\text{FPR} = \frac{1}{\#\{y=-1\}} \sum_{(\boldsymbol{x},y)\in\mathcal{D} \text{ s.t. } y=-1} \mathbb{I}_{\widehat{y}(\boldsymbol{x})=1} = \mathbb{P}_n\left(\widehat{y}(\boldsymbol{x}) = 1 \mid y(\boldsymbol{x}) = -1\right).$$

Table 8.3 we provide an overview of the most common reported ratios and their definitions.

Apart from the metrics shown in the table, another quantity is used to assess a classification model, called the *F1-score*. This is defined as

$$\frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}},$$

and it is useful when we want to evaluate the performance of the model on the *positive class*.

Indeed, to achieve a large F1-score, we need both precision and recall to be large. This means that, whenever the true label of a point $\boldsymbol{x}$ is +1, we will classify it correctly with large probability, and conversely, whenever a point is classified as -1, its true label is -1 with high probability.

Remark. The F1-score is defined in this way exactly because of its purpose to measure the performance on the positive class. For example, if we define the F1-score by the simplest formula

$$\frac{1}{2}\left(\text{precision} + \text{recall}\right),$$

then it is possible to achieve a large F1-score even if the model is not good on the positive class, e.g. if the precision is very large but the recall is almost zero.

	Want large		Want small
Precision	$\frac{\#\text{TP}}{\#\{\widehat{y}=+1\}} = \mathbb{P}_n(y=1 \mid \widehat{y}=1)$	FDR (1 - Precision)	$\frac{\#\text{FP}}{\#\{\widehat{y}=+1\}} = \mathbb{P}_n(y=-1 \mid \widehat{y}=+1)$
Recall (TPR, power)	$\frac{\#\text{TP}}{\#\{y=+1\}} = \mathbb{P}_n(\widehat{y}=1 \mid y=1)$	FPR (type I error)	$\frac{\#\text{FP}}{\#\{y=-1\}} = \mathbb{P}_n(\widehat{y}=+1 \mid y=-1)$

Table 8.3: This table summarizes the error metrics that are used in binary classification.

As we mentioned earlier, we now have to use the new error function

$$\frac{c_{\text{FN}}}{\#\{y=+1\}} \sum_{(x,y):y=1} \mathbb{I}_{\widehat{y}(x)=-1} + \frac{c_{\text{FP}}}{\#\{y=-1\}} \sum_{(x,y):y=-1} \mathbb{I}_{\widehat{y}(x)=+1}.$$

In general, the use of a different loss function can give rise to many difficulties, especially during optimization. Interestingly, instead of changing the loss function (and so the optimization problem too), we can make a small modification to the binary classification techniques that we saw in the previous sections to make them suitable in the asymmetric setting.

Recall that, if $\widehat{f} : \mathbb{R}^d \to \mathbb{R}$ is a binary classification model, we use the rule

$$\widehat{y}(x) = \operatorname{sign} \widehat{f}(x) = \begin{cases} +1 & \text{if } \widehat{f}(x) > 0 \\ -1 & \text{if } \widehat{f}(x) < 0 \end{cases}$$

to predict the class of a point x.

As we have mentioned, larger (positive) values of $\widehat{f}(x)$ show greater confidence that $y(x) = +1$, and analogously, smaller (negative) values show greater confidence that $y(x) = -1$. If one wants to eliminate false positives, their strategy should classify as positives only those points for which $\widehat{f}(x)$ is *very* large.

This way, the model will predict $\widehat{y}(x) = +1$ only when it is very confident that the true label is also $y(x) = +1$. Thus, the number of false positives will decrease. To incorporate this strategy into the above prediction rule, we introduce a new variable τ that denotes the threshold that $\widehat{f}(x)$ should exceed so that the point x is classified as positive. In other words, the prediction rule now becomes

$$\widehat{y}_\tau(x) = \operatorname{sign}\left(\widehat{f}(x) - \tau\right) = \begin{cases} +1 & \text{if } \widehat{f}(x) > \tau \\ -1 & \text{if } \widehat{f}(x) < \tau \end{cases}.$$

If type I errors are more undesirable than type II errors, then τ will be positive. Analogously, it will be negative if type II errors are more important, and zero if the two types of error have equal weight. This new prediction rule and its comparison with the original one are illustrated in Figure 8.19.

8.7.2 ROC curves

Oftentimes, for a particular application we don't know which FPR or TPR is desired. Instead, we want to the practitioner to be able to

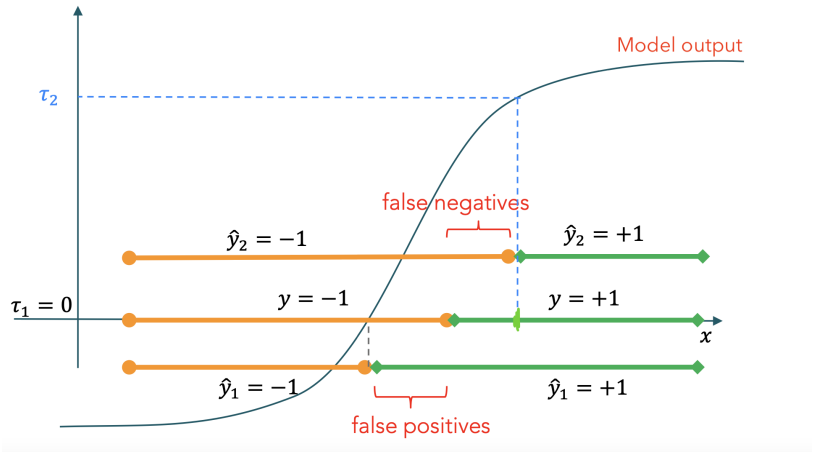


Figure 8.19: This plot illlustrates how increasing the threshold τ leads to more false negatives as in the top horizontal line. On the other hand, decreasing it leads to more false positives as we see in the lower horizontal line.

tweak a model according to their desired FPR or TPR. One tool that visualizes this performance and helps us compare different models for many values of τ simultaneously is the so-called *ROC curve* (ROC stands for *receiver operating characteristic*). This curve plots the *true positive rate* $\text{TPR} = \frac{\#\text{TP}}{\#\text{P}}$ against the false positive rate $\text{FPR} = \frac{\#\text{FP}}{\#\text{N}}$ for the different values of the threshold τ.

In Figure 8.20 we look at four different ROC curves. To understand what the ROC curve of a *good* model looks like, it might be useful to look at the mark of the *ideal* model at the top-left corner.

This is a model that classifies all points correctly, but of course, such a model is practically never achievable. Intuitively, a *good* model has a ROC curve that is *close* to the mark of the ideal model.

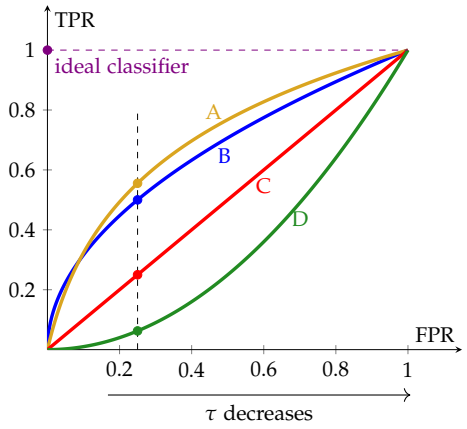


Figure 8.20: The ROC curves of four different classifiers. The dashed vertical line shows the TPR of each classifier when we require that $\text{FPR} \leq 0.25$.

The variable τ does not directly enter the diagram, but moving along each curve is equivalent to varying τ, so τ has an implicit influence on each curve. Given that each curve represents a classifier

$$\widehat{y}_\tau(\boldsymbol{x}) = \begin{cases} +1 & \text{if } \widehat{f}(\boldsymbol{x}) > \tau \\ -1 & \text{if } \widehat{f}(\boldsymbol{x}) < \tau, \end{cases}$$

it follows that τ decreases as we move along a ROC curve from the left to the right. Indeed, decreasing τ makes it easier to classify a point as $+1$, so the number of false positives, and hence the FPR, rises.

Let us now focus on the comparison of the four classifiers shown in Figure 8.20. Clearly, classifier D is the one with the lowest performance. Indeed, if we control the FPR at a given level, classifier D achieves the lowest TPR. Next comes classifier C. This one has equal TPR and FPR for any value of τ, so it is equivalent to random guessing.

Bonus Material

To see this equivalence, fix some threshold τ and denote by α the common value of TPR, FPR for that value of τ. For a given a point $x \in \mathcal{X}$, the probability of C classifying it as positive is

$$\mathbb{P}\left[\widehat{y} = 1\right] = \mathbb{P}\left[\widehat{y} = 1 | y = 1\right] \mathbb{P}\left[y = 1\right] + \mathbb{P}\left[\widehat{y} = 1 | y = -1\right] \mathbb{P}\left[y = -1\right].$$

For n large enough, the latter quantity is approximately equal to

$$\begin{aligned}\mathbb{P}_n\left[\widehat{y} = 1 | y = 1\right] \mathbb{P}\left[y = 1\right] + \mathbb{P}_n\left[\widehat{y} = 1 | y = -1\right] \mathbb{P}\left[y = -1\right] &= \text{TPR} \cdot \mathbb{P}\left[y = 1\right] + \text{FPR} \cdot \mathbb{P}\left[y = -1\right] \\ &= \alpha \left(\mathbb{P}\left[y = 1\right] + \mathbb{P}\left[y = -1\right]\right) \\ &= \alpha.\end{aligned}$$

In other words, the classifier does not make use of the information provided by x, but it classifies it as $+1$ uniformly at random, with probability α. Thus, C is indeed equivalent to random guessing.

Classifiers A and B are both better than random guessing. However, none of them dominates the other uniformly. For very small values of the FPR, classifier B is better since its curve is higher than that of A. For a larger FPR, classifier A outperforms B.

This example shows that depending on the value of τ, different classifiers might be preferable. However, using many different classifiers could be inconvenient, so we often want to keep only one of them and discard all the others. This creates the need for a one single measure of a classifier's performance, that will not depend on τ.

One such measure is the *area under the ROC curve (AUROC)*. From Figure 8.20, it becomes clear that this area is always between zero and one. The closer it is to one, the better the model. The AUROC is equal to 1 for the ideal classifier and equal to $1/2$ for random guessing. Note that in Figure 8.20, classifiers A and B might have the same AUROC. However, B performs better for a low FPR, whereas A outperforms it for a high FPR.

Finally, it is worth mentioning that there exists a different type of ROC curve that is sometimes used in practice, in which the precision is plotted against the recall of a model [8]. Two such curves are

[8] See Table 8.3 for the definitions of these quantities.

shown in Figure 8.21.

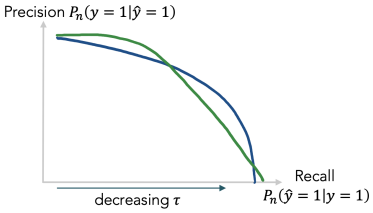


Figure 8.21: The recall-precision ROC curves of two models. The idea of choosing a model with large recall and precision simultaneously is the same as the one behind the use of the F1-score.

8.7.3 *Distribution shifts and robustness*

So far, both in regression and classification, we have made an important implicit assumption without which all our results become invalid. This assumption is that the model $\widehat{y}$ is used on data that come from the same distribution as the training and test sets. As Example 8.8 shows, this might not always be true.

Example 8.8. Suppose that an agricultural lab in Australia wants to select a model $M \in \{M_1, \dots, M_r\}$ that detects if a plant has a disease based on various measurements. In other words, the test is given a vector $\boldsymbol{x}$ of measurements taken from a plant, and it predicts whether this plant has the disease ($\widehat{y}(\boldsymbol{x}) = -1$) or not ($\widehat{y}(\boldsymbol{x}) = +1$).

The training and test datasets are large enough, and they mostly consist of plants located in the lab's establishment in Australia, as well as some other plants on different continents. The lab chooses the model $M = \widehat{y}$ with the lowest cross-validation error and releases it for mass use.

Some other labs in Asia and Europe test this model and discover that it performs very poorly despite the careful model selection process that it passed through.

The underlying reason for this inconsistency might simply be the fact that the data used by the Asian and European labs come from a different distribution than the ones for the selection of the model. More specifically, the climate, the atmospheric conditions, and the soil composition are different in Australia, Europe, and Asia, so the measurements taken from healthy and infected plants in different locations might not follow a consistent pattern.

Thus, although the model might be very accurate in plants located in Australia, it can still happen that it completely fails to classify plants on different continents correctly.

Let us now look at Figure 8.22. In this figure, the dataset consists of a majority and a minority group.[9] Points in the majority group g_1 are generated from a different distribution than points in the minority group 2. The average error of classifier ① is relatively low. More specifically, it classifies 23 out of the 29 points correctly, so the average error is $6/29 \approx 0.207$.

[9] In terms of Example 8.8, the majority group consists of the plants in Australia and the minority group of the rest of the plants used for model selection.

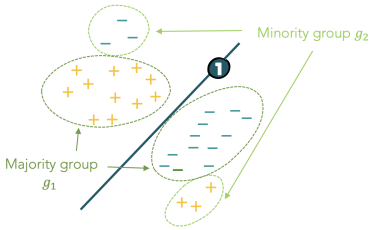


Figure 8.22: The dataset consists of a majority and a minority group. Classifier ① appears as a reasonable choice, but it is absolutely redundant if we are interested in the minority group.

However, if we later apply the model to make predictions for points coming from the same distribution as the ones in g_2, it will obviously give us very poor results.

To treat this problem, we can use the classifier with the optimal *worst-group accuracy*. Instead of

$$\frac{1}{|g_1 \cup g_2|} \sum_{(\boldsymbol{x},y)\in g_1\cup g_2} \mathbb{I}_{\widehat{y}(\boldsymbol{x})\neq y},$$

this classifier minimizes

$$\max\left\{ \frac{1}{|g_1|} \sum_{(\boldsymbol{x},y)\in g_1} \mathbb{I}_{\widehat{y}(\boldsymbol{x})\neq y}, \frac{1}{|g_2|} \sum_{(\boldsymbol{x},y)\in g_2} \mathbb{I}_{\widehat{y}(\boldsymbol{x})\neq y} \right\}.$$

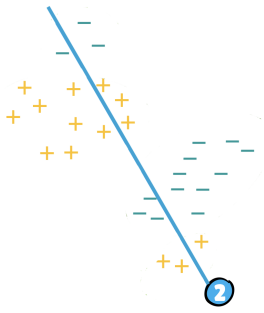


Figure 8.23: Although the optimal worst-group classifier might seem unintuitive at first sight, it achieves accuracy over 65% in both the minority and the majority groups.

For the particular dataset shown in Figure 8.22, this classifier would be the one shown in Figure 8.23.

8.7.4 Fairness

Closely related to worst-group accuracy, more recently, evaluation metrics of machine learning methods that take into account *fairness* have received more attention.

Example 8.9. Say we train a classifier $\hat{Y}$ that classifies whether a student X applying to university should be admitted. Ideally, we would like this classifier to be fair with respect to some attributes of X, for example, a binary attribute S whether they are Swiss or not. We can represent this through a binary variable $S \in \{-1, 1\}$. If we naively train $\hat{Y}$ on past admissions data, the predictor might learn an existing bias, for example, that a disproportionate amount of non-Swiss students are admitted solely based on their nationality. This classifier might achieve maximum accuracy when predicting whether a new student is admitted by the *existing* admissions system, but it is not fair if it should make the decision. Therefore, we would like to optimize accuracy only under the constraint of being fair, which needs to be formalized.

There are multiple notions of fairness, and usually, whether they accurately describe a natural understanding of fairness depends highly on the problem context. We will name a few common ones.

Equal Odds. We could claim that a classifier is fair, if all groups S (Swiss or non-Swiss in Example 8.9) have the same false-positive and false-negative rate, that is, for all $y, s, s' \in \{-1, 1\}$

$$\mathbb{P}(\hat{Y} = 1 | Y = y, S = s) = \mathbb{P}(\hat{Y} = 1 | Y = y, S = s').$$

Demographic Parity. We could also claim that a classifier is only fair, if the probability of getting a positive prediction (getting admitted to university in Example 8.9) does not vary between groups, that is, for all $s, s' \in \{-1, 1\}$

$$\mathbb{P}(\hat{Y} = 1 | S = s) = \mathbb{P}(\hat{Y} = 1 | S = s').$$

Equal Opportunity. In many settings, Demographic Parity and Equal Odds are viewed as too restrictive and unfair, because achieving high accuracy becomes impossible. This is known as the *accuracy-fairness tradeoff*. One less restrictive alternative fairness notion is to only demand that every group has the same opportunity to get a positive prediction, given that the ground truth is also positive. In mathematical terms, this reads as

$$\mathbb{P}(\hat{Y} = 1 | S = s, Y = 1) = \mathbb{P}(\hat{Y} = 1 | S = s', Y = 1).$$

There are many more definitions of fairness (e.g., Fairness Through Unawareness, Conditional Fairness, Worst-group Accuracy, Balanced Accuracy), and choosing which one to use is highly context-dependent. Therefore, we suggest that the reader

thinks about the following question: Which mathematical notion of fairness would be most reflective of what you might consider fairness in Example 8.9?

9
Kernels

Roadmap

In this chapter, we explore a special method for learning with non-linear features: kernels. In Section 9.1 we revisit featurization and and motivate the kernel trick, which is described in Subsection 9.1.2. In Section 9.2 we provide some example Kernels that are often used practice, as described in Section 9.3.

Learning Objectives

After reading this chapter you should:

- Know what the kernel trick refers to and state its main idea.
- Understand the main mathematical principle that justifies the reparametrization used in kernelization.
- Know how gradient descent relates to this justification.
- Be able to characterize valid kernels and how they can be constructed from other kernels.
- Know example kernels and their main characteristics.
- Understand how kernels are used in practice for regression and classification.

9.1 The kernel trick

Recall the objective of linear regression or classification, where we aim to fit a linear function $f : \mathbb{R}^d \to \mathbb{R}$ with $f(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$ to a set of datapoints $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ with $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ or $y_i \in \{-1, 1\}$, respectively. We discussed the approach of minimizing the mean of some loss function ℓ across all data points, that is, to solve

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle).$$

To extend this framework, we also explored the concept of featurization, e.g., in Section 4.3, where inputs are first transformed into a feature space using a feature map $\phi : \mathbb{R}^d \to \mathbb{R}^p$. The function f

then takes the form $f(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle$, with the weights now being p-dimensional and the modified objective is

$$\min_{\boldsymbol{w} \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle). \tag{9.1}$$

This approach is very powerful, but if applied naïvely can be computationally infeasible: If computing or storing $\phi(\boldsymbol{x})$ is expensive, for example when p is very large (or even infinite), or when searching for a minimizer $\widehat{\boldsymbol{w}}$ in $\mathbb{R}^p$ is computationally hard, this method quickly renders intractable.

Example 9.1. In order to represent a polynomial of degree m of a $d-$dimensional input, we require $p = \mathcal{O}(d^m)$ features, and hence, when storing the representations of n data points, we require memory of the order of $\mathcal{O}(nd^m)$. Even for moderately large m and d, this order is extremely poor.

Bonus Material

We will demonstrate a simple trick for counting polynomial features of degree m, to derive the stated order $p = \mathcal{O}(d^m)$. The argument is known as a *stars and bars* argument: There is a bijection between m-th order monomials and the set of binary strings of length $d+m$ with exactly d zeros. Since the number of such strings is $\binom{d+m}{d} = \binom{d+m}{m}$, we know that the number of m-th order monomials is also $\binom{d+m}{d} = \mathcal{O}(d^m)$. It remains to construct the bijection.

Note that we can express every m-th order monomial as $1^{\alpha_0} x_1^{\alpha_1} \cdot \ldots \cdot x_d^{\alpha_d}$, where $\alpha_i \in \mathbb{N}$ for all i and $\sum_{i=0}^{d} \alpha_i = m$. The corresponding binary string is constructed by starting with the empty string s, and then following this algorithm: For $l = 0, \ldots, d$:

1. Append $s \leftarrow (s, \underbrace{1, \ldots, 1}_{\alpha_l - \text{times}})$. Note that if $\alpha_l = 0$, we append nothing, $s \leftarrow s$.
2. If $l < d$, append $s \leftarrow (s, 0)$.

What we end up with is a string of length $d+m$ that has exactly d zeros, m ones and at most d chunks of ones. Each zero is a deliminator between each chunk of ones—including "empty chunks". The length of the i-th chunk corresponds to the power of x_i—the "empty chunk" corresponds to raising to the power of zero.

For example, if we take $d = 5$, $m = 7$ and the monomial $x_1^2 x_2 x_4^3 = 1^1 x_1^2 x_2^1 x_3^0 x_4^3 x_5^0$, we get the encoding 101101001110. Or, if we have the monomial 110100110011, it corresponds to the monomial $1^2 x_1^1 x_2^0 x_3^2 x_4^0 x_5^2 = x_1 x_3^2 x_5^2$.

It is easy to see that this correspondence between binary strings and monomials is a bijection.

This chapter discusses how the kernel trick provides a way to handle general nonlinear features while keeping computational requirements at bay.

9.1.1 Kernelization

We will start by presenting a three-step instruction detailing how to execute *kernelization* before formally justifying it in Subsection 9.1.3.

Reparametrization. We start by reparameterizing the weights $\boldsymbol{w}$, by assuming they take the form $\boldsymbol{w} = \Phi^\top \boldsymbol{\alpha}$ with $\boldsymbol{\alpha} \in \mathbb{R}^n$ and Φ being

the feature matrix as usual, that is,

$$\Phi = \begin{pmatrix} -\phi(\boldsymbol{x}_1)^\top - \\ \vdots \\ -\phi(\boldsymbol{x}_n)^\top - \end{pmatrix} \in \mathbb{R}^{n \times p}.$$

It is noteworthy that this reparametrization constrains $\boldsymbol{w}$ to be within the subspace $\operatorname{span}(\Phi^\top) \subset \mathbb{R}^p$. As we will discuss in Subsection 9.1.3, we can do this without "losing anything" because at least one minimizer of Equation 9.1 is in that space.

Inner product formulation of the loss. Plugging this parameterization into $f(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle$ we can see how $f(\boldsymbol{x})$ relies on $\boldsymbol{x}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ solely through the *inner products of the features* since

$$f(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle = \left\langle \Phi^\top \boldsymbol{\alpha}, \phi(\boldsymbol{x}) \right\rangle = \sum_{i=1}^{n} \alpha_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}) \rangle .$$

Specifically, within the objective of Equation 9.1, the covariates $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ only appear within the inner products $\langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle$ for $i, j = 1, \ldots, n$.

Replacing inner products. We can then define a bivariate function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ with values $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle$. Further, we define a matrix $\boldsymbol{K} \in \mathbb{R}^{n \times n}$ which has as its components $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. This yields a reformulation of the objective as

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{n} \sum_{i=1}^{n} \ell \left(y_i, \sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \right) = \min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, (\boldsymbol{K}\boldsymbol{\alpha})_i). \tag{9.2}$$

Comparing Equation 9.1 and Equation 9.2, instead of storing $\phi(\boldsymbol{x}_i) \in \mathbb{R}^p$ for $i = 1, \ldots, n$ we now only need to store $k(\boldsymbol{x}_i, \boldsymbol{x}_j) \in \mathbb{R}$ for $i, j = 1, \ldots, n$, thereby diminishing the memory requirements from $\mathcal{O}(np)$ to $\mathcal{O}(n^2)$. That is much better when $p \gg n$. However, when computing $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ naïvely by first computing $\phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j)$ and then taking the inner product separately, the computation time did not change—we still need to perform $\mathcal{O}(n^2 p)$ operations.

9.1.2 *The kernel trick and valid kernel functions*

The "beauty" of the so-called *kernel trick* is to circumvent ϕ altogether, and instead come up with functions k that are easy to compute and only implicitly correspond to some feature mapping. Specifically, we can restart from the beginning with the linear model $f(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$ and perform the reparametrization $\boldsymbol{w} = \boldsymbol{X}^\top \boldsymbol{\alpha}$. Again, this leads to the model $f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i \langle \boldsymbol{x}_i, \boldsymbol{x} \rangle$. The shortcut is to directly replace all inner products with some function k, known as the *kernel function*, that are both easy to evaluate and can "guarantee" existence of a feature map ϕ. The model changes to $f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$. Given the implicit nature of ϕ in this formulation, even the representation of inner products of an infinite number of features, i.e., "$p = \infty$", becomes feasible. We might not

know what exactly ϕ is, nor how to compute it, but we know that such a ϕ exists.

Next, we discuss for which kernel functions k this is true and then provide some examples that are easy to compute.

It turns out that some $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ implicitly represents a feature mapping ϕ, if and only if k is symmetric and positive semi-definite, as formalized in the next definition:

Definition 9.2 (Kernels)**.** A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is called a *kernel* if

1. k is *symmetric*: For all $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d$ it holds $k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$.
2. k is *positive semi-definite*: For all $n \in \mathbb{N}$ and all $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n \in \mathbb{R}^d$ the kernel matrix

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_n, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{pmatrix}$$

   is positive semidefinite.

This definition is motivated by the following Theorem 9.3.

Bonus Material

Theorem 9.3. *Let $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be any function. There exists a Hilbert space $(\mathcal{H}, \langle \cdot, \cdot \rangle_{\mathcal{H}})$ and a function $\phi : \mathbb{R}^d \to \mathcal{H}$ such that $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle_{\mathcal{H}}$ for all $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d$ if and only if k is a kernel.*

The proof is somewhat involved, so we do not show it here. However, the interested reader is referred to Theorem 4.16 in Steinwart and Christmann 2008.

For our purposes, you can think of a Hilbert space $\mathcal{H}$ as simply being $\mathbb{R}^p$ with $p \in \mathbb{N}$, or $\mathcal{H}$ being $\ell^2 \subset \mathbb{R}^{\mathbb{N}}$, the space of all sequences $\mathbf{h} = (h_i)_{i \in \mathbb{N}}$ with real values $h_i \in \mathbb{R}$ whose squared sum is finite, $\sum_{i=1}^{\infty} h_i^2 < \infty$. In the first case, the inner product is just the standard Euclidean inner product, and in the second case, we have $\langle h, h' \rangle_{\ell^2} = \sum_{i=1}^{\infty} h_i h_i'$.

Some useful properties of kernels follow directly from this definition, together with Theorem 9.3.

1. Feature maps can be *composed*. Given two maps $\phi : \mathbb{R}^d \to \mathbb{R}^p$ and $\psi : \mathbb{R}^p \to \mathbb{R}^q$, these two maps can be composed as $\psi \circ \phi$ to induce a valid kernel given by $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \psi(\phi(\boldsymbol{x})), \psi(\phi(\boldsymbol{x}')) \rangle$.
2. Kernels can be *added* in two ways. Let $k_1 : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and $k_2 : \mathbb{R}^{d'} \times \mathbb{R}^{d'} \to \mathbb{R}$ be kernels. Then $k : \mathbb{R}^{d+d'} \times \mathbb{R}^{d+d'} \to \mathbb{R}$ with

$$k((\boldsymbol{x}, \boldsymbol{y}), (\boldsymbol{x}', \boldsymbol{y}'))) = k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{y}, \boldsymbol{y}')$$

   is a valid kernel, and if $d = d'$, $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ with

$$k(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{x}, \boldsymbol{x}')$$

   is also a valid kernel.

3. Analogously, kernels can also be *multiplied* in two ways. Let $k_1 : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and $k_2 : \mathbb{R}^{d'} \times \mathbb{R}^{d'} \to \mathbb{R}$ be kernels. Then $k : \mathbb{R}^{d+d'} \times \mathbb{R}^{d+d'} \to \mathbb{R}$ with

$$k((\boldsymbol{x}, \boldsymbol{y}), (\boldsymbol{x}', \boldsymbol{y}'))) = k_1(\boldsymbol{x}, \boldsymbol{x}')k_2(\boldsymbol{y}, \boldsymbol{y}')$$

is a valid kernel, and if $d = d'$, $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ with

$$k(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}')k_2(\boldsymbol{x}, \boldsymbol{x}')$$

is also a valid kernel.

So, beyond memory savings from $\mathcal{O}(np)$ to $\mathcal{O}(n^2)$, *how much does the computational complexity change using kernels?* Generally, computing $k(\boldsymbol{x}, \boldsymbol{x}')$ directly can be significantly more efficient that computing $\phi(\boldsymbol{x}), \phi(\boldsymbol{x}')$ first and then computing the inner product $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle$ separately. However, the exact savings depend on the chosen kernel.

Example 9.4. In the example of polynomials (Example 9.1), we can see a significant computational gain when picking the right k (analogous to picking the "right ϕ"). If we use vanilla polynomial features of degree m on d-dimensional inputs, the computation time is $\mathcal{O}(n^2 d^m)$, since we have to compute n^2 inner products of $p = \binom{d+m}{m} = \mathcal{O}(d^m)$ features. Instead, we can pick the kernel function $k(\boldsymbol{x}, \boldsymbol{x}') = (1 + \langle \boldsymbol{x}, \boldsymbol{x}' \rangle)^m$, which corresponds to polynomial features of degree m up to some constant coefficients. We only need $\mathcal{O}(n^2(d+m))$ computations to evaluate $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for $i, j = 1, \ldots, n$, since we have to compute an inner product of d features, and then perform m multiplications, for all n^2 pairs of samples.[1] This presents a significant speed-up over the naïve approach. For instance, when $n = 5$, $d = 5$ and we take polynomial features of degree $m = 4$, the naïve approach would perform $n^2 p = 25 \times \binom{5+4}{4} = 31500$ computations. Using k, we only need $n^2(d+m) = 25 \times (5+4) = 225$ computations.

[1] Exploiting the symmetry of k one can do better, but the order $\mathcal{O}(n^2(d+m))$ remains the same.

9.1.3 *Validity of the Reparameterization*

In Subsection 9.1.2, in the first step of kernelization, we chose to reparameterize $\boldsymbol{w} = \Phi^\top \boldsymbol{\alpha}$. This reparameterization allowed us to solve the n-dimensional optimization problem

$$\widehat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \boldsymbol{\alpha}^\top \Phi \phi(\boldsymbol{x}_i)) \tag{9.3}$$

instead of the original p-dimensional problem

$$\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w} \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \boldsymbol{w}^\top \phi(\boldsymbol{x}_i)). \tag{9.4}$$

However, the question remains:

Why is this a valid parameterization?

After all, Equation 9.4 searches over a larger space of possible optimizers $\boldsymbol{w} \in \mathbb{R}^p$ compared to Equation 9.3, which only searches over the smaller space $\tilde{\boldsymbol{w}} = \Phi^\top \boldsymbol{\alpha}$, $\boldsymbol{\alpha} \in \mathbb{R}^n$.

To answer this question, we will show that (perhaps surprisingly) at least one of the minimizers of Equation 9.4 can be expressed as

a linear combination of the featurized datapoints $\{\phi(\boldsymbol{x}_1), ..., \phi(\boldsymbol{x}_n)\}$, that is,

$$\widehat{\boldsymbol{w}} = \sum_{i=1}^{n} \hat{\alpha}_i \phi(\boldsymbol{x}_i).$$

Thus, searching over the smaller space $S := \{\sum_{i=1}^{n} \alpha_i \phi(\boldsymbol{x}_i) : \boldsymbol{\alpha} \in \mathbb{R}^n\} \subset \mathbb{R}^p$ is indeed sufficient!

To gain some intuition, we start with the following Theorem:

Theorem 9.5. *Suppose that we perform gradient descent with an arbitrary step-size on the training loss $L(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, \boldsymbol{w}^\top \phi(\boldsymbol{x}_i))$ initialized at $\boldsymbol{w}^0 = \boldsymbol{0}$, where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a continuously differentiable function. Then all iterates $\boldsymbol{w}^t$ can be expressed as*

$$\boldsymbol{w}^t = \sum_{i=1}^{n} \alpha_i^t \phi(\boldsymbol{x}_i)$$

for some $\boldsymbol{\alpha}^t \in \mathbb{R}^n$.

Proof. Let us write down the gradient step for the training loss. For any $t \in \mathbb{N}$, it holds that

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta \left(\frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{w}} \ell(y_i, \langle \boldsymbol{w}^t, \phi(\boldsymbol{x}_i) \rangle) \right).$$

Let us denote $u := \langle \boldsymbol{w}^t, \phi(\boldsymbol{x}_i) \rangle$. Then, according to the chain rule, it holds that

$$\nabla_{\boldsymbol{w}} \ell(y_i, \langle \boldsymbol{w}^t, \phi(\boldsymbol{x}_i) \rangle) = \frac{\partial}{\partial u} \ell(y_i, u) \phi(\boldsymbol{x}_i).$$

Thus we obtain for the gradient step:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \left(\frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial u} \ell(y_i, u) \phi(\boldsymbol{x}_i) \right).$$

The partial derivative $\frac{\partial}{\partial u} \ell(y_i, u)$ is a scalar dependent on y_i and $u = \langle \boldsymbol{w}^t, \phi(\boldsymbol{x}_i) \rangle$. This implies that the vector $-\eta \left(\frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial u} \ell(y_i, u) \phi(\boldsymbol{x}_i) \right)$ is contained in $S = \{\sum_{i=1}^{n} \alpha_i \phi(\boldsymbol{x}_i) : \boldsymbol{\alpha} \in \mathbb{R}^n\}$. Therefore, whenever $\boldsymbol{w}^t \in S$, then also $\boldsymbol{w}^{t+1} \in S$. Since we have assumed that $\boldsymbol{w}^0 = \boldsymbol{0} \in S$, the claim follows by induction. □

In Theorem 5.18, we have shown that for a strongly convex smooth function L, gradient descent with an appropriate step size converges to a global minimizer. Let $\widehat{\boldsymbol{w}}$ be the limit of the gradient descent sequence $\{\boldsymbol{w}^t, t \in \mathbb{N}\}$ that we have considered in Theorem 9.5 (and assume for now that this limit exists). Since every element of this sequence is contained in S and S is a closed subset of $\mathbb{R}^p$, it follows that the limit of the sequence is also contained in S. In other words, there exists $\widehat{\boldsymbol{\alpha}} \in \mathbb{R}^n$ for which

$$\widehat{\boldsymbol{w}} = \Phi^\top \widehat{\boldsymbol{\alpha}}.$$

Thus, if the loss satisfies certain conditions, it follows from Theorem 9.5 that there exists a global minimizer $\widehat{\boldsymbol{w}}$ which is fully contained in S.

Instead of verifying these conditions (which hold for many losses, such as the squared loss), we will illustrate a different perspective on why the kernel trick works: we will show that the training loss does not depend on the part of w which is not in S. We will conclude that there must exist a global minimizer $\hat{w}$ of Equation 9.4 fully contained in S.

Theorem 9.6 (Sufficiency of reparameterization)**.** *Among the global minimizers of Equation 9.4 there is at least one that has the form*

$$\hat{w} = \Phi^\top \hat{\alpha},$$

where $\hat{\alpha} \in \mathbb{R}^n$ is a global minimizer of the kernelized optimization problem Equation 9.3.

Note that we only consider the case where the training loss has no regularization penalty ($\lambda = 0$), however, the Theorem also extends to this general case (see bonus section).

Proof. Recall that the training loss $L(w) = \frac{1}{n}\sum_{i=1}^n \ell(y_i, w^\top \phi(x_i))$ only depends on w through $w^\top \phi(x_i)$. Any vector $w \in \mathbb{R}^p$ can be decomposed into its orthogonal projection onto the space $S = \{\sum_{i=1}^n \alpha_i \phi(x_i) : \alpha \in \mathbb{R}^n\}$ and its projection onto $S^\perp$, the orthogonal complement of S:

$$w = \Pi_S w + \Pi_{S^\perp} w.$$

In particular, if we take any global minimizer $\hat{w}$ of the training loss, we have

$$\begin{aligned} L(\hat{w}) &= \frac{1}{n}\sum_{i=1}^n \ell(y_i, (\Pi_S \hat{w} + \Pi_{S^\perp}\hat{w})^\top \phi(x_i)) \\ &= \frac{1}{n}\sum_{i=1}^n \ell(y_i, (\Pi_S \hat{w})^\top \phi(x_i)) \\ &= L(\Pi_S \hat{w}), \end{aligned}$$

where we have used that $\Pi_{S^\top}\hat{w}$ is orthogonal to all $\phi(x_i)$. Thus, it follows that $\Pi_S \hat{w}$ is also a global minimizer. Since $\Pi_S \hat{w} \in S$, it has the form $\Pi_S \hat{w} = \Phi^\top \hat{\alpha}$.

More generally, since the loss does not depend on $\Pi_{S^\perp} w$ for any w, it holds that

$$\begin{aligned} \min_{w \in \mathbb{R}^p} \frac{1}{n}\sum_{i=1}^n \ell(y_i, w^\top \phi(x_i)) &= \min_{w \in S} \frac{1}{n}\sum_{i=1}^n \ell(y_i, w^\top \phi(x_i)) \\ &= \min_{\alpha \in \mathbb{R}^n} \frac{1}{n}\sum_{i=1}^n \ell(y_i, \alpha^\top \Phi \phi(x_i)). \end{aligned}$$

This concludes the proof. □

Thus, we have shown that to find a global minimizer of the original optimization problem Equation 9.4, it is sufficient to search in the smaller subspace $S = \{\sum_{i=1}^n \alpha_i \phi(x_i) : \alpha \in \mathbb{R}^n\}$ instead of $\mathbb{R}^p$. In other words, the kernel trick reparameterization is valid.

Bonus Material

Our Theorem 9.6 admits a more general formulation, called the *Representer Theorem*. The following Theorem shows that for any empirical loss that only depends on $\boldsymbol{w}$ via its scalar product with the datapoints $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$ and its norm $\|\boldsymbol{w}\|$, a minimizer of the empirical loss is contained in the span $\mathcal{X} = \text{span}(\boldsymbol{x}_1, ..., \boldsymbol{x}_n)$ of the (featurized) datapoints. Thus, the kernel trick "works" for a very general class of loss functions.

Theorem 9.7 (Representer Theorem)**.** *Let $\mathcal{H}$ be a Hilbert space, $g : \mathbb{R} \to \mathbb{R}$ a non-decreasing function, $f : \mathbb{R}^n \to \mathbb{R}$. Let $\boldsymbol{x}_1, ..., \boldsymbol{x}_n \in \mathcal{H}$ be the (featurized) datapoints. Define $\mathcal{H}_x = \text{span}(\boldsymbol{x}_1, ..., \boldsymbol{x}_n)$. Let the empirical loss function $F : \mathcal{H} \to \mathbb{R}$ be given by $F(\boldsymbol{w}) = g(\|\boldsymbol{w}\|_{\mathcal{H}}) + f(\langle \boldsymbol{w}, \boldsymbol{x}_1 \rangle_{\mathcal{H}}, ..., \langle \boldsymbol{w}, \boldsymbol{x}_n \rangle_{\mathcal{H}})$. Then it holds that*

$$\inf_{\boldsymbol{w} \in \mathcal{H}} F(\boldsymbol{w}) = \inf_{\boldsymbol{w} \in \mathcal{H}_x} F(\boldsymbol{w}).$$

In other words, if the training loss can be expressed in this form—where $g(\|\boldsymbol{w}\|_{\mathcal{H}})$ could for example be the ridge penalty and f from kernel regression—there is a minimizer of the training loss which lies in the span of the (featurized) datapoints $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$ and can be thus expressed as their linear combination.

Proof. We decompose the Hilbert space $\mathcal{H}$ as the direct sum of the span of the datapoints and its orthogonal complement:

$$\mathcal{H} = \mathcal{H}_x \oplus \mathcal{H}_x^{\perp}.$$

Then we can write any $\boldsymbol{w} \in \mathcal{H}$ as $\boldsymbol{w} = \boldsymbol{w}_x + \boldsymbol{v}$, where $\boldsymbol{w}_x \in \mathcal{H}_x$ and $\boldsymbol{v} \in \mathcal{H}_x^{\perp}$. It holds that

$$f(\langle \boldsymbol{w}, \boldsymbol{x}_1 \rangle_{\mathcal{H}}, ..., \langle \boldsymbol{w}, \boldsymbol{x}_n \rangle_{\mathcal{H}}) = f(\langle \boldsymbol{w}_x, \boldsymbol{x}_1 \rangle_{\mathcal{H}}, ..., \langle \boldsymbol{w}_x, \boldsymbol{x}_n \rangle_{\mathcal{H}}),$$

and from Pythagoras we obtain

$$g(\|\boldsymbol{w}\|_{\mathcal{H}}) = g(\|\boldsymbol{w}_x + \boldsymbol{v}\|_{\mathcal{H}}) = g(\sqrt{\|\boldsymbol{w}_x\|_{\mathcal{H}}^2 + \|\boldsymbol{v}\|_{\mathcal{H}}^2}) \geq g(\|\boldsymbol{w}_x\|_{\mathcal{H}}), \tag{9.5}$$

and the claim follows. □

In Equation 9.5, strict inequality holds if g is strictly increasing. Hence, for strictly increasing regularizers g, we can even conclude that any minimizer of the empirical loss has to be in the span of the training datapoints.

9.2 Examples of kernels

We will now discuss some of the most prominent kernels used in machine learning.

9.2.1 Inner product kernels

One family of kernels is commonly referred to as *inner product* kernels, where the inner product is not to be confused with the inner product of $\phi(\boldsymbol{x}), \phi(\boldsymbol{x}')$. Consider a function $g : \mathbb{R} \to \mathbb{R}$ with exclusively non-negative coefficients within its Taylor series. Then, the kernel function $k(\boldsymbol{x}, \boldsymbol{x}') = g(\langle \boldsymbol{x}, \boldsymbol{x}' \rangle)$ is a valid kernel. You may take it as an exercise to prove this.

Exercise 9.8. Prove the above claim that for any $g : \mathbb{R} \to \mathbb{R}$ that has exclu-

sively non-negative coefficients within its Taylor series, $k(\boldsymbol{x}, \boldsymbol{x}') = g(\langle \boldsymbol{x}, \boldsymbol{x}' \rangle)$ is a valid kernel.

For example, taking $g(x) = (1+x)^m$ with $m \in \mathbb{N}$, we can see that the Taylor series

$$g(x) = \sum_{k=0}^{\infty} \frac{1}{k!} \frac{d^k g}{(dx)^k}(x) = \sum_{k=0}^{m} \binom{m}{k} x^k$$

only has non-negative coefficients. The corresponding kernel is known as a *polynomial kernel*, cf. Example 9.4. It corresponds to the feature map ϕ that contains all polynomial features up to degree m. So, for example, when $d = 2$ and $k(\boldsymbol{x}, \boldsymbol{x}') = (1 + \langle \boldsymbol{x}, \boldsymbol{x}' \rangle)^2$, the feature map such that $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle$ is

$$\phi(\boldsymbol{x}) = \left(1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2, x_1^2, x_2^2\right)^\top \in \mathbb{R}^6.$$

9.2.2 RBF kernels

Another class of kernels, called *radial basis function (RBF) kernels*, can be represented as functions of the distance between two points, denoted as $k(\boldsymbol{x}, \boldsymbol{x}') = g(\|\boldsymbol{x} - \boldsymbol{x}'\|)$, where $\|\cdot\|$ is an arbitrary norm. Within this category, a frequently encountered example is the α-exponential kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_p^\alpha}{\tau}\right),$$

where τ denotes a bandwidth parameter.

Specifically, when $\alpha = 2$ and $p = 2$, the kernel is commonly referred to as *Gaussian*, while for $\alpha = 2$ and $p = 1$, it is termed *Laplacian*. Note that sometimes the term "Gaussian kernel" is used synonymously with "RBF", and the Laplacian kernel sometimes is defined with $\alpha = 1$ instead of $\alpha = 2$ (e.g., within the context of libraries such as `scikit-learn`).

Figure 9.1 shows instances of the Gaussian and Laplacian kernel in the setting where $d = 1$ and we vary the first argument of k for two bandwidths each. We can see the influence of the bandwidth on the kernel: A larger value of τ renders $k(x, z)$ increasingly "flat". Interpreting the kernel as a measure of *similarity*, a larger τ indicates greater similarity or correlation between points equidistant from each other. The distinction between Laplacian and Gaussian kernels lies in their respective shapes.

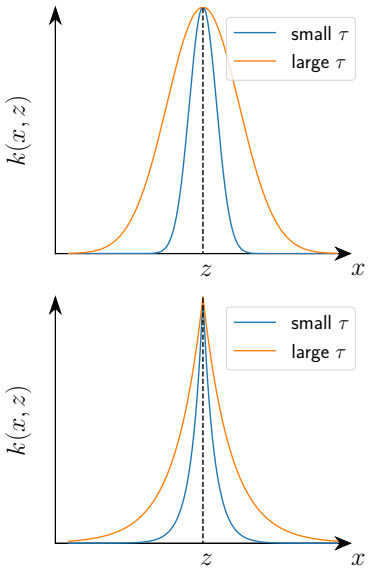


Figure 9.1: Gaussian kernel (first picture) and Laplacian kernel (second picture) with small (blue) and large (orange) bandwidth values.

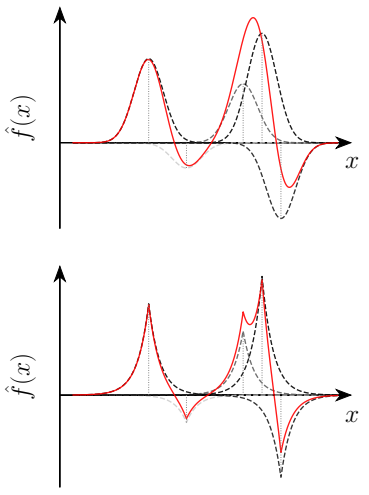


Figure 9.2: Kernel functions $f(x)$ (red line) for Gaussian (first plot) and Laplacian (second plot) kernels with same data points and alphas

Recall that a function in the kernelized function space will take the form

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}).$$

where $\boldsymbol{x}_i$ are given data points and α_i are weights from the reparametrization. Figure 9.2 shows two examples of this function when $d = 1$ and we have 5 datapoints (at the vertical dotted lines). The gray

dashed lines represent individual kernel functions for the provided data points, which are then aggregated with weights α_i to yield $f(x)$ (depicted by the red line). We can clearly see the effect that choosing different kernels has on the shape of the function f.

So what feature map does an RBF kernel correspond to? As alluded to previously, one strength of kernelization in general is that we can even have infinite-dimensional representations of our data. RBF kernels are an example of this: The corresponding feature map's co-domain is infinite-dimensional. This means that it maps to an infinite sequence of features, which we would not be able to compute explicitly. As an example, we provide a formal statement of this for the Gaussian kernel and $d = 1$ in the following bonus material.

Bonus Material

Theorem 9.9. *Let k be the Gaussian kernel for $d = 1$. Then, for $\phi : \mathbb{R} \to \ell^2 \subset \mathbb{R}^{\mathbb{N}}$ with*

$$\phi(x) = \left(\exp\left(-x^2/\tau\right) \sqrt{\frac{(2/\tau)^k}{k!}} x^k \right)_{k \in \mathbb{N}}$$

it holds $k(x, x') = \langle \phi(x), \phi(x') \rangle_{\ell^2}$.

Proof. The proof is a straight-forward consequence from the series expansion of the exponential function. We can rewrite the kernel as

$$\begin{aligned}
k(x, x') &= \exp\left(-(x - x')^2/\tau\right) \\
&= \exp\left(-x^2/\tau\right) \exp\left(-x'^2/\tau\right) \exp\left(2xx'/\tau\right) \\
&= \exp\left(-x^2/\tau\right) \exp\left(-x'^2/\tau\right) \sum_{k=0}^{\infty} \frac{(2xx'/\tau)^k}{k!} \\
&= \sum_{k=0}^{\infty} \left(\exp\left(-x^2/\tau\right) \sqrt{\frac{(2/\tau)^k}{k!}} x^k \right) \left(\exp\left(-x'^2/\tau\right) \sqrt{\frac{(2/\tau)^k}{k!}} x'^k \right) \\
&= \langle \phi(x), \phi(x') \rangle_{\ell^2}
\end{aligned}$$

where the last line holds by the definition of the inner product on ℓ^2. Note also that the calculation above automatically implies that $\text{range}(\phi) \subset \ell^2$, which justifies the last equality. This concludes the proof. □

9.3 Using the kernel trick in practice

We will now discuss how to use kernels in machine learning practice when performing regression or classification.

9.3.1 Kernelized vs. Linear Least Squares Regression

As previously discussed, the main trick is to never explicitly compute the feature transformation and instead directly substitute all inner products with kernels.

Assume that we use squared loss $\ell(y, \hat{y}) = (y - \hat{y})^2$. Then the objective from Equation 9.2 becomes

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{n} \|\boldsymbol{y} - \boldsymbol{K}\boldsymbol{\alpha}\|_2^2$$

where we, again, used the kernel matrix

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_n, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{pmatrix}.$$

We will call a solution of this optimization problem $\hat{\boldsymbol{\alpha}}$ and the resulting function $\hat{f}(\boldsymbol{x}) = \sum_{i=1}^n \hat{\alpha}_i k(\boldsymbol{x}_i, \boldsymbol{x})$.

This objective resembles the one we used in the linear regression setting discussed in Chapter 4, given by $\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$. Indeed, one can view conventional linear regression as a special case of kernel regression: If we take k to be the linear kernel, $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle$, and we replace $\boldsymbol{w}$ by $\boldsymbol{X}^\top \boldsymbol{\alpha}$ (as justified in Subsection 9.1.3), the modified objective becomes $\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \frac{1}{n} \left\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{\alpha}\right\|_2^2$. Notice that

$$\boldsymbol{X}\boldsymbol{X}^\top = \begin{pmatrix} \langle \boldsymbol{x}_1, \boldsymbol{x}_1 \rangle & \dots & \langle \boldsymbol{x}_1, \boldsymbol{x}_n \rangle \\ \vdots & \ddots & \vdots \\ \langle \boldsymbol{x}_n, \boldsymbol{x}_1 \rangle & \dots & \langle \boldsymbol{x}_n, \boldsymbol{x}_n \rangle \end{pmatrix},$$

so that $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^\top$.

However, two main distinctions between linear and kernelized regression remain: Firstly, we search $\hat{\boldsymbol{\alpha}}$ in $\mathbb{R}^n$ instead of $\mathbb{R}^d$—this is good in high-dimensional settings, but computationally hard for very large sample sizes. Secondly, although the optimization objectives have similar form, the resulting functions are vastly different, and can be highly non-linear in general.

Since the objectives are of the same structural form, we can apply the arsenal of optimization methods discussed in Chapter 5 as well as the regularization techniques discussed in Chapter 7, such as ridge regression.

9.3.2 *Kernel Ridge Regression*

Just like linear and featurized regression, kernelized regression can be affected by overfitting. So, the natural question arises: Can we add regularization to the kernelized version of the regression?

Recall that ridge regression in the linear model uses the objective $\frac{1}{n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2$ for some $\lambda \geq 0$. Again, after reparametrizing $\boldsymbol{w}$ with $\boldsymbol{X}^\top \boldsymbol{\alpha}$, we can equivalently write this objective as

$$\frac{1}{n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2 = \frac{1}{n} \left\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{\alpha}\right\|_2^2 + \lambda \boldsymbol{\alpha}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{\alpha}.$$

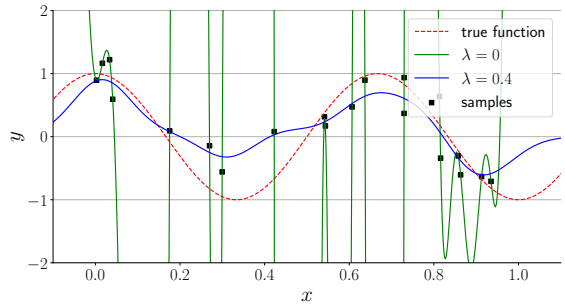
Subsequently, as discussed before, transitioning from inner products to general kernels involves substituting $\boldsymbol{X}\boldsymbol{X}^\top$ with the kernel

matrix $\boldsymbol{K}$, yielding the modified objective:

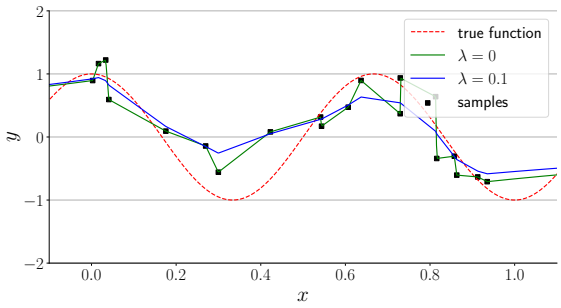
$$\frac{1}{n}\|\boldsymbol{y}-\boldsymbol{K}\boldsymbol{\alpha}\|_2^2+\lambda\boldsymbol{\alpha}^\top\boldsymbol{K}\boldsymbol{\alpha}.$$

This is the objective of the *kernel ridge regression*.

Figure 9.3 shows that this regularization technique has an effect on the fitted function $\widehat{f}$ that is similar to the effect in featurized regression in Chapter 7. As we can see, the regularization indeed helps recover the true function. Note that the function fitted with an unregularized Laplacian kernel (Figure 9.3(b)) looks very different to the one fitted with an unregularized Gaussian kernels (Figure 9.3(a)).



(a) Gaussian kernel.



(b) Laplacian kernel.

Figure 9.3: Kernelized ridge regression where the ground-truth is the cosine functions. We observe 20 samples and λ is the ridge regularization strength. The red dotted line is the true function, the green is unregularized kernel regression and blue is the kernel regression with a ridge penalty.
(a): We fit the data with the Gaussian kernel where we choose $\tau = 0.01$.
(b): We fit the data with the Laplacian kernel with $\tau = 1$.

9.3.3 *Classification with kernels*

Similarly, in the classification setting where we assign labels via $\hat{y} = \operatorname{sign}\widehat{f}(\boldsymbol{x})$, we can use the logistic loss from Equation 8.2, which is

$$\ell(y_i,\widehat{f}(\boldsymbol{x}_i)) = \log\left(1+\exp(-y_i\langle\boldsymbol{w},\boldsymbol{x}_i\rangle)\right).$$

We can also simply replace $\langle\boldsymbol{w},\boldsymbol{x}_i\rangle$ with $\sum_{j=1}^n \alpha_j k(\boldsymbol{x}_i,\boldsymbol{x}_j)$ (as justified in the previous section) and obtain the new objective

$$\min_{\boldsymbol{\alpha}\in\mathbb{R}^n}\frac{1}{n}\sum_{i=1}^n\log\left(1+\exp\left(-y_i\sum_{j=1}^n\alpha_j k(\boldsymbol{x}_i,\boldsymbol{x}_j)\right)\right).$$

We can run the optimization methods in Chapter 5 and obtain a solution $\widehat{\boldsymbol{\alpha}}$, to obtain the classifier

$$\hat{y} = \operatorname{sign}\sum_{j=1}^n\widehat{\alpha}_j k(\boldsymbol{x},\boldsymbol{x}_j).$$

Again, linear classification can be seen as a special case of kernel-based classification with $k(\boldsymbol{x},\boldsymbol{x}') = \langle\boldsymbol{x},\boldsymbol{x}'\rangle$.

10

Neural Networks

In this chapter, we introduce the notion of a neural network (NN). We study the general architecture of neural networks, highlight the critical differences between neural networks and the previously studied regression and classification techniques, and analyze each step of the construction of a network - the choice of the size of the network, the choice of activation functions, and the solution of the optimization problems that show up. Finally, we present some methods that prevent overfitting, and we briefly touch upon deep learning and convolutional neural networks (CNNs).

Roadmap

In Section 10.1 we present the general form of a neural network and explain how neural networks use the training data to come up with meaningful features. In Section 10.2 we discuss the inference algorithm used by a neural network, known as forward propagation. In Section 10.3 we analyze the algorithm used for weight training, commonly referred to as backward propagation. In Section 10.4 we discuss the weight optimization further, emphasizing weight initialization and learning rates for the SGD algorithm. We also provide insight into methods that prevent overfitting, such as early stopping and batch normalization. Finally, in section Section 10.6, we introduce convolutional neural networks, explain their importance, and analyze their internal mechanisms.

Learning Objectives

After reading this chapter you should:

- Know the general form of a neural network, the basic activation functions, and the way in which activation functions interact with linear transformations throughout the network.
- Be able to describe and apply forward and backward propagation.
- Know why initialization and choice of the learning rate play a significant role in weight optimization.
- Be able to explain early stopping, dropout, and batch normalization, as well as their purpose.
- Know how to describe a convolutional neural network and compute the number of parameters and the output dimensions of such a network.

10.1 Introduction to Neural Networks

In the standard setting of a supervised learning problem, we are given a set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of training data, where $x_1, \ldots, x_n \in \mathbb{R}^d$, and our purpose is the detection of the most suitable prediction (regression/ classification) function[1] $\hat{f} : \mathbb{R}^d \to \mathbb{R}$.

We restrict our search to a large pool F of potential prediction functions[2], parametrized by a parameter $w \in \mathbb{R}^m$. The choice of a particular $f_w \in F$ is then subject to the minimization of the training loss. [3]

$$L(w; \mathcal{D}) := \frac{1}{n} \sum_{i=1}^{n} \ell\left(f_w(x_i), y_i\right).$$

Our first attempt was the use of affine prediction functions, that is, linear functions with a bias parameter. However, as shown in Figure 10.1 and Figure 10.2, such functions are often unable to suitably capture simple classification tasks.

Later, we attempted to capture nonlinear relations by introducing feature transformations that map the input $x = \left(x_{[1]}, \ldots, x_{[d]}\right)$ to a vector $\phi(x) \in \mathbb{R}^m$ and looked at affine functions of $\phi(x)$ instead of x. This gave us a lot of additional flexibility. For example, the classification tasks in Figure 10.1 ($x \in \mathbb{R}^2$) could now easily be solved with the use of $\phi(x) = x_{[1]}^2 + x_{[2]}^2$ and $\phi(x) = x_{[1]} x_{[2]}$ respectively. Moreover, we used the kernel trick to deal with the large computational complexity that this last method induces.

What do all the above methods have in common? One close look will reveal that, regardless of their form, the features are already determined prior to the training of the model. Indeed, whether they are x or $\phi(x)$, the features are hand-designed and do not depend on the particular training data set that we have at our disposal. In other words, we used prediction functions of the form $f_w(x) = w^T \phi(x)$, where ϕ was fixed *a priori*, and the training involved only fitting w.

Is this ideal? In fact, a set of features that is very meaningful for a specific prediction task might be significantly less useful when performing a different task, or when using a different dataset. Also, real datasets are usually more complex than the datasets in Figure 10.1. Consequently, in many cases, just by looking at such datasets, it might be impossible to think of specific features that would match.

Kernels are able to approximate *any* decision function given *infinite* amount of data, but in real scenarios when we only have access to finite sample points, the choice of a specific kernel and its parameters matter a lot. Again, simply looking at the dataset will rarely give us any information about the kernel that we should use.

To illustrate the difficulty of feature selection, we can look at the classification task in Figure 10.3. The goal is to train a model that

[1] For regression problems, we always assume that the relation of x with y derives from a *ground-truth* function f^*. More specifically, we assume that $y = f^*(x) + \varepsilon$, where ε is a noise random variable, independent from x. For binary classification problems we assume that $y = \text{sign}\,(\varepsilon + f^*(x))$.

[2] F is often the class $\{f : f(x) = w^T x + w_0\}$ of affine functions, or the class $\{f : f(x) = g(\phi(x))$ with $g(z) = w^T z + w_0\}$, where $\phi : \mathbb{R}^d \to \mathbb{R}^m$ is a feature transformation.

[3] $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a loss function. Common choices are $\ell(f(x), y) = (y - f(x))^2$ and $\ell(f(x), y) = \mathbf{1}_{\{yf(x)<0\}}$.

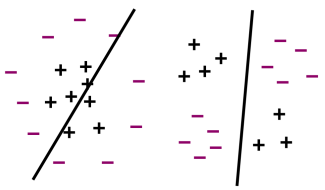


Figure 10.1: A linear classifier is unable to separate certain sets of points.

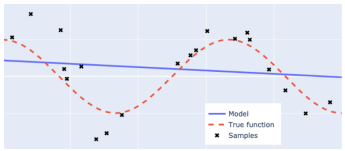
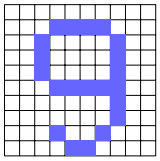


Figure 10.2: A linear regressor is unable to describe nonlinear relations.

recognizes a handwritten input digit.

Suppose that in our training set, we have a handwritten digit 9, as in Figure 10.4. What would be an effective way to create features from this digit so that we can train our classification model?

One seemingly good idea is to *digitalize* the given digit, that is, design an $m \times n$ grid and represent the digit with a pattern of colored cells.



Then, we could represent this cell by a sequence of length $m \times n$ and entries in $\{0, 1\}$ that will represent the color of each cell. In other words, the features would be the intensity of the cells in this $m \times n$ digitalization of each digit in the training set.

Although this looks like a promising idea, there are several reasons why it ultimately fails to produce effective features. For instance, imagine the same handwritten digit 9, but shifted a little towards some direction. We would like our model to recognize that this new digit is the same as the previous one. However, using the same grid as above will give a completely different sequence, so the feature values of the two digits will not be similar, even though we are talking about the same digit.

Hence, it is evident that hand-designing features is a difficult task, even for this simple classification task. [4] Ideally, we would prefer to have the features be determined by the data, or, in other words, to have the model learn good features from the training data.

This is exactly the idea behind neural networks. Instead of fixing ϕ and optimizing over $\boldsymbol{w}$, neural networks allow us to optimize jointly over ϕ and $\boldsymbol{w}$. This way, we can learn good features *directly from the data*. To formalize this new framework, it would be useful to recall some elements from the case of hand-designed features. In that case, we had a mapping $\phi(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), \dots, \phi_m(\boldsymbol{x}))$ and tried to solve the optimization problem

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w} \in \mathbb{R}^m}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \ell\left(\boldsymbol{w}^T \phi(\boldsymbol{x}_i), y_i\right).$$

As explained earlier, the idea is to allow the feature map ϕ to vary and to optimize over ϕ as well. The easiest way to do this is to parameterize ϕ by some set of parameters $\boldsymbol{\Theta}$ and to optimize jointly over $\boldsymbol{w}$ and $\boldsymbol{\Theta}$. Hence, the optimization problem is going to be

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w} \in \mathbb{R}^m, \boldsymbol{\Theta} \in \mathbb{R}^{m \times d}}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \ell\left(\boldsymbol{w}^T \phi(\boldsymbol{x}_i; \boldsymbol{\Theta}), y_i\right).$$

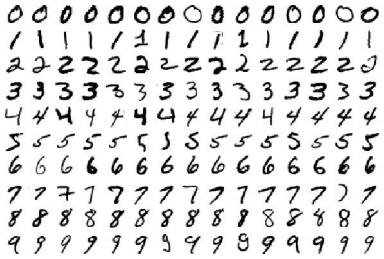


Figure 10.3: As we have seen in Chapter 8, the MNIST dataset is a famous dataset containing hand-written digits. We should train a model in a part of this dataset, and test its performance on a different part.

Figure 10.4: A handwritten digit 9.

[4] We ultimately want to be able to classify complex images, so this task is indeed significantly simpler.

Now, with $\boldsymbol{\theta}_j$ denoting the j-th row of $\boldsymbol{\Theta}$, the feature map $\phi(\boldsymbol{x};\boldsymbol{\Theta}) = (\phi_1(\boldsymbol{x};\boldsymbol{\theta}_1),\dots,\phi_m(\boldsymbol{x};\boldsymbol{\theta}_m))$ is not designed a priori, but it is determined by the training data points themselves, through the solution of the above optimization problem.

The next obstacle is the parametrization of the feature maps. More specifically, the functions $\phi_j(\boldsymbol{x};\boldsymbol{\theta}_j)$ must be determined prior to model training. Once again, we will employ a simple, yet effective idea: we will use feature maps of the form $\phi_j(\boldsymbol{x};\boldsymbol{\theta}_j) = \varphi(\boldsymbol{\theta}_j^T\boldsymbol{x})$, where $\boldsymbol{\theta}_j \in \mathbb{R}^d$ and $\varphi : \mathbb{R} \to \mathbb{R}$ is a nonlinear *activation function*. Equivalently, we can use matrix multiplication for a more concise notation, with $\phi(\boldsymbol{x};\boldsymbol{\Theta}) = \varphi(\boldsymbol{\Theta}\boldsymbol{x})$ where $\boldsymbol{\Theta} \in \mathbb{R}^{m\times d}$ and φ is applied element-wise.

Lastly, we need to choose the activation function φ. The following functions are commonly utilized:

- Identity: $\varphi(z) = z$
- Sigmoid: $\varphi(z) = \frac{1}{1+\exp(-z)}$
- Hyperbolic tangent: $\varphi(z) = \tanh z = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- Rectified Linear Unit (ReLU): $\varphi(z) = \max(0,z)$.

To conclude, given the weights $\boldsymbol{w}$ and the parameters $\boldsymbol{\theta}$, the neural network outputs a nested function of the form

$$f(\boldsymbol{x};\boldsymbol{w},\boldsymbol{\theta}) = \sum_{j=1}^{m} w_j \varphi\left(\boldsymbol{\theta}_j^T\boldsymbol{x}\right).$$

Such functions are called *artificial neural networks (ANNs)* or *multilayer perceptrons (MLP)*. More generally, the term *artificial neural network* refers to nonlinear functions which are nested *compositions of (learnable) linear functions composed with fixed nonlinear (activation) functions*. A neural network is commonly illustrated as in Figure 10.8.

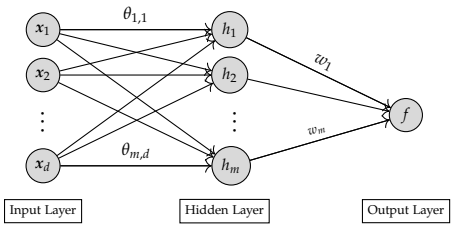


Figure 10.8: The graphical representation of a neural network with one hidden layer.

Before moving further, let us devote some time to ensuring we understand the network's internal functioning in Figure 10.8.

1. The network is given an input consisting of d real numbers $x_1,\dots,x_d$. These numbers are the entries of the input vector $\boldsymbol{x}$.

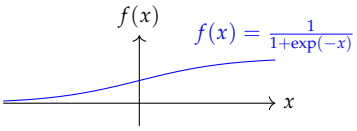


Figure 10.5: The sigmoid activation.

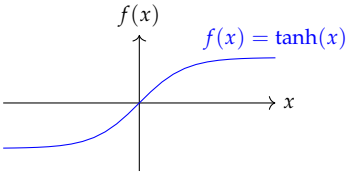


Figure 10.6: The hyperbolic tangent activation.

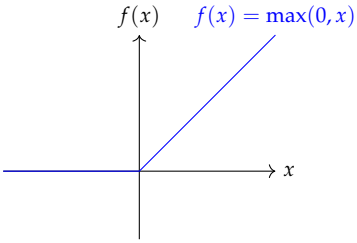


Figure 10.7: The ReLU activation.

2. At first, the inputs are linearly transformed into

$$z_i := \sum_{j=1}^{d} \theta_{i,j} x_j, \; i = 1, \ldots, m.$$

   Using matrix multiplication, we can neatly summarize this in a single formula with $\boldsymbol{z} = \boldsymbol{\Theta}\boldsymbol{x}$.

3. These linear combinations are then passed through a nonlinear activation function:

$$h_i := \varphi(z_i), \; i = 1, \ldots, m.$$

   These are the nodes of the hidden layer, and they will also be referred to as the *hidden/activation units*, or simply *units*. The number m of such units is called the *width of the layer*. Using vector notation and element-wise application of φ, this is often abbreviated to $\boldsymbol{h} = \varphi(\boldsymbol{z})$.

4. Lastly, $h_1, \ldots, h_m$ are linearly transformed into the output

$$f := \sum_{i=1}^{m} w_i h_i,$$

   or using vector notation: $f = \boldsymbol{w}^T \boldsymbol{h}$.

For simplicity, we will start using the term *parameters* to refer to all parameters of the entire network, whereas *weights* will be referring to the weight matrices of any given layer as in Figure 10.9. Further, to make the distinction between weight vectors and weight matrices more clear, we will be using uppercase $\boldsymbol{W}$ to denote weight matrices going forward.

Another modification that needs to be made is the inclusion of bias terms. Just like in linear regression, where we used affine functions of the form $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0$ instead of linear functions $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$, we need to account for the fact that the prediction function might not pass through the origin. Hence, we can include bias terms as shown in Figure 10.10. For neural networks, it is common to denote bias terms by $\boldsymbol{b} \in \mathbb{R}^m$, which we will adopt for the remainder of this chapter.

Instead of drawing these additional nodes for the bias terms, we could alternatively augment *both the input and the hidden units* with an extra entry equal to 1. Hence, in our subsequent illustrations, we are going to omit the bias nodes.

A plausible question following the above construction could be whether a neural network actually achieves its goal. In other words, given an arbitrary *ground truth* function f^*, are we able to approximate it using neural networks? The following theorem, proved by Cybenko 1989 gives an affirmative answer to this question when f^* is assumed to be continuous. Before stating the theorem, we define

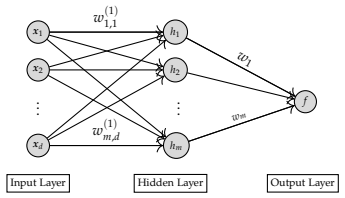


Figure 10.9: The illustration of a neural network with the new compact notation. The utility of the exponent in some weights will become evident later when we discuss about networks with multiple hidden layers.

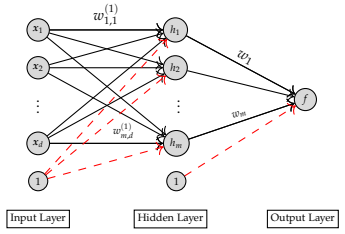


Figure 10.10: The illustration of a neural network with the new compact notation. The utility of the exponent in some weights will become evident later when we discuss about networks with multiple hidden layers.

a *sigmoidal function* $\sigma : \mathbb{R} \to \mathbb{R}$ to be any continuous function with the property

$$\lim_{t\to\infty} \sigma(t) = 1 \text{ and } \lim_{t\to-\infty} \sigma(t) = 0.$$

Notice that the sigmoid activation function is a sigmoidal function.

Theorem 10.1 (Universal Approximation Theorem)**.** *Let $f : [0,1]^d \to \mathbb{R}$ be a continuous function, and let φ be any sigmoidal function. Then, f can be uniformly approximated by a finite sum of the form*

$$\hat{f}(\boldsymbol{x}) = \boldsymbol{W}^{(2)} \varphi\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

with $\boldsymbol{W}^{(1)} \in \mathbb{R}^{m\times d}$, $\boldsymbol{b}^{(1)} \in \mathbb{R}^m$, and $\boldsymbol{W}^{(2)} \in \mathbb{R}^{1\times m}$ for some $m \in \mathbb{N}$. In other words, given any continuous function $f : [0,1]^d \to \mathbb{R}$ and any $\varepsilon > 0$, there exists a function $\hat{f}$ of the above form such that

$$\sup_{\boldsymbol{x}\in[0,1]^d} \left|\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})\right| \leq \varepsilon.$$

Remark. Since $f, \hat{f}$ are continuous functions and $[0,1]^d$ is a compact space, the above sup is actually a max.

Although Theorem 10.1 guarantees that neural networks with one hidden layer can approximate any continuous function, the theorem does not make statements about the number of hidden units m that must be used (i.e., the width of the network). It can be a good idea to consider more complex *neural network architectures*, e.g., by increasing the depth instead of the width. In particular, we can introduce the following modifications:

- Increase the number of hidden layers. This will make the network able to learn the decision function f^* more efficiently, due to the compositional structure. The number of hidden layers is called the *depth* of the network.
- Reorganize the network so that it has multiple outputs. This will render it suitable for more complex tasks such as multi-class classification. [5]
- Use different activation functions across different units and layers.

[5] For example, the multiple outputs will denote the probability that the input belongs to each of $K \geq 2$ classes. Then, we can assign the input to the class corresponding to the largest of the outputs.

The form of a neural network after introducing all these generalizations is illustrated in Figure 10.11. Such networks are often called *fully connected* because every unit is connected to all the units of the adjacent layers.

We will denote by $\boldsymbol{W}^{(i)}$ and $\boldsymbol{b}^{(i)}$ the weights $\left(w_{k,l}^{(i)}\right)$ and biases $b_k^{(i)}$ of the i-th layer, where $i = 1, 2, \ldots, L$, and by $\boldsymbol{\Theta}$ all the parameters of the network, consisting of the weights and biases $\left(\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)}, \boldsymbol{b}^{(1)}, \ldots, \boldsymbol{b}^{(L)}\right)$. We can think of $\boldsymbol{\Theta}$ either as a collection of vectors as above or as a long vector that results from gluing all the parameters together.

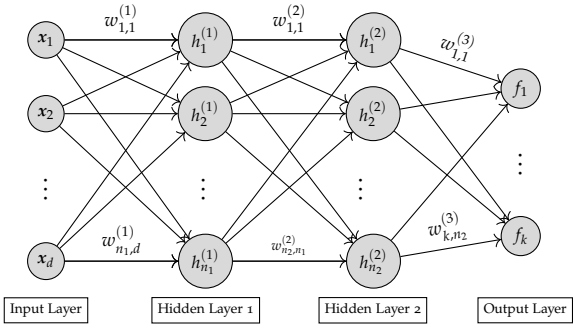


Figure 10.11: A neural network with two hidden layers and k outputs.

Except for increasing the depth of a network and varying the activation functions and the number of outputs, there are several other techniques that increase the power and flexibility of neural networks and allow them to perform surprisingly well in tasks that, until recently, only the human brain used to be able to execute. Such tasks are image classification, image captioning, and translation, as shown in Figure 10.12, Figure 10.13, and Figure 10.14.

Figure 10.12: An image classification task that can be executed with the use of neural networks. See Krizhevsky, Sutskever, and G. E. Hinton 2017 for additional details.

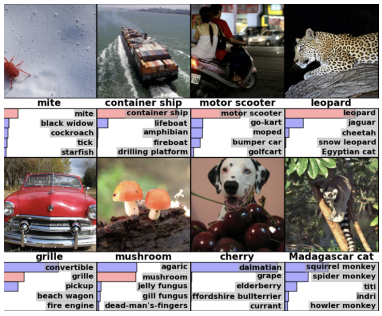


Figure 10.13: An image captioning task. See Vinyals et al. 2015 for additional details.

10.2 Forward propagation

In this section, we are going to delve deeper into the internal functioning of a neural network. More specifically, the next few pages are devoted to a more analytic study of the computations a neural network executes in order to make inferences about the input.

Although the graphical representation of neural networks is very intuitive and easy to use, we should bear in mind that, as a mathematical object, a neural network is a function $f(\cdot;\boldsymbol{\Theta})$, which accepts vectors $\boldsymbol{x}$ as inputs, and outputs real values y.

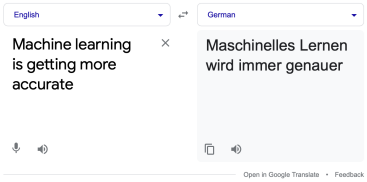
If the parameters $\boldsymbol{\Theta}$ are given (or learned), and the neural network is given an input point $\boldsymbol{x}$, how is it going to compute $y := f(\boldsymbol{x};\boldsymbol{\Theta})$?

For NNs with only one hidden layer, the answer was given in Section 10.1. Suppose now that we have a neural network with $L-1$ hidden layers. Then the computation is made with an algorithm known as *forward propagation* or *forward pass*. By writing out the unrolled mathematical formula

$$f(\boldsymbol{x};\boldsymbol{\Theta}) = \boldsymbol{W}^{(L)}\varphi(\boldsymbol{W}^{(L-1)}\varphi(\dots\varphi(\boldsymbol{W}^{(1)}\boldsymbol{x}+\boldsymbol{b}^{(1)})\dots)+\boldsymbol{b}^{(L-1)})+\boldsymbol{b}^{(L)},$$

it becomes clear that this can be in a simple while-loop that iterates through the layers. A pseudocode implementation is given in Algorithm 6.

Figure 10.14: Translation from English to German is an example of a machine learning application.



Exercise 10.2. Consider the neural network shown in Figure 10.15. All the activation functions are equal to the sigmoid activation $\varphi(x) = \frac{1}{1+\exp(-x)}$.

1. What is $h^{(1)}_i$?
2. What is $h^{(2)}_i$?

Algorithm 6: Forward propagation of an L-layer neural network.

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $l = 1, \dots, L-1$ **do**
    $\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}$
    $\boldsymbol{h}^{(l)} = \varphi(\boldsymbol{z}^{(l)})$
end for
$f = \boldsymbol{W}^{(L)}\boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)}$
return f

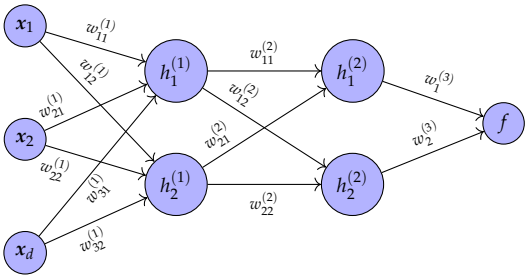


Figure 10.15: A NN with two hidden layers.

3. What is f?

Solution. 1. We compute $h_i^{(1)}$ in two steps:

- First, we construct the linear combination $z_i^{(1)} = \sum_{k=1}^{3} w_{ki}^{(1)} x_k$.
- Secondly, we pass this linear combination through the sigmoid activation function, and we obtain

$$h_i^{(1)} = \varphi\left(z_i^{(1)}\right) = \frac{1}{1+\exp\left(-z_i^{(1)}\right)} = \frac{1}{1+\exp\left(-\sum_{k=1}^{3} w_{ki}^{(1)} x_k\right)}$$

2. We proceed similarly:
   - First, we compute the linear combination $z_i^{(2)} = \sum_{k=1}^{2} w_{ki}^{(2)} h_k^{(1)}$.
   - Secondly, we apply φ:

$$h_i^{(2)} = \varphi\left(z_i^{(2)}\right) = \frac{1}{1+\exp\left(-\sum_{k=1}^{2} w_{ki}^{(2)} h_i^{(1)}\right)}.$$

3. This step is different only in that we do not need to apply the activation function in the end. In other words,

$$f = w_1^{(3)} h_1^{(2)} + w_2^{(3)} h_2^{(2)}.$$

□

10.3 Backward propagation

Forward propagation describes the inference procedure within a neural network. In other words, given a set of parameters $\boldsymbol{\Theta}$ and an input $\boldsymbol{x}$, a neural network uses forward propagation and outputs a label $y = f(\boldsymbol{x}; \boldsymbol{\Theta})$ that corresponds to the prediction of the NN for the given input.

However, as in most supervised learning approaches we have considered, one of the most important steps is the training of our model.

10.3.1 Optimizing over weights in neural networks

As we discussed earlier, training a neural network corresponds to finding the optimal parameters $\boldsymbol{\Theta}^*$ of the minimization problem

$$\boldsymbol{\Theta}^* := \arg\min_{\boldsymbol{\Theta}} L(\boldsymbol{\Theta}; \mathcal{D}) = \arg\min_{\boldsymbol{\Theta}} \frac{1}{n} \sum_{i=1}^{n} \ell(\boldsymbol{\Theta}; \boldsymbol{x}_i, y_i),$$

where ℓ is a given loss function, e.g. the squared loss: $(y - f(\boldsymbol{x}; \boldsymbol{\Theta}))^2$, or the logistic loss: $\log(1 + \exp\{-y \cdot f(\boldsymbol{x}; \boldsymbol{\Theta})\})$.

In general, the objective function $L(\boldsymbol{\Theta}; \mathcal{D})$ is not convex, so gradient descent might converge to a stationary point or a local minimum instead of a global minimum.

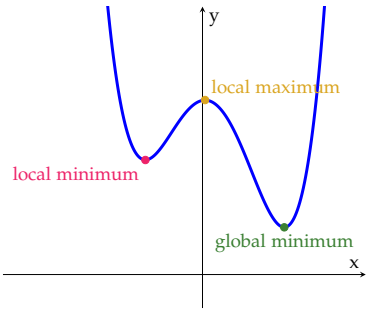


Figure 10.16: The objective function of a neural network is not necessarily convex. Depending on the initialization, GD might converge to the global minimum, but also to a local minimum, a saddle point, or even stay at a local maximum.

Despite potential degeneracies of the loss function like saddle points, local maxima, and local minima, we typically use gradient-based optimization, in particular, variants of (minibatch) stochastic gradient descent (SGD) (Algorithm 3). [6]

[6] The randomness induced by the use of SGD can help evade saddle points.

SGD is also more practical from a computational point of view. The dimensionality of the vector $\boldsymbol{\Theta}$ is equal to the number of parameters of the whole neural network, which is usually very large. Thus, it becomes very costly to compute the gradients of the n terms

$$\ell(\boldsymbol{\Theta}; \boldsymbol{x}_i, y_i), \ i = 1, \ldots, n.$$

Minibatch SGD requires the computation of $|S| \ll n$ of these gradients, making optimization significantly faster.

10.3.2 Computation of the gradient

Still, the computation of the gradient $\nabla_{\boldsymbol{\Theta}} \ell(\boldsymbol{\Theta}; \boldsymbol{x}, y)$ is costly as it requires the computation of all partial derivatives of the form

$$\frac{\partial}{\partial w_{i,j}^{(l)}} \ell(\boldsymbol{\Theta}; \boldsymbol{x}, y),$$

where $w_{i,j}^{(l)}$ denotes a single weight of the l-th layer. Below, we will introduce the backward propagation (a.k.a. backpropagation, or *backprop*) algorithm which utilizes the network structure to efficiently compute all of the partial derivatives. In Example 10.3, we present a simple special case that will help us understand the underlying computational mechanism.

Example 10.3. Consider an ANN with one output, one hidden layer, and one input unit, as shown in Figure 10.17. In this case, the neural network is a function that depends only on w, w' so it can be written as $f(\boldsymbol{x}; \boldsymbol{\Theta}) = f(x; [w, w'])$. The input unit x passes through the following

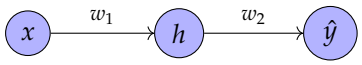


Figure 10.17: An simple ANN with one output, one hidden layer, and one input unit.

transformations:

$$x \to w_1 x \to \varphi(w_1 x) \to w_2 \cdot \varphi(w_1 x) = \hat{y}.$$

The linear transformation $w_1 x$ is denoted by z, $\varphi(z)$ is denoted by h, and the final result $f(x; \boldsymbol{\Theta})$ is denoted by $\hat{y}$. For this example, we are using the square loss. Then,

$$\ell(\boldsymbol{\Theta}; x, y) = (y - \hat{y})^2 =: \ell_y(\hat{y}).$$

Suppose that we are using a gradient-based optimization method and that we have initialized the weights at some fixed values. We aim to perform the next iteration of the algorithm as efficiently as possible, i.e., with as few computations as possible.

We first execute forward propagation as in Algorithm 6, and we store z, w_2, and $\hat{y}$. The gradient is two-dimensional, and it is equal to

$$\nabla_{\boldsymbol{\Theta}} \ell(\boldsymbol{\Theta}; x, y) = \left[\frac{\partial \ell}{\partial w_2}, \frac{\partial \ell}{\partial w_1} \right]^T.$$

From the chain rule,

$$\frac{\partial \ell}{\partial w_2} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2}.$$

The first factor is equal to $2(\hat{y} - y)$. Since $\hat{y} = w_2 \cdot h$, the second factor is equal to h. Hence,

$$\frac{\partial \ell}{\partial w_2} = 2(\hat{y} - y) \cdot h.$$

Notice that h has been computed during forward propagation as shown in Algorithm 6, so it need not be computed again here. For the second partial derivative, we use the chain rule again:

$$\begin{aligned} \frac{\partial \ell}{\partial w_1} &= \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h} \cdot \frac{\partial h}{\partial z} \cdot \frac{\partial z}{\partial w_1} \\ &= 2(\hat{y} - y) \cdot w_2 \cdot \varphi'(z) \cdot x. \end{aligned}$$

The first factor has already been computed in the previous step. The other equalities result from the relations

$$\hat{y} = w_2 \cdot h, \; h = \varphi(z), \text{ and } z = w_1 x.$$

After these computations, we can update the weights and repeat the same process for their new values.

To conclude, instead of six factors, two for the first and four for the second term, we only needed to compute two, $2(\hat{y} - y)$ and $\varphi'(z)$. The rest were either common between the two terms so they only had to be computed once, or they had been computed during the forward pass.

The above example shows how gradient computation can be accelerated by taking advantage of the forward pass and memoizing intermediate results that will be required by the backpropagation algorithm. To generalize these ideas to more complex neural networks, it is essential to recall some elements from multivariate calculus that were discussed in Section 2.

In particular, it is common to use $\nabla_{\boldsymbol{x}} f$ to denote taking the gradient of some function $f : \mathbb{R}^n \to \mathbb{R}$ (usually of the loss function) with respect to $\boldsymbol{x} \in \mathbb{R}^n$, which can be a vector (e.g. hidden units or biases of some intermediate layer) or a matrix (e.g. weight matrices). Let us recall the definition of the gradient from multivariate calculus:

$$\nabla_x f = \frac{\partial f}{\partial \boldsymbol{x}} = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]^T.$$

Notice how the canonical definition of the gradient always is with respect to a (column-)*vector*, but what if we want to take the gradient w.r.t. a *matrix* (or a row-vector)? One way to think about this is that we just take the partial derivative w.r.t. each element and arrange the result into the original shape, so for some matrix $\boldsymbol{W}^{(l)} \in \mathbb{R}^{m \times n}$, we have

$$\nabla_{\boldsymbol{W}^{(l)}} f = \frac{\partial f}{\partial \boldsymbol{W}^{(l)}} = \begin{bmatrix} \partial f / \partial w_{1,1}^{(l)} & \dots & \partial f / \partial w_{1,n}^{(l)} \\ \vdots & \ddots & \vdots \\ \partial f / \partial w_{m,1}^{(l)} & \dots & \partial f / \partial w_{m,n}^{(l)} \end{bmatrix}.$$

10.3.3 Optimization in a general neural network

Let us now consider a NN with its parameters initialized at

$$\boldsymbol{\Theta} = \left(\boldsymbol{W}^{(1)}, \dots, \boldsymbol{W}^{(L)}, \boldsymbol{b}^{(1)}, \dots, \boldsymbol{b}^{(L)}\right).$$

Our goal is to compute the gradients

$$\nabla_{\boldsymbol{W}^{(1)}} \ell, \dots, \nabla_{\boldsymbol{W}^{(L)}} \ell, \nabla_{\boldsymbol{b}^{(1)}} \ell, \dots, \nabla_{\boldsymbol{b}^{(L)}} \ell$$

and update the weights using GD, SGD, or any other iterative optimization algorithm. Here, $\ell = \ell(\boldsymbol{\Theta}; \boldsymbol{x}, y)$ is the loss function as usual. For simplicity, in the following we focus on computing the gradients of the weight matrices, but the same principles apply to computing the gradients of the bias terms.

- Step 1: Like in Example 10.3, we start with the last layer's weights and use the chain rule:

  $$\nabla_{\boldsymbol{W}^{(L)}} \ell = \frac{\partial \ell}{\partial \boldsymbol{W}^{(L)}} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial \boldsymbol{W}^{(L)}}.$$

  The first factor can be computed as in Example 10.3. According to Algorithm 6, $f = \boldsymbol{W}^{(L)} \boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)}$, so the second factor equals

  $$\frac{\partial f}{\partial \boldsymbol{W}^{(L)}} = \begin{bmatrix} \left(\boldsymbol{h}^{(L-1)}\right)^T \\ \vdots \\ \left(\boldsymbol{h}^{(L-1)}\right)^T \end{bmatrix},$$

  a matrix with as many rows as $\boldsymbol{W}^{(L)}$, all equal to $\boldsymbol{h}^{(L-1)}$. [7] Note that in the case of binary classification, $k = 1$ and $\boldsymbol{W}^{(L)}$ consists of a single row, in which case the result simply becomes the transposed gradient w.r.t. $\boldsymbol{W}^{(L)}$. Also notice that $\boldsymbol{h}^{(L-1)}$ has been computed during the forward pass already, which can be reused.

[7] To recall how we differentiate with respect to a matrix, such as $\boldsymbol{W}^{(L)}$, review the bonus material in Section 2.1.

Remark. Notice that the dimensions of the two factors match: f is a k-dimensional vector and ℓ is real-valued, so

$$\frac{\partial \ell}{\partial f} = \nabla_f \ell$$

is a $1 \times k$-dimensional vector. On the other side,

$$f = \boldsymbol{W}^{(L)} \boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)} \in \mathbb{R}^k,$$

so $\boldsymbol{W}^{(L)}$ has k rows. Thus, $\frac{\partial f}{\partial \boldsymbol{W}^{(L)}}$ also has k rows.

The number of columns of $\frac{\partial f}{\partial \boldsymbol{W}^{(L)}}$ is equal to the dimension of $\boldsymbol{h}^{(L-1)}$, which is equal to n_{L-1}.

- Step 2: We go one layer back and we compute $\nabla_{\boldsymbol{W}^{(L-1)}} \ell$.

  Recall that, with the notation of Algorithm 6, ℓ is a function of $f = \boldsymbol{W}^{(L)} \boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)}$. In its turn, $\boldsymbol{h}^{(L-1)}$ is related to $\boldsymbol{W}^{(L-1)}$ through

  $$\boldsymbol{h}^{(L-1)} = \varphi(\boldsymbol{z}^{(L-1)})$$

  and

  $$\boldsymbol{z}^{(L-1)} = \boldsymbol{W}^{(L-1)} \boldsymbol{h}^{(L-2)} + \boldsymbol{b}^{(L-1)}.$$

  These relations show us how we need to apply the chain rule to get the desired result:

  $$\nabla_{W^{(L-1)}} \ell = \frac{\partial \ell}{\partial f} \cdot \underbrace{\frac{\partial f}{\partial \boldsymbol{h}^{(L-1)}} \cdot \frac{\partial \boldsymbol{h}^{(L-1)}}{\partial \boldsymbol{z}^{(L-1)}}}_{=\partial f / \partial \boldsymbol{z}^{(L-1)}} \cdot \frac{\partial \boldsymbol{z}^{(L-1)}}{\partial \boldsymbol{W}^{(L-1)}}.$$

  The first factor has been computed in Step 1. The second one is equal to $\boldsymbol{W}^{(L)}$, given the form of f, so we also do not need to compute it. Like in Step 1, the last factor is equal to the matrix

  $$\begin{bmatrix} \left(\boldsymbol{h}^{(L-2)}\right)^T \\ \vdots \\ \left(\boldsymbol{h}^{(L-2)}\right)^T \end{bmatrix},$$

  which has been computed during forward propagation. Thus, it remains to compute $\frac{\partial \boldsymbol{h}^{(L-1)}}{\partial \boldsymbol{z}^{(L-1)}}$. From the element-wise application of φ in $\boldsymbol{h}^{(L-1)} = \varphi(\boldsymbol{z}^{(L-1)})$ it follows that

  $$\frac{\partial \boldsymbol{h}^{(L-1)}}{\partial \boldsymbol{z}^{(L-1)}} = \begin{bmatrix} \varphi'\left(z_1^{(L-1)}\right) & 0 & \dots & 0 \\ 0 & \varphi'\left(z_2^{(L-1)}\right) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \varphi'\left(z_{n_{L-1}}^{(L-1)}\right) \end{bmatrix}.$$

  This is sometimes also written as $\operatorname{diag}\left(\varphi'(\boldsymbol{z}^{(L-1)})\right)$ and is the only factor that has not been computed before.

- Steps 3,4,...: We continue with

  $$\nabla_{\boldsymbol{W}^{(L-2)}} \ell, \dots, \nabla_{\boldsymbol{W}^{(1)}} \ell$$

  in a similar fashion.

As illustrated in Figure 10.18, the algorithm's efficiency is achieved by reusing computations from previous steps and from the forward pass.

$$
\begin{aligned}
\nabla_{\boldsymbol{W}^{(L)}}\ell = \quad & \frac{\partial \ell}{\partial \boldsymbol{W}^{(L)}} && = \frac{\partial \ell}{\partial f}\cdot\frac{\partial f}{\partial \boldsymbol{W}^{(L)}} \\
\nabla_{\boldsymbol{W}^{(L-1)}}\ell = \quad & \frac{\partial \ell}{\partial \boldsymbol{W}^{(L-1)}} && = \frac{\partial \ell}{\partial f}\cdot\frac{\partial f}{\partial \boldsymbol{z}^{(L-1)}}\cdot\frac{\partial \boldsymbol{z}^{(L-1)}}{\partial \boldsymbol{W}^{(L-1)}} \\
\nabla_{\boldsymbol{W}^{(L-2)}}\ell = \quad & \frac{\partial \ell}{\partial \boldsymbol{W}^{(L-2)}} && = \frac{\partial \ell}{\partial f}\cdot\frac{\partial f}{\partial \boldsymbol{z}^{(L-1)}}\cdot\frac{\partial \boldsymbol{z}^{(L-1)}}{\partial \boldsymbol{z}^{(L-2)}}\cdot\frac{\partial \boldsymbol{z}^{(L-2)}}{\partial \boldsymbol{W}^{(L-2)}} \\
& \vdots && \vdots \\
\nabla_{\boldsymbol{W}^{(l)}}\ell = \quad & \frac{\partial \ell}{\partial \boldsymbol{W}^{(l)}} && = \frac{\partial \ell}{\partial f}\cdot\frac{\partial f}{\partial \boldsymbol{z}^{(L-1)}}\cdot\frac{\partial \boldsymbol{z}^{(L-1)}}{\partial \boldsymbol{z}^{(L-2)}}\cdot\ldots\cdot\frac{\partial \boldsymbol{z}^{(l+1)}}{\partial \boldsymbol{z}^{(l)}}\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{W}^{(l)}}
\end{aligned}
$$

Figure 10.18: At each step we only need to compute one factor, the red one. This way, the number of computations is minimized and optimization becomes faster.

It now starts to become apparent why this algorithm is called *backpropagation*: Through repeated application of the chain rule, we propagate the gradients through the hidden layers one layer at a time, starting at the final layer and going backwards form there.

10.4 Optimization and training techniques in NNs

As discussed in the previous section, the objective function

$$L(\boldsymbol{\Theta};\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}\ell\left(\boldsymbol{\Theta};\boldsymbol{x}_i,y_i\right)$$

of a neural network is not convex in general. This raises one major issue during optimization: The convergence of gradient descent depends on the initialization of the weights. [8]

[8] This would not be an issue had the objective function been convex. Indeed, by Theorem 5.14, every stationary point of a convex function is a global minimum.

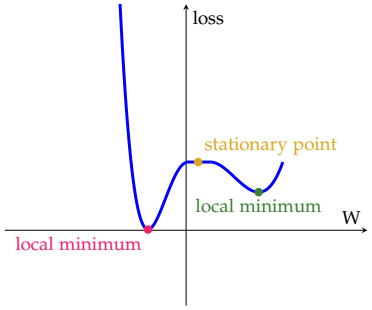


Figure 10.19: If we initialize the weights at (or close to) a stationary point, then gradient descent might get stuck at this point.

In Figure 10.19 it becomes clear that, depending on the initial value of the weights, gradient-based methods might converge to a local or global minimum, but might also get stuck at a stationary point which is not a local minimum. Indeed, since the gradient

$$\nabla_{\boldsymbol{\Theta}}L(\boldsymbol{\Theta};\mathcal{D})$$

of the objective function is zero in all these kinds of points, the update

$$\boldsymbol{\Theta}^{t+1} = \boldsymbol{\Theta}^{t} - \eta_t\cdot\nabla_{\boldsymbol{\Theta}}L(\boldsymbol{\Theta};\mathcal{D}) = \boldsymbol{\Theta}^{t} - \eta_t\cdot 0 = \boldsymbol{\Theta}^{t} \tag{10.1}$$

will be static. Additionally, computing the gradient over the entire dataset at every step quickly becomes computationally prohibitive as the number of data points and NN parameters grows.

One way to avoid both of these issues is to use (minibatch) SGD instead of GD. This way, we use the update

$$\boldsymbol{\Theta}^{t+1} = \boldsymbol{\Theta}^t - \eta_t \cdot \nabla_{\boldsymbol{\Theta}^t} \left(\frac{1}{B} \sum_{i \in S} \ell \left(\boldsymbol{\Theta}^t; \boldsymbol{x}_i, y_i \right) \right), \tag{10.2}$$

where the minibatch $S \subset \{1, \ldots, n\}$ with batch-size $|S| = B$ might change from iteration to iteration. [9]

[9] Usually, instead of drawing B i.i.d. samples after each step, we shuffle the dataset once at the beginning of training and then iterate through all data points in batches of size B. One iteration through the dataset is called an *epoch*, and usually the dataset is reshuffled after every epoch. More information about the exact mechanism can be found in Section 5.4.4.

Hence, even if $\boldsymbol{\Theta}^t$ is a stationary point of the objective function, the gradient

$$\nabla_{\boldsymbol{\Theta}^t} \left(\frac{1}{B} \sum_{i \in S} \ell \left(\boldsymbol{\Theta}^t; \boldsymbol{x}_i, y_i \right) \right) = \frac{1}{B} \sum_{i \in S} \nabla_{\boldsymbol{\Theta}^t} \ell \left(\boldsymbol{\Theta}^t; \boldsymbol{x}_i, y_i \right) \tag{10.3}$$

of the restricted sum might not be equal to zero, so we might be able to escape the stationary point.

Even so, there are still some problems that might persist, and these problems will be discussed in the following subsections.

10.4.1 Vanishing and exploding gradients

From Equation 10.2 and Equation 10.3 it becomes clear that the size $\left\| \boldsymbol{\Theta}^{t+1} - \boldsymbol{\Theta}^t \right\|_2$ and the direction of the update from $\boldsymbol{\Theta}^t$ to $\boldsymbol{\Theta}^{t+1}$ are determined by the individual terms

$$\nabla_{\boldsymbol{\Theta}^t} \ell \left(\boldsymbol{\Theta}^t; \boldsymbol{x}_i, y_i \right), \; i \in S,$$

which are in turn composed by

$$\nabla_{\boldsymbol{W}^{(l)}} \ell \left(\boldsymbol{W}^{(l)}; \boldsymbol{x}_i, y_i \right), \; l = 1, \ldots, L.$$

If these components grow too much $\left(\left\| \nabla_{\boldsymbol{W}^{(l)}} \ell \right\|_2 \to \infty \right)$ or shrink too much $\left(\left\| \nabla_{\boldsymbol{W}^{(l)}} \ell \right\|_2 \to 0 \right)$, then optimization can fail. [10] How is it possible to control the size of these terms and make optimization more stable?

[10] Too large updates can drive optimization away from the critical points (local/global minima) and too small ones make optimization too slow.

One solution to the problem of controlling the size of the gradient is to choose the activation functions in a careful way. More specifically, we saw that the matrix

$$\operatorname{diag} \left(\varphi' \left(\boldsymbol{z}^{(l)} \right) \right)$$

appears during optimization as a multiplicative factor, so the derivative φ' directly affects the size of the gradient. If $\varphi' \left(z_i^{(l)} \right)$ becomes too small or too large (in terms of its absolute value), the gradients $\nabla_{\boldsymbol{W}^{(l)}} \ell$ also tend to follow the same behavior.

For example, the derivative of the sigmoid activation function

$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$

is equal to

$$\varphi'(z) = \frac{\exp(-z)}{(1+\exp(-z))^2},$$

which can also be written as $\varphi(z) \cdot (1 - \varphi(z))$. The graph of this function is shown in Figure 10.20.

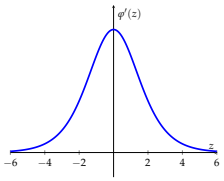


Figure 10.20: The derivative of the sigmoid function is nonzero, but it is *essentially zero* everywhere but in a small region around $z = 0$.

The first advantage of this derivative is that it is easy to compute. Thus, we do not need to resort to numerical approximations that would most likely make optimization much slower. Another advantage is that the derivative is nonzero. This prevents the gradient $\nabla_{\boldsymbol{W}^{(l)}} \ell$ from becoming equal to zero, so it is harder for GD (or SGD) to get stuck on stationary points. [11]

[11] Of course the gradient contains more factors, so it might actually get equal to zero because some of them are zero. Still, since $\varphi'(z) \neq 0$, the chances that $\nabla_{\boldsymbol{W}^{(l)}} \ell = 0$ are lower.

On the other hand, $\varphi'(z)$ takes very small values everywhere except for a small region around zero which might lead to vanishing gradients. Consequently, the sigmoid function might be a safer choice when we know a priori that

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}$$

is more likely to concentrate around zero.

Another option for the activation function is the ReLU, which is defined as

$$\varphi(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{else} \end{cases}$$

The ReLU activation is not differentiable at $z = 0$. At the rest of the points, its derivative is

$$\varphi'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{else} \end{cases}$$

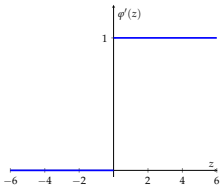


Figure 10.21: The derivative of the ReLU activation is piecewise constant.

Since z will rarely be exactly equal to zero, the non-differentiability of the function at $z = 0$ is not so restrictive. Moreover, while the derivative is equal to zero for $z < 0$, it is strictly bounded away from zero for all positive z, which often helps in avoiding vanishing gradients.

10.4.2 *Random weight initialization*

Even if we choose the activation functions in the best possible way, the gradients might still vanish or explode, as they also depend on other factors. For instance, we saw in Subsection 10.3.3 that

$$\nabla_{W^{(l)}} \ell = \frac{\partial \ell}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial \boldsymbol{h}^{(l)}} \cdot \frac{\partial \boldsymbol{h}^{(l)}}{\partial \boldsymbol{z}^{(l)}} \cdot \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{W}^{(l)}},$$

and that the last two factors are equal to

$$\operatorname{diag}\left(\varphi'\left(\boldsymbol{z}^{(l)}\right)\right) \text{ and } \begin{bmatrix} (\boldsymbol{h}^{(l-1)})^T \\ \vdots \\ (\boldsymbol{h}^{(l-1)})^T \end{bmatrix}.$$

Hence, the gradients also depend on $\boldsymbol{h}^{(l-1)}$ for $l = 1, \ldots, L$. If the norms of these vectors grow or shrink too much, this will also affect the norms of the gradients.

One way to control the norm of $\boldsymbol{h}^{(l-1)}$ is to *randomly* initialize the weights. In that case, $\boldsymbol{h}^{(l-1)}$ becomes a random variable. Below we analyze the variance of this random variable, giving us some guarantees about its size.

Bonus Material

The fact that a bound on the variance of $\boldsymbol{h}^{(l-1)}$ allows us to control $\left|\boldsymbol{h}^{(l-1)}\right|$ can be justified by Chebyshev's inequality. This inequality says that, if X is a random variable with mean μ and variance σ^2, then for any $k > 0$,

$$\Pr\left(|X - \mu| \geq k\sigma\right) \leq \frac{1}{k^2}.$$

Hence, if $\mu = 0$ and σ^2 is bounded by a constant C, it holds that $|X|$ is bounded by kC with probability at least $1 - \frac{1}{k^2}$.

In general, due to the complex nature of a neural network and the complete freedom in the choice of the activation function, it is quite hard to derive exact theoretical expressions for the choice of the weights and the bounds for $\operatorname{Var}\left(\boldsymbol{h}^{(l-1)}\right)$. However, we can look at simple networks to obtain some intuition. This is demonstrated in Example 10.4.

Example 10.4. Consider the simple one-layer network shown in Figure 10.22. Suppose that this network uses the ReLU activation function $\varphi(z) = \max\{z, 0\}$. Suppose that the inputs $x_1, \ldots, x_d$ are i.i.d. zero-mean random variables with variance $\operatorname{Var}(x_i) = \mathbb{E}(x_i^2) = 1$. [12]

Suppose that the weights are initialized randomly, independently from each other, and independently from $x_1, \ldots, x_d$. More specifically, assume that

$$w_1, \ldots, w_d \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2),$$

where $\sigma > 0$ a constant. Observe that the random variable

$$Z = \sum_{i=1}^{d} w_i x_i$$

[12] The assumptions about the mean and variance are not really restrictive. One can achieve them by standardizing the input variables i.e. subtracting their mean and normalizing with their variance.

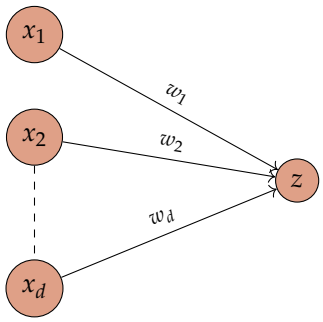


Figure 10.22: A simple NN with one hidden layer.

is symmetric, i.e.

$$\Pr(Z = z) = \Pr(Z = -z)$$

for all $z \in \mathbb{R}$. This is due to the symmetry of $w_1, \dots, w_d$ and their independence with $x_1, \dots, x_d$. Also,

$$\mathbb{E}[Z] = \sum_{i=1}^{d} \mathbb{E}[w_i x_i] = \sum_{i=1}^{d} \underbrace{\mathbb{E}[w_i]}_{0} \underbrace{\mathbb{E}[x_i]}_{0} = 0.$$

Notice that

$$\begin{aligned}\mathrm{Var}(w_i x_i) &= \mathbb{E}\left[w_i^2 x_i^2\right] - (\mathbb{E}[w_i x_i])^2 \\ &= \mathbb{E}\left[w_i^2\right] \mathbb{E}\left[x_i^2\right] \\ &= \sigma^2,\end{aligned}$$

so

$$\mathrm{Var}[Z] = \mathrm{Var}\left(\sum_{i=1}^{d} w_i x_i\right) = \sum_{i=1}^{d} \mathrm{Var}(w_i x_i) = d \cdot \sigma^2.$$

Finally, we have

$$\begin{aligned}\mathbb{E}\left[v^2\right] &= \mathbb{E}\left[\max\{Z, 0\}^2\right] \\ &= \mathbb{E}\left[Z^2 \mid Z > 0\right] \cdot \Pr(Z > 0) + \mathbb{E}[0 \mid Z < 0] \cdot \Pr(Z < 0) \\ &= \frac{1}{2}\mathbb{E}\left[Z^2 \mid Z < 0\right] \\ &= \frac{1}{2}\mathbb{E}\left[Z^2\right] \\ &= \frac{1}{2} d\sigma^2,\end{aligned}$$

where the second to last equality holds due to the symmetry of Z. Hence, setting $\sigma = \sqrt{2/d}$ ensures that the variance of z will be bounded by 1. In other words, the weights should be sampled from the distribution $\mathcal{N}\left(0, \frac{2}{d}\right)$.

This example gives us some heuristics about weight initialization. Of course, general NNs are much deeper and much more complex, but we can follow the same ideas to ensure that z has a bounded variance, and so that it does not lead to exploding gradients.

Two initialization schemes that usually work well in practice are the following:

- Glorot and Bengio (2010): $w_i \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$ or $w_i \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}+n_{\text{out}}}\right)$, where $n_{\text{in}}, n_{\text{out}}$ denote the number of nodes of the layers on

which these weights act. These initialization schemes, known as *Xavier* initialization, are commonly used for the hyperbolic tangent (tanh) activation function.

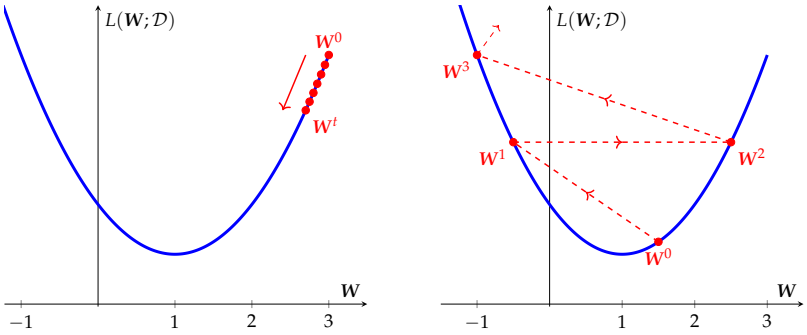
- He et al. (2015): $w_i \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$. This scheme is followed whenever the network uses the ReLU activation, which explains the similarity with the result we found in Example 10.4.

10.4.3 *Learning rate and weight updates*

As we have seen in Equation 10.2, SGD updates the weights after each iteration according to the rule

$$\boldsymbol{\Theta}^{t+1} = \boldsymbol{\Theta}^t - \eta_t \cdot \nabla_{\boldsymbol{\Theta}^t} \left(\frac{1}{B} \sum_{i \in S} \ell\left(\boldsymbol{\Theta}^t; \boldsymbol{x}_i, y_i\right) \right).$$

One question that we have left unanswered is the choice of the learning rate η_t. In Remark 5.2, we explained that this is a crucial part that affects both the convergence and the speed of the algorithm. This is illustrated in Figure 10.23.



(a) If η is too small, it will take longer to reach the minimum.

(b) If η becomes too large, then (stochastic) gradient descent might even fail to converge.

Figure 10.23: Failing to choose the learning rate carefully can have catastrophic effects on optimization.

Like weight initialization, there are no specific criteria for the choice of η that guarantee fast convergence of the optimization algorithm. However, several heuristic methods have been found to work well in practice.

One of them is to decrease the learning rate η_t as the number t of iterations increases. The idea behind this method is that, being randomly selected, the initial parameters $\boldsymbol{\Theta}^0$ are most likely far away from the global minimum, so allowing for large steps would quickly bring us close to it. After some iterations, we will get closer to the minimum so large steps may lead to drifting away from it. This is illustrated in Figure 10.24.

In practice, popular choices include piecewise constant learning rate, as shown in Figure 10.4.3, or linear/cosine decay.

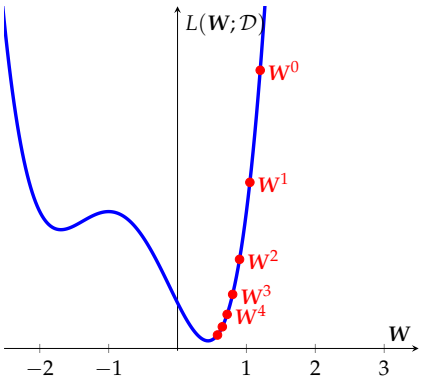


Figure 10.24: In the beginning, the stepsize is very large, which quickly brings us close to the minimum. Later on, we make smaller and smaller steps to make sure that we do not overshoot the minimum point.

Oftentimes, SGD does not converge to the minimum monotonically. Instead, it oscillates around it before getting close enough. Hence, constantly decreasing the learning rate might not be an ideal strategy. One common alternative is to use the ratio

$$\frac{\left|\nabla_{\boldsymbol{\Theta}^t} L\left(\boldsymbol{\Theta}^t; \mathcal{D}\right)\right|}{\left\|\boldsymbol{\Theta}^t\right\|_2}$$

as an indicator. This measures the size of the weight update compared to the magnitude of the weights. If this ratio is very small, this is an indication that we should increase the learning rate, and vice versa.

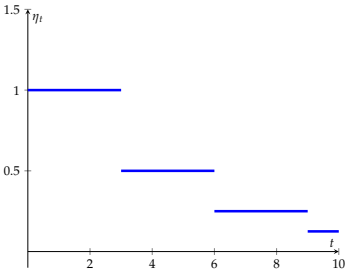


Figure 10.25: A piecewise constant learning rate.

Another method that is commonly used to speed up the convergence of SGD is the use of *momentum*. This method was discussed independently of NNs in Subsection 5.4.1. The main issue that this method tries to address is that SGD does not always move in a descent direction, so the trajectory

$$\boldsymbol{\Theta}^0 \to \boldsymbol{\Theta}^1 \to \ldots \to \boldsymbol{\Theta}^t$$

might oscillate without actually making significant progress toward the minimum. This is illustrated in Figure 10.26.

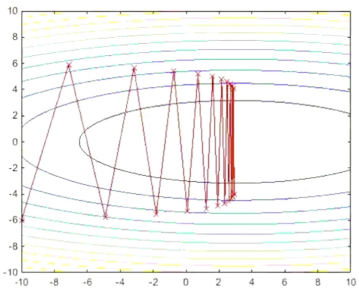


Figure 10.26: Because SGD does not generally move in a descent direction, the weights might follow an oscillating trajectory and need a lot of iterations to reach the minimum, or even overshoot it.

However, if we combine the update direction $-\nabla_{\boldsymbol{\Theta}^t} L\left(\boldsymbol{\Theta}^t; \mathcal{D}\right)$ with the direction of the previous updates, it is more likely that we will move closer to the direction of the minimum. In other words, instead of

$$\boldsymbol{\Theta}^{t+1} \leftarrow \boldsymbol{\Theta}^t - \eta_t \nabla_{\boldsymbol{\Theta}^t} L\left(\boldsymbol{\Theta}^t; \mathcal{D}\right),$$

we update the weights according to

$$\begin{aligned} \boldsymbol{v}^{t+1} &\leftarrow m \cdot \boldsymbol{v}^t + \eta_t \nabla_{\boldsymbol{\Theta}^t} L\left(\boldsymbol{\Theta}^t; \mathcal{D}\right) \\ \boldsymbol{\Theta}^{t+1} &\leftarrow \boldsymbol{\Theta}^t - \boldsymbol{v}^{t+1}, \end{aligned}$$

where $m > 0$ is a constant that regulates the degree of contribution of the past update directions to the current one. The effect of momentum is shown in Figure 10.27.

10.5 Regularization in NNs

Neural networks with several layers usually have thousands or millions of parameters so we might be worried about overfitting. There

are several techniques that help avoid the danger of overfitting. Some of them follow the same logic as the general regularization techniques that we saw in Section 7.3, and some others are heuristic ideas that have proved to work well in practice.

Adding a penalty term is a regularization technique that we have talked about earlier, and which is also very popular in NNs. More specifically, instead of solving the optimization problem

$$\arg\min_{\boldsymbol{\Theta}\in\mathbb{R}^d} L(\boldsymbol{\Theta};\mathcal{D}),$$

we solve

$$\arg\min_{\boldsymbol{\Theta}\in\mathbb{R}^d} L(\boldsymbol{\Theta};\mathcal{D}) + \lambda \left\|\boldsymbol{\Theta}\right\|_2^2,$$

where $\lambda > 0$ is a constant whose choice follows the same principles as in Subsection 7.5.

Another way to prevent overfitting is to stop training earlier. Normally, as we have seen in Algorithm 3, training stops when a stopping criterion of the form $\left\|\boldsymbol{\Theta}^{t+1} - \boldsymbol{\Theta}^t\right\|_2 \leq \varepsilon$ is satisfied. [13] One alternative stopping criterion is the performance of the current model on the validation set.

When the network starts to overfit, its training error continues to go down, but its validation error starts to increase. One way to detect when overfitting starts to occur is to compute the validation error of the network after some number of iterations of SGD (e.g., after each epoch, or pass through the data set). When the validation error starts rising, we stop training even if the stopping criterion $\left\|\boldsymbol{\Theta}^{t+1} - \boldsymbol{\Theta}^t\right\|_2 \leq \varepsilon$ is not satisfied yet. This technique is known as *early stopping* and is illustrated in Figure 10.28.

10.5.1 Dropout regularization

The introduction of penalty terms and early stopping are methods that are not restricted to NNs but can be used in several other machine-learning models. On the other hand, there are some regularization techniques that have been inspired by the specific nature of NNs and only apply to them.

One popular technique of this type is the so-called *dropout*, which was introduced by Srivastava et al. (2014). The key idea of dropout is to reduce the number of parameters during training by randomly omitting some of the network units during each iteration of SGD.

More specifically, we fix a number $p \in (0,1)$ and, before the start of each iteration of SGD, we eliminate ("drop out") each hidden unit with probability $1-p$. [14] We also freeze all weights associated with the deleted hidden units. Then, we consider the sub-network with the hidden units that remain and update only the weights associated with them. Finally, we bring back the deleted hidden units

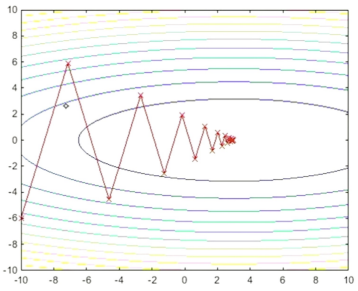


Figure 10.27: Momentum flattens out oscillations and speeds up convergence. Compare with Figure 10.26.

[13] ε is called the *tolerance* and is often set to 10^{-r}, where $r = 3, 4, 5, 6$.

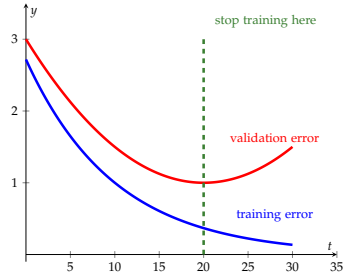


Figure 10.28: As we have seen in Figure 7.7, the validation error starts to increase when the model overfits. Stopping training right before this happens often prevents overfitting.

[14] In other words, for each hidden unit we flip a coin whose probability of heads is $1-p$, and cross it out if the coin brings heads.

and repeat the same process. An illustration of dropout regularization can be found in Figure 10.29 and one example of its effect on generalization error is shown in Figure 10.30

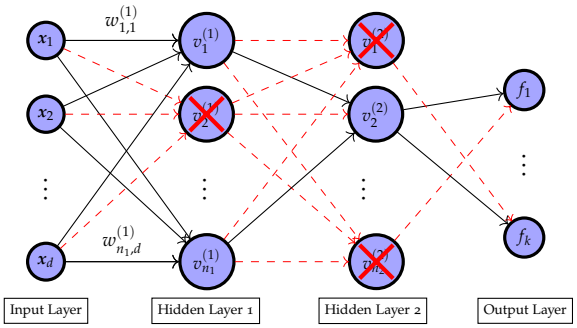


Figure 10.29: Some hidden units are deleted for the particular iteration of SGD. Their associated weights are frozen and they do not get updated in this iteration.

One question that emerges has to do with the hidden units that are used during test time. Should we keep all of them or do we still need to cross some out?

The answer is that during test time all the hidden units must be used. The omission of some of them during training was solely due to the large number of parameters, the need to avoid overfitting, and to make optimization faster. These problems do not appear during test time, so it is advisable to use the full set of learned features to make predictions.

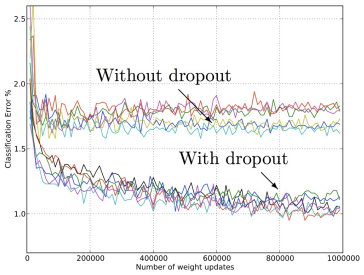


Figure 10.30: This example from Srivastava et al. (2014) illustrates how the use of dropout reduces the classification error.

However, we must be careful with the values of the weights. Looking at a single neuron (before applying the activation function), which normally is given by

$$z_j^{(l)} = \sum_{i=0}^{n_{i-1}} w_{j,i}^{(l)} h_i^{(l-1)} + b_j^{(l)},$$

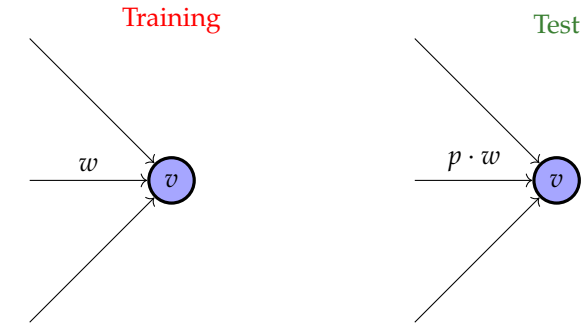
the forward pass now uses the equation

$$z_j^{(l)} = \sum_{i=0}^{n_{i-1}} w_{j,i}^{(l)} h_i^{(l-1)} \cdot \mathbb{I}_{C_i} + b_j^{(l)},$$

where $C_i = \{\text{unit } h_i^{(l-1)} \text{ is not deleted in this iteration}\}$. Given that $\mathbb{E}\left[\mathbb{I}_{C_i}\right] = \mathbb{P}\left(C_i\right) = p$, we obtain

$$\begin{aligned}
\mathbb{E}\left[z_j^{(l)}\right] &= \sum_{i=0}^{n_{i-1}} \mathbb{E}\left[w_{j,i}^{(l)} h_i^{(l-1)} \cdot \mathbb{I}_{C_i} + b_j^{(l)}\right] \\
&= \sum_{i=0}^{n_{i-1}} w_{j,i}^{(l)} h_i^{(l-1)} \cdot \mathbb{E}\left[\mathbb{I}_{C_i}\right] + b_j^{(l)} \\
&= \sum_{i=0}^{n_{i-1}} w_{j,i}^{(l)} h_i^{(l-1)} \cdot p + b_j^{(l)} \\
&= \left(p \cdot \boldsymbol{w}_j^{(l)}\right)^T \cdot \boldsymbol{h}^{(l-1)} + b_j^{(l)},
\end{aligned}$$

where $\boldsymbol{w}_j^{(l-1)}$ denotes the j-th row of $\boldsymbol{W}^{(l-1)}$. Hence, to compensate for using dropout, we need to multiply all weights with the constant p during test time. On average, this will scale down all units by the correct factor, as shown by the above equation.



(a) The unit and the weights are present with probability p.

(b) The unit and the weights are always present.

Figure 10.31: During test time we do not omit any hidden units but we scale all weights down by the constant p.

10.5.2 *Batch normalization*

In Subsection 10.4.2, we analyzed the problem of vanishing and exploding gradients and explained how important weight initialization is. Controlling the variance of the weights played a central role in restricting the norm of the gradients. Both weight-initialization schemes that were introduced in Subsection 10.4.2 were inspired by this idea - keeping weight variance low.

However, after the first few iterations of SGD, the updated weights have often nothing to do with their initial values. Hence, their mean might drift away from zero and their variance might become too large. [15] This will possibly bring the danger of vanishing and exploding gradients back up.

[15] This distribution shift during training is known as *internal covariate shift*.

One technique that addresses this problem is the standardization of the activation units, not only in the beginning but also during training. This procedure can be incorporated into the network as an extra layer, where the input will be the raw activation units and the output will be the standardized ones.

It is common practice to standardize, not all the units, but certain *batches* of them, which is why this technique is known as *batch normalization*. The standardization follows the classic rule – subtraction of the mean of the batch and normalization by its standard deviation. It is presented in Algorithm 7.

One might wonder what the role of β and γ is. As shown in Algorithm 7, batch normalization sets the mean of the units equal to zero and their variance equal to one, but this is not necessarily optimal - perhaps training would become more efficient if the mean and variance were equal to some other constants. For this reason, we add a final step in which $\widehat{h}_j$ is scaled by γ and shifted by β. These parameters were characterized as *learnable* because they are also included in the loss function and updated together with the weights during optimization.

Except for alleviating internal covariate shift, batch normalization also allows for larger learning rates, so it makes SGD converge

1: **Input:** Unprocessed batch $\{x_{i_1}, \ldots, x_{i_k}\}$ of units for some minibatch $S = \{i_1, \ldots, i_k\}$ of indices. ε and momentum α are hyperparameters.
2: **Learnable parameters**: β, γ
3: **Persistent buffers**: $\mu_{EMA}, \sigma^2_{EMA}$
4: **Normalization step (at training time)**:
- compute minibatch mean $\mu_S := \frac{1}{|S|} \sum_{j \in S} x_j$
- compute minibatch variance $\sigma^2_S := \frac{1}{|S|} \sum_{j \in S} (x_i - \mu_S)^2$
- update moving average mean $\mu_{EMA} = (1 - \alpha)\mu_{EMA} + \alpha \mu_S$
- update moving average variance $\sigma^2_{EMA} = (1 - \alpha)\sigma^2_{EMA} + \alpha \sigma^2_S$
- normalize each point: $\widehat{x}_j = \frac{x_j - \mu_S}{\sqrt{\sigma^2_S + \varepsilon}}$, where ε prevents the fraction from exploding.
- scaling and shifting: $\overline{x}_j = \gamma \widehat{x}_j + \beta$ for all $j \in S$.

5: **Normalization step (at test time)**:
- normalize each point using the moving averages computed during training: $\widehat{x}_j = \frac{x_j - \mu_{EMA}}{\sqrt{\sigma^2_{EMA} + \varepsilon}}$.
- scaling and shifting: $\overline{x}_j = \gamma \widehat{x}_j + \beta$ for all $j \in S$.

6: **Output:** $\overline{x}_j$ for all $j \in S$.

Algorithm 7: Batch Normalization.

faster [16]. Finally, it has a regularizing effect since it controls the magnitude of the activation units h_j.

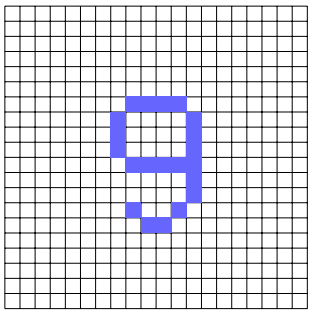
[16] When gradients do not explode, we can opt for larger learning rates. When they get very large, even small steps might make SGD diverge from the minimum.

An important implementation detail of batch normalization is how it is applied at test time. During inference, we often do not have access to an entire batch of data points to compute mean and variance statistics over. In order to still be able to use a model trained with batch normalization in situations like that, the algorithm keeps track of the moving average of mean and variances during training, such that they can be to normalize the activation units at test time.

10.6 Convolutional Neural Networks

As shown in Figure 10.12 and Figure 10.13, neural networks are extensively used in several computer vision tasks, like image classification or image captioning. The general approach in such tasks consists of two main steps.

Step 1: Each digital image is formed by an $m \times n$ grid of small squares, called pixels. Each pixel has its own color. We assign to each pixel a number, which denotes the pixel's intensity or saturation. In grayscale images, each pixel is assigned to a number in $[0, 1]$, from white to black. In RGB images, where each color is a combination of three *color channels*, red, green, and blue, each pixel corresponds to a triplet. Each entry of the triplet denotes the intensity of the respective color channel for that pixel.



Step 2: After the first step, the image is represented either as an $m \times n$ matrix (in greyscale images), with each entry denoting the color of the corresponding pixel, or as an $m \times n \times 3$ *cube*, with each triplet denoting again the RGB representation of each pixel. This *cube* is called a *tensor*. An example of a $5 \times 5 \times 3$ tensor is shown in Figure 10.32.

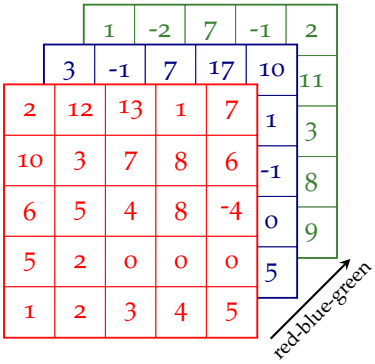
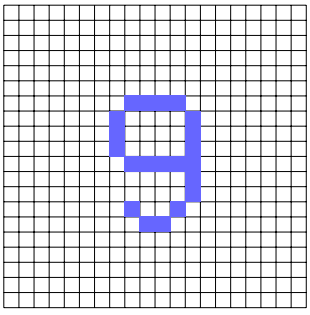


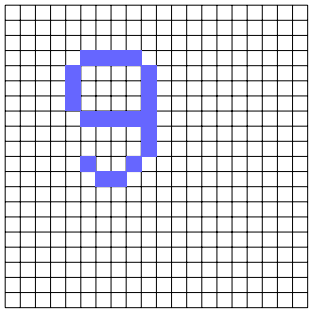
Figure 10.32: A tensor can be thought of as a cube, or as a triplet of matrices placed one behind the other. Each matrix corresponds to one of three color channels - red, blue, and green. For example, this tensor represents a picture whose upper leftmost pixel is a mix of red, blue, and green, with intensities 2,3, and 1 respectively.

In any case, each image is represented as a numerical structure, which can be given as input to a neural network. Image classification and image captioning are based on the analysis of these matrices and tensors.

The problem with this general strategy is that images from the same class might have completely different matrix or tensor representations. For example, consider the two images in Figure 10.33.



(a) This is a picture of a digit.



(b) This pictures the same digit, only shifted upwards and leftwards.

Figure 10.33: These two images should be put in the same class, but their tensor representations will be completely different since almost no matching pixels have the same color.

Although these two images depict the same digit, their tensor representations will be completely different. Indeed, many of the

blue cells in the left picture become white in the right one, and vice versa. This contradiction confuses the model, whose strategy is classifying images based on their tensor representation.[17]

[17] Formally, this is referred to as the 'translational-invariance', which in simple words, means if you move the position of the digit 9 in the image either to the left or to the right, the underlying meaning/label of the image does not vary.

The size of this problem is magnified if we think that, besides getting shifted, the content of an image could also be rotated, and scaled up or down. In all these cases, the tensor representation would completely change, making it impossible for the model to distinguish different digits based on their associated tensors.

10.6.1 A new NN architecture

One solution to this problem is the augmentation of the training set by adding extra images that are generated by spatial transformations (translations, rotations, scaling up/down) of the original ones. For instance, if the original training set contains the image shown in Figure 10.33(a), we could also add the image in Figure 10.33(b), or some other transformed versions of it.

This way, the model becomes more flexible and learns to recognize additional patterns. However, in large-scale problems, where the training dataset is already very large, augmenting it, even more, makes training very costly from a computational viewpoint.

Another potential solution would be to pre-process the images so that their content is centered and straightened. For simple images like Figure 10.33 this strategy might work, but this is harder for more complex ones where the objects' exact location and relative location might play a more critical role.

One key observation is that the color of each pixel is mostly related to the color of the nearby pixels and less to the color of the distant ones. This brings up the idea of designing a different architecture of neural networks, not like the fully connected NNs that were discussed in the previous sections.

The difference in the new architecture is that the units in the first few layers share some of the weights with each other. Also, each unit is connected only to a few of the units of the next layer and not all of them.

This new architecture is illustrated on a small scale in Figure 10.34. Notice that this figure only contains the first layer. As we move deeper, the network might also contain some fully connected layers.

Another evident advantage of the new architecture is that the number of parameters is smaller in comparison with the fully connected NNs. Indeed, the first layer of the network in Figure 10.34 contains only two parameters compared to $3 \times 4 = 12$ of its fully connected analog. This might help make optimization faster and prevent overfitting.

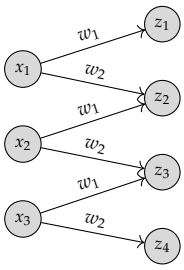


Figure 10.34: In this new architecture, each unit in the input layer interacts only with its nearby units. For example, z_2 depends only on x_1 and x_2. Just like in fully connected NNs, the units z_1, z_2, z_3, z_4 are passing through a nonlinear activation function φ before moving to the next layer.

10.6.2 *One-dimensional convolution*

In fully connected NNs like the one in Figure 10.11, each hidden layer $\boldsymbol{v}^{(l)}$ is related to the previous one through the relation

$$\boldsymbol{v}^{(l)} = \varphi\left(\boldsymbol{W}^{(l)}\boldsymbol{v}^{(l-1)}\right),$$

where $\boldsymbol{W}^{(l-1)}$ denotes the weights between these two hidden layers and φ is a nonlinear activation function. [18]

[18] Since $\boldsymbol{W}^{(l)}\boldsymbol{v}^{(l-1)}$ is a vector, we use the convention $\varphi(\boldsymbol{x}) = (\varphi(x_1), \ldots, \varphi(x_d))$.

In the new architecture, the algebraic relation between x_1, x_2, x_3 and z_1, z_2, z_3, z_4 is given by

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} w_1 & 0 & 0 \\ w_2 & w_1 & 0 \\ 0 & w_2 & w_1 \\ 0 & 0 & w_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \tag{10.4}$$

as it can be easily inferred from Figure 10.34. The difference is that the weight matrix now has a special structure[19]. To take advantage of this structure, we need to use the notion of *convolution*.

[19] For the interested readers, such matrices are called Toeplitz matrices.

Definition 10.5. Consider two vectors $\boldsymbol{w} \in \mathbb{R}^k$ and $\boldsymbol{x} \in \mathbb{R}^d$. Their *convolution* is defined to be the vector $\boldsymbol{z} \in \mathbb{R}^{k+d-1}$ with

$$z_i := \sum_{j=\max\{1,i-d+1\}}^{\min\{i,k\}} w_j x_{i-j+1}.$$

For example, if $\boldsymbol{w} = (w_1, w_2)^T$ and $\boldsymbol{x} = (x_1, x_2, x_3)^T$, then

$$\boldsymbol{w} * \boldsymbol{x} = \begin{bmatrix} w_1 \cdot x_1 \\ w_1 \cdot x_2 + w_2 \cdot x_1 \\ w_1 \cdot x_3 + w_2 \cdot x_2 \\ w_2 \cdot x_3 \end{bmatrix}.$$

It is now evident that Equation 10.4 can be rewritten in a more compact form as $\boldsymbol{z} = \boldsymbol{w} * \boldsymbol{x}$, where $\boldsymbol{w} = (w_1, w_2)^T$. Taking into account the nonlinear activation function that $\boldsymbol{z}$ will go through, we can now derive the relation of two consecutive layers in our new architecture:

$$\boldsymbol{v}^{(l)} = \varphi\left(\boldsymbol{w}^{(l)} * \boldsymbol{v}^{(l-1)}\right).$$

For that reason, a neural network that includes layers with this architecture is known as a *convolutional neural network*, commonly abbreviated as *CNN*.

Notice that, as mentioned earlier, a CNN does not necessarily contain *only* convolutional layers. It might also contain a few fully connected layers, especially as we move closer to the output.

Before looking at convolutions more closely from a mathematical point of view, let us first look at their effect at an intuitive level. As we discussed at the beginning of the section, the purpose of CNNs is to utilize the fact that there are strong associations between nearby pixels, as opposed to between distant ones. In other words, the weights in a CNN are used to *filter* the information that the network receives from an image. For that reason, the weights in CNNs are most commonly known as *filters*.

Figure 10.35 illustrates the effect of a certain type of filter, called a *Gaussian filter*. The visual result of applying this filter is a *blurred image*, where nearby pixels have the same color. In other words, the filter has a smoothening effect on the image, mitigating the color differences between neighboring pixels.

Figure 10.35: A Gaussian filter is a specific type of filter that has a smoothening effect. When convolved with such a filter, an image gets blurred, and neighboring pixels end up having the same color.

10.6.3 *Multidimensional convolution*

The CNNs that were introduced in the previous subsections are not sufficient for image analysis tasks. Indeed, these networks were only defined for one-dimensional input, whereas, as discussed at the beginning of this section, images are represented as two-dimensional (matrices) or three-dimensional objects (tensors).

Fortunately, convolutions can also be defined for higher-dimensional objects. However, due to a large number of indices, giving a formal definition for high-dimensional convolutions is not as useful as understanding visually how such convolutions operate. To do this, let us look at the following example:

Example 10.6. Consider the matrix

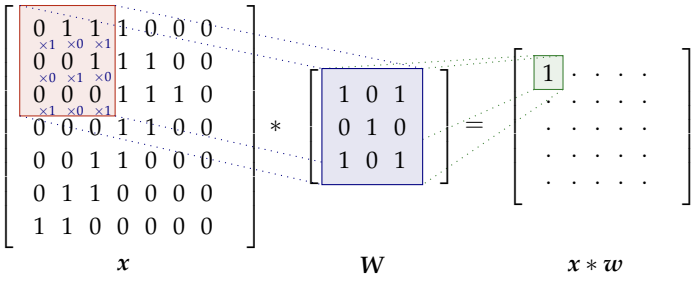
$$\boldsymbol{x} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{7\times 7}.$$

This can be thought of as the matrix representation of a grayscale image. Consider also the filter

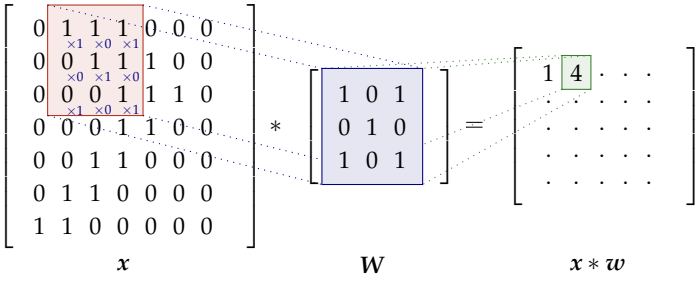
$$\boldsymbol{W} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{3\times 3}.$$

Their convolution is a matrix $x * w \in 5 \times 5$ defined in the following way:

- The entry on the position $(1,1)$ is computed as a *dot-product* of w and the upper leftmost 3×3 submatrix of x. [20]



- We now slide w to the right and execute the same operation. This gives us the element on the position $(1,2)$.

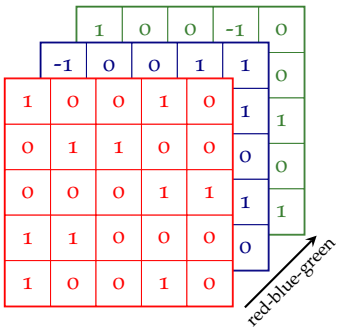


- We compute the rest of the entries of the convolution by sliding w across all 3×3 submatrices of x.

Example 10.6 illustrates the computation of the convolution $x * W$ when x and W are two-dimensional objects (matrices). This covers the case of grayscale images, whose numerical representation is an object of the above form.

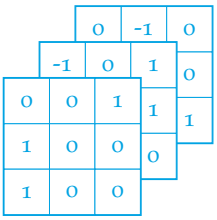
What about RGB images? Recall that such images are represented as tensors, which are three-dimensional objects. Fortunately, convolutions can be naturally generalized to this scenario as well. Again, instead of a formal definition, a numerical example is much more helpful in understanding the functioning of high-dimensional convolutions.

Example 10.7. Consider an RGB image whose tensor representation $x = \left(x_{ijk}\right) \in \mathbb{R}^{5\times5\times3}$ is the following:



Consider also the filter w shown below.

[20] To compute the dot-product, consider these 3×3 matrices as nine-dimensional vectors.



Just like in Example 10.6, we place w on top of the upper leftmost subtensor of x and compute their inner product (seeing these two $3 \times 3 \times 3$ tensors are seen as 27-dimensional vectors). This gives us the $(1,1)$-entry of the resulting matrix. We then slide w over x and compute the rest of the entries likewise. This is illustrated more clearly in Figure 10.36.

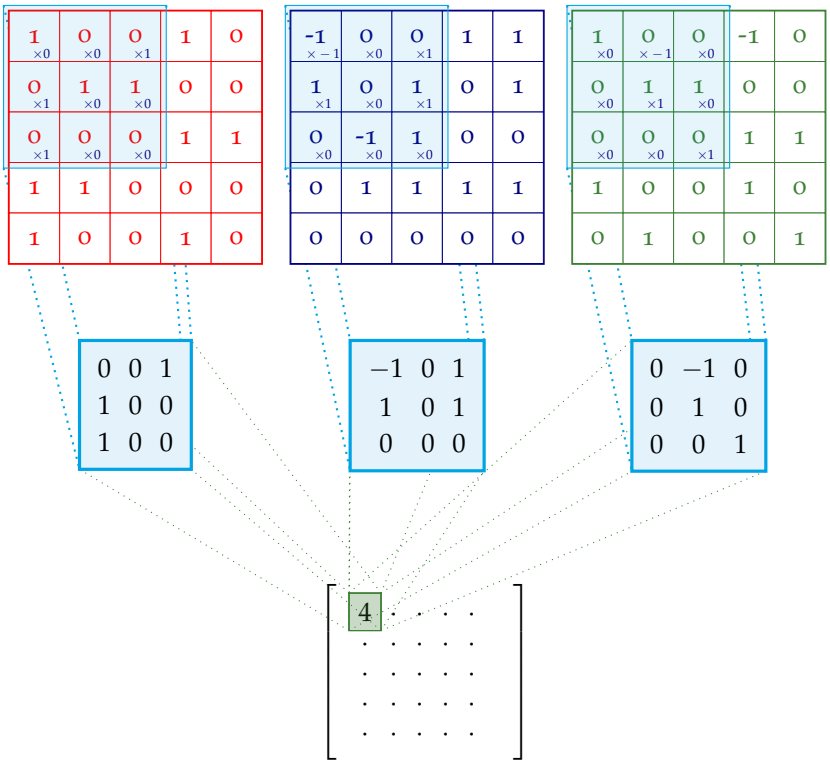


Figure 10.36: Like in two-dimensional convolutions, we place the tensor w *on top* of x and, considering them as 27-dimensional vectors, we compute their inner product.

As shown in Figure 10.36, convolving x with w results in a matrix. In practice, each unit x is convolved with multiple tensors $w_1, \ldots, w_m$ in each layer. Hence, we obtain k different matrices, which we glue together and form a tensor of depth k. As a result, a convolutional layer takes tensors as input and outputs tensors of possibly different dimensions.

This gives rise to the following question: how can we compute the output dimensions of a convolutional layer? To answer this question, consider an $n \times n$ RGB image and suppose that we would like to apply m filters of dimension $f \times f$.

One parameter that needs to be taken into account is the so-called *padding*. As seen in Figure 10.36, the pixels on the boundary of the image, or close to it, do not participate in the convolution as much as the ones in the middle of the image. For instance, as

we slide the filter $\boldsymbol{w}$ on top of the image tensor $\boldsymbol{x}$, the pixel at the middle of $\boldsymbol{x}$ (third row, third column) is always covered by $\boldsymbol{w}$, so it passes its information to many of the entries of the convolution. On the contrary, the top left pixel is only covered when $\boldsymbol{w}$ is at the position shown in Figure 10.36. When $\boldsymbol{w}$ moves around, it never covers this pixel again.

To address this situation, we often add a frame to each image, that is, a couple of white rows and columns around it (i.e., append zeros to image borders). This frame is called a *padding*. If we denote the dimension of the padding by p, then the dimension of the entire image is $(n + 2p) \times (n + 2p)$.

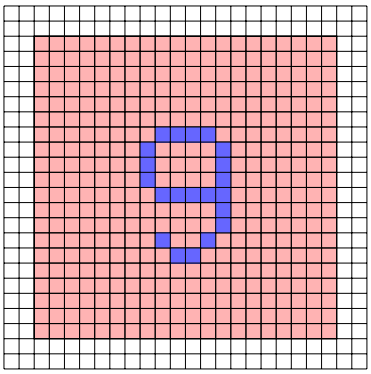
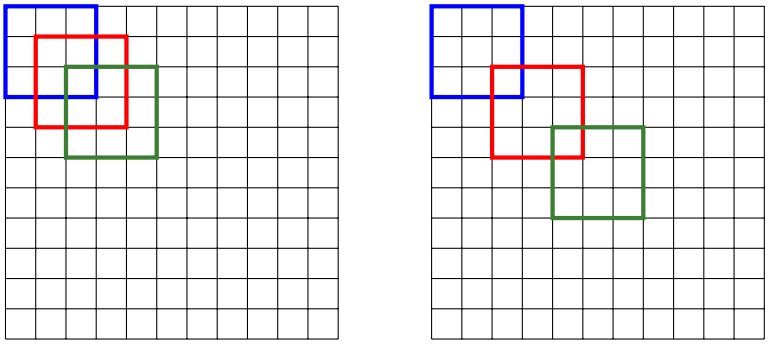


Figure 10.37: A 20×20 image with a padding of depth 2. The dimension of the entire image if 22×22.

Another parameter that we must take into account is the *stride*. In Example 10.6 and Example 10.7 we explained that, to compute the convolution $\boldsymbol{x} * \boldsymbol{w}$, the filter $\boldsymbol{w}$ makes unitary steps on top of the image tensor $\boldsymbol{x}$ until it exhausts all possible positions. However, we often want to reduce the dimensions of the output tensor, so the stepsize of the filter might be larger.

For example, the stepsize of the filter could be equal to 2. In that case, the filter stops at the positions shown in Figure 10.38. The stepsize of the filter is otherwise known as the *stride*.



(a) The stride is equal to 1. (b) The stride is equal to 2.

Figure 10.38: The left side shows a filter whose stride is equal to 1, like the ones shown in Example 10.6 and Example 10.7. The right one shows a filter with a stride equal to 2.

One basic prerequisite is that the stride matches the dimensions of the image and the filter. For example, in Figure 10.38, where the image has dimension $n + 2p = 11$, the filter has dimension $f = 3$, and the stride is equal to $s = 2$, the filter can move freely around the image, without having to cross its boundaries. On the contrary, in Figure 10.39, it is evident that the filter cannot cover the whole image.

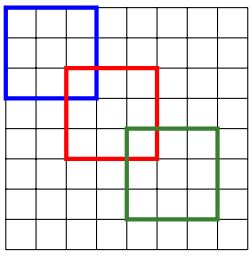


Figure 10.39: After reaching the position in green, the filter cannot move further. Hence, it cannot cover the last row and the last column.

In general, the required condition[21] is that the stride s divides $n + 2p - f$, where n, p, f are the dimension of the image, the depth of the padding, and the dimension of the filter respectively. In that

[21] Otherwise, one would process all that is possible, so we have similar formulas but with the floor operation.

case, it can be shown that the dimensions of the output tensor are

$$\left(\frac{n+2p-f}{s}+1\right)\times\left(\frac{n+2p-f}{s}+1\right)\times m,$$

where m is the number of filters applied to the image.

Bonus Material

The fact that s must divide $n+2p-f$ can be easily explained if we think of all the positions that the filter takes on top of the image. When we place it at the top left corner, it occupies the rectangle with corners

$$(1,1),(f,1),(1,f),(f,f).$$

We then move it s squares to the right, so now its corners are

$$(s+1,1),(f+s,1),(s+1,f),(f+s,f).$$

When we move it again, its corners will be

$$(2s+1,1),(f+2s,1),(2s+1,f),(f+2s,f).$$

Following the same pattern, after k steps the corners will be

$$(ks+1,1),(f+ks,1),(ks+1,f),(f+ks,f),$$

where k is a positive integer.

Eventually, the top right corner of the filter must reach the top right corner of the image, whose coordinates are $(n+2p,1)$. Hence, we must have

$$f+ks=n+2p$$

for some positive integer k. This equation can be rewritten as

$$k=\frac{n+2p-f}{s}.$$

This proves that s must divide $n+2p-f$. The first two dimensions of the output are determined by the number of shifted filters applied to the image. The shifted filters correspond to $s=0,...,s=k$. This is why the final dimension is equal to $k+1$. In total, we obtain the dimensions of the output tensor

$$\left(\frac{n+2p-f}{s}+1\right)\times\left(\frac{n+2p-f}{s}+1\right)\times m.$$

Part III

Unsupervised Learning

11

Clustering

In this chapter, we introduce the field of unsupervised learning, a branch of machine learning that deals with learning from unlabelled data. We then examine clustering, a central problem in unsupervised learning, analogous to classification in supervised learning. Specifically, we study the k-means algorithm, a model-based approach to perform clustering.

Roadmap

In Section 11.1, we present a concise overview of unsupervised learning. We introduce the task of clustering and explore different approaches in Section 11.2. In Section 11.3, we discuss the k-means algorithm together with Lloyd's heuristic. Section 11.4 provides the reader with a theoretical analysis of the performance guarantees of k-means, while highlighting important limitations. In Section 11.5 we compare several initialization schemes. In the final section, Section 11.6, we delve into the challenging task of selecting the most suitable number of clusters.

Learning Objectives

By the end of this chapter, you should be able to:

- Explain the difference between the branches of supervised and unsupervised learning.
- Describe the core concepts of clustering and outline the various existing approaches.
- Present the k-means clustering algorithm, along with Lloyd's heuristic.
- Show that k-means monotonically decreases its cost function.
- Illustrate the different challenges and limitations of k-means.
- Employ effective initialization techniques for the k-means algorithm.
- Understand the inherent difficulty of model selection in clustering and apply common heuristics to determine the optimal number of clusters.

11.1 Unsupervised Learning

Unsupervised learning is the subfield of machine learning that deals with learning without supervision, where the term "supervision" refers to the availability of labeled data. In other words, we consider a dataset $\mathcal{D} = \{\boldsymbol{x}_i \in \mathbb{R}^d : i = 1, \dots, n\}$ that lacks target values y_i's. By exploring the inherent structure of the input data, unsupervised learning techniques enable the extraction of meaningful patterns, facilitating further analysis and informed decision-making.

Unsupervised learning techniques exhibit a wide range of use cases in practical applications. These include learning compact data representations (i.e. compression), discovering latent variables and categorizing data into sub-groups, inducing new features for improved performance in downstream supervised tasks, identifying abnormal observations (i.e. anomaly detection), and facilitating exploratory data analysis (e.g. visualization).

Recall that supervised learning tasks fall under two main categories: regression (when the target values are continuous) and classification (for discrete labels). As it will become clear at the end of this and the following chapters, there exist unsupervised analogs to these two tasks. Namely, *dimensionality reduction* can be seen as the unsupervised counterparts of regression, and the same can be said about the relationship between *clustering* and classification. Additionally, another fundamental problem in unsupervised learning is *density estimation*, which involves estimating the underlying probability distribution of the data. This task is deeply connected to *generative modeling*, a branch of machine learning with high disruptive potential that has recently ignited heated debates about AI's role in society.

11.2 Standard Approaches to Clustering

The goal of *clustering* is to group data points into clusters based on their similarity. More precisely, similar points should be grouped together, while ensuring that dissimilar points belong to different clusters. The notion of similarity and dissimilarity depends on the specific context and objective of the clustering task. One common approach is to represent the data points in Euclidean space (e.g. $\mathbb{R}^d$ for a given dimension d) and to use the ℓ_2 distance as a measure of similarity between two points. Datapoints that are closer to each other are considered more similar, while those that are farther apart are considered dissimilar. It is worth noting that other metrics or similarity functions can also be employed, allowing clustering to be performed on various types of data beyond Euclidean spaces.

Once the similarity measure is determined, there still exist several different ways in which to proceed with the actual clustering. One widely used technique is *Hierarchical Clustering*. The algorithm

begins by forming an individual cluster for each data point, and then iteratively merges pairs of clusters that are considered to be the closest according to a similarity measure. A common example of such a measure is the average distance between a pair of points, with one point drawn from each cluster. This process results in a cluster hierarchy that can be visually represented by a dendrogram (or tree). Clusters of varying sizes can be obtained by cutting the dendrogram at different heights, yielding different levels of granularity. This provides some flexibility in choosing the final cluster configuration. Figure 11.1 further illustrates the method with a simple practical example.

Another rich family of clustering techniques is given by the so-called *Partitioning Approaches*, which rely on principles from graph theory. These methods involve constructing a (weighted) graph, where each data point serves as a node. The graph is then partitioned into clusters using various approaches. For instance, *Graph-Cut based approaches* aim to find cuts in the graph with certain optimality properties, while *Spectral Clustering* leverages spectral analysis tools to determine the clusters.

Both of these method classes are primarily intended for a fixed dataset, usually referred to as the training set. As a result, they typically do not provide a natural way to assign cluster membership to points in a test set. For this reason, we will shift our focus to *Model-Based Approaches* for the remainder of this chapter. These methods enable us to learn clusters from a training set and subsequently generalize the learned model to points sampled from the same distribution as the training set. Examples of model-based approaches include *Gaussian Mixture Models* (where the model is a probabilistic one) and *k-Means Clustering*, which will be the focus of the rest of this chapter.

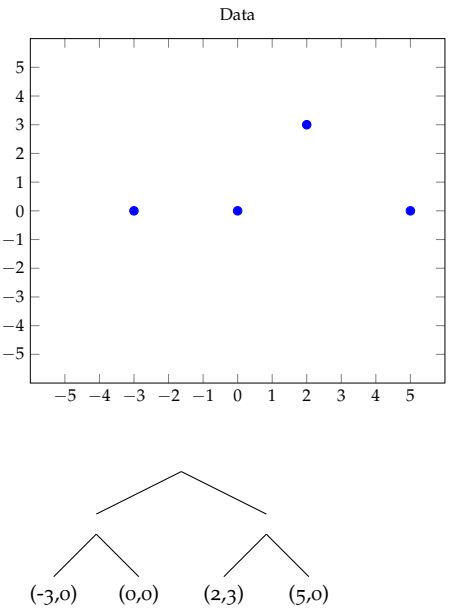


Figure 11.1: We demonstrate the application of hierarchical clustering on a toy dataset consisting of 4 points in $\mathbb{R}^2$. In this case, the distance between a cluster and a point is measured using the average distance of the point to the cluster points (this method is known as *Average Linkage Clustering*). The first step involves clustering the points $(-3, 0)$ and $(0, 0)$ together since they are the closest pair. In the second step, by considering all the point-to-point and cluster-to-point distances, we group together the points $\{(2, 3)\}$ and $\{(5, 0)\}$. This leads to the resulting tree diagram, as shown above.

11.3 k-Means Clustering and Lloyd's Heuristic

Consider the dataset $\mathcal{D} = \{\boldsymbol{x}_i \in \mathbb{R}^d : i = 1, \ldots, n\}$. In k-means clustering, each cluster is represented by a single point, the center of the cluster. In what follows, we will denote the center of the j-th cluster by $\boldsymbol{\mu}_j \in \mathbb{R}^d$ and the cluster assignment for $\boldsymbol{x}_i$ by $z_i \in 1, \ldots, k$. Given the cluster centers $\boldsymbol{\mu}_j$, the point $\boldsymbol{x}_i \in \mathbb{R}^d$ is assigned to the cluster whose center is the closest to it. In other words, z_i, the cluster assignment for the i-th point, is determined by the following expression:

$$z_i = \arg \min_{j=1,\ldots,k} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|_2. \tag{11.1}$$

It is worth noting that this assignment rule applies not only to the points in the original dataset, but also to any point $\boldsymbol{x}$ in $\mathbb{R}^d$. This induces a partition of $\mathbb{R}^d$, called a *Voronoi partition*, that enables us to potentially extend the learned clustering model to newly sampled points.

But how are the cluster centers $\boldsymbol{\mu}_j$ selected? Let $\boldsymbol{\mu} = (\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k)^\top$ denote the vector of cluster centers. A desirable property for $\boldsymbol{\mu}$ is to minimize the sum of squared distances between each point and its assigned cluster center. This can be expressed as the following objective:

$$\hat{R}(\boldsymbol{\mu}) = \sum_{i=1}^{n} \min_{j \in \{1,\dots,k\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|_2^2 = \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|_2^2. \tag{11.2}$$

While the squared distance between a data point and its cluster center is a convex function, the problem of finding the optimal centroids to use as cluster centers, defined by

$$\arg\min_{\boldsymbol{\mu}} \hat{R}(\boldsymbol{\mu}), \tag{11.3}$$

is non-convex. In fact, it is known to be NP-hard, which means that finding the global optimal solution is often computationally infeasible. This is why practical implementations of k-means clustering typically rely on heuristic approaches.

The most well-known heuristic for k-means clustering is Lloyd's heuristic, which consists of two iterative steps performed until convergence. In the first step, the cluster assignments are updated while assuming the cluster centers are known. In the second step, the new cluster assignments are used to compute improved estimates of the cluster centers. It can be easily shown that the optimal center $\boldsymbol{\mu}_j$ for the j-th cluster can be obtained by taking the mean of the points in that cluster, i.e.

$$\boldsymbol{\mu}_j = \frac{1}{n_j} \sum_{i:\, z_i = j} \boldsymbol{x}_i, \tag{11.4}$$

where $n_j = |\{i = 1, \dots, n : z_i = j\}|$ denotes the number of points in the j-th cluster. Observe that this coincides with the *centroid* (or center of mass) of the cluster. Incidentally, note that the name of the algorithm comes from the fact that at every step the *means* of the points in each cluster are computed.

Bonus Material

Lemma 11.1. *Given a set of points $\mathcal{D} = \{\boldsymbol{x}_i \in \mathbb{R}^d : i = 1, \dots, n\}$, the cost function*

$$\mathcal{L}(\boldsymbol{\mu}) = \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}\|_2^2 \tag{11.5}$$

is minimized by the center of mass of $\mathcal{D}$, i.e. by

$$\boldsymbol{\mu}^* = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i. \tag{11.6}$$

Proof. Observe that,

$$\mathcal{L}(\mu) = \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}^* + \boldsymbol{\mu}^* - \boldsymbol{\mu}\|_2^2 \tag{11.7}$$

$$= \frac{1}{n}\sum_{i=1}^{n} [(\boldsymbol{x}_i - \boldsymbol{\mu}^*) + (\boldsymbol{\mu}^* - \boldsymbol{\mu})]^\top [(\boldsymbol{x}_i - \boldsymbol{\mu}^*) + (\boldsymbol{\mu}^* - \boldsymbol{\mu})] \tag{11.8}$$

$$= \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}^*\|_2^2 + \frac{2}{n}\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu}^*)\top(\boldsymbol{\mu}^* - \boldsymbol{\mu}) + \|\boldsymbol{\mu}^* - \boldsymbol{\mu}\|_2^2 \tag{11.9}$$

$$= \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}^*\|_2^2 + \|\boldsymbol{\mu}^* - \boldsymbol{\mu}\|_2^2, \tag{11.10}$$

since

$$\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu}^*) = \sum_{i=1}^{n} \boldsymbol{x}_i - n\boldsymbol{\mu}^* = 0. \tag{11.11}$$

Note that $\|\boldsymbol{\mu}^* - \boldsymbol{\mu}\|_2^2$ is always non-negative. Hence, we conclude that for all $\boldsymbol{\mu}$ it holds that,

$$\mathcal{L}(\boldsymbol{\mu}) \geq \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}^*\|_2^2 = \mathcal{L}(\boldsymbol{\mu}^*). \tag{11.12}$$

□

Define $\boldsymbol{z}^{(t)} = \left(z_1^{(t)}, \ldots, z_n^{(t)}\right)^\top$, where $z_i^{(t)}$ denotes the cluster assignment of the i-th data point $\boldsymbol{x}_i$ at iteration t. Similarly, let $\boldsymbol{\mu}^{(t)} = \left(\boldsymbol{\mu}_1^{(t)}, \ldots, \boldsymbol{\mu}_k^{(t)}\right)^\top$ be the vector of cluster centers at iteration t and $n_j^{(t)} = \left|\{i = 1, \ldots, n \colon z_i^{(t)} = j\}\right|$ the total number of points in the j-th cluster at iteration t. Then, the pseudo-code for the algorithm is as follows.

Algorithm 8: Lloyd's heuristic

Initialization: $\boldsymbol{\mu}^{(0)} \leftarrow \left(\boldsymbol{\mu}_1^{(0)}, \ldots, \boldsymbol{\mu}_k^{(0)}\right)^\top$
$t \leftarrow 1$
repeat
  $z_i^{(t)} \leftarrow \arg\min_{j \in \{1,\ldots,k\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t-1)}\|_2, \qquad i = 1, \ldots, n$
  $\boldsymbol{\mu}_j^{(t)} \leftarrow \frac{1}{n_j^{(t)}} \sum_{i \colon z_i^{(t)} = j} \boldsymbol{x}_i, \qquad j = 1, \ldots, k$
  $t \leftarrow t + 1$
until convergence

Observe that the computational complexity of each iteration is of order $\mathcal{O}(nkd)$. Figure 11.2 further illustrates the algorithm for a toy-example.

11.4 Convergence Analysis and Limitations

In the following section, we examine the main theoretical guarantees and limitations of k-means clustering.

One important property of the k-means algorithm is that, in each iteration, the average squared distance between a point and its as-

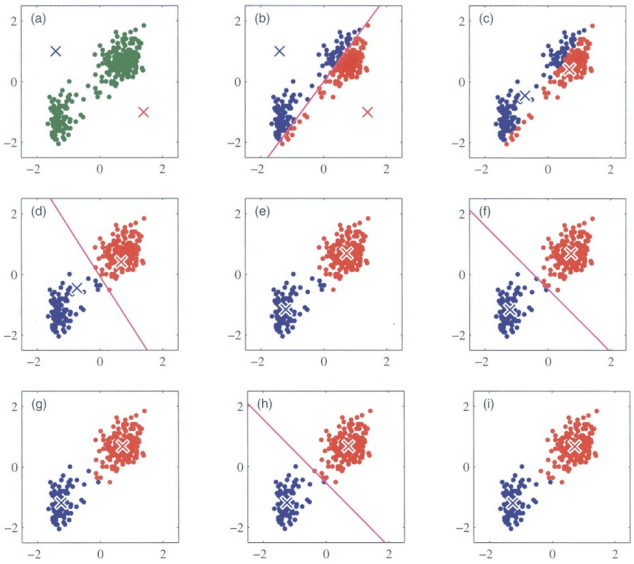


Figure 11.2: First 4 rounds of the k-means algorithm. The centroids are drawn as crosses and the cluster memberships are represented by the color of the points. The straight line depicts the partition of $\mathbb{R}^2$ induced by the clustering model (source: Bishop and Nasrabadi 2006).

signed cluster center does not increase. This fact can be formally stated as follows.

Theorem 11.2. *Write* $\boldsymbol{z} = (z_1, \ldots, z_n)^\top$ *and* $\boldsymbol{\mu} = (\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_n)^\top$, *and define*

$$\hat{R}(\boldsymbol{\mu}, \boldsymbol{z}) = \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|_2^2. \tag{11.13}$$

Then, at any iteration $t = 1, 2, \ldots$, *it holds that:*

$$\hat{R}(\boldsymbol{\mu}^{(t)}, \boldsymbol{z}^{(t)}) \geq \hat{R}(\boldsymbol{\mu}^{(t+1)}, \boldsymbol{z}^{(t+1)}). \tag{11.14}$$

Proof. The proof can be decomposed into two parts mirroring the two steps of the algorithm.

For the first part, observe that

$$\hat{R}(\boldsymbol{\mu}^{(t)}, \boldsymbol{z}^{(t)}) \geq \min_{\boldsymbol{z}} \hat{R}(\boldsymbol{\mu}^{(t)}, \boldsymbol{z}) = \hat{R}(\boldsymbol{\mu}^{(t)}, \boldsymbol{z}^{(t+1)}), \tag{11.15}$$

since by definition we have that $z_i^{(t+1)} = \arg\min_{j \in \{1,\ldots,k\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t)}\|_2$.

Similarly, it holds that

$$\hat{R}(\boldsymbol{\mu}^{(t)}, \boldsymbol{z}^{(t+1)}) \geq \min_{\boldsymbol{\mu}} \hat{R}(\boldsymbol{\mu}, \boldsymbol{z}^{(t+1)}) = \hat{R}(\boldsymbol{\mu}^{(t+1)}, \boldsymbol{z}^{(t+1)}). \tag{11.16}$$

This can be shown by rewriting $\hat{R}(\boldsymbol{\mu}, \boldsymbol{z}^{(t+1)})$ as

$$\hat{R}(\boldsymbol{\mu}, \boldsymbol{z}^{(t+1)}) = \sum_{j=1}^{k} \sum_{i:\, z_i^{(t+1)} = j} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|_2^2, \tag{11.17}$$

and observing that, by Lemma 11.1, each term of the second sum is minimized by setting $\boldsymbol{\mu}_j$ equal to $\boldsymbol{\mu}_j^{(t+1)}$, where

$$\boldsymbol{\mu}_j^{(t+1)} = \frac{1}{n_j^{(t+1)}} \sum_{i:\, z_i^{(t+1)} = j} \boldsymbol{x}_i. \tag{11.18}$$

To conclude the argument, note that (11.14) can be immediately obtained by combining (11.15) and (11.16).

□

The algorithm is therefore guaranteed to converge, but only to a *locally* optimal solution. For this reason, the performance of k-means strongly depends on the initialization of the cluster centers. Different initialization schemes are discussed in Section 11.5.

Additionally, the number of iterations required to converge can theoretically be exponential, even in lower dimensions. In practice, however, the algorithm often converges quickly.

It is important to note that k-means can lead to suboptimal solutions when the clusters have non-spherical shapes, as illustrated by Figure 11.3. This is due to the strong inductive bias introduced by using the Euclidean norm to measure distances between data points and cluster centers. To overcome this limitation, kernel-based methods can be employed.

Finally, model selection—and more specifically selecting the appropriate number of clusters k—remains an open problem. Section 11.6 presents various approaches to address this issue.

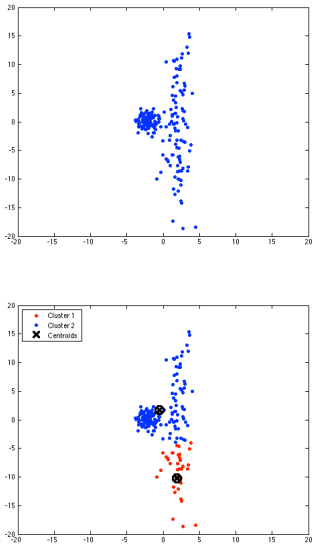


Figure 11.3: Due to the elongated shape of one of the clusters, the dataset is partitioned in an unnatural way. In this case, the horizontal axis is more informative than the vertical axis when it comes to separating the two clusters. This fact is not captured by the usual Euclidean distance. To address this limitation, kernelized methods offer a natural solution.

11.5 Initialization Schemes and k-means++

Since convergence to the global optimum is not guaranteed in k-means clustering, the initialization of cluster centers plays a crucial role in the performance of the algorithm. Several initialization schemes have been proposed to tackle this challenge. These heuristics often involve a random component, where cluster centers are sampled from a specific distribution defined over the points in the dataset. By running the algorithm multiple times with different initial configurations, it is possible to select the solution that yields the most satisfactory outcome. This approach helps mitigate the sensitivity of k-means to the initial placement of cluster centers and increases the chances of finding a better solution.

A simple approach for initializing cluster centers in k-means is to consider the *uniform* distribution over the given data points. However, this approach may lead to suboptimal results when dealing with unbalanced clusters. In such cases, where some clusters contain significantly more points than others, the uniform distribution may result in multiple centers being sampled from the larger clusters while the smaller clusters are left without any initial center. This imbalance in initialization can impact the subsequent clustering process and potentially lead to biased or inaccurate results.

To address this issue, an alternative scheme called the *furthest point heuristic* has been introduced. The first cluster center is drawn

uniformly at random among the points in the dataset. Each successive centroid is chosen sequentially, considering the maximum distance to the nearest cluster center selected so far. By prioritizing points that are farthest from existing centers, this heuristic attempts to ensure a more balanced distribution of cluster centers. It often works well in practice, even though it can be sensitive to outliers.

The *k-means++* heuristic—introduced by Arthur and Vassilvitskii (2007)—improves upon previous initialization methods by introducing additional randomness. The first centroid is again selected uniformly at random. The remaining cluster centers are sampled sequentially with probability proportional to the distance to the closest centroid. This means that data points that are further away from all of the current centroids are more likely to be selected as centers for the remaining clusters. At the same time, an isolated outlier that happens to be the furthest point from all of the clusters will not be selected as the next centroid with probability 1, as would have been the case with the furthest point heuristic. By incorporating additional randomness, the *k*-means++ heuristic provides a more robust and balanced initialization scheme. The procedure can be summarized as follows.

1. Sample the first cluster center uniformly at random among the points in the dataset:

$$\boldsymbol{\mu}_1^{(0)} = \boldsymbol{x}_i \in \mathcal{D}, \quad i \sim \text{Uniform}(\{1, \dots, n\}). \tag{11.19}$$

2. For $j = 2, \dots, k$, draw the j-th cluster center according to the following rule:

$$\boldsymbol{\mu}_j^{(0)} = \boldsymbol{x}_i \in \mathcal{D}, \qquad i \sim p(i) \propto \min_{m \in \{1,\dots,j-1\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_m\|_2^2. \tag{11.20}$$

It can be shown that the expected cost of *k*-means++ is optimal within a factor of $\mathcal{O}(\log k)$, i.e.

$$\hat{R}(\boldsymbol{\mu}_{k-\text{means}++}) \leq \mathcal{O}(\log k) \min_{\mu} \hat{R}(\boldsymbol{\mu}). \tag{11.21}$$

Nowadays, most libraries have adopted the *k*-means++ heuristic as their default initialization scheme, including *scikit-learn*.

Implementation

```
from sklearn.cluster import KMeans
# Create a KMeans instance specifying the desired number of clusters and the initialization scheme
kmeans = KMeans(n_clusters=2, init="k-means++")
# Fit the KMeans model to the training set X_train
kmeans.fit(X_train)
# Assign cluster memberships to the test set X_test
predictions = kmeans.predict(X_test)
```

11.6 Model Selection: choosing k

Model selection in clustering primarily involves finding the optimal number of clusters k. Unfortunately, this turns out to be a challenging task. Unlike supervised learning, where increasing the complexity of the model may lead to overfitting, in clustering, both the training and validation loss will decrease as k increases. This characteristic, illustrated in Figure 11.4, makes traditional cross-validation methods unsuitable. To tackle this challenge, several alternative approaches have been introduced in the literature.

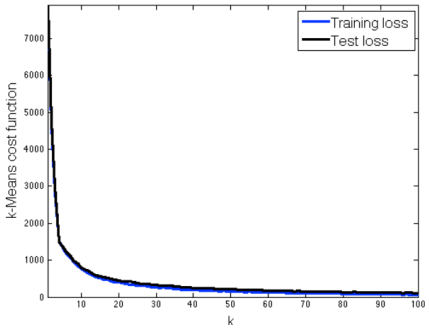


Figure 11.4: As opposed to what typically happens in the context of supervised learning, the training and validation loss both monotonically decrease when the number of clusters increases. This prevents cross-validation from being used for model selection in clustering.

A common heuristic is to plot the cost function $\hat{R}$ against the parameter k, as shown in Figure 11.5. This plot typically exhibits a concave shape, indicating diminishing returns: initially, increasing k leads to a sharp decrease in the cost function, but beyond a certain point, the benefit of adding more clusters to the model becomes negligible. The idea is to identify the number of clusters around the *elbow* (or kink) of the curve, where further increasing k would not significantly reduce the cost function.

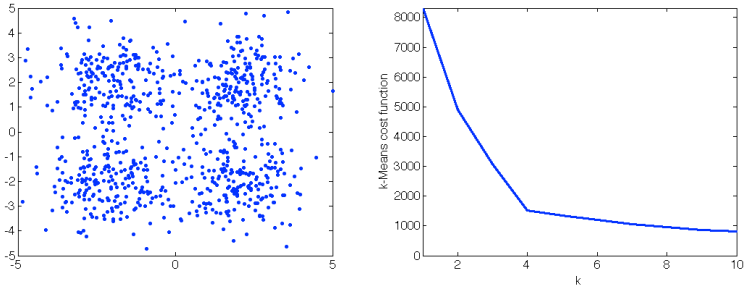


Figure 11.5: On the right, the k-means cost function $\hat{R}$ is plotted against the number of clusters k. Observe that the elbow is located around the value k = 4, which seems appropriate when looking at the scatterplot of the dataset on the left.

A related approach to model selection in clustering involves adding a regularization term to the cost function, which penalizes increasing the model complexity, i.e. the number of clusters. The modified loss function can be written as follows:

$$\hat{R}(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k) + \lambda \cdot k, \tag{11.22}$$

where $\lambda \geq 0$ is a hyperparameter that balances the trade-off between the goodness of fit and the model complexity. Although there is no principled way to determine the optimal value for λ, it

is relatively easier to select compared to directly choosing the number of clusters k. This is because multiple values of λ can lead to the same clustering solution, making this approach more robust. Figure 11.6 compares this regularization approach with the elbow method, suggesting their formal equivalence.

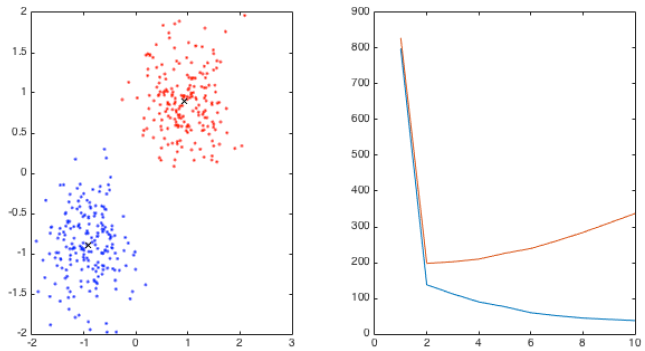


Figure 11.6: The plot illustrates the empirical cost function $\hat{R}$ (blue curve) and the regularized loss (orange curve). Both the elbow method and the regularization heuristic suggest selecting a value for k equal to 2, which arguably corresponds to the correct number of clusters for this dataset.

It is worth mentioning that there exist several other approaches that are based, for instance, on information-theoretic concepts such as stability and informativeness. Within this context, two commonly used criteria are the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC).

12

Principal Component Analysis

In this chapter, we delve deeper into the world of unsupervised learning techniques by introducing the field of dimensionality reduction. While clustering can be viewed as the unsupervised counterpart of classification, dimensionality reduction can be regarded as the unsupervised equivalent of regression. Our primary focus will be on one of the most widely used techniques in this field, Principal Component Analysis (PCA).

Roadmap

In Section 12.1, we introduce the field of dimensionality reduction and provide motivating examples of its applications. We then dive into Principal Component Analysis (PCA), which is not only an effective method on its own, but also has several connections to more advanced unsupervised learning methods. Section 12.2 presents a formal derivation of the solution to the PCA problem for the one-dimensional case, while Section 12.3 extends this result to the general case and discusses various strategies to select the number of components. The connection between PCA and the Singular Value Decomposition (SVD) is explored in Section 12.4. Finally, in Section 12.5, we introduce a general framework that unifies both PCA and k-means clustering under a common formalism.

Learning Objectives

By the end of this chapter, you should be able to:

- Explain the motivation behind dimensionality reduction techniques and provide examples of their applications.
- Derive the formal solution for the one-dimensional case and extend this result to the general case.
- Identify the first principal components given the scatterplot of a dataset.
- Select the appropriate number of components using different strategies based on the specific application at hand.
- Show how the Singular Value Decomposition (SVD) can be used to compute the principal components.
- Illustrate the relationship between PCA and k-means clustering through the lens of a more general framework.

12.1 Dimensionality Reduction

Given the dataset $x_1, \ldots, x_n \in \mathbb{R}^d$, the goal of dimensionality-reduction techniques is to find low-dimensional representations (or *embeddings*) $z_1, \ldots, z_n \in \mathbb{R}^k$, typically with $k \ll d$.

These methods have a broad range of practical applications (see for instance Figure 12.1). Clearly, compressing the data can be useful to save memory and speed up computations. Furthermore, by reducing the dimensionality of the data to $k \leq 3$, we can produce intuitive plots and potentially gain insights through exploratory data analysis. Finally, it is worth noting that dimensionality reduction techniques can sometimes improve the performance of downstream supervised tasks, such as regression or classification, by removing noisy or irrelevant features and simplifying the model. This can lead to improved interpretability and generalization performance. Additionally, dimensionality reduction methods can help identify meaningful structures in the data that can be leveraged to extract new and potentially more informative features in an unsupervised manner.

The effectiveness of dimensionality reduction techniques can be attributed to the *manifold hypothesis*, which posits that high-dimensional data often lie on or near a lower-dimensional manifold. This implies that it is possible to represent the data in a lower-dimensional space while preserving much of its structure. The manifold hypothesis is supported by the fact that high-dimensional data frequently exhibit redundant and highly correlated features. Think for example of images of faces. These are characterized by several symmetries, which can be understood as constraints on the support of their distribution. By leveraging this structure in the data, dimensionality reduction techniques can effectively reduce the complexity of the data while retaining a substantial amount of information.

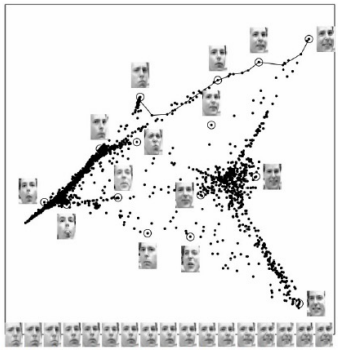


Figure 12.1: In this example (Roweis and Saul 2000), dimensionality-reduction techniques have been applied to gray-scale images of faces. The result is a two dimensional embedding that exhibits some interesting properties in terms of semantics. One can clearly see that images depicting different facial expressions get mapped to different locations. Furthermore, slowly varying the coordinates of a point on this two dimensional embedding results in a smooth transition between facial expressions.

In what follows we will focus on *model-based approaches* to dimensionality reduction. In other words, we will fit a model to our training data, with the expectation that this model will also provide useful representations for data freshly sampled from the same distribution as the training set. As is customary in this course, we will begin by presenting a *linear* method for dimensionality reduction, while *non-linear* approaches based on kernels and neural networks will be explored in the subsequent chapter.

12.2 Principal Component Analysis with $k = 1$

For simplicity, let us first consider the one-dimensional case where $k = 1$. Additionally, assume, without loss of generality, that the data is centered, i.e. $\sum_{i=1}^n x_i = 0$ (if this is not the case, subtract the empirical mean $\mu = n^{-1} \sum_{i=1}^n x_i$ from the data). We are interested in

representing any given data point $\boldsymbol{x} \in \mathbb{R}^d$ as a point on a line. In other words, we will approximate $\boldsymbol{x}$ as:

$$\boldsymbol{x} \approx z\boldsymbol{w}, \tag{12.1}$$

where $z \in \mathbb{R}$ is the one-dimensional representation (or *encoding*) of x, and the line associated with $\boldsymbol{w} \in \mathbb{R}^d$ corresponds to our one-dimensional model. The idea is to select z and $\boldsymbol{w}$ so that a certain criterion is optimized. In Principal Component Analysis (or PCA) we aim to minimize the *reconstruction error*, i.e.

$$\|\boldsymbol{x} - z\boldsymbol{w}\|_2^2. \tag{12.2}$$

Therefore, dimensionality-reduction via PCA can be viewed as the problem of finding the *linear* compression algorithm that provides optimal reconstruction of the inputs (as measured by the Euclidean distance). Figure 12.2 illustrates the method for a two-dimensional dataset. As a side note, observe that, in a way, the original data-point $\boldsymbol{x}$ assumes the role of a surrogate label in the reconstruction error loss. This clever trick of using the data itself for supervision is a recurrent theme in unsupervised learning.

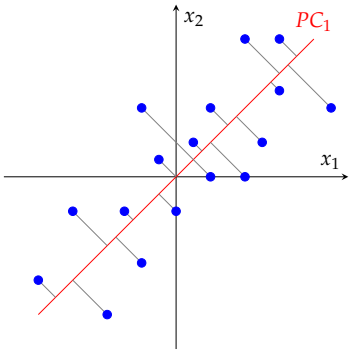


Figure 12.2: Example of PCA with $d = 2$ and $k = 1$. Note that the principal component line will in general be different from the regression line. This is because the regression line measures the error distances vertically, while the principal component line measures them perpendicularly to itself.

Note that the current parametrization of the solution gives rise to an identifiability issue. For instance, given any non-zero constant α, the pairs $(z, \boldsymbol{w})$ and $(z' = \alpha z, \boldsymbol{w}' = \alpha^{-1}\boldsymbol{w})$ are equivalent. Therefore, in order to ensure uniqueness, we restrict our search over the space of normalized vectors, i.e. $\boldsymbol{w} \in \mathbb{R}^d$ such that $\|\boldsymbol{w}\|_2 = 1$.

We can now instantiate the PCA problem in the setting where $k = 1$. The goal is to minimize the reconstruction error on the dataset $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ jointly over $\boldsymbol{w}$ and $z_1, \ldots, z_n$. That is, if we denote with $\boldsymbol{w}^*$ and $z_1^*, \ldots, z_n^*$ the optimal solutions, we are interested in solving the following optimization problem:

$$\boldsymbol{w}^*, z_1^*, \ldots, z_n^* = \arg \min_{\substack{\boldsymbol{w} \in \mathbb{R}^d : \|\boldsymbol{w}\|_2 = 1 \\ z_1, \ldots, z_n \in \mathbb{R}}} \sum_{i=1}^{n} \|\boldsymbol{x}_i - z_i \boldsymbol{w}\|_2^2 \tag{12.3}$$

Observe that for any given vector $\boldsymbol{w}$, the optimal representation is given by the orthogonal projection of the data-point onto the direction of $\boldsymbol{w}$, i.e.

$$z_i^* = \boldsymbol{w}^\top \boldsymbol{x}_i. \tag{12.4}$$

This allows us to simplify 12.3 so that only the vector $\boldsymbol{w}$ appears:

$$\boldsymbol{w}^* = \arg \min_{\boldsymbol{w} \in \mathbb{R}^d : \|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i\|_2^2. \tag{12.5}$$

We can now solve for the optimal $\boldsymbol{w}^*$.

$$\begin{aligned}
\boldsymbol{w}^* &= \arg \min_{\|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i\|_2^2 && (12.6)\\
&= \arg \min_{\|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} \left(\boldsymbol{x}_i - \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i\right)^\top \left(\boldsymbol{x}_i - \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i\right) && (12.7)\\
&= \arg \min_{\|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} \boldsymbol{x}_i^\top \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i - 2\boldsymbol{x}_i^\top \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i + \boldsymbol{x}_i^\top \boldsymbol{x}_i && (12.8)\\
&= \arg \min_{\|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} -\boldsymbol{x}_i^\top \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i + \boldsymbol{x}_i^\top \boldsymbol{x}_i && (12.9)\\
&= \arg \max_{\|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} \left(\boldsymbol{w}^\top \boldsymbol{x}_i\right)^2, && (12.10)
\end{aligned}$$

where we have used the fact that $\boldsymbol{w}^\top \boldsymbol{w} = \|\boldsymbol{w}\|_2^2 = 1$. This tells us that the optimal $\boldsymbol{w}^*$ is aligned with the direction that maximizes the empirical variance of the projected data, i.e. $n^{-1} \sum_{i=1}^{n} \|\boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i\|_2^2 = n^{-1} \sum_{i=1}^{n} \left(\boldsymbol{w}^\top \boldsymbol{x}_i\right)^2$. Figure 12.3 illustrates further this alternative perspective.

Making use of the fact that

$$(\boldsymbol{w}^\top \boldsymbol{x}_i)^2 = (\boldsymbol{w}^\top \boldsymbol{x}_i)(\boldsymbol{w}^\top \boldsymbol{x}_i)^\top = \boldsymbol{w}^\top (\boldsymbol{x}_i \boldsymbol{x}_i^\top) \boldsymbol{w}, \tag{12.11}$$

we get that

$$\begin{aligned}
\boldsymbol{w}^* &= \arg \max_{\|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} \left(\boldsymbol{w}^\top \boldsymbol{x}_i\right)^2 && (12.12)\\
&= \arg \max_{\|\boldsymbol{w}\|_2 = 1} \sum_{i=1}^{n} \boldsymbol{w}^\top (\boldsymbol{x}_i \boldsymbol{x}_i^\top) \boldsymbol{w} && (12.13)\\
&= \arg \max_{\|\boldsymbol{w}\|_2 = 1} \boldsymbol{w}^\top \left(\sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^\top \right) \boldsymbol{w} && (12.14)\\
&= \arg \max_{\|\boldsymbol{w}\|_2 = 1} \boldsymbol{w}^\top \Sigma \boldsymbol{w}, && (12.15)
\end{aligned}$$

where $\Sigma = n^{-1} \sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^\top = n^{-1} X^\top X$ is the *empirical covariance matrix* (recall that we have assumed the data to be centered). Observe that, since Σ is a symmetric and positive semi-definite, it admits the following eigen-decomposition:

$$\Sigma = \sum_{i=1}^{d} \lambda_i \boldsymbol{v}_i \boldsymbol{v}_i^\top, \tag{12.16}$$

where the eigenvalues are $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d \geq 0$ and the eigenvectors $\boldsymbol{v}_i$'s form an orthonormal basis of $\mathbb{R}^d$.

We can finally present the PCA solution for the one-dimensional case.

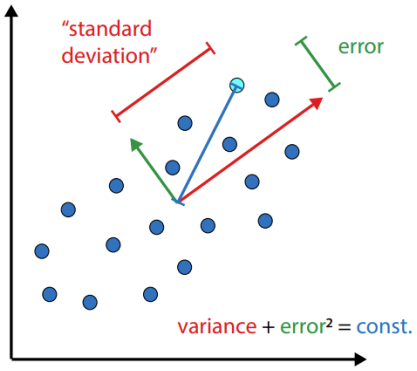


Figure 12.3: Since PCA consists of an orthogonal projection, minimizing the reconstruction error is perfectly equivalent to maximizing the empirical variance of the projected data.

Lemma 12.1. *For the PCA problem*

$$\boldsymbol{w}^* = \arg\max_{\|\boldsymbol{w}\|_2=1} \boldsymbol{w}^\top \Sigma \boldsymbol{w}, \tag{12.17}$$

it holds that

$$\boldsymbol{w}^* = \boldsymbol{v}_1, \tag{12.18}$$

i.e. the optimal vector $\boldsymbol{w}^*$, which we will refer to as the first principal component vector*, is given by the eigenvector of Σ associated with the largest eigenvalue.*

Proof. Since the eigenvectors of Σ form an orthonormal basis of $\mathbb{R}^d$, we can represent the vector $\boldsymbol{w}$ as

$$\boldsymbol{w} = \sum_{i=1}^{d} \alpha_i \boldsymbol{v}_i. \tag{12.19}$$

We can now rewrite the objective function and the constraint of problem 12.17 using this representation. For the objective function, we get

$$\boldsymbol{w}^\top \Sigma \boldsymbol{w} = \left(\sum_{i=1}^{d} \alpha_i \boldsymbol{v}_i\right)^\top \left(\sum_{j=1}^{d} \lambda_j \boldsymbol{v}_j \boldsymbol{v}_j^\top\right) \left(\sum_{k=1}^{d} \alpha_k \boldsymbol{v}_k\right) \tag{12.20}$$

$$= \sum_{i,j,k=1}^{d} \alpha_i \alpha_k \lambda_j \boldsymbol{v}_i^\top \boldsymbol{v}_j \boldsymbol{v}_j^\top \boldsymbol{v}_k \tag{12.21}$$

$$= \sum_{i=1}^{d} \alpha_i^2 \lambda_i, \tag{12.22}$$

where we have used the fact that $\boldsymbol{v}_i^\top \boldsymbol{v}_j = 0$ if $i \neq j$ and $\boldsymbol{v}_i^\top \boldsymbol{v}_j = 1$ when $i = j$. Similarly, for the constraint we have

$$\|\boldsymbol{w}\|_2^2 = \boldsymbol{w}^\top \boldsymbol{w} = \left(\sum_{i=1}^{d} \alpha_i \boldsymbol{v}_i\right)^\top \left(\sum_{i=j}^{d} \alpha_i \boldsymbol{v}_j\right) \tag{12.23}$$

$$= \sum_{i,j=1}^{d} \alpha_i \alpha_j \boldsymbol{v}_i^\top \boldsymbol{v}_j \tag{12.24}$$

$$= \sum_{i=1}^{d} \alpha_i^2 \stackrel{!}{=} 1. \tag{12.25}$$

This means that we are interested in the optimal allocation of the weights α_i's so that the quantity $\sum_{i=1}^{d} \alpha_i^2 \lambda_i$ is maximized, subject to the constraint that $\sum_{i=1}^{d} \alpha_i^2 = 1$, which tells us that the α_i^2's define a probability mass function over the λ_i's. Clearly, the optimal solution is given by placing all of the probability mass on λ_1, i.e. by setting $\alpha_1^2 = 1$ and $\alpha_j^2 = 0$ for $j \in \{2, \ldots, d\}$ (recall that the eigenvalues λ_i's are arranged in decreasing order). This results in $\boldsymbol{w}^* = \pm \boldsymbol{v}_1$.

□

We conclude this section with two technical remarks. Firstly, note that if w^* is the normalized vector that defines the first principal direction, then $-w^*$ is also an optimal choice for the first principal vector. Secondly, if the largest eigenvalue has multiplicity greater than one, i.e. if there are multiple largest eigenvalues such that $\lambda_1 = \ldots = \lambda_m > \lambda_{m+1} \geq \ldots \lambda_d \geq 0$, then any convex combination of their corresponding eigenvectors will result in an optimal solution.

12.3 Principal Component Analysis with arbitrary k

The previous discussion can be easily generalized to the case where $1 \leq k \leq d$. Instead of a line, which is a one-dimensional object, we are now interested in finding the k-dimensional linear subspace minimizing the reconstruction error. As before, suppose we are given the data $x_1, \ldots, x_n \in \mathbb{R}^d$ and assume that $\mu = n^{-1} \sum_i x_i = 0$. Define $\Sigma = n^{-1} \sum_{i=1}^n x_i x_i^\top$, and consider its eigen-decomposition given by $\Sigma = \sum_{i=1}^d \lambda_i v_i v_i^\top$ where $\lambda_1 \geq \ldots \geq \lambda_d$. We can now present the main result of this chapter.

Theorem 12.2. *The solution to the PCA problem*

$$W^*, z_1^*, \ldots, z_n^* = \arg \min_{\substack{W \in \mathbb{R}^{d \times k} : W^\top W = I_k \\ z_1, \ldots, z_n \in \mathbb{R}^k}} \sum_{i=1}^n \|x_i - W z_i\|_2^2, \tag{12.26}$$

is given by

$$W^* = (v_1 | \cdots | v_k), \qquad z_i^* = W^{*\top} x_i. \tag{12.27}$$

In other words, a basis for the k-dimensional subspace minimizing the reconstruction error can be obtained from the first k eigenvectors $v_1, \ldots, v_k$ of the empirical covariance matrix Σ. A graphical example with two principal components is shown in Figure 12.4.

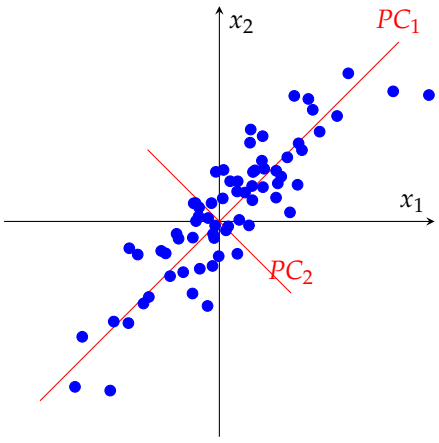


Figure 12.4: Example of PCA with $d = 2$ and $k = 2$. Note that the two principal component vectors form an orthonormal basis for $\mathbb{R}^2$. This choice of basis has a special property: the ellipse described by the cloud of points would appear axis-aligned in this basis.

As an aside, observe that the empirical variance along the j-th principal component direction is equal to the j-th eigenvalue λ_j of the matrix Σ:

$$\frac{1}{n} \sum_{i=1}^n \|v_j v_j^\top x_i\|_2^2 = \frac{1}{n} \sum_{i=1}^n \left(v_j^\top x_i\right)^2 = v_j^\top \Sigma v_j = \lambda_j. \tag{12.28}$$

We will sometime refer to this quantity as the variance *explained* by the j-th principal component direction. In certain situations, it may be more intuitive to view PCA as a method that maximizes the empirical variance captured by the projected data, rather than minimizing the reconstruction error.

Note that PCA can be described as an orthogonal projection onto the subspace spanned by $v_1, \dots, v_k$. The linear mapping associated with this projection can be expressed as follows,

$$P \colon \mathbb{R}^d \to \mathbb{R}^d \tag{12.29}$$

$$\boldsymbol{x} \mapsto W^* W^{*\top} \boldsymbol{x}. \tag{12.30}$$

It can be easily verified that this operator is indeed *idempotent*, i.e. $P^2 = P$, which tells us that applying the operator a second time on the result of the first application does not alter the outcome.

The optimal number of components k is a crucial aspect of PCA, and the appropriate method for selecting this number depends on the specific objectives of the analysis. For visualization, it may be sufficient to inspect the possible plots. When PCA is used in a larger machine-learning pipeline, such as for feature induction, cross-validation can be used to select the optimal number of components (k) that maximizes the performance of the overall model. For compression, it is common practice to choose k such that a predefined percentage (e.g. 95%) of the original variance is retained by the compressed data. Another popular heuristic is based on the so-called *scree* plot, which shows the explained variance as a function of the number of principal components (see Figure 12.5). A reasonable approach is to select a value of k around the point where the line becomes almost horizontal, meaning that the variance explained by the excluded components is negligible. This point is sometime referred to as the *elbow* of the plot. We expect the marginal increase in the explained variance to get smaller and smaller the more components are added to the model. This phenomenon of diminishing returns is well exemplified by Figure 12.6.

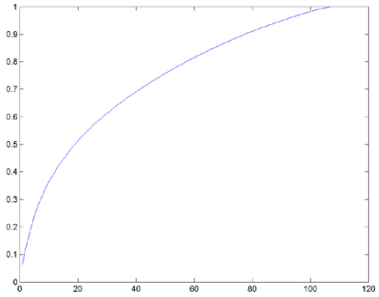


Figure 12.5: A typical scree plot. The explained variance is plotted against the number of principal components retained (source: http://chrisdecoro.com/eigenfaces).

Figure 12.6: This is a sequence of reconstructed images of faces generated using an increasing number of principal components. As the number of components increases, the reconstructed images become more accurate and better resemble the original images. However, there is a noticeable diminishing marginal return in terms of image quality as more components are added. Therefore, it is reasonable to assume that a downstream supervised model, such as one used for facial recognition, may perform well even with a reduced number of principal components, which can result in faster and more efficient computations (source: http://chrisdecoro.com/eigenfaces).

12.4 Connection to SVD

A numerically stable implementation of PCA can be obtained via the Singular-Value Decomposition (SVD) of the data-matrix $X = (\boldsymbol{x}_1 | \cdots | \boldsymbol{x}_n)^\top \in \mathbb{R}^{n \times d}$. Recall that for any rectangular matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, we have the following factorization:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\top, \tag{12.31}$$

where $\boldsymbol{U} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{V} \in \mathbb{R}^{d \times d}$ are orthogonal matrices, and $S \in \mathbb{R}^{n \times d}$ is a diagonal matrix whose diagonal elements σ_i's are arranged in decreasing order. We can now plug this representation in the expression for the empirical covariance matrix Σ to get an explicit characterization of its eigen-decomposition:

$$n \cdot \Sigma = \boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{V}\boldsymbol{S}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{S}^\top \boldsymbol{S}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^\top, \qquad (12.32)$$

where $\boldsymbol{D} := \boldsymbol{S}^\top \boldsymbol{S}$ is a diagonal matrix in $\mathbb{R}^{d\times d}$. This tells us that the first k eigenvectors $v_1, \ldots, v_k$ of the empirical covariance matrix Σ coincide with the first k right singular vectors of the data matrix X. In other words, we can read off the PCA basis from the first k columns of the matrix V.

Bonus Material

In this section we build upon the deep connection between PCA and SVD to present yet another perspective on the optimality of PCA.

Recall that, when $d \leq n$, the SVD allows us to rewrite the matrix $X = USV^\top$ as the sum of d rank-1 matrices :

$$X = \sum_{j=1}^{d} \sigma_j \boldsymbol{u}_j \boldsymbol{v}_j^\top. \qquad (12.33)$$

We can now define the *truncated SVD* of X as follows,

$$\tilde{X}_k = \sum_{j=1}^{k} \sigma_j \boldsymbol{u}_j \boldsymbol{v}_j^\top, \qquad (12.34)$$

for some $k \leq d$. For this low-rank approximation, the following fundamental result holds:

Theorem 12.3 (Eckart-Young Theorem)**.** *Let X be a matrix with dimensions $n \times d$ and* rank *r. For any $k \leq r$, we have that:*

$$\tilde{X}_k \in \arg \min_{\substack{Y \in R^{n\times d}, \\ \text{rank}(Y) \leq k}} \|X - Y\|_F^2, \qquad (12.35)$$

where $\|Y\|_F^2 = \text{trace}(YY^\top)$.

It can be easily shown that, if we denote with $\tilde{\boldsymbol{x}}_i = W^* W^{*\top} \boldsymbol{x}_i$ the reconstruction of the i-th datapoint via PCA, then we have that,

$$(\tilde{\boldsymbol{x}}_i | \ldots | \tilde{\boldsymbol{x}}_n)^\top = \tilde{X}_k, \qquad (12.36)$$

i.e. the output of PCA arranged in matrix form coincide with the *truncated SVD* of X. Therefore, we can interpret PCA as providing the *best rank-k approximation* to the data-matrix X. Furthermore, we have the following characterization for the minimum reconstruction error:

$$\frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \tilde{\boldsymbol{x}}_i\|_2^2 = \frac{1}{n} \|X - \tilde{X}\|_F^2 = \sum_{j=k+1}^{d} \lambda_j, \qquad (12.37)$$

where $\lambda_j = \sigma_j^2 / n$ is the j-th eigenvalue of the empirical covariance matrix Σ. This tells us that the reconstruction error amounts to the variance of the data along the excluded components.

There exist several libraries implementing the PCA algorithm for practitioners. Usually, they rely on efficient and stable subroutines that compute the SVD of the data-matrix X. One popular example is the *scikit-learn* library, which provides a user-friendly API for performing PCA. Below is an example code snippet that showcases some of the key features of the scikit-learn API for PCA.

Implementation

```
from sklearn.decomposition import PCA
# Create a PCA instance specifying the desired number of components
pca = PCA(n_components = 1)
# Fit the PCA model to the data
pca.fit(X)
# Transform the data using the fitted PCA model
X_transformed = pca.transform(X)
```

12.5 A general framework for PCA and k-Means

At first glance, k-means clustering and PCA may appear to be completely unrelated techniques. However, it is possible to show that they are both solving the same fundamental problem, although under different constraints.

Recall the PCA optimization problem:

$$W^*, z_1^*, \ldots, z_n^* = \arg \min_{\substack{W \in \mathbb{R}^{d \times k} : \, W^\top W = I_k \\ z_1, \ldots, z_n \in \mathbb{R}^k}} \sum_{i=1}^{n} \|x_i - W z_i\|_2^2. \tag{12.38}$$

Observe, that it is possible to rewrite the k-means clustering optimization problem as follows:

$$W^*, z_1^*, \ldots, z_n^* = \arg \min_{\substack{W \in \mathbb{R}^{d \times k} \\ z_1, \ldots z_n \in E_k}} \sum_{i=j}^{n} \|x_i - W z_i\|_2^2, \tag{12.39}$$

where $E_k = \{e_1, \ldots, e_k\}$ is the set of unit vectors e_i's such that $e_{i,j} = 0$ if $i \neq j$ and $e_{i,i} = 1$. Note that if we denote by μ_i the i-th column of W, it holds that $We_i = \mu_i$. With this in mind, one can show the equivalence of problem 12.39 to the conventional definition of the k-means clustering optimization problem.

By comparing equations 12.38 and 12.39, it becomes apparent that both PCA and k-means clustering are seeking to compress the data with maximum fidelity. The primary difference between the two techniques lies in the type of constraints imposed on the model complexity. This insight gives rise to a much broader class of techniques known as *matrix factorization* methods, whose goal is to approximately decompose the data matrix X as the product of two separate matrices W and Z:

$$X \approx WZ, \tag{12.40}$$

The specific constraints imposed on W and Z will depend on the application and the desired properties of the resulting decomposition. In the case of PCA, the matrix W represents the orthonormal basis of the optimal k-dimensional subspace, and it must therefore be orthogonal. The matrix Z, on the other hand, is given by the coefficients of the projections onto this subspace and is arbitrary. For

k-means clustering, the constraints on W and Z are different. In this case, the matrix W contains the cluster centroids and is thus arbitrary, while the matrix Z encodes the cluster assignments. By imposing different constraints on W and Z, matrix factorization methods can be adapted to a wide range of applications beyond PCA and k-means clustering, making them a powerful tool in many areas of machine learning.

Part IV

Probabilistic Methods

13

Probabilistic Modeling & Inference

We often have knowledge about how the data we observe is generated. Previously, we have used this assumption to define the generalization error in Section 6.1 or when discussing model bias and variance in Section 7.2. However, so far, this was only for the sake of analyzing the machine learning method that we proposed.

In this chapter, we consider the data-generating distribution, denoted $\mathbb{P}^\star$, not only when analyzing or evaluating our machine learning methods, but also when *designing new probabilistic learning methods*. This chapter is devoted to modeling and learning $\mathbb{P}^\star$ using probabilistic methods and statistical inference. We further connect this new approach with previously discussed methods for regression and classification, and draw analogies to their probabilistic equivalents.

Roadmap

In Section 13.1 we present the probabilistic perspective on the ML pipeline and compare it with the algorithmic perspective. In Section 13.2 we introduce probabilistic models to describe the underlying data distribution. Section 13.3 covers how to estimate the data distribution from a given dataset using a probabilistic model and suitable inference methods such as MLE or MAP. Section 13.4 briefly touches upon the field of decision theory by introducing the notion of Bayes optimal predictors. In Section 13.5 and Section 13.6, we revisit previously discussed regression and classification methods from the probabilistic perspective.

Learning Objectives

After reading this chapter you should know

- how a data distribution can be modeled with a probabilistic model.
- the difference between discriminative and generative models.
- how a data distribution can be learned using statistical inference methods.
- the difference between the frequentist and the Bayesian approach.

- how to compute the MLE estimator.
- how to compute the MAP estimator.
- the statistical interpretation of regression methods and how they relate to the algorithmic perspective.
- the statistical interpretation of classification methods and how they relate to the algorithmic perspective.
- the definition of a Bayes optimal predictor.
- Bayes optimal predictors for regression with square loss and classification with different 0-1 losses.
- the Gaussian naive Bayes method and Fisher's linear discriminant analysis.

13.1 Algorithmic vs. Probabilistic Perspective

So far, we have considered the learning problem from an *algorithmic perspective*. Our training methods M were defined by choosing a function class F as the optimization domain and a training loss $L(f; \mathcal{D})$ as the objective, and then minimizing L over F using optimization methods, such as gradient descent. As the output, we then obtained a function $\widehat{f} = M(\mathcal{D})$ that can take in an input vector and map it to either a label (in supervised learning) or a transformed representation (unsupervised learning). Crucially, F and L were selected to possess favorable optimization properties (e.g., being convex or differentiable) and to ensure that $\widehat{f}$ effectively fulfills its intended purpose: for instance, predicting on test data.

In particular, the choice was made independently of any underlying randomness in the data. In practice, such randomness in the dataset can arise from various sources. For example, certain values of the input features $\boldsymbol{x}$ may intrinsically have different frequencies of appearing. For a regression task, there may be randomness in the observations y due to the inherent noise in physical measurements. For a classification task, different subjective perceptions of human labellers can lead to randomness in the labels y. The kind of randomness can be (approximately) known to us, and one may want to incorporate it into the learning pipeline explicitly.

This is what we do when we take the *probabilistic perspective* on machine learning, where we make probabilistic assumptions about the data-generating mechanism and, most crucially, use it to motivate the learning method M for the corresponding dataset. Specifically, we assume that the data consists of samples from an underlying distribution $\mathbb{P}^\star$. Even though $\mathbb{P}^\star$ is unkown in practice, we might be confident that it is in a family of distributions $\mathcal{P}$ that share some structural commonalities. Choosing this family is known as *probabilistic modeling*, which formalizes the set of assumptions on the underlying distribution. Using *statistical inference*, we can then choose $\widehat{\mathbb{P}} \in \mathcal{P}$ using the data $\mathcal{D}$ to estimate $\mathbb{P}^\star$.

Using the probabilistic perspective, we can

- make assumptions on the origin of the data and the data genera-

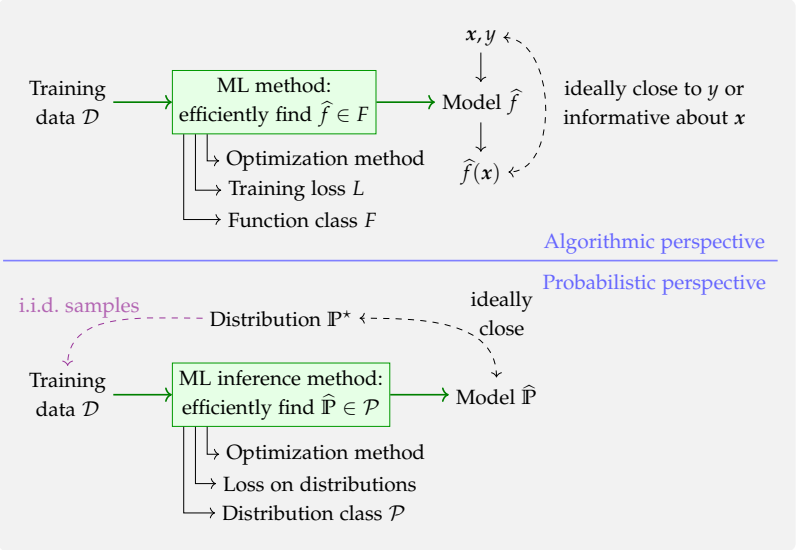


Figure 13.1: Comparison of the algorithmic perspective with the probabilistic perspective. Instead of learning a function $\widehat{f}$, we learn the entire distribution $\widehat{\mathbb{P}}$ that approximates the data-generating distribution $\mathbb{P}^\star$.

tion process (e.g., "What is the likelihood of sampling $\mathcal{D}$?").

- understand why methods work or not (e.g., "For which distributions does the square loss work well?") (see also Chapter 6).
- encode prior knowledge into our model (e.g., "A coin is more likely to have a 50% chance of landing heads up than a 100% chance").
- quantify the uncertainty in predictions made by our models (e.g., "How confident is our model in predicting that a photo contains a cat?").
- develop new decision rules (e.g., only decide for a label when having a minimum confidence, otherwise abstain).
- generate new samples from the distribution (e.g., create new, unseen images of cats).

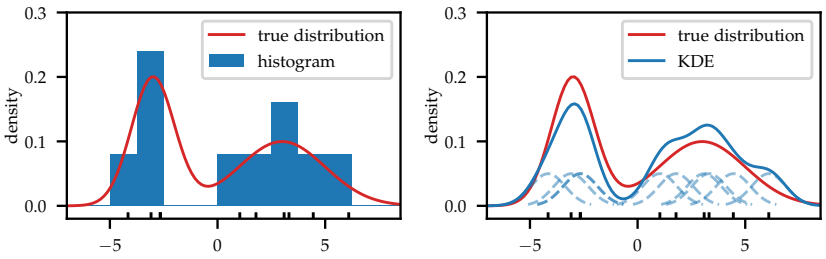
A visualization of the main conceptual difference between the probabilistic and algorithmic perspective is summarized in Figure 13.1. In the remainder of the chapter, we discuss how to learn a distribution $\widehat{\mathbb{P}}$ from data that is close to the true distribution $\mathbb{P}^\star$.

13.2 Probabilistic Modeling

We first describe in mathematical terms what the probabilistic perspective entails. Assume that we observe a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$. The first and fundamental assumption we make is that the data points in $\mathcal{D}$ consist of random variables that are *independently and identically distributed (i.i.d.)* samples drawn from some distribution $\mathbb{P}^\star_{X,Y}$.[1] We can then always express the distribution for the entire dataset by $\mathbb{P}_\mathcal{D} = \prod_{i=1}^n \mathbb{P}_{X_i,Y_i}$ due to the independence of the data points.

[1] Be aware that the i.i.d. assumption is often violated in practice (e.g. sampling with temporal/spatial dependencies, sampling bias, strategic sampling, etc.). However, these dependencies are hard to model and hence i.i.d. is a standard assumption that is used in most cases.

One approach to modeling $\mathbb{P}^\star$ is to make no further assumptions and use "general-purpose" estimators. For example, the well-known *histogram* estimator for a density of the distribution $\mathbb{P}^\star$ can always be applied, and (essentially) does not make any implicit assumptions on $\mathbb{P}^\star$. A smooth analog to that is Kernel density estimation (KDE). Both are visualized in Figure 13.2.



(a) Histogram density estimate.

(b) Kernel density estimate.

Figure 13.2: Histogram and Kernel Density Estimation. The black ticks above the x-axis indicate the points in the dataset $\mathcal{D}$. The left figure shows a discrete approximation of the true PDF by histogram binning, the right figure shows a smooth approximation of the true PDF by kernel density estimation with a Gaussian kernel.

However, such general-purpose estimators can come at a cost. Usually, to have an accurate estimate of the true distribution, they need a lot of samples, because they cannot exploit further structure that $\mathbb{P}^\star$ might have. Therefore, the second assumption we make is that the distribution $\mathbb{P}^\star$ lies in a family of parameterized models.

Similar to parametric function classes for supervised learning (like in Chapter 4, Chapter 8 or Chapter 10), we *model* the distribution to be from a parametric family of distributions.

Definition 13.1. A family of distributions $\mathcal{P}$ is called *parametric*, if it is fully described by a finite number of parameters[2] $\theta \in \Theta \subset \mathbb{R}^p$ with $p \in \mathbb{N}$, that is,

$$\mathcal{P} = \left\{ \mathbb{P}^\theta \mid \theta \in \Theta \right\}.$$

Note that previously, we used the term "model" solely to refer to functions. In the probabilistic viewpoint, we use "model" to refer to entire distributions, and we use the term "modeling" to refer to the act of choosing the distributional family $\mathcal{P}$. Notably, we usually choose the parametric family of distributions $\mathcal{P}$ in a way such that it likely includes the true distribution, i.e., that there exists a *true* parameter $\theta^\star \in \Theta$ for which[3]

$$\mathbb{P}^\star = \mathbb{P}^{\theta^\star} \in \mathcal{P}.$$

Hence, we can estimate $\mathbb{P}^\star$ by estimating $\theta^\star$.[4]

Example 13.2. An example for a parametric family of distributions is the family of normal distributions, defined as

$$\mathcal{P} = \left\{ \mathcal{N}(\mu, \sigma^2) \mid (\mu, \sigma) \in \mathbb{R}^2 \right\}.$$

It is parametric because its distributions are fully described by the parameters $\theta = (\mu, \sigma) \in \Theta = \mathbb{R}^2$.

[2] We assume that the mapping $\theta \mapsto \mathbb{P}^\theta$ is (in some sense) continuous.

[3] It is not necessarily a problem if $\mathbb{P}^\star \notin \mathcal{P}$ but we output $\widehat{\mathbb{P}} \in \mathcal{P}$. Such a mismatch between $\mathbb{P}^\star$ and $\mathcal{P}$ is called *misspecification*, and the field that studies learning in this regime is called *agnostic learning*.

[4] Here the continuity of $\theta \mapsto \mathbb{P}^\theta$ is important. Intuitively, if θ is close to $\theta^\star$, then $\mathbb{P}^\theta$ is also close to $\mathbb{P}^{\theta^\star}$ (in some sense).

Notably, there exist families of distributions that are *not* parametric. For example, the family of all distributions on $\mathbb{R}$ that have a density, $\mathcal{P} = \{\mathbb{P} \mid \mathbb{P} \text{ has a density on } \mathbb{R}\}$, is *not* a parametric family. There exists no way to assign a parameter $\theta \in \mathbb{R}^p$ to every density on $\mathbb{R}$.

13.3 Statistical Inference

Broadly speaking, statistical inference is the act of choosing an estimator $\widehat{\mathbb{P}}$ from the distribution family $\mathcal{P}$ – based on the data $\mathcal{D}$ – that, ideally, approximates $\mathbb{P}^\star$ as well as possible. In *parametric estimation methods*, we can achieve this by searching over Θ and finding the distribution

$$\widehat{\mathbb{P}} = \mathbb{P}^{\widehat{\theta}} \in \mathcal{P}$$

with some estimated parameter $\widehat{\theta} \in \Theta$ that approximates $\theta^\star$.

13.3.1 Inference Paradigms

There are two major paradigms for designing statistical inference methods: the *frequentist* paradigm and the *Bayesian* paradigm. They follow a different design idea to infer properties about $\mathbb{P}^\star$.

We start with a simple example to illustrate the main difference between frequentist and Bayesian thinking.

Example 13.3. Imagine a scientist has a (possibly imperfect) coin, and wants to determine the probability that when the coin is flipped, it lands with "heads" facing up. The scientist chooses to model the outcome of the coinflip with a random variable $X \in \{0, 1\}$ that follows a Bernoulli distribution $\mathrm{Ber}(\theta)$. Hence, the parametric model is $\mathbb{P}_X^\theta \in \mathcal{P} = \{\mathrm{Ber}(\theta) \mid \theta \in [0, 1]\}$. Now, suppose the scientist flips his coin 5 times to determine $\theta^\star$, the parameter of the true underlying distribution, and it just so happens that all coinflips $X_1, \ldots, X_5$ land heads up. What is a good estimate for the probability $\theta^\star$? A subscriber to the *frequentist* paradigm would argue that, since we only have observed these 5 coinflips, the best estimator must be based on the observed *frequency*, so that $\widehat{\theta} = 1$. On the other hand, a subscriber to the *Bayesian* paradigm would argue that, since we have only observed 5 coinflips, and we know that coins usually have a 0.5 probability of landing heads up, we should only update our previous belief to a certain extend and estimate, e.g., $\widehat{\theta} = 6/7$.

In general, both the frequentist and Bayesian viewpoints are valid, depending on the context. However, in Example 13.3, the frequentist approach is questionable, since we are all familiar with coin tosses, so we have a strong *prior belief*. This notion of prior belief can be modeled by assuming that $\theta^\star$ *itself is an unobserved random variable* from a distribution $\mathbb{P}_{\theta^\star}$ that is chosen by the scientist.[5,6] In turn, the notion of updating the belief is captured in *Bayes' rule* (see Theorem 3.39)[7]

$$\underbrace{p(\theta \mid \mathcal{D})}_{\text{posterior belief}} = \underbrace{\frac{p(\mathcal{D} \mid \theta)}{p(\mathcal{D})}}_{\text{update}} \underbrace{p(\theta)}_{\text{prior belief}}$$

[5] There is also the possibility to define the parameters of the prior distribution as random variables themselves, which in turn can be estimated using Bayesian inference. Such methods are called *hierarchical Bayes models*, which go beyond this course.

[6] $\mathbb{P}_{\theta^\star}$ is not to be confused with $\mathbb{P}^{\theta^\star}$. The former is a distribution on the parameter space Θ, the latter is a distribution on the data space.

[7] A quick remark on *notation*. For the rest of the chapter, we neglect the subscripts on PDFs and PMFs to specify which random variables they refer to. Specifically, for any random variables Z whose law is in a parametric family $\mathbb{P}_Z \in \mathcal{P} = \{\mathbb{P}_Z^\theta \mid \theta \in \Theta\}$ that have PDFs or PMFs p_Z^θ with $\theta \in \Theta$, we write

$$p(z; \theta) = p_Z^\theta(z).$$

In the Bayesian setting, where the parameter $\theta^\star$ is a random variable with prior distribution $\mathbb{P}_{\theta^\star}$ that has PDF or PMF $p_{\theta^\star}$, we will write

$$\begin{aligned} p(\theta) &= p_{\theta^\star}(\theta), \\ p(z \mid \theta) &= p_{Z \mid \theta^\star = \theta}(z), \\ p(\theta \mid z) &= p_{\theta^\star \mid Z = z}(\theta). \end{aligned}$$

This eases readability but one should be aware that this presents an abuse of notation.

where p denotes the densities (PDFs) or probability mass functions (PMFs), respectively. Note that $p(\mathcal{D}) = \int p(\mathcal{D} \mid \theta)p(\theta)d\theta$. The prior distribution $\mathbb{P}_{\theta^\star}$ with PDF or PMF $p(\theta)$ describes the Bayesian's belief, that θ is the true parameter before receiving any information from the observed data $\mathcal{D}$. After seeing the samples in $\mathcal{D}$, Bayesian inference allows us to obtain a posterior distribution $\mathbb{P}_{\theta^\star|\mathcal{D}}$ describing our updated belief that θ is the true parameter. This happens through the multiplicative update term $p(\mathcal{D} \mid \theta)/p(\mathcal{D})$. In Example 13.3, we could choose the prior $6 \cdot \theta(1-\theta)$. The corresponding posterior is visualised in Figure 13.3.

Exercise 13.4. In Example 13.3, compute the posterior of $\theta^\star$ given that all 5 coinflips are heads, using Bayes rule with a prior $p(\theta) = 6 \cdot \theta(1-\theta)$. Why might the Bayesian scientist have picked $\widehat{\theta} = 6/7$?

Solution. We have that $p(\mathcal{D} \mid \theta) = \prod_{i=1}^{5} p(X_i = 1 \mid \theta) = \theta^5$ and

$$p(\mathcal{D}) = \int p(\mathcal{D} \mid \theta)p(\theta)d\theta = 6\int_0^1 \theta^5 \cdot \theta(1-\theta)d\theta = 3/28.$$

Thus, the posterior takes the form

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)}{p(\mathcal{D})}p(\theta) = \frac{\theta^5}{3/28} \cdot 6 \cdot \theta(1-\theta) = 56 \cdot \theta^6(1-\theta).$$

The maximum of this posterior is given by $\widehat{\theta} = 6/7$. This choice will be discussed in Subsection 13.3.3. □

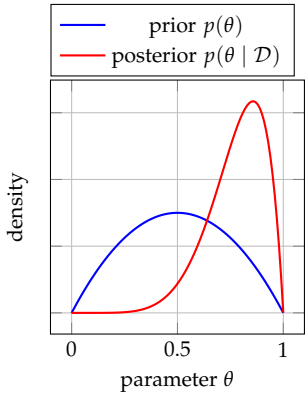


Figure 13.3: Prior and posterior distribution of Example 13.3 with prior $6 \cdot \theta(1-\theta)$.

There are many pros and cons one can name about the Bayesian approach over the frequentist approach, here is an incomplete list:

1. It lets the scientist incorporate prior information into the estimator. However, this prior information can also bias the results.
2. Various statements can only be formulated in a Bayesian approach (e.g., "What is the expectation of $\theta^\star$ given the data $\mathcal{D}$?").
3. The precision of the estimator is unknown to Frequentists whenever an experiment can only be performed once.
4. Bayesian estimators are optimal from a decision-theoretic point, compare with Section 13.4.
5. Approximating the posterior can be computationally hard, albeit the existence of rather universal tools designed for that purpose (such as Markov chain Monte Carlo).

Notably, for both the frequentist and Bayesian approach, we still need to come up with rules on how to pick $\widehat{\theta}$. This will be discussed next.

13.3.2 *Maximum Likelihood Estimation (MLE)*

A common frequentist approach, that considers an a priori fixed $\theta^\star$, is *maximum likelihood estimation (MLE)*. As the name suggests, MLE finds the parameter $\widehat{\theta}_{\text{MLE}}$ which maximizes the likelihood of observing $\mathcal{D}$ over possible distributions in $\left\{\mathbb{P}^{\theta}_{X,Y} \mid \theta \in \Theta\right\}$. This likelihood

is defined as the density (or probability mass function) of the model $\mathbb{P}_{\mathcal{D}}^{\theta}$ evaluated at the dataset $\mathcal{D}$, denoted $p(\mathcal{D};\theta)$.[8,9] Intuitively, it makes sense to choose the distribution in the parametric model, for which it is most likely to sample the given $\mathcal{D}$. A more rigorous proof for this intuition is given below as bonus material.

[8] Recall the abuse of notation by not subscribing p.

[9] In the Bayesian framework, the likelihood refers to the conditional probabilities of the form $p(\mathcal{D} \mid \theta)$, but since $\theta^\star$ is not a random variable in the frequentist setting, we write $p(\mathcal{D};\theta)$ and still refer to it as the likelihood.

Definition 13.5. Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \mid i = 1, \dots, n\}$ with i.i.d. samples and a parametric statistical model with parameter space Θ, the *maximum likelihood estimator* is defined as

$$\widehat{\theta}_{\text{MLE}} := \arg\max_{\theta \in \Theta} p(\mathcal{D};\theta) \overset{\text{i.i.d.}}{=} \arg\max_{\theta \in \Theta} \prod_{i=1}^{n} p(\boldsymbol{x}_i, y_i; \theta)$$

and corresponds to the estimated distribution $\widehat{\mathbb{P}}_{X,Y} = \mathbb{P}_{X,Y}^{\widehat{\theta}_{\text{MLE}}}$.

It is common to consider the equivalent minimization problem using the negative log-likelihood[10]

[10] Since the logarithm is strictly monotonically increasing, the maximizer of the log-likelihood is also a maximizer of the likelihood.

$$\begin{aligned}
\arg\max_{\theta \in \Theta} \prod_{i=1}^{n} p(\boldsymbol{x}_i, y_i; \theta) &= \arg\max_{\theta \in \Theta} \log \left(\prod_{i=1}^{n} p(\boldsymbol{x}_i, y_i; \theta) \right) \\
&= \arg\max_{\theta \in \Theta} \sum_{i=1}^{n} \log p(\boldsymbol{x}_i, y_i; \theta) \\
&= \arg\min_{\theta \in \Theta} \sum_{i=1}^{n} -\log p(\boldsymbol{x}_i, y_i; \theta) \qquad (13.1)
\end{aligned}$$

where we transform the product into a sum using the logarithm.

Note how this problem corresponds to a typical loss minimization problem from the algorithmic perspective as we have seen in the previous chapters. Taking the negative logarithm of the likelihood maps densities (or probability mass functions) to a decreasing loss as visualized in Figure 13.4.

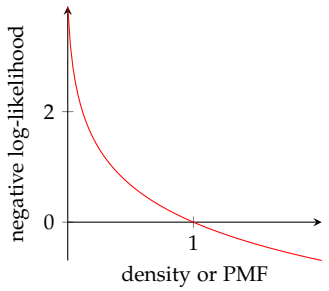


Figure 13.4: The negative log-likelihood transforms the density or PMF at a datapoint $(\boldsymbol{x}_i, y_i)$ (which we want to maximize) to a loss decreasing in this density, which we can now minimize.

Equation 13.1 is the optimization problem to be solved if θ parameterizes the complete joint distribution $\mathbb{P}_{X,Y}^{\theta}$ as it is the case for generative models. Often, we are interested in a discriminative model, which parameterizes only the conditional distribution $\mathbb{P}_{Y|X}^{\theta}$. Then we can further simplify Equation 13.1 to

$$\begin{aligned}
&\arg\min_{\theta \in \Theta} \sum_{i=1}^{n} -\log p(\boldsymbol{x}_i, y_i; \theta) \\
&\quad = \arg\min_{\theta \in \Theta} \sum_{i=1}^{n} -\log(p(y_i \mid \boldsymbol{x}_i; \theta) p(\boldsymbol{x}_i)) \\
&\quad = \arg\min_{\theta \in \Theta} \sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i; \theta) + \sum_{i=1}^{n} -\log p(\boldsymbol{x}_i) \\
&\quad = \arg\min_{\theta \in \Theta} \sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i; \theta), \qquad (13.2)
\end{aligned}$$

where the last equality follows from the fact that the marginal $p(x_i)$ does not depend on θ. Based on our statistical model, we have closed-form expressions for $p(\boldsymbol{x}_i, y_i; \theta)$ or $p(y_i \mid \boldsymbol{x}_i; \theta)$ which we can insert into Equation 13.1 or Equation 13.2 and we can find the

minimum analytically (e.g., using the first-order condition from Theorem 2.4) or numerically (e.g., using gradient descent).

Example 13.6. To illustrate MLE with a toy example, assume we are given a dataset $\mathcal{D} = \{2, 6, 1.5\}$ of i.i.d. samples from the true distribution $\mathbb{P}_X^\star = \mathcal{N}(3, 4)$. Without knowing $\mathbb{P}_X^\star$, we correctly assume that the data is sampled from a Gaussian distribution and use the family of Gaussian distributions $\{\mathcal{N}(\mu, \sigma^2) \mid (\mu, \sigma) \in \mathbb{R}^2\}$ as our statistical model. For computing the MLE estimators $\widehat{\mu}_{MLE}$ and $\widehat{\sigma}_{MLE}$ as in Equation 13.1, we minimize the negative log-likelihood

$$\begin{aligned} -\log p(\mathcal{D}; \mu, \sigma) &= \sum_{i=1}^{n} -\log p(x_i; \mu, \sigma) \\ &= \sum_{i=1}^{n} -\log \left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x_i - \mu)^2} \right) \\ &= \sum_{i=1}^{n} \left(\frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(x_i - \mu)^2 \right) \\ &= c + n \log(\sigma) + \frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i - \mu)^2 \end{aligned}$$

with constant $c = \frac{n}{2} \log(2\pi)$. In this example, we can obtain an analytic solution for the minimizer by setting the first derivatives of $-\log p(\mathcal{D}; \mu, \sigma)$ with respect to μ and σ to zero. We obtain

$$\begin{aligned} \frac{\partial}{\partial \mu} -\log p(\mathcal{D}; \mu, \sigma) &= -\frac{1}{\sigma^2} \sum_{i=1}^{n} (x_i - \mu) \stackrel{!}{=} 0 \quad &\iff \quad \mu &= \frac{1}{n} \sum_{i=1}^{n} x_i \\ \frac{\partial}{\partial \sigma} -\log p(\mathcal{D}; \mu, \sigma) &= \frac{n}{\sigma} - \frac{1}{\sigma^3} \sum_{i=1}^{n} (x_i - \mu)^2 \stackrel{!}{=} 0 \quad &\iff \quad \sigma^2 &= \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2. \end{aligned}$$

We can observe that the empirical mean $\widehat{\mu}_{MLE} = \frac{1}{n} \sum_{i=1}^{n} x_i$ and empirical variance $\widehat{\sigma}^2_{MLE} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \widehat{\mu}_{MLE})^2$ of the given dataset $\mathcal{D}$ uniquely satisfy the first-order condition and therefore correspond to the MLE estimators, which globally minimize the negative log-likelihood. In our case, we arrive at $\widehat{\mathbb{P}}_X = \mathcal{N}(3.17, 4.06)$ by plugging in the given $\mathcal{D}$ which is quite close to the true distribution $\mathbb{P}_X^\star$.

Bonus Material

In the following, we provide a formal proof for the intuition that choosing $\widehat{\theta}_{\text{MLE}}$, for which the likelihood of sampling the given $\mathcal{D}$ is largest, is a good strategy, because it converges to $\theta^\star$.

Proof. We first show that the true parameter $\theta^\star$ maximizes the expected log-likelihood $\log p(\boldsymbol{x}, y; \theta)$

over the true distribution $\mathbb{P}_{X,Y}^{\theta^\star}$.

$$
\begin{aligned}
&\arg\max_{\theta\in\Theta} \mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log p(\boldsymbol{x},y;\theta)\right] \\
&\quad = \arg\max_{\theta\in\Theta} \mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log \frac{p(\boldsymbol{x},y;\theta)}{p(\boldsymbol{x},y;\theta^\star)} + \log p(\boldsymbol{x},y;\theta^\star)\right] && (1)\\
&\quad = \arg\max_{\theta\in\Theta} \mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log \frac{p(\boldsymbol{x},y;\theta)}{p(\boldsymbol{x},y;\theta^\star)}\right] + \mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log p(\boldsymbol{x},y;\theta^\star)\right] && (2)\\
&\quad = \arg\max_{\theta\in\Theta} \mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log \frac{p(\boldsymbol{x},y;\theta)}{p(\boldsymbol{x},y;\theta^\star)}\right] && (3)\\
&\quad = \arg\min_{\theta\in\Theta} \mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log \frac{p(\boldsymbol{x},y;\theta^\star)}{p(\boldsymbol{x},y;\theta)}\right] && (4)\\
&\quad = \arg\min_{\theta\in\Theta} D_{\mathrm{KL}}(\mathbb{P}_{X,Y}^{\theta^\star} \parallel \mathbb{P}_{X,Y}^{\theta}) && (5)\\
&\quad = \theta^\star && (6)
\end{aligned}
$$

where we have (1) by property of logarithm, (2) by linearity of expectation, (3) since $\mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log p(\boldsymbol{x},y;\theta^\star)\right]$ is independent of θ and hence constant, (4) because negating maximization yields minimization and property of logarithm, (5) by definition of Kullback-Leibler divergence (see remark below) and (6) since $D_{\mathrm{KL}}(\mathbb{P}_{X,Y}^{\theta^\star} \parallel \mathbb{P}_{X,Y}^{\theta})$ is minimized for $\mathbb{P}_{X,Y}^{\theta^\star} = \mathbb{P}_{X,Y}^{\theta}$ (see remark below).

Since by the law of large numbers (see Theorem 3.51) this expected log-likelihood is empirically approximated by the average log-likelihood of the dataset

$$
\mathbb{E}_{\boldsymbol{x},y\sim\mathbb{P}_{X,Y}^{\theta^\star}} \left[\log p(\boldsymbol{x},y;\theta)\right] \approx \frac{1}{n}\sum_{i=1}^{n} \log p(\boldsymbol{x}_i,y_i;\theta) = \frac{1}{n}\log p(\mathcal{D};\theta)
$$

for large n, the maximizer of the average log-likelihood of the dataset converges to $\theta^\star$ asymptotically. □

Remark. The *Kullback-Leibler (KL) divergence* is an asymmetric measure for the "difference" (or divergence) between a reference distribution P and another distribution Q. It is defined as

$$
D_{\mathrm{KL}}(P \parallel Q) := \mathbb{E}_{X\sim P}\left[\log \frac{p(X)}{q(X)}\right] = \int_{-\infty}^{\infty} p(x)\cdot \log \frac{p(x)}{q(x)} dx \tag{13.3}
$$

where $p(x)$ and $q(x)$ refer to the PDF or PMF of the respective distributions P and Q. Intuitively, it measures the expected difference in the log-probabilities $\log \frac{p(X)}{q(X)} = \log p(X) - \log q(X)$ of the two distributions. However, this expected difference is asymmetric in the sense of $D_{\mathrm{KL}}(P \parallel Q) \neq D_{\mathrm{KL}}(Q \parallel P)$, since the expectation is taken with respect to the reference distribution given as the first argument. This means this quantity cannot be interpreted as a symmetric distance metric. The important property of the KL-divergence is that it is non-negative and it attains its minimum zero if both distributions are equal, precisely

$$
\begin{aligned}
&D_{\mathrm{KL}}(P \parallel Q) \geq 0 \quad \text{for all distributions } P, Q \\
&D_{\mathrm{KL}}(P \parallel Q) = 0 \quad \iff P = Q.
\end{aligned} \tag{13.4}
$$

Bonus Material

The resulting MLE estimator has multiple favorable statistical properties, which can be summarized in this Theorem:

Theorem 13.7. *Suppose that $\Theta \subset \mathbb{R}^d$. Under some technical assumptions, the distribution of MLE estimators $\widehat{\theta}_{\text{MLE}}$ with randomness in $\mathcal{D}$ is asymptotically normal. Using the notation from Definition 3.50, we can formulate this as, for $|\mathcal{D}| = n \to \infty$,*

$$\sqrt{n}(\widehat{\theta}_{\text{MLE}} - \theta^\star) \xrightarrow{law} \mathcal{N}(0, I^{-1}(\theta^\star)).$$

Here $I(\theta^\star)$ denotes the so-called Fisher information matrix—we will not define it and only discuss its meaning. Theorem 13.7 implies *consistency*: The MLE estimator $\widehat{\theta}_{\text{MLE}}$ converges in probability to the true parameters $\theta^\star$ for $|\mathcal{D}| = n \to \infty$. Using the notation from Definition 3.50, we can formulate this as $\widehat{\theta}_{\text{MLE}} \xrightarrow{\mathbb{P}} \theta^\star$. Further, it implies *asymptotic efficiency*: The variance of MLE estimator $\widehat{\theta}_{\text{MLE}}$ with randomness in $\mathcal{D}$ is asymptotically equal to $I^{-1}(\theta^\star)$ for $|\mathcal{D}| = n \to \infty$. A famous result in classical statistics, known as the *Cramér-Rao lower bound (CRLB)* establishes that this is optimal—no unbiased estimator can have a lower variance.

However, all these properties are asymptotic and only hold for $n \to \infty$. Given a dataset of finite size n, one cannot take these properties for granted and must be careful to avoid overfitting.

13.3.3 Maximum A Posteriori Estimation (MAP)

A similar approach based on the Bayesian paradigm, which treats $\theta^\star$ as a random variable, is called *maximum a posteriori estimation (MAP)*. The MAP estimator $\widehat{\theta}_{\text{MAP}}$ is the maximizer of the posterior belief $p(\theta \mid \mathcal{D})$ for the true parameter after observing $\mathcal{D}$. Intuitively, the Bayesian believes that this parameter has highest density or PMF after observing $\mathcal{D}$.

Definition 13.8. Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \mid i = 1, \dots, n\}$ with i.i.d. samples and a parametric statistical model with parameter space Θ, the *maximum a posteriori* estimator is defined as[11]

$$\begin{aligned}\widehat{\theta}_{\text{MAP}} &:= \arg\max_{\theta \in \Theta} p(\theta \mid \mathcal{D}) = \arg\max_{\theta \in \Theta} p(\mathcal{D} \mid \theta) p(\theta) \\ &\overset{\text{i.i.d.}}{=} \arg\max_{\theta \in \Theta} \left(\prod_{i=1}^{n} p(\boldsymbol{x}_i, y_i \mid \theta) \right) \cdot p(\theta)\end{aligned}$$

and estimates the true distribution with $\widehat{\mathbb{P}}_{X,Y} = \mathbb{P}_{X,Y}^{\widehat{\theta}_{\text{MAP}}}$.

Intuitively, the prior belief influences the posterior belief by putting more or less weight $p(\theta)$ on certain θ. Hence, this allows us to encode prior knowledge in the prior distribution of the parameter, as discussed in Subsection 13.3.1.

As for MLE in Equation 13.1, we transform the maximization problem into an equivalent minimization problem by taking the

[11] The second equality is obtained by Bayes rule (see Theorem 3.30), where the denominator $p(\mathcal{D})$ is irrelevant for the arg max.

negative logarithm on both sides

$$\begin{aligned}&\arg\max_{\theta\in\Theta}\left(\prod_{i=1}^{n} p(\boldsymbol{x}_i, y_i \mid \theta)\right)\cdot p(\theta)\\ &\quad=\arg\min_{\theta\in\Theta}\sum_{i=1}^{n} -\log p(\boldsymbol{x}_i, y_i \mid \theta) - \log p(\theta). \qquad (13.5)\end{aligned}$$

For discriminative models, it can be further simplified to

$$\begin{aligned}&\arg\min_{\theta\in\Theta}\sum_{i=1}^{n} -\log p(\boldsymbol{x}_i, y_i \mid \theta) - \log p(\theta)\\ &\quad=\arg\min_{\theta\in\Theta}\sum_{i=1}^{n} -\log(p(y_i \mid \boldsymbol{x}_i,\theta)p(\boldsymbol{x}_i \mid \theta)) - \log p(\theta)\\ &\quad=\arg\min_{\theta\in\Theta}\sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i,\theta) + \sum_{i=1}^{n} -\log p(\boldsymbol{x}_i) - \log p(\theta)\\ &\quad=\arg\min_{\theta\in\Theta}\sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i,\theta) - \log p(\theta). \qquad (13.6)\end{aligned}$$

Similar to Equation 13.2, the justification for the third equality is that $\boldsymbol{x}_i$ is independent from the random variable θ, since θ only parameterizes $\mathbb{P}_{Y|X}$ in a discriminative model.

Summary and Comparison. Recall Example 13.3. In Exercise 13.4, we actually showed that the scientist chose the MAP estimator, without calling it that. We can now complete Figure 13.3 and get Figure 13.5. We see that the Frequentist chose to maximize the likelihood, whereas the Bayesian chose to maximize the posterior distribution.

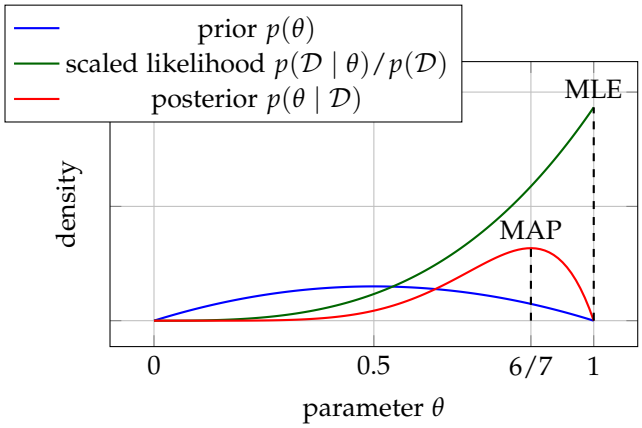


Figure 13.5: Prior, scaled likelihood and posterior distribution of Example 13.3 with prior $p(\theta) = 6 \cdot \theta(1-\theta)$. The MAP is the maximizer of the posterior, whereas the MLE is the maximizer of the likelihood.

Next, let us highlight an interesting relation between MLE and MAP. Assume that we use MAP without having any prior knowledge about $\theta^\star$. This is expressed by using a uniform distribution $\mathbb{P}_{\theta^\star} = \text{Uniform}(\Theta)$ over the parameter space Θ for the prior belief, which returns the same prior $p(\theta)$ for all $\theta \in \Theta$.[12] Hence, $p(\theta)$ does not play a role for the arg max in Definition 13.8 and can be eliminated, which then yields

$$\widehat{\theta}_{\text{MAP}} = \arg\max_{\theta\in\Theta}\prod_{i=1}^{n} p(\boldsymbol{x}_i, y_i \mid \theta) = \arg\max_{\theta\in\Theta}\prod_{i=1}^{n} p(\boldsymbol{x}_i, y_i; \theta) = \widehat{\theta}_{\text{MLE}}.$$

[12] Note that the uniform distribution does not always exist. In that case, setting $p(\theta) = c$ for some constant c is called an *improper* prior.

One can see that using MAP with a uniform prior distribution coincides with using MLE.

Bonus Material

MLE and MAP have something in common. They infer the most fitting $\widehat{\theta}$ from $\mathcal{D}$ by maximizing some quantity, say $f(\theta, \mathcal{D})$, be it the likelihood of observing $\mathcal{D}$ or the posterior probability of θ given $\mathcal{D}$, and then use this estimate $\widehat{\theta}$ to compute quantities related to the modeled distribution, such as $p(y \mid \boldsymbol{x}; \widehat{\theta})$.

These "optimization" based probabilistic learning methods, which can be formulated as

$$p(y \mid \boldsymbol{x}, \widehat{\theta}) \quad \text{with} \quad \widehat{\theta} = \arg\max_{\theta \in \Theta} f(\theta, \mathcal{D})$$

are typically efficient, since there are many methods for maximization or minimization at your disposal. However, these methods ignore the uncertainty in the obtained estimate $\widehat{\theta}$, since they only maximize the likelihood or posterior.

An alternative are "integration" based probabilistic learning methods, also called *Bayesian model averaging*, which can be formulated as

$$p(y \mid \boldsymbol{x}, \mathcal{D}) = \int p(y \mid \boldsymbol{x}, \theta) p(\theta \mid \mathcal{D}) d\theta$$

for the case of parametric discriminative models. Intuitively, given the dataset $\mathcal{D}$, it does not only consider the model with the largest posterior probability $p(\theta \mid \mathcal{D})$, but takes the average over all models weighted by their corresponding posterior probability into account. Hence, these methods take the uncertainty of each model into account. Although this kind of integration is typically intractable, approximation methods exist. More details are provided in the course *Probabilistic Artificial Intelligence*.

13.4 Bayes Optimal Predictors

Once we have estimated the underlying data-generating distribution, we might still have to make predictions. We now discuss how to transfer knowing or estimating the distribution leads to optimal prediction rules.

Assume we are in a supervised learning setting with the space of possible predictions $\mathcal{Y}$ and the loss function $\ell(\cdot, y)$ with respect to the ground truth data y. Using statistical inference, we can estimate the distribution of $Y \mid X = x$, and derive the optimal decision rule $f^\star(x)$ from it. This is called the *Bayesian optimal decision rule* or *Bayes' optimal predictor*.

Definition 13.9. The *Bayes' optimal predictor* is defined as

$$f^\star(x) := \arg\min_{a \in \mathcal{Y}} \mathbb{E}\left[\ell(a, Y) \mid X = x\right] = \arg\min_{a \in \mathcal{Y}} \int p(y \mid x) \cdot \ell(a, y) dy$$

for all inputs x and is the theoretically best possible predictor one can achieve when knowing the conditional distribution $\mathbb{P}_{Y|X}$.

The reason this is called *Bayes'* optimal predictor is that minimizes the posterior expected loss given the input x. This forms the foundation of *Bayesian decision theory* turning the Bayesian's belief into actions.

In practice, the true conditional distribution $\mathbb{P}_{Y|X}$ is unknown to us due to the finite dataset and we can only use the inferred distribution $\widehat{\mathbb{P}}_{Y|X}$ for estimating the Bayes' optimal predictor with

$$\widehat{f}(x) := \arg\min_{a \in \mathcal{Y}} \widehat{\mathbb{E}}\left[\ell(a, Y) \mid X = x\right] = \arg\min_{a \in \mathcal{Y}} \int \widehat{p}(y \mid x) \cdot \ell(a, y) dy$$

where $\widehat{p}(y \mid x)$ is obtained from $\widehat{\mathbb{P}}_{Y|X}$. This is where probabilistic modeling and statistical inference comes into play.

Example 13.10. A common example is the classification of spam emails. Let $\mathcal{Y} = \{\text{not spam}, \text{spam}\}$ be the set of actions the ML model can choose from and assume we are given a model $\widehat{\mathbb{P}}_{Y|x}$ which gives us an estimate $\widehat{p}(\text{spam} \mid x)$ for the probability that a given email x is spam. Assuming that classifying spam emails as non-spam is much more severe than the other way around, we design a loss function

$$\ell(a, y) = \begin{cases} 0, & a = y \\ 10, & a = \text{not spam and } y = \text{spam} \\ 1, & a = \text{spam and } y = \text{not spam} \end{cases}$$

which penalizes false negatives much stronger than false positives. Assume we are now given an email x_1 with $\widehat{p}(\text{spam} \mid x_1) = 0.2$. The expected loss for each of the possible actions would be

$$\begin{aligned} \widehat{\mathbb{E}}\left[\ell(\text{not spam}, Y) \mid X = x_1\right] &= 0.2 \cdot 10 + 0.8 \cdot 0 = 2 \\ \widehat{\mathbb{E}}\left[\ell(\text{spam}, Y) \mid X = x_1\right] &= 0.2 \cdot 0 + 0.8 \cdot 1 = 0.8 \end{aligned}$$

and the Bayes' optimal predictor would decide for $\widehat{f}(x_1) = \text{spam}$ which has the lowest expected loss. However, if we are given another email x_2 with even lower probability $\widehat{p}(\text{spam} \mid x_2) = 0.05$ for being spam, the expected loss for each possible action would be

$$\begin{aligned} \widehat{\mathbb{E}}\left[\ell(\text{not spam}, Y) \mid X = x_2\right] &= 0.05 \cdot 10 + 0.95 \cdot 0 = 0.5 \\ \widehat{\mathbb{E}}\left[\ell(\text{spam}, Y) \mid X = x_2\right] &= 0.05 \cdot 0 + 0.95 \cdot 1 = 0.95 \end{aligned}$$

and the Bayes' optimal predictor would decide for $\widehat{f}(x_2) = \text{not spam}$ which now has the lowest expected loss.

Remark. Observe that our supervised ML pipeline uses the loss function to first find a predictor

$$\widehat{f} := \arg\min_{f \in F} \sum_{i=1}^{n} \ell(f(\boldsymbol{x}_i), y_i)$$

within a fixed function class F which minimizes the total loss on the given inputs $\{x_1, \ldots, x_n\}$. It then uses this function $\widehat{f}$ for making all later predictions on unseen inputs x, which therefore heavily depend on the characteristics of the function class and the previously given inputs. The conceptual difference to the Bayes' optimal predictor is that the latter is defined as a function

$$f^\star \colon x \mapsto \arg\min_{a \in \mathcal{Y}} \mathbb{E}_{Y|x}\left[\ell(a, Y)\right]$$

which itself minimizes the expected loss on every possible input x separately. Therefore it corresponds to the *theoretically* best possible predictor over all functions since it is not constrained to a specific function class. However, this is only achievable with the knowledge of the true conditional distribution $\mathbb{P}_{Y|X}$. The question is whether we can find a closed-form solution for the optimal decision rule when choosing a concrete loss function.

Bonus Material

It might sound intuitive to you that minimizing the expected loss results in a good predictor, but you might ask yourself, why it results in the *best* possible predictor for supervised learning tasks. To this end, we provide a short proof that the Bayes' optimal predictor indeed minimizes the generalization error over all possible functions/decisions.

Proof. We calculate for the generalization error that

$$
\begin{aligned}
L(f^\star; \mathbb{P}_{X,Y}) &= \mathbb{E}_{X,Y}\left[\ell(f^\star(X), Y)\right] && (1)\\
&= \mathbb{E}_X\left[\mathbb{E}_{Y|X}\left[\ell(f^\star(X), Y)\right]\right] && (2)\\
&= \mathbb{E}_X\left[\min_{f(X)} \mathbb{E}_{Y|X}\left[\ell(f(X), Y)\right]\right] && (3)\\
&= \min_{f \text{ arbitrary}} \mathbb{E}_X\left[\mathbb{E}_{Y|X}\left[\ell(f(X), Y)\right]\right] && (4)\\
&= \min_{f \text{ arbitrary}} \mathbb{E}_{X,Y}\left[\ell(f(X), Y)\right] && (5)\\
&= \min_{f \text{ arbitrary}} L(f; \mathbb{P}_{X,Y}) && (6)
\end{aligned}
$$

where (1) holds by Definition 6.1 (generalization error), (2) by Theorem 3.42 (law of total expectation), (3) by Definition 13.9 (Bayes' optimal predictor), (4) since the expectation of $\mathbb{E}_{Y|X}\left[\ell(f(X), Y)\right]$ over X is minimized over an arbitrary function f and this minimizer exactly corresponds to the function which pointwise minimizes $\mathbb{E}_{Y|X}\left[\ell(f(X), Y)\right]$ over $f(X)$ for all $X = x$ as in (3), (5) by Theorem 3.42 (law of total expectation) and (6) by Definition 6.1 (generalization error). □

13.5 Probabilistic Perspective on Regression

In this section we provide concrete examples of how to define a discriminative statistical model and apply MLE and MAP inference for regression tasks, and we derive the Bayes' optimal decision rule for square loss. We will revisit previously discussed methods and gain a deeper statistical understanding of them.

13.5.1 Regression with MLE

Given a dataset $\mathcal{D}$, the first step is to define a statistical model capturing all our assumptions on the underlying data generation process. For example, we can assume that the outputs y are generated by

$$y = f(\boldsymbol{x}; \theta^\star) + \varepsilon \tag{13.7}$$

where ε models additive noise that is independent of $\boldsymbol{x}$ and with some true parameter $\theta^\star$. The function f is a function from some parameterized function class F. For example, the class of linear functions can be described with $F = \left\{\boldsymbol{x} \mapsto \boldsymbol{x}^\top \boldsymbol{w} \mid \boldsymbol{w} \in \mathbb{R}^d\right\}$ and θ would then correspond to the weight vector $\boldsymbol{w}$. It turns out that modeling the noise distribution is crucial in this setting.

Gaussian Noise. The most common assumption is that we have Gaussian noise $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, where for simplicity we assume that the noise variance σ^2 is fixed and known to us.[13]

[13] If the noise variance is unknown to us, we can include it as an unknown parameter in our statistical model. This means, we define our model with the parameters $\theta = (\gamma, \sigma)$ where γ are the unknown parameters of the function f and σ^2 is the unknown noise variance.

For now, we are only interested in predicting y from the inputs $\boldsymbol{x}$ and hence focus on discriminative models describing the conditional distribution $\mathbb{P}^\theta_{Y|X}$. From our statistical assumptions in Equation 13.7, we know that the randomness in y for a given $\boldsymbol{x}$ only comes from the random noise ε. Therefore, we model the conditional distribution with the parametric statistical model

$$Y \mid X = \boldsymbol{x} \sim \mathcal{N}(f(\boldsymbol{x}; \theta), \sigma^2) \tag{13.8}$$

with the conditional density

$$p(y \mid \boldsymbol{x}; \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y - f(\boldsymbol{x};\theta))^2}$$

and some unknown parameter θ.

We can now apply MLE to infer a parameter $\widehat{\theta}$ which estimates the true conditional distribution with $\theta^\star$ best. To this end, minimize the negative conditional log-likelihood, which by Equation 13.2 is equivalent to minimizing the sum over

$$\begin{aligned} -\log p(y_i \mid \boldsymbol{x}_i; \theta) &= -\log\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2}\right) \\ &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2 \end{aligned}$$

and by plugging the final expression into Equation 13.2 we obtain

$$\begin{aligned} \widehat{\theta}_{\text{MLE}} &= \operatorname*{arg\,min}_{\theta \in \Theta} \sum_{i=1}^n -\log p(y_i \mid \boldsymbol{x}_i; \theta) \\ &= \operatorname*{arg\,min}_{\theta \in \Theta} \sum_{i=1}^n \left(\frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2\right) \\ &= \operatorname*{arg\,min}_{\theta \in \Theta} \sum_{i=1}^n (y_i - f(\boldsymbol{x}_i;\theta))^2. \end{aligned} \tag{13.9}$$

As we can see, maximizing the likelihood of the data under i.i.d. Gaussian noise with constant and known variance as seen from the probabilistic perspective corresponds to minimizing the square loss $\ell(f(\boldsymbol{x};\theta), y) = (y - f(\boldsymbol{x};\theta))^2$ as seen from the algorithmic perspective. Hence, we arrived at a statistical motivation for the square loss, which from an optimization standpoint was only motivated by its convexity and differentiability. Note that if we instantiate the model in Equation 13.7 with linear functions f, the MLE estimator corresponds to linear regression as in Equation 4.1.

Exercise 13.11. Consider the statistical model $y_i = f(\boldsymbol{x}_i;\theta) + \varepsilon_i$ with $\varepsilon_i \sim \mathcal{N}(0, \sigma_i^2)$, where each sample includes Gaussian noise with a different variance σ_i^2. Find the corresponding algorithmic perspective for applying MLE to this statistical model.

Solution. Incorporating the change from $y_i = f(\boldsymbol{x}_i;\theta) + \varepsilon$ to $y_i = f(\boldsymbol{x}_i;\theta) + \varepsilon_i$ into the MLE estimate is straightforward, as it only involves replacing the Gaussian noise variance σ^2 with the sample-specific noise variance σ_i^2. Hence,

$$-\log p(y_i \mid \boldsymbol{x}_i;\theta) = \frac{1}{2}\log(2\pi\sigma_{\varepsilon,i}^2) + \frac{1}{2\sigma_{\varepsilon,i}^2}(y_i - f(\boldsymbol{x}_i;\theta))^2$$

and the MLE estimator corresponds to

$$\begin{aligned}\widehat{\theta}_{\text{MLE}} &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \frac{1}{2\sigma_i^2}(y_i - f(x_i;\theta))^2 \\ &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \frac{1}{\sigma_i^2}(y_i - f(x_i;\theta))^2\end{aligned}$$

similar to Equation 13.9. Thus, we have a weighted square loss - the larger the variance in the noise of label y_i, the less we should weigh it. □

Heavy-tailed Noise. Alternatively, one can model the noise in Equation 13.7 with a more heavy-tailed *non-standardized Student's t-distribution*, such that the conditional distribution of the true outputs y is assumed to be

$$p(y \mid \boldsymbol{x};\theta) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\pi\nu\sigma^2}\,\Gamma\left(\frac{\nu}{2}\right)}\left(1 + \frac{(y - f(\boldsymbol{x};\theta))^2}{\nu\sigma^2}\right)^{-\frac{\nu+1}{2}}, \qquad (13.10)$$

with parameters $\nu, \sigma > 0$, where a smaller ν corresponds to heavier tails.[14] The resulting MLE estimator is given by

$$\widehat{\theta}_{\text{MLE}} = \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \log\left(1 + \frac{(y_i - f(\boldsymbol{x}_i;\theta))^2}{\nu\sigma^2}\right)$$

and corresponds to regression with the loss

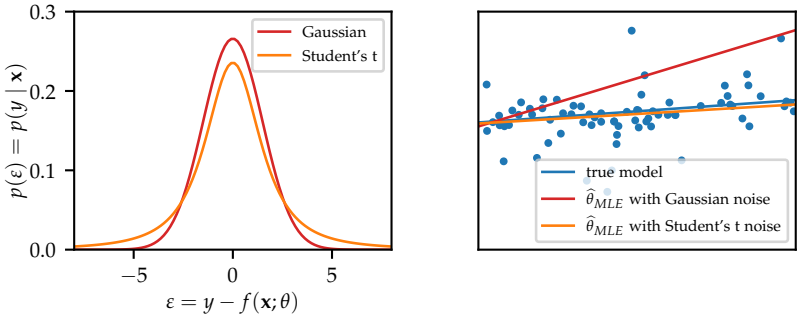
$$\ell(f(\boldsymbol{x};\theta), y) = \log\left(1 + \frac{(y - f(\boldsymbol{x};\theta))^2}{\nu\sigma^2}\right)$$

as seen from the algorithmic perspective. This modeling approach can lead to vastly different predictors, as can be seen in Figure 13.6.

[14] Be aware that the variance of a Student's t-distributed random variable is $\sigma^2\frac{\nu}{\nu-2}$, where σ^2 acts as a scaling parameter.

Noise Models and Loss Functions. In general for MLE, the choice of the likelihood function through the statistical model as in Equation 13.8 implicitly corresponds to the choice of the loss function as in Equation 13.9. This insight is important, since people deciding between potential loss functions and without statistical background are often not aware of the implicit statistical assumptions behind their decisions. Making wrong statistical assumptions can have severe impact on the model performance.

If the true noise in the data generating mechanism follows a heavy-tailed distribution, choosing the square loss might result in



(a) Gaussian and Student's t.

(b) Regressors fitted with MLE.

Figure 13.6: Gaussian vs. Student's t noise model. Observe in the left figure how the PDF of the Student's t-distribution decays much slower, which is referred to as a heavy tail. Observe in the right figure how the frequent outliers in the dataset caused by the heavy-tailed noise in the measurement process negatively impacts the performance of the MLE estimator based on the Gaussian noise model contrary to the one based on the Student's t noise model.

bad performance due to the mismatch between the true noise distribution and the family of distributions used to model the noise and induced by the square loss. From the algorithmic perspective, this bad performance can be explained with the squared distances to the frequent outliers having much larger influence on the estimator as depicted in Figure 13.6. Hence, a more robust choice against outliers would be the loss function based on the Student's t-distribution.

13.5.2 *Regression with MAP*

As in Equation 13.7, we again assume

$$y = f(\boldsymbol{x};\theta^\star) + \varepsilon \tag{13.11}$$

with some true parameter $\theta^\star$ and i.i.d. noise $\varepsilon \sim \mathcal{N}(0,\sigma^2)$, where the noise variance σ^2 is assumed to be fixed and known to us. Then our parametric statistical model for the conditional distribution corresponds to

$$Y \mid X = \boldsymbol{x} \sim \mathcal{N}(f(\boldsymbol{x};\theta),\sigma^2) \tag{13.12}$$

with conditional density

$$p(y \mid \boldsymbol{x},\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y-f(\boldsymbol{x};\theta))^2}$$

and some unknown parameter θ.

Gaussian Prior. From the Bayesian perspective, the parameter $\theta^\star$ is now considered as a random variable. This means, we also need a prior distribution for the parameter θ based on our prior knowledge or additional assumptions besides the statistical model. A simple choice is a *Gaussian prior* of the form

$$\theta^\star \sim \mathcal{N}(0,\sigma_\theta^2 I_d) \tag{13.13}$$

I_d denoting the d-dimensional identity matrix, with density

$$p(\theta) = \frac{1}{(2\pi\sigma_\theta^2)^{d/2}} \exp\left(-\frac{1}{2\sigma_\theta^2}\|\theta\|_2^2\right).$$

We apply the negative logarithm to both terms and get

$$\begin{aligned}
-\log p(y_i \mid \boldsymbol{x}_i, \theta) &= -\log\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2}\right) \\
&= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2 \\
-\log p(\theta) &= -\log\left(\frac{1}{(2\pi\sigma_\theta^2)^{d/2}} e^{-\frac{1}{2\sigma_\theta^2}\|\theta\|_2^2}\right) \\
&= \frac{d}{2}\log(2\pi\sigma_\theta^2) + \frac{1}{2\sigma_\theta^2}\|\theta\|_2^2
\end{aligned}$$

and by plugging them into Equation 13.6 we obtain

$$\begin{aligned}
\widehat{\theta}_{\text{MAP}} &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i, \theta) - \log p(\theta) \\
&= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \left(\frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2\right) \\
&\qquad + \frac{d}{2}\log(2\pi\sigma_\theta^2) + \frac{1}{2\sigma_\theta^2}\|\theta\|_2^2 \qquad (13.14) \\
&= \arg\min \theta \in \Theta \sum_{i=1}^{n} \frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2 + \frac{1}{2\sigma_\theta^2}\|\theta\|_2^2 \\
&= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} (y_i - f(\boldsymbol{x}_i;\theta))^2 + \frac{\sigma^2}{\sigma_\theta^2}\|\theta\|_2^2 .
\end{aligned}$$

This shows that applying MAP on our regression model in Equation 13.11 with i.i.d. Gaussian noise and a Gaussian prior on θ_i corresponds to ℓ_2-*regularized regression* with square loss and regularization parameter $\lambda = \frac{\sigma^2}{\sigma_\theta^2}$. Note that if we instantiate our model in Equation 13.11 with a linear function f, the MAP estimator corresponds to the solution obtained by *ridge regression*.

This explicit expression for the regularization parameter in terms of the data noise variance σ^2 and the prior parameter variance σ_θ^2 provide us some more insights and motivation for certain choices of the regularization parameter λ from a probabilistic perspective. Recall from 7.3.2 that we used ridge regression to regulate the model complexity with the choice of λ to prevent the models from fitting the noise in the data. Knowing that λ implicitly corresponds to $\frac{\sigma^2}{\sigma_\theta^2}$ allows us to further reason about this effect from the probabilistic perspective. When choosing a large λ, we implicitly assume that the data is quite noisy and we should not believe individual data points too much, and/or that the parameters have rather small variance corresponding to simpler models as explained in Subsection 7.2.1. In contrast, by choosing a small λ we assume that the data is quite accurate, such that it is preferred to interpolate the data points, and/or we assume that the parameters have a rather large variance corresponding to more complex models.

Laplace Prior. Another option for the prior distribution of θ_i is using a *Laplace prior* on the parameters

$$\theta_i^\star \sim \text{Laplace}(0, b) \quad \text{i.i.d. for all } i = 1, \ldots, d \qquad (13.15)$$

with density of the prior being

$$p(\theta_i) = \frac{1}{2b} e^{-\frac{|\theta_i|}{b}}$$

and $\mathrm{Var}(\theta_i) = 2b^2$. Applying negative logarithm for the prior yields

$$\begin{aligned} -\log p(\theta) &= -\log \prod_{i=1}^{d} p(\theta_i) = \sum_{i=1}^{d} -\log p(\theta_i) \\ &= \sum_{i=1}^{d} -\log\left(\frac{1}{2b} e^{-\frac{|\theta_i|}{b}}\right) = \sum_{i=1}^{d} \log(2b) + \sum_{i=1}^{d} \frac{|\theta_i|}{b} \\ &= \frac{1}{b}\sum_{i=1}^{d} |\theta_i| + d \cdot \log(2b) = \frac{1}{b}\|\theta\|_1 + d \cdot \log(2b) \end{aligned}$$

and thus, the MAP estimator is given by

$$\begin{aligned} \widehat{\theta}_{\mathrm{MAP}} &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i, \theta) - \log p(\theta) \\ &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \frac{1}{2\sigma^2}(y_i - f(\boldsymbol{x}_i;\theta))^2 + \frac{1}{b}\|\theta\|_1 \\ &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} (y_i - f(\boldsymbol{x}_i;\theta))^2 + \frac{2\sigma^2}{b}\|\theta\|_1 . \end{aligned} \tag{13.16}$$

This corresponds to ℓ_1-*regularized regression* with square loss and regularization parameter $\lambda = \frac{2\sigma^2}{b}$. If we instantiate our model in Equation 13.11 with a linear function f, the MAP estimator corresponds to the solution obtained by *LASSO regression*.

Priors and Regularization. Deriving ℓ_2 and ℓ_1 regularization from the probabilistic perspective provides us with a new interpretation of their regularization behaviors based on the used priors as seen in Figure 13.7. Regularizing with the ℓ_2-norm assumes that the weights are distributed with a Gaussian prior from Equation 13.13, which leads to weights closer to mean zero. In contrast, ℓ_1-regularization assumes that the weights are distributed with a Laplace prior from Equation 13.15 and assigns higher probability to weights being exactly zero, which leads to sparser weight vectors. An empirical example in the context of ridge and lasso regression was previously shown in Figure 7.19.

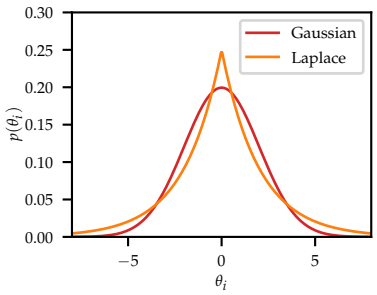


Figure 13.7: Gaussian vs. Laplace prior for weights of linear regression.

In general for MAP and MLE, the choice of the likelihood function through the statistical model is equivalent to the choice of the loss function to be minimized. In addition for MAP, the choice of the prior distribution of parameter θ is equivalent to the choice of the regularization.

13.5.3 *Bayes Optimal Regression using Square Loss*

Let us come back to the decision-theoretic view on regression, and derive closed-form optimal decision rules for square loss $\ell(f(x), y) = (f(x) - y)^2$. Based on this model, we can make predictions using the Bayesian optimal decision rule

$$f^\star(x) := \arg\min_{a\in\mathcal{Y}} \mathbb{E}\left[\ell(a, Y) \mid X = x\right]$$

as defined in Definition 13.9, where the minimization happens over $\mathcal{Y} = \mathbb{R}$.

The optimal decision rule can be derived by setting the first derivative of $\mathbb{E}[\ell(a, Y) \mid X = x]$ to zero (Theorem 2.4), yielding

$$\begin{aligned}\frac{\partial}{\partial a}\mathbb{E}[\ell(a, Y) \mid X = x] &= \frac{\partial}{\partial a}\left(a^2 - 2a\mathbb{E}[Y \mid X = x] + \mathbb{E}\left[Y^2 \mid X = x\right]\right) \\ &= 2a - 2\mathbb{E}[Y \mid X = x] \\ &\stackrel{!}{=} 0.\end{aligned}$$

Solving for a yields the Bayes' optimal predictor

$$f^\star(x) = \arg\min_{a \in \mathcal{Y}} \mathbb{E}[\ell(a, Y) \mid X = x] = \mathbb{E}[Y \mid X = x] = \int_{-\infty}^{\infty} y p_{Y|x}(y) dy$$

where $\mathbb{E}[Y \mid X = x]$ is the mean of the conditional distribution $\mathbb{P}_{Y|X=x}$.

As we can see in Figure 13.8, the Bayes' optimal predictor under the squared loss always predicts the conditional mean. This makes sense since the squared loss penalizes larger deviations stronger than smaller deviations.

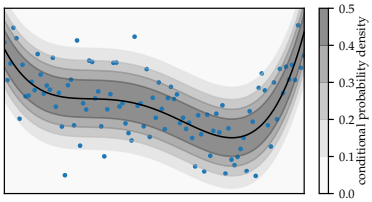


Figure 13.8: Optimal decision rule for squared loss. This figure shows the conditional probability densities (contour lines) of our model for $\mathbb{P}_{Y|X=x}$ estimated from the given dataset (blue dots). By using the squared loss, we can observe that the resulting Bayes' optimal predictor (black line) always predicts the conditional mean.

13.6 Probabilistic Perspective on Classification

We now discuss how the probabilistic modeling perspective gives rise to different methods to solve classification tasks.

13.6.1 Discriminative and Generative Modeling for Classification

For regression, we assumed that the data generation follows Equation 13.7. In particular, when trying to find $f^\star$, we only needed to model the observation noise ε, effectively modeling the conditional probability distribution $Y|X$, cf. Equation 13.8. For classification, one can either do the same or, alternatively, model the joint distribution of X and Y by first modeling the distribution of Y and then modeling the conditional distribution of $X|Y = y$ for each class y.

The first approach is generally referred to as *discriminative modeling*, and the latter as *generative modeling*. More concretely, discriminative modeling entails explicitly modeling or parametrizing the conditional distribution $\mathbb{P}_{Y|X}$, while modeling the marginal $\mathbb{P}_X$ for completeness is optional, whereas generative modeling explicitly parametrizes both the conditional $\mathbb{P}_{X|Y}$ and marginal $\mathbb{P}_Y$. This difference can be seen in Figure 13.9.

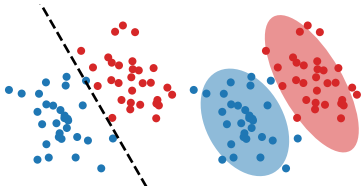


Figure 13.9: The discriminative model (left) uses $\mathbb{P}_{Y|X}$ to find a decision boundary between different y. The generative model (right) uses $\mathbb{P}_{X|Y}$ to model the distributions of x for different y.

Note that while generative modeling automatically models the full joint distribution $\mathbb{P}_{X,Y}$, discriminative models can also be used to do so if we also parameterize the marginal $\mathbb{P}_X$. The distinguishing factor is how the joint distribution is factorized. Using discriminative modeling, one may decompose $\mathbb{P}_{X,Y}$ into the marginal

distribution $\mathbb{P}_X$ and the conditional distribution $\mathbb{P}_{Y|X}$ and model each of them separately

$$\mathbb{P}_{X,Y} = \mathbb{P}_{Y|X} \cdot \mathbb{P}_X.$$

Using generative modeling, one may decompose $\mathbb{P}_{X,Y}$ into the marginal distribution $\mathbb{P}_Y$ and the conditional distribution $\mathbb{P}_{X|Y}$ such that the joint probabilities read

$$\mathbb{P}_{X,Y} = \mathbb{P}_{X|Y} \cdot \mathbb{P}_Y.$$

For modeling joint distributions, this option has the advantage that the prior $\mathbb{P}_Y$ over the outputs (e.g., a finite set of class labels) is typically less complex and easier to model than the prior $\mathbb{P}_X$ over the inputs (e.g. space of images).

Typically, the choice between discriminative and generative modeling depends on the task at hand. For example, when the input features X are symptoms of a patient (headache, loss of taste, fever,...), and the output Y is the disease that the patient has (e.g, the flu, Covid-19 etc.), there might be a lot of overlap between the symptoms of different diseases, and hence the relationship $X|Y$ might be more natural to model or parametrize than $Y|X$.

- If we only want to predict y from $\boldsymbol{x}$ (e.g., given some image $\boldsymbol{x}$, what is the probability of y = cat?), discriminative models can sometimes be more favorable in terms of performance, as they directly learn $\mathbb{P}_{Y|X}$ which can be more computationally efficient than learning $\mathbb{P}_{X|Y}$ and $\mathbb{P}_Y$.[15] However, generative models can also be used for prediction, as discussed in Subsection 13.6.3.
- If we want to generate new samples x from a certain class y (e.g., given the label y = cat, sample a new image $\boldsymbol{x}$ of a cat, see Figure 13.10), generative models can sometimes be more favorable since they explicitly parametrize $\mathbb{P}_{X|Y}$ which we can sample from. However, in principle, discriminative models can also be used for sampling new datapoints when also learning $\mathbb{P}_X$.[16]

Figure 13.10: Images generated by conditioning on the classes y=cat and y=tiger using DuDGAN (Yeom, Gu, and Lee 2024).

13.6.2 *Classification using Discriminative Models*

We first focus on binary classification with labels $y \in \{+1, -1\}$. The first step is to define a statistical model to capture all assumptions on the underlying data generation process of a given dataset $\mathcal{D}$. We assume that the labels are generated based on a parametric function $f(\cdot, \theta^\star) \in F$, sometimes referred to as the *discriminator*, with some true parameter $\theta^\star$. If we assume that F is the class of linear functions $\boldsymbol{x} \mapsto \boldsymbol{x}^\top \boldsymbol{w}$, the parameter $\theta^\star$ will correspond to the weight vector $\boldsymbol{w}^\star$, for example. We define a model for the conditional probability $p(y \mid \boldsymbol{x})$, using a *logistic model* based on the logistic function[17] $\sigma(z) = \frac{1}{1+e^{-z}}$ as our parametric statistical model which defines the conditional distribution[18]

$$Y \mid X = \boldsymbol{x} \sim \text{Ber}(\sigma(f(\boldsymbol{x}; \theta))) \tag{13.17}$$

[15] This is for example discussed in the seminal work by Ng and Jordan (2001).

[16] In total, the distinction between generative and discriminative models can be a bit misleading, since both are capable of generating and discriminating datapoints.

[17] The logistic function is a member of the class of sigmoid functions which are characterized by their S-shaped curve. You have already encountered them as activation functions for neural networks in Section 10.1.

[18] The Bernoulli distribution is commonly used for binary random variables with outcome $\{0, 1\}$, but we can similarly use it for $\{+1, -1\}$.

such that the probability mass function on $\{-1, +1\}$ is

$$\begin{aligned} p(y \mid \boldsymbol{x}; \theta) &= \begin{cases} \sigma(f(\boldsymbol{x}; \theta)), & y = +1 \\ 1 - \sigma(f(\boldsymbol{x}; \theta)), & y = -1 \end{cases} \\ &= \sigma(y f(\boldsymbol{x}; \theta)) \\ &= \frac{1}{1 + e^{-y f(\boldsymbol{x}; \theta)}} \end{aligned}$$

and can be used for classification by predicting the most likely class label

$$\widehat{y} = \operatorname*{arg\,max}_{y \in \{+1, -1\}} p(y \mid \boldsymbol{x}; \theta).$$

Compared to the Gaussian noise model for regression as given in Equation 13.8, one can think of Equation 13.17 as "Bernoulli noise" with the noisy outputs limited to the discrete set $\{+1, -1\}$.

As visualized in Figure 13.11, the logistic function returns probability 0.5 for $f(\boldsymbol{x}; \theta) = 0$ and approaches probability 1 for positive $y f(\boldsymbol{x}; \theta)$ and 0 for negative $y f(\boldsymbol{x}; \theta)$. We can interpret $\sigma(y f(\boldsymbol{x}; \theta))$ as the probability that the discriminator function predicts the given y (i.e. both have the same sign).[19]

[19] The formal justification is that the logistic function satisfies the symmetric property $1 - \sigma(z) = \sigma(-z)$.

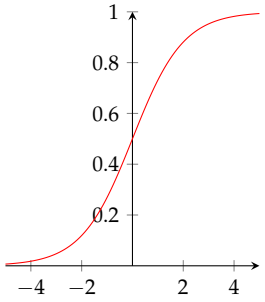


Figure 13.11: Logistic Function

Maximum Likelihood Estimation. As before, we apply MLE to find the parameter $\widehat{\theta}$ which estimates the true conditional distribution with $\theta^\star$ best. We start with deriving the negative log-likelihood

$$\begin{aligned} -\log p(y_i \mid \boldsymbol{x}_i; \theta) &= -\log \left(\frac{1}{1 + e^{-y_i f(\boldsymbol{x}_i; \theta)}} \right) \\ &= \log \left(1 + e^{-y_i f(\boldsymbol{x}_i; \theta)} \right) \end{aligned}$$

and by plugging this into Equation 13.2 we obtain

$$\begin{aligned} \widehat{\theta}_{\text{MLE}} &= \operatorname*{arg\,min}_{\theta \in \Theta} \sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i; \theta) \\ &= \operatorname*{arg\,min}_{\theta \in \Theta} \sum_{i=1}^{n} \log \left(1 + e^{-y_i f(\boldsymbol{x}_i; \theta)} \right). \end{aligned} \tag{13.18}$$

As we can see, maximizing the likelihood under the logistic model as seen from the probabilistic perspective corresponds to minimizing the logistic loss $\ell(f(\boldsymbol{x}; \theta), y) = \log \left(1 + e^{-y f(\boldsymbol{x}; \theta)} \right)$ as seen from the algorithmic perspective, both known as *logistic regression*. Hence, we arrived at a new statistical interpretation and motivation for the logistic loss, which we motivated in Section 8.3 purely from an optimization view.

This observation again underlines our insight from Subsection 13.5.1 that the choice of the loss function implicitly contains statistical assumptions on the data. In this case, the logistic loss assumes that the class probabilities are given by the logistic function as visualized in Figure 13.11.

Maximum a Posteriori Estimation. As before in Subsection 13.5.2, we can adopt the Bayesian's perspective and consider the unknown parameter $\theta^\star$ of the logistic model as a random variable with a prior distribution. By applying MAP we can infer the parameter $\widehat{\theta}$ which best estimates the true conditional distribution with $\theta^\star$. In the case of regression with a Gaussian noise model, we observed that a *Gaussian prior* for θ of the form

$$\theta^\star \sim \mathcal{N}(0, \sigma_\theta^2 I_d) \tag{13.19}$$

leads to ℓ_2-regularized regression with the square loss. Under the logistic model, using the same prior, we obtain the MAP estimator

$$\begin{aligned}
\widehat{\theta}_{\text{MAP}} &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i, \theta) - \log p(\theta) \\
&= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \log\left(1 + e^{-y_i f(\boldsymbol{x}_i;\theta)}\right) + \frac{d}{2}\log(2\pi\sigma_\theta^2) + \frac{1}{2\sigma_\theta^2}\|\theta\|_2^2 \\
&= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \log\left(1 + e^{-y_i f(\boldsymbol{x}_i;\theta)}\right) + \frac{1}{2\sigma_\theta^2}\|\theta\|_2^2
\end{aligned} \tag{13.20}$$

corresponding to ℓ_2-*regularized logistic regression* with regularization parameter $\lambda = \frac{1}{2\sigma_\theta^2}$.

Similarly, using a *Laplace prior* for θ of the form

$$\theta_i^\star \sim \text{Laplace}(0, b) \quad \text{i.i.d. for all } i = 1, \ldots, d \tag{13.21}$$

with prior density

$$p(\theta_i) = \frac{1}{2b} e^{-\frac{|\theta_i|}{b}}$$

we obtain the MAP estimator

$$\begin{aligned}
\widehat{\theta}_{\text{MAP}} &= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} -\log p(y_i \mid \boldsymbol{x}_i, \theta) - \log p(\theta) \\
&= \arg\min_{\theta\in\Theta} \sum_{i=1}^{n} \log\left(1 + e^{-y f(\boldsymbol{x};\theta)}\right) + \frac{1}{b}\|\theta\|_1 .
\end{aligned} \tag{13.22}$$

corresponding to ℓ_1-*regularized logistic regression* with regularization parameter $\lambda = \frac{1}{b}$.

This again shows that regularizing can be equivalent to placing prior assumptions on the distribution of parameter $\theta^\star$.

13.6.3 Classification using Generative Models

So far, we have only discussed discriminative models, seen in Chapter 13, which aim to estimate the conditional probability $p(y \mid \boldsymbol{x})$ for given $\boldsymbol{x}$ and y. Instead of modeling $\mathbb{P}_{Y|X}$, generative modeling also tries to model the distribution over the data itself, by aiming to estimate the *joint distribution* $\mathbb{P}_{X,Y}$. As discussed in Section 13.2, the generative model is more expressive, as the conditional distribution can be derived from the joint distribution by

$$p(y \mid \boldsymbol{x}) = \frac{p(y, \boldsymbol{x})}{p(\boldsymbol{x})} = \frac{p(y, \boldsymbol{x})}{\int_{y'} p(y', \boldsymbol{x})\, dy'}.$$

However, deriving the joint distribution from the conditional distribution is not possible. We can apply Bayes' rule (Theorem 3.30) and get

$$p(y \mid \boldsymbol{x}) \propto p(y)p(\boldsymbol{x} \mid y).$$

So, one way to approach this is by first estimating the prior probabilities over the classes $p(y)$, and afterwards estimate $p(\boldsymbol{x} \mid y)$, the conditional distribution over $\boldsymbol{x}$ for each class y.

Gaussian Naive Bayes Model. The *naive Bayes* models the class distribution as a categorical random variable with

$$\mathbb{P}(Y = y) = p_y \quad \text{where } y \in \mathcal{Y} = \{1, \dots, c\} \text{ and } \sum_{y=1}^{c} p_y = 1.$$

For the appearance of the given class Y, the naive Bayes model makes the conditional independence assumption

$$p(x_1, \dots, x_d \mid y) = \prod_{j=1}^{d} p(x_j \mid y),$$

i.e., given a class label, each feature (dimension) is generated independently of the other features.[20] This dramatically reduces the model complexity of the joint distribution, as instead of forming an arbitrary complicated joint distribution over d-dimensional feature vectors, we can describe only conditional distributions over one-dimensional features for each class label.

[20] This independence assumption is the reason it is called *naive* Bayes. There are generalizations that do not use this independence assumption, such as the *Gaussian Bayes classifier*.

This one-dimensional conditional distribution still needs to be defined. If one uses a Gaussian distribution,

$$X_j \mid Y = y \sim \mathcal{N}(\mu_{y,j}, \sigma^2_{y,j}),$$

the corresponding MLE is called the *Gaussian Naive Bayes* (GNB) classifier. For this, we need to estimate $c - 1$ parameters for the prior on the labels (the class frequencies) and then for each class we have to estimate a mean and a variance for the respective conditional Gaussian distribution. These parameters can be estimated using maximum likelihood estimators, see Definition 13.5.

Lemma 13.12. *Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\}$, the maximum likelihood estimates in the Gaussian Naive Bayes model are*

$$\widehat{p}_y = \frac{n_y}{n}, \quad \widehat{\mu}_{y,j} = \frac{1}{n_y} \sum_{i:y_i=y} x_{i,j}, \quad \widehat{\sigma}^2_{y,j} = \frac{1}{n_y} \sum_{i:y_i=y} \left(x_{i,j} - \hat{\mu}_{y,j}\right)^2$$

where n_y denotes the frequency of label y in the dataset $\mathcal{D}$.

Proof. We start by estimating the class frequencies p_y. The log-likelihood is given by

$$\log p(\mathcal{D} \mid \{p_y\}_{y=1}^{c}) = \sum_{i=1}^{n} \log p_{y_i} = \sum_{y=1}^{c} n_y \log p_y,$$

where the first equality holds by the categorical model, and n_y denotes the frequency of label y in the dataset $\mathcal{D}$. To maximize the likelihood while keeping the constraint that $\sum_y p_y = 1$, we can use the method of Lagrange multipliers to find a maximum value for the parameters p_y. The Lagrangian is given by

$$\mathcal{L}(\{p_y\}_{y=1}^c, \lambda) = \sum_{y=1}^{c} n_y \log(p_y) - \lambda \left(\sum_{y=1}^{c} p_y - 1 \right),$$

and differentiating this with respect to the parameters p_y and λ yields the gradient

$$\frac{\partial \mathcal{L}(\{p_y\}_{y=1}^c, \lambda)}{\partial p_y} = \frac{n_y}{p_y} - \lambda, \quad \frac{\partial \mathcal{L}(\{p_y\}_{y=1}^c, \lambda)}{\partial \lambda} = -\left(\sum_{y=1}^{c} p_y - 1 \right).$$

Setting all gradients to zero, some simple algebra yields

$$p_y = \frac{n_y}{\lambda}, \quad \sum_{y=1}^{c} p_y = \frac{\sum_{y=1}^{c} n_y}{\lambda} = 1,$$

from which we can conclude that $\lambda = n$ and thus the relative frequencies of the classes $\widehat{p}_y = n_y/n$ are the maximum likelihood estimators for the class frequency parameters.

We continue with the means and variances. The log-likelihood of the parameters for the one-dimensional Gaussians is given by

$$\begin{aligned}
&\log p\left(\mathcal{D} \mid \left\{\mu_{y,j}, \sigma_{y,j}^2\right\}_{y,j}\right) \\
&= \sum_{i=1}^{n} \log p\left((x_i, y_i) \mid \left\{\mu_{y,j}, \sigma_{y,j}^2\right\}_{y,j}\right) \\
&= \sum_{y=1}^{c} \sum_{i:y_i=y} \log p\left(x_i \mid \left\{\mu_{y,j}, \sigma_{y,j}^2\right\}_{y,j}, y\right) p(y) \\
&= \sum_{y=1}^{c} \sum_{i:y_i=y} \sum_{j=1}^{d} \log p\left(x_{i,j} \mid \mu_{y,j}, \sigma_{y,j}^2, y\right) p(y) && (13.23) \\
&= \sum_{y=1}^{c} \sum_{i:y_i=y} \sum_{j=1}^{d} \log \frac{1}{\sqrt{2\pi\sigma_{y_i,j}^2}} \exp\left(-\frac{1}{2}\frac{(x_{i,j} - \mu_{y_i,j})^2}{\sigma_{y_i,j}^2}\right) p(y) && (13.24) \\
&= \sum_{y=1}^{c} \sum_{j=1}^{d} \left(-\frac{n_y}{2}\log(2\pi) - \frac{n_y}{2}\log\left(\sigma_{y,j}^2\right) - \sum_{i:y_i=y} \frac{(x_{i,j} - \mu_{y,j})^2}{2\sigma_{y,j}^2} - \log p(y) \right)
\end{aligned}$$

where Equation 13.23 holds by the independence assumption between features and Equation 13.24 holds because we assume a Gaussian distribution in each feature. Hence, the partial derivatives are given by

$$\begin{aligned}
\frac{\partial \log p\left(\mathcal{D} \mid \left\{\mu_{y,j}, \sigma_{y,j}^2\right\}_{y,j}\right)}{\partial \mu_{y,i}} &= \sum_{i:y_i=y} \frac{x_{i,j} - \mu_{y,j}}{\sigma_{y,j}^2} \\
\frac{\partial \log p\left(\mathcal{D} \mid \left\{\mu_{y,j}, \sigma_{y,j}^2\right\}_{y,j}\right)}{\partial \sigma_{y,i}^2} &= -\frac{n_y}{2\sigma_{y,j}^2} + \sigma_{y,j}^{-4} \sum_{i:y_i=y} \frac{(x_{i,j} - \mu_{y,j})^2}{2}.
\end{aligned}$$

Setting the derivative to zero and rearranging leads to

$$\widehat{\mu}_{y,j} = \frac{1}{n_y} \sum_{i:y_i=y} x_{i,j} \quad \text{and} \quad \widehat{\sigma}^2_{y,j} = \frac{1}{n_y} \sum_{i:y_i=y} (x_{i,j} - \widehat{\mu}_{y,j})^2 ,$$

which concludes the proof. □

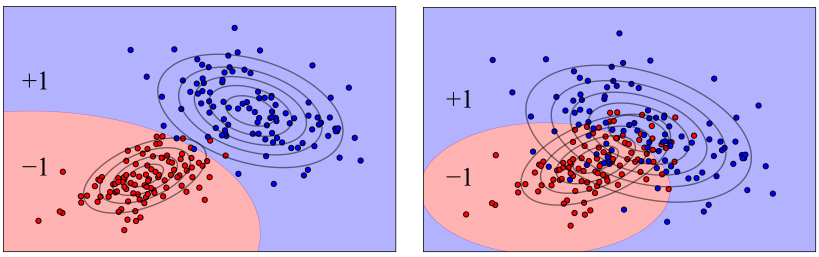
Once we have estimated the parameters, as presented in Lemma 13.12, we would like to make predictions on new datapoints. Given a new data point $\boldsymbol{x}$, the predicted label is the maximizer of the conditional distribution, which we can rewrite using Bayes Theorem as

$$\begin{aligned}
\widehat{y} &= \arg\max_y \ \log \widehat{p}(y \mid \boldsymbol{x}) \\
&= \arg\max_y \ \log \widehat{p}(y) + \sum_{j=1}^{d} \log \widehat{p}(x_j \mid y) \\
&= \arg\max_y \ \log \widehat{p}_y - \frac{1}{2} \sum_{j=1}^{d} \left(\log \widehat{\sigma}^2_{y,j} + \frac{1}{\widehat{\sigma}^2_{y,j}} (x_j - \widehat{\mu}_{y,j})^2 \right) .
\end{aligned}$$

For the special case of binary classification, it is easy to see that this decision rule is equivalent to the prediction

$$\widehat{y} = \operatorname{sign}(\widehat{f}(\boldsymbol{x})) := \operatorname{sign}\left(\log \frac{\widehat{p}(+1 \mid \boldsymbol{x})}{\widehat{p}(-1 \mid \boldsymbol{x})} \right),$$

where the function $\widehat{f}$ denotes the estimated *log-posterior-odds* of the class probability, and acts as the *discriminant function*. Intuitively, $\widehat{p}(+1 \mid \boldsymbol{x})$ is greater than $\widehat{p}(-1 \mid \boldsymbol{x})$ if and only if $\widehat{f}(\boldsymbol{x}) > 0$, and thus taking the sign of discriminant function picks the class which is more likely under $\boldsymbol{x}$. An example of the corresponding decision boundary can be seen in Figure 13.12.



(a) Means further apart. (b) Closer means.

Figure 13.12: Gaussian Naive Bayes model fitted via MLE to samples from two Gaussian mixture models with different means, indicated by the contour lines of each cluster. The closer the means, the more curved the decision boundary becomes.

Note that the decision boundary can be curved, as the different estimated Gaussian distributions have different covariance matrices.

Fisher's Linear Discriminant Analysis. Now let's consider the special case of the GNB binary classifier when the variance is shared across the classes, i.e., $\sigma^2_{+1,j} = \sigma^2_{-1,j} = \sigma^2_j$ for all $j \in \{1, \ldots, d\}$). We use the short-hand notation $+/-$ for the classes

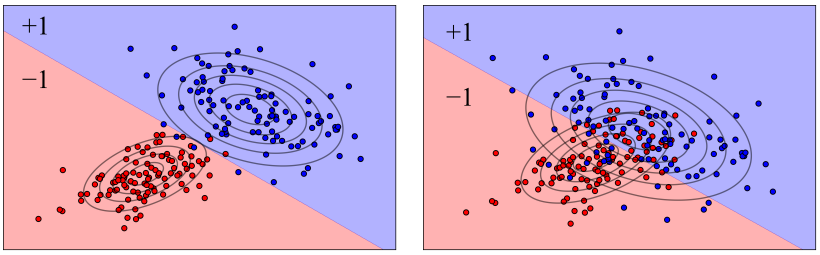
$+1/-1$. The discriminant function takes the form

$$\begin{aligned}
\widehat{f}(\boldsymbol{x}) &= \log \frac{\widehat{p}(+ \mid \boldsymbol{x})}{\widehat{p}(- \mid \boldsymbol{x})} \\
&= \log \frac{\widehat{p}_+ \prod_{j=1}^d p(x_j \mid +)/p(\boldsymbol{x})}{\widehat{p}_- \prod_{j=1}^d p(x_j \mid -)/p(\boldsymbol{x})} \\
&= \log \frac{\widehat{p}_+}{1-\widehat{p}_+} + \log \frac{\prod_{j=1}^d \frac{1}{\sqrt{2\pi\widehat{\sigma}_j^2}} \exp\left(-\frac{(x_j-\widehat{\mu}_{+,j})^2}{2\widehat{\sigma}_j^2}\right)}{\prod_{j=1}^d \frac{1}{\sqrt{2\pi\widehat{\sigma}_j^2}} \exp\left(-\frac{(x_j-\widehat{\mu}_{-,j})^2}{2\widehat{\sigma}_j^2}\right)} \\
&= \log \frac{\widehat{p}_+}{1-\widehat{p}_+} + \sum_{j=1}^d \frac{1}{2\widehat{\sigma}_j^2} \left(-(x_j-\widehat{\mu}_{+,j})^2 + (x_j-\widehat{\mu}_{-,j})^2\right) \\
&= \underbrace{\sum_{j=1}^d \frac{1}{\widehat{\sigma}_j^2} \left(\widehat{\mu}_{+,j} - \widehat{\mu}_{-,j}\right) x_j}_{=:\boldsymbol{w}^\top \boldsymbol{x}} + \underbrace{\log \frac{\widehat{p}_+}{1-\widehat{p}_+} + \sum_{j=1}^d \frac{\widehat{\mu}_{-,j}^2 - \widehat{\mu}_{+,j}^2}{2\sigma_j^2}}_{=:w_0}.
\end{aligned}$$

Therefore, when performing binary classification with a GNB classifier with shared variance across the classes, then in the end we fit a linear classifier

$$\widehat{y} = \operatorname{sign}(\widehat{f}(\boldsymbol{x})) = \operatorname{sign}(\boldsymbol{w}^\top \boldsymbol{x} + w_0),$$

just like we have seen in Section 8.2. This is called *Fisher's linear discriminant analysis*. In Figure 13.13, one can see the resulting model in the same example we have discussed for general naive Gaussian Bayes. Notably, the decision boundary is not curved.



(a) Means further apart. (b) Closer means.

Figure 13.13: Fisher's linear discriminant analysis model fitted to samples from two Gaussian mixture models with different means, indicated by the contour lines of the distribution. The decision boundary is always linear.

13.6.4 *Bayes Optimal Classification*

In this section, we derive closed-form optimal decision rules for concrete loss functions used in the context of classification tasks. Given any joint probability distribution,[21] we can make predictions using the Bayesian optimal decision rule for that distribution, more commonly referred to as the *Bayes' optimal classifier*

$$f^\star(x) := \operatorname*{arg\,min}_{a \in \mathcal{Y}} \mathbb{E}\left[\ell(a, Y) \mid X = x\right]$$

as defined in Definition 13.9, where the minimization happens over a finite domain $\mathcal{Y}$ corresponding to the set of possible labels. Therefore, the optimal decision rule can be evaluated by case distinction

[21] In what follows we assume that $\mathbb{P}$ is the true distribution - if we only know the learned distribution $\widehat{\mathbb{P}}$, we only obtain an estimate of the optimal decision rule.

over all possible, finitely many predictions $a \in \mathcal{Y}$. In addition, the finite domain $\mathcal{Y}$ allows one to derive decision boundaries between different predictions a, where the expected loss $\mathbb{E}\left[\ell(a, Y) \mid X = x\right]$ is equal for different a.

0-1-Loss. Recall that the most intuitive loss for classification tasks is the 0-1-loss defined as

$$\ell(f(x), y) = \mathbb{I}_{f(x) \neq y} = \begin{cases} 0, & f(x) = y \\ 1, & f(x) \neq y \end{cases}.$$

We can derive that the expected loss

$$\mathbb{E}\left[\ell(a, Y) \mid X = x\right] = \mathbb{P}(Y \neq a \mid X = x) = 1 - p(a \mid x)$$

corresponds to the probability of falsely predicting a. This means that the Bayes' optimal classifier

$$\begin{aligned} f^\star(x) &= \arg\min_{a \in \mathcal{Y}} \mathbb{E}\left[\ell(a, Y) \mid X = x\right] \\ &= \arg\min_{a \in \mathcal{Y}} 1 - p(a \mid x) \\ &= \arg\max_{a \in \mathcal{Y}} p(a \mid x) \end{aligned} \tag{13.25}$$

always predicts the label with the lowest misprediction probability, which is equivalent to predicting the class with the highest conditional probability.

Observe that the Bayes' optimal classifier might not be unique for certain x, since there can be multiple labels $a \in \mathcal{Y}$ with the same conditional probability $p(a \mid x)$. These x are known as the *decision boundary*, where the optimal action is ambiguous due to different actions having equal expected loss.

In the case of binary classification with $\mathcal{Y} = \{-1, +1\}$ the Bayes' optimal classifier is given by

$$f^\star(x) = \begin{cases} -1, & p(+1 \mid x) \leq p(-1 \mid x) \\ +1, & p(+1 \mid x) > p(-1 \mid x) \end{cases}$$

with decision boundary at $p(-1 \mid x) = p(+1 \mid x) = 0.5$ as visualized in Figure 13.14.

Asymmetric 0-1-loss. One can also define an asymmetric loss for classification tasks which penalizes the wrong prediction of different labels differently. The asymmetric loss can be defined as

$$\ell(f(x), y) = c_{f(x)} \mathbb{I}_{f(x) \neq y} = \begin{cases} 0, & f(x) = y \\ c_{f(x)}, & f(x) \neq y \end{cases}$$

which penalizes the incorrect prediction of the label $f(x)$ with the cost $c_{f(x)}$. The expected loss

$$\mathbb{E}\left[\ell(a, Y) \mid X = x\right] = \mathbb{E}_{Y|x}\left[\mathbb{I}_{a \neq y}\right] c_a = (1 - p(a \mid x)) c_a$$

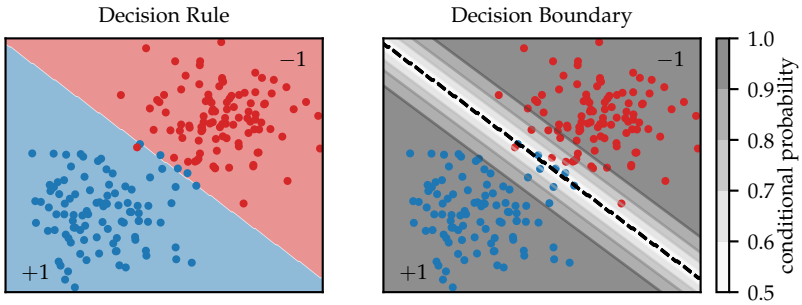


Figure 13.14: Optimal decision rule for 0-1-loss. The left figure shows the decision regions for the binary labels $\mathcal{Y} = \{-1, +1\}$ (red and blue regions) and the right figure visualizes the conditional probabilities (contour lines) of the most likely label in addition to the decision boundary (dotted line). Observe that this decision rule makes both false negatives (blue dots in red region) and false positives (red dots in blue region).

is then given by the misprediction probability when predicting the label a weighted by the misprediction cost c_a. This results into the Bayes' optimal classifier

$$f^\star(x) = \arg\min_{a \in \mathcal{Y}} (1 - p(a \mid x)) c_a$$

which has to tradeoff between minimizing the conditional misprediction probability $1 - p(a \mid x)$ and minimizing cost c_a.

In the context of binary classification, the asymmetric loss reduces to

$$\ell(f(x), y) = \begin{cases} 0, & f(x) = y \\ c_{\text{FN}}, & f(x) = -1 \text{ and } y = +1 \\ c_{\text{FP}}, & f(x) = +1 \text{ and } y = -1 \end{cases}$$

with c_{FN} corresponding to the cost of falsely predicting $f(x) = -1$ (i.e. a false negative) and c_{FP} corresponding to the cost of falsely predicting $f(x) = +1$ (i.e. a false positive). The expected costs are then given as

$$\mathbb{E}\left[\ell(-1, Y) \mid X = x\right] = (1 - p(-1 \mid x)) c_{\text{FN}} = p(+1 \mid x) c_{\text{FN}}$$
$$\mathbb{E}\left[\ell(+1, Y) \mid X = x\right] = (1 - p(+1 \mid x)) c_{\text{FP}} = p(-1 \mid x) c_{\text{FP}}$$

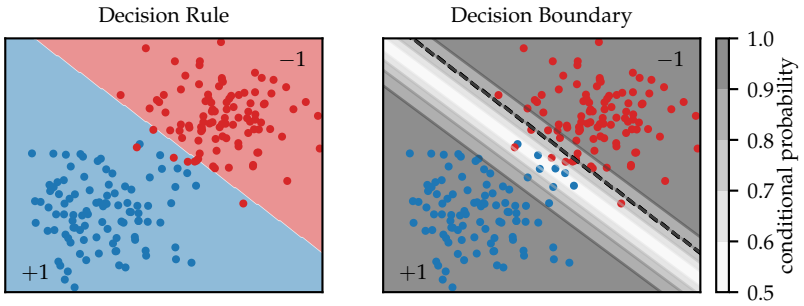
which results into the Bayes' optimal classifier

$$f^\star(x) = \begin{cases} -1, & p(+1 \mid x) c_{\text{FN}} < p(-1 \mid x) c_{\text{FP}} \\ +1, & p(+1 \mid x) c_{\text{FN}} > p(-1 \mid x) c_{\text{FP}} \end{cases}$$

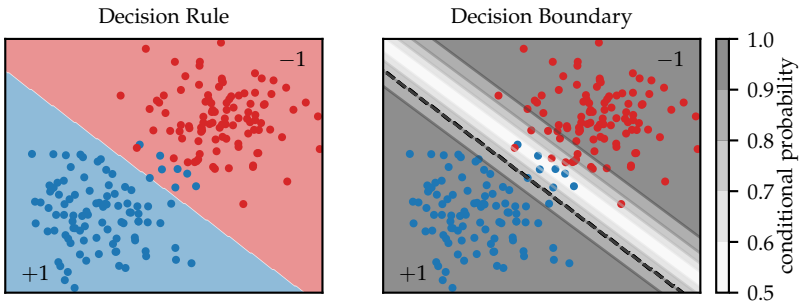
with decision boundary at $p(-1 \mid x) = \frac{c_{\text{FN}}}{c_{\text{FN}} + c_{\text{FP}}}$ or equivalently $p(+1 \mid x) = \frac{c_{\text{FP}}}{c_{\text{FN}} + c_{\text{FP}}}$.

As one can observe in Figure 13.15, the different costs for false negatives and false positives cause the decision boundary to be shifted.

Exercise 13.13. Derive the decision boundary $p(+1 \mid x) = \frac{c_{FP}}{c_{FN} + c_{FP}}$ for the asymmetric 0-1-loss in the case of binary classification.



(a) Penalizing false negatives.



(b) Penalizing false positives.

Figure 13.15: Optimal decision rules for different asymmetric o-1-losses. In the upper figure, we penalize false negatives stronger by choosing $c_{\text{FN}} = 4$ and $c_{\text{FP}} = 1$. We can observe that the decision rule prefers to predict $+1$ and only predicts -1 if $p(-1 \mid x) > \frac{4}{4+1} = 0.8$. For the given dataset, this decision rule does not make any false negative predictions, since all points in the red region indeed have label -1. In the figure below, we chose $c_{\text{FN}} = 1$ and $c_{\text{FP}} = 4$ resulting in no false positive predictions on the given dataset.

Solution. The Bayes' optimal classifier has its decision boundary at

$$\begin{aligned} p(+1 \mid x)c_{\text{FN}} &= p(-1 \mid x)c_{\text{FP}} \\ &= (1 - p(+1 \mid x))c_{\text{FP}} \\ \iff \quad p(+1 \mid x)(c_{\text{FN}} + c_{\text{FP}}) &= c_{\text{FP}} \\ \iff \quad p(+1 \mid x) &= \frac{c_{\text{FP}}}{c_{\text{FN}} + c_{\text{FP}}} \end{aligned}$$

This is the desired expression. □

o-1-loss with Abstention. For some tasks, it is better to abstain from predictions or output "unsure" instead of making wrong predictions. For instance, ML models for autonomous driving are not always able to predict the next best action and it is sometimes better to alert and hand off control to the human driver instead of making potentially harmful decisions. This is known as *selective classification*, which extends the label set $\mathcal{Y}$ with a *reject option* r and allows the model to classify certain inputs and to reject others. The reject option is associated with a rejection cost c used to tradeoff between predicting and abstaining. The corresponding loss can then be defined as

$$\ell(f(x), y) = \begin{cases} 0, & f(x) \neq r \text{ and } f(x) = y \\ 1, & f(x) \neq r \text{ and } f(x) \neq y \\ c, & f(x) = r \end{cases}$$

and the expected loss is given by

$$\mathbb{E}\left[\ell(a, Y) \mid X = x\right] = \begin{cases} 1 - p(a \mid x), & a \neq r \\ c, & a = r, \end{cases}$$

i.e., the conditional misprediction probability when predicting a label or by the rejection cost when abstaining. The resulting Bayes' optimal classifier is

$$f^\star(x) = \arg\min_{a \in \mathcal{Y}} \mathbb{E}\left[\ell(a, Y) \mid X = x\right]$$
$$= \begin{cases} \arg\max_{a \in \mathcal{Y}} p(a \mid x), & \max_{a \in \mathcal{Y}} p(a \mid x) > 1 - c \\ r, & \max_{a \in \mathcal{Y}} p(a \mid x) < 1 - c. \end{cases}$$

One can perceive $1 - c$ as the lower threshold on the conditional probability $p(a \mid x)$ necessary to predict (and not abstain), i.e., the minimum confidence level necessary to predict. If the highest confidence level among all labels is below this threshold, the Bayes' optimal classifier decides to abstain. Otherwise, the most likely label with highest confidence level is predicted.

Observe that it makes sense to choose $c \in (0, 1 - 1/K]$ with K being the number of possible labels. On the one hand, if $c = 0$, the predictor will always abstain, as there is no cost associated with it. Thus, we should have $c > 0$. On the other hand, the most likely label cannot have confidence less than $\frac{1}{K}$, which would correspond to all labels having uniform confidence. Equivalently, the most likely label cannot have a misprediction probability higher than $1 - \frac{1}{K}$. Hence, choosing $c > 1 - \frac{1}{K}$ would result in the optimal decision rule never taking the reject option.

In the context of binary classification, the Bayes' optimal classifier reduces to

$$f^\star(x) = \begin{cases} -1, & p(+1 \mid x) < c \\ r, & p(+1 \mid x) \in (c, 1 - c) \\ +1, & p(+1 \mid x) > 1 - c \end{cases}$$

with decision boundaries at $p(+1 \mid x) \in \{c, 1 - c\}$. Intuitively, one can see the reject option as a way of adding a "safety margin" between the decision regions of the labels in which it is better to abstain from making predictions as seen in Figure 13.16.

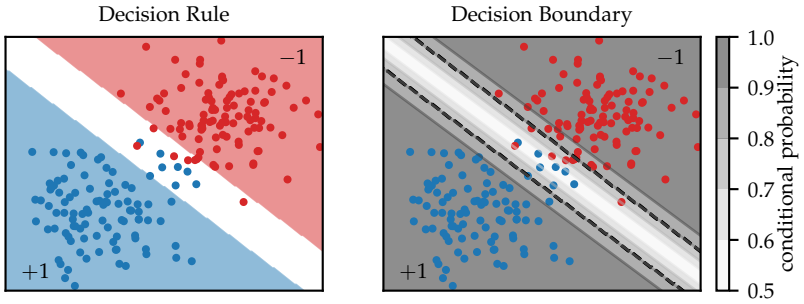


Figure 13.16: Optimal decision rule for 0-1-loss with abstention with $c = 0.2$, meaning that predictions are only made if the most likely label has conditional probability $p(a \mid x) > 1 - c = 0.8$. This decision rule does not predict any wrong labels on the given dataset, but it abstains on some points.

14

Gaussian Mixture Models

In this chapter, we explore the value of Gaussian Bayes classifiers in the context of missing data, and how they support both unsupervised and semi-supervised learning in such settings. Our modeling approach focuses on Gaussian Mixture Models (GMMs), which are widely known for their use in unsupervised learning. We explain how to estimate these models and introduce the Expectation-Maximization (EM) algorithm, a standard method for fitting the parameters of GMMs. Additionally, we distinguish between the hard EM and soft EM versions of the algorithm.

Roadmap

In Section 14.1, we introduce semi-supervised learning and provide a solution to working with incomplete data sets through generative modeling. In Section 14.2 we will introduce a popular generative model used for unsupervised and semi-supervised learning, known as the Gaussian mixture model. In Section 14.3, we elaborate on two versions of the Expectation-Maximization (EM) algorithm, the hard- and the soft-EM algorithm, to construct a Gaussian mixture model. In later subsections, we will explore theoretical convergence and model selection for Gaussian mixture models.

Learning Objectives

After reading this chapter, you should:

- Understand the promise of semi-supervised learning with missing labels.
- Understand how Gaussian mixture models can be used in unsupervised, supervised, and semi-supervised learning with potentially incomplete data sets.
- Understand how Gaussian mixture models are estimated and what assumptions they require.
- Understand the Expectation-Maximization algorithm and why it is preferred over standard optimization approaches such as maximum likelihood estimation with stochastic gradient descent.

14.1 Missing Data and Semi-Supervised Learning

We will continue our exploration of generative models and their further utility. Let us start with a motivating example when working with missing data. Up until now, we have made the assumption that all features are observed for every instance. However, this is not always the case in practical scenarios. For example, in medical tests, the features can represent the results of various medical tests conducted on patients. But what happens when a patient has not undergone a specific medical test? How should a model respond to such missing values? One possible approach is to use generative models to infer the relationships between features and fill the missing fetures. Generative modeling, therefore, offers a means of addressing the gaps present in incomplete datasets. For example, see Figure 14.1. The same premise holds for *semi-supervised* learning in which a small amount of labeled data is paired with a large amount of unlabeled data. The goal is to build a classifier or, in general, a machine learning model that is superior to only using the small labeled data set. Most approaches follow the same data-generating modeling procedure in which the correlation between labeled and unlabeled is established in order to utilize the unlabeled sets by inferring labels.

X_1	X_2	...	X_d	Y
1.1	N/A	...	.5	1
N/A	.4	...	.7	1
...	...	...	...	...
-.4	-.5	...	N/A	7

Figure 14.1: Example of a data set that misses some feature values.

Naturally, a special case of missing data occurs when all features are present within the dataset, but the labels are completely absent. This is unsupervised learning, as discussed in Chapter 11. For an example of such a data set, see Figure 14.2.

X_1	X_2	...	X_d	Y
1.1	.3	...	.5	N/A
.5	.4	...	.7	N/A
...	...	...	...	...
-.4	-.5	...	.3	N/A

Figure 14.2: Example of a data set that misses its labels.

The goal of generative modeling in the presence of incomplete data sets is to generate data such that the imputed features are statistically indistinguishable from the originally observed ones. Achieving this requires making certain assumptions about the underlying data-generation process. These assumptions typically involve conditions on the joint distribution of all relevant variables, including both observed and potentially unobserved components.

In particular, we may assume that the labeled data are generated according to a Gaussian Bayes classifier (see Subsection 13.6.3). However, in practice, we observe only the features, while the corresponding labels are missing. Illustrations of data sets with and without labels can be found in Figure 14.3 and Figure 14.4, respectively. This raises the question of how the joint distribution, over both features and labels, differs from the marginal distribution, over features alone. We will clarify this distinction using an example based on Gaussian assumptions similar to those made in the Gaussian Bayes classifier.

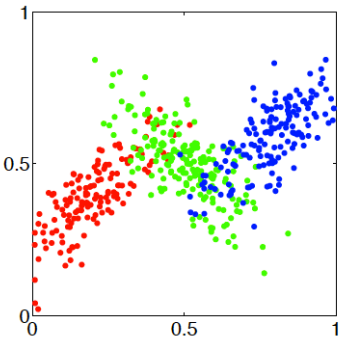


Figure 14.3: If we assume $p(X|Y)$ to be Gaussian in accordance with the assumptions, this is how a data set is observed when its labeling is known in advance Bishop et al. 1995.

14.2 Gaussian Mixture Model

Building upon Bayesian Gaussian classifier, we assume that the features are a convex combination of Gaussian distributions, a mixture

of models. This is known as a Gaussian mixture model (GMM). The resulting marginal density $p(\boldsymbol{X})$ from summing out the possible labels, we find that

$$\begin{aligned} P(\boldsymbol{x} \mid \theta) &= P(\boldsymbol{x} \mid \mu, \Sigma, w) \\ &= \sum_{j=1}^{k} P(Z = j) P(\boldsymbol{x} \mid Z) \\ &= \sum_{j=1}^{k} w_j \mathcal{N}(\boldsymbol{x}; \mu_j, \Sigma_j), \end{aligned}$$

where the variable θ contains all the model parameters, including the class label probabilities, Gaussian mean, and covariance parameters. As the labels are unknown, but a distribution over the labels, in which case, we will denote them with the letter Z instead of Y to refer to them as general latent variables.

The class labels are distributed according to a categorical distribution in the k classes, with $P(Z = j) = w_j$ ($w_j \geq 0$ and $\sum_{j=1}^{k} w_j = 1$) and the conditional of the label are distributed Gaussian. See Figure 14.5. However, as a spectator, we only get to see the marginal distribution, summing out the hidden variable set. See Figure 14.6.

The parameters of this model include the following.

- The weights w_j of each class, telling about the likelihood of encountering a data point in this class.
- The means μ_j and covariances Σ_j for the appearance of each class j of our samples.

In the absence of the observed labels, the maximum likelihood estimate as in Definition 13.5 is still well defined. In fact, the negative log-likelihood is as follows,

$$L(w_{1:k}, \mu_{1:k}, \Sigma_{1:k}) = -\sum_{i=1}^{n} \log \sum_{j=1}^{k} w_j \mathcal{N}(\boldsymbol{x}_i \mid \mu_j, \Sigma_j). \tag{14.1}$$

When we observe labels, maximum likelihood estimation is straightforward, involving simple calculations of means and covariances, as illustrated in Example 13.6. Unfortunately, when labels are hidden, minimizing Equation 14.1 becomes NP-hard due to the non-convex nature of the objective. However, we can still potentially employ (stochastic) gradient descent to find a locally optimal solution for Equation 14.1. However, this optimization process must be done with caution, as there are conditions that must be met. For instance, the covariance matrices are required to be positive definite. Due to these conditions, an iterative procedure known as expectation-maximization is preferred, which will be subject of the Section 14.3.

Now that we have discussed the two extremes, suppose that some labels are known but the majority is unknon – the typical

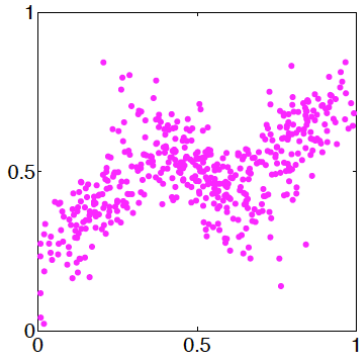


Figure 14.4: If we assume $p(X|Y)$ to be Gaussian in accordance with the assumptions, this is how the data set from Figure 14.3 is observed when its labeling is unknown Bishop et al. 1995.

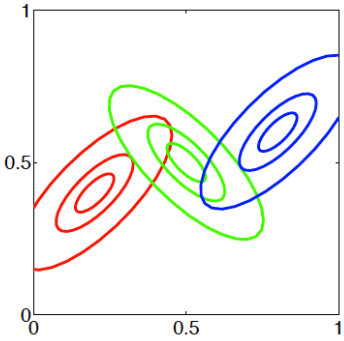


Figure 14.5: If we assume $p(X \mid Y)$ to be Gaussian in accordance with the assumptions, an abstract depiction of the distribution $p(\boldsymbol{x}, y)$ of the data set from Figure 14.3 Bishop et al.

semi-supervised setting. In this case, we can add hard constraints such that a specific x_j for which the labels z_j are known is assigned together or to different clusters. As the clusters are permutation independent, we can pick the cluster alignment arbitrarily just ensuring that the two with the same label fall into the same cluster.

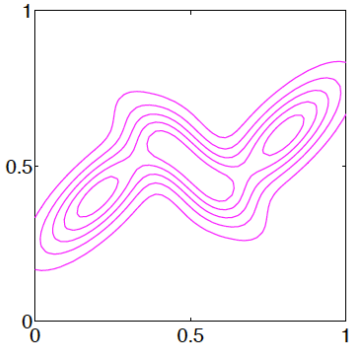


Figure 14.6: If we assume $p(X \mid Y)$ to be Gaussian in accordance with the assumptions, the marginal $p(\boldsymbol{x})$ of the distribution from Figure 14.5 Bishop et al.

14.2.1 Classification with GMMs

Before we explain a tractable method for fitting these models, we will briefly demonstrate a Bayesian classifier using the GMM assumption. That is, how can we use the GMM model for classifications?

Suppose that we have obtained a fitted procedure and have θ, that is, the probabilities w_j, means μ_j and covariances Σ_j. Now, in order to classify a specific x_i, we will use the maximum a posterior estimate to get

$$\hat{y} = \arg\max_y p(y|x) = \arg\max_y p(y)p(x|y)$$

which, in the context of GMMs, leads upon taking the logarithm of the above to the following classification rule:

$$\hat{y} = \arg\max_j \log(w_j) + \frac{1}{2}\log(\det(2\pi\Sigma_j)) + \frac{1}{2}(x-\mu_j)^\top \Sigma_j^{-1}(x-\mu_j).$$

Notice the close similarity to the Gaussian Bayes classifier. In fact, the difference is only the use of nonisotropic variance and multiple classes. Should we

14.3 Expectation-maximization Algorithm

Interestingly, one possible approach to optimizing the cost function in Equation 14.1 is closely connected to the earlier discussions in Chapter 11. The key idea is to exploit the relationship between a Gaussian Mixture Model (GMM) and the Gaussian Bayes classifier. Furthermore, we can consider clustering or fitting a GMM as the task of fitting a Gaussian Bayes classifier in the presence of missing labels, as previously discussed in this chapter. If we somehow obtain the labels, we would know how to estimate the parameters using closed-form solutions, as elaborated in Section 13.6.3.

Consequently, a natural approach is to employ an iterative method. We begin with an initial guess of the model parameters and use them to predict the unknown labels. Subsequently, we impute the missing data using these label predictions, resulting in a dataset with label assignments. This enables closed-form updates of the model parameters, as fitting a Gaussian Bayes classifier is straightforward with labeled data. Consequently, we can obtain improved parameter estimates compared to the initial guess. These refined estimates can then be used to further improve the label predictions, and the process continues iteratively. This idea forms the

basis of the expectation-maximization (or minimization) algorithm (EM algorithm), which can be viewed as a generalization of Lloyd's heuristic (the k-means algorithm), discussed in Chapter 11.

14.3.1 *Hard EM algorithm*

There are two versions of the algorithm. First, the simpler version of the algorithm is *Hard-EM algorithm*, which can be seen below.

- Initialize the parameters $\theta^{(0)}$, consisting of the initial weights $w_{1:k}^{(0)}$, initial means $\mu_{1:k}^{(0)}$ and initial covariance matrices $\Sigma_{1:k}^{(0)}$.
- For $t = 1, 2, \ldots$
  - **E-step:** predict the most-likely class for each data point.

$$z_i^{(t)} = \arg\max_z P(z \mid \boldsymbol{x}_i, \theta^{(t-1)}) \tag{14.2}$$

$$= \arg\max_z P(z \mid \theta^{(t-1)}) P(\boldsymbol{x}_i \mid z, \theta^{(t-1)}) \tag{14.3}$$

  - Now we got the complete data

$$D^{(t)} = \left\{ \left(\boldsymbol{x}_1, z_1^{(t)} \right), \ldots, \left(\boldsymbol{x}_n, z_n^{(t)} \right) \right\}.$$

  - **M-step:** we can compute the MLE as for the Gaussian Bayes classifier in closed form

$$\theta^{(t)} = \arg\max_\theta P(D^{(t)} \mid \theta).$$

The hard prefix stems from the fact that we assign a fixed class to a given label in E-step. Unfortunately, there are some issues with this even though there often are good results with this algorithm and it can be applied in a broader setting than Gaussian Bayes classifiers only. The drawbacks is that the data points are assigned a fixed label, even though the model is uncertain about the class label of the data point. Intuitively, in this way the algorithm tries to extract too much information from a single data point at every interation when there is uncertainty about the class label. In practice, this may work poorly if the clusters overlap, as shown in example Example 14.1.

Example 14.1. Take a look at the data set in figure Figure 14.7. We are interested in the relationship between different Gaussian mixtures as we increase the number of classes. In particular, it is true that any distribution with k Gaussians can always be expressed as a mixture of Gaussian distributions involving l Gaussians, where $l \geq k$. This is true because the weights of the additional classes could be set to zero of the classes we do not care about, or letting two Gaussians collide with each other (having the same mean and covariances). Therefore, if we would be able to estimate the parameters correctly, we should be able to recover that the number of classes is, in reality, lower than the number of classes the model used. Therefore, the final solution of a fitting algorithm is expected to closely resemble Figure 14.8.

However, if you the hard-EM algorithm is executed, it makes a deterministic prediction for each class label, and the fit is displayed in Figure 14.9.

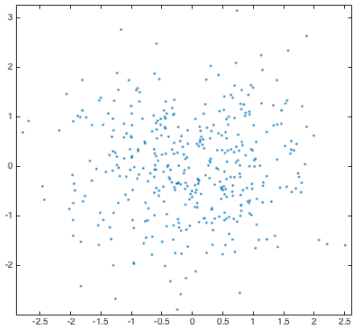


Figure 14.7: Data considered in Example 14.1.

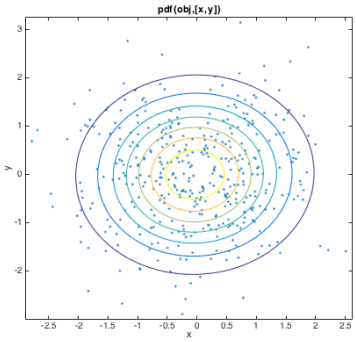


Figure 14.8: True solution of fitting a (single) Gaussian onto the data of Figure 14.7.

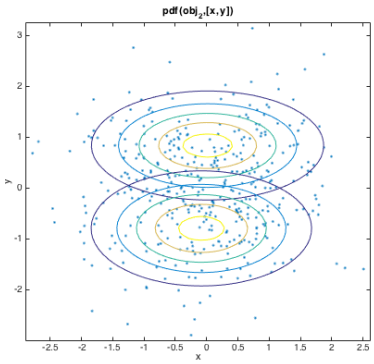


Figure 14.9: Hard EM algorithm solution when applied on the data of Figure 14.7.

The reason for this is in a way that the algorithm asks too much of a single data point. We saw that the decision boundary ($P(Y \mid \boldsymbol{x})$) that comes from a Gaussian Bayes classifier is a akin to logistic regression. That means, around the decision boundary, you are very unsure about class membership. On the other hand, the hard-EM algorithm will force the predicted model, to assign a class to data points. This creates this artificial separation into two clusters.

One way to fix this overconfidence of the hard EM algorithm is to be more careful about modeling the class membership, in particular reasoning about partial membership with classes. One possible approach when we have a guess of the model parameters of the GMM (weights, means, covariances), would be to calculate not just the most likely membership class, but the full posterior distribution of the class labels, for a given data point.

14.3.2 Soft-EM algorithm

Suppose that we are given a model $P(z \mid \theta)$ and $P(\boldsymbol{x} \mid z, \theta)$. For each data point $\boldsymbol{x}$, we are able to compute a posterior distribution over the cluster membership, inferring distributions over the latent variables z. This comes down to

$$\gamma_j(\boldsymbol{x}) := P(Z = j \mid \boldsymbol{x}, \Sigma, \mu, \boldsymbol{w}) = \frac{w_j P(\boldsymbol{x} \mid \mu_j, \Sigma_j)}{\sum_{\ell=1}^{k} w_\ell P(\boldsymbol{x} \mid \mu_\ell, \Sigma_\ell)} \tag{14.4}$$

$$= \frac{w_j N(\boldsymbol{x} \mid \mu_j, \Sigma_j)}{\sum_{\ell=1}^{k} w_\ell N(\boldsymbol{x} \mid \mu_\ell, \Sigma_\ell)}. \tag{14.5}$$

We can show that with the use of the MLE, see Definition 13.5, for a solution of

$$(\mu^*, \Sigma^*, w^*) = \arg\min_{w,\mu,\Sigma} - \sum_{i=1}^{n} \sum_{j=1}^{k} w_j N(\boldsymbol{x}_i \mid \mu_j, \Sigma_j),$$

it must hold that

$$\mu_j^* = \frac{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i) \boldsymbol{x}_i}{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)},$$

$$\Sigma_j^* = \frac{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)(\boldsymbol{x}_i - \mu_j^*)(\boldsymbol{x}_i - \mu_j^*)^\top}{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)},$$

$$w_j^* = \frac{1}{n} \sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i).$$

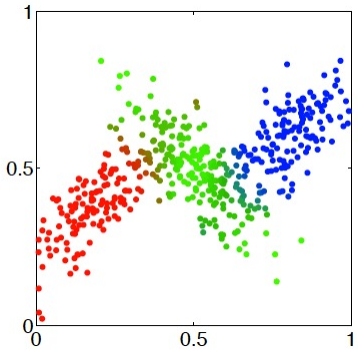


Figure 14.10: In the soft EM-algorithm, data points have partial class membership instead of having their class assigned to the most probable class. At the boundaries of the clusters, the colors overlap, as two of the class membership probabilities.

Proof. For the use of maximum likelihood estimation, since the class-labels are no longer fixed but probabilistic, we have to make use of the expected (log-)likelihood, from which we get

$$\mathbb{E}_{P(Z|\boldsymbol{x})} \left[\sum_{i=1}^{n} \log \left(P(\boldsymbol{x}_i, z_i \mid \mu, \Sigma, \boldsymbol{w}) \right) \right]$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{k} \gamma_j(\boldsymbol{x}) \left[\log \left(P(z_i = j \mid \mu, \Sigma, \boldsymbol{w}) \right) + \log \left(P(\boldsymbol{x}_i \mid z_i = j, \mu, \Sigma, \boldsymbol{w}) \right) \right]$$

$$= \sum_{j=1}^{k} \sum_{i=1}^{n} \gamma_j(\boldsymbol{x}) \log \left(w_j \right) + \sum_{i=1}^{n} \sum_{j=1}^{k} \gamma_j(\boldsymbol{x}) \log \left(N(\boldsymbol{x}_i \mid \mu_j, \Sigma_j) \right).$$

From here, since the first term only contains $\boldsymbol{w}$ while the second term contains the parameters μ and Σ, we can split up the derivation of the maximum likelihood estimators. For the class weights $\boldsymbol{w}$, and since $\sum_{j=1}^{k} w_j = 1$, from a similar derivation as in Chapter **??**, we can conclude that

$$w_j^* = \frac{1}{n}\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i).$$

For the mean and covariances, this is again simply maximum likelihood estimation involving Gaussians, with the exception that each data point $\boldsymbol{x}$ has a weight $\gamma_j(\boldsymbol{x})$ rather than all considered data point having the same importance. Performing maximum likelihood estimation on the log-likelihood function

$$\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)\left(-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\left(\det(\Sigma_j)\right) - \frac{1}{2}(\boldsymbol{x}_i - \mu_j)^\top \Sigma_j^{-1}(\boldsymbol{x} - \mu_j)\right)$$

results in

$$\mu_j^* = \frac{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)\boldsymbol{x}_i}{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)},$$

and

$$\Sigma_j^* = \frac{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)(\boldsymbol{x}_i - \mu_j^*)(\boldsymbol{x}_i - \mu_j^*)^\top}{\sum_{i=1}^{n} \gamma_j(\boldsymbol{x}_i)}.$$

□

Therefore, the final algorithm, known as the *soft-EM* algorithm, can be seen as follows.

- Initialize parameters $\mu^{(0)}, \Sigma^{(0)}, \boldsymbol{w}^{(0)}$.
- While not converged for t:
  - **E-step**: calculate the cluster membership weights $\gamma_j^{(t)}(\boldsymbol{x}_i)$ for each i and j, given the estimates of $\mu^{(t-1)}, \Sigma^{(t-1)}, \boldsymbol{w}^{(t-1)}$ according to the posterior prediction formula

$$\gamma_j^{(t)}(\boldsymbol{x}) = \frac{w_j^{(t-1)} N(\boldsymbol{x} \mid \mu_j^{(t-1)}, \Sigma_j^{(t-1)})}{\sum_{\ell=1}^{k} w_\ell^{(t-1)} N(\boldsymbol{x} \mid \mu_\ell^{(t-1)}, \Sigma_\ell^{(t-1)})}.$$

  - **M-step**: fit the clusters to the weighted data points. This is a closed-form maximum likelihood solution resulting in

$$w_j^{(t)} \leftarrow \frac{1}{n}\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i), \quad \mu_j^{(t)} \leftarrow \frac{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)\boldsymbol{x}_i}{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)}$$

$$\Sigma_j^{(t)} \leftarrow \frac{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)(\boldsymbol{x}_i - \mu_j^{(t)})(\boldsymbol{x}_i - \mu_j^{(t)})^\top}{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)}.$$

Example 14.2. An example of a selection of steps in the soft EM algorithm is depicted in Figure 14.11.

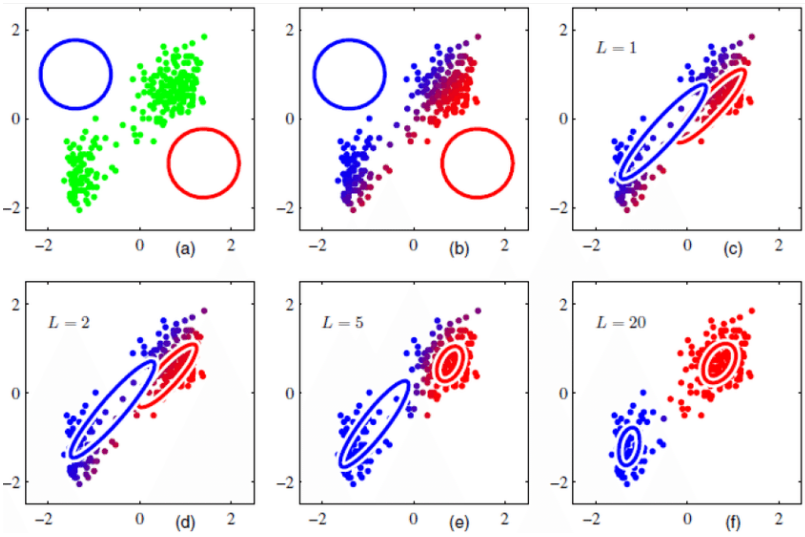


Figure 14.11: Highlighted steps in the soft EM algorithm.

(a) First, Gaussian Bayes classifiers are initialized. No fitting of the data has been done yet.

(b) For each of the data points, the class probabilities $\gamma_j(\boldsymbol{x})$ are calculated, according to Equation 14.4.

(c) The Gaussian Bayes classifiers are fitted to the weighted data points. This is the Gaussian mixture model that has been generated after one round of the soft EM algorithm.

(d) After another round of both reweighting the data points and refitting the Gaussian Bayes classifiers, this is the Gaussian mixture model, which has been generated after two rounds of the soft EM algorithm.

(e) This is the Gaussian mixture model, which has been generated after five rounds of the soft EM-algorithm.

(f) This is the Gaussian mixture model, which has been generated after twenty rounds of the soft EM-algorithm. This solution is stable.

14.3.3 Model Selection

To use Gaussian Mixture Models (GMMs) effectively for clustering, it is essential to determine the optimal number of components (or clusters) that best capture the underlying structure of the data.

As we saw in the Figure 14.9, the number of clusters is important in order to elucidate the underlying structure properly. Selecting too few components can lead to underfitting, where the model is too simple to capture the complexity of the data. Conversely, choosing too many components may result in overfitting, where the model becomes overly flexible and starts modeling noise in the data rather than meaningful structure. Thus, selecting the right number of components is critical for building a well-generalized model.

Also, notice that by increasing the number of components n, we are increasing the likelihood of the data, as every point can be regarded as a separate cluster and in that way maximize the likelihood. Thus, we see that likelihood alone is not sufficient to guide us to select number of clusters often referred to as model selection.

Similar to when selecting a kernel or neural network architecture in supervised learning most common methods resort to criteria such as the Bayesian Information Criterion (BIC) and the Akaike Information Criterion (AIC). Both metrics balance model fit against complexity by penalizing excessive numbers of parameters. In practice, GMMs are fitted for a range of cluster counts (e.g., 1 to 100), and AIC or BIC is calculated for each. The number of components corresponding to the lowest value of these criteria is generally selected.

- $\text{AIC}(k) = k - \log L$
- $\text{BIC}(k) = k \log n - \log L$

where $\log L$ is the loglikelihood, k is number of clusters and n is the number of datapoints.

Another intuitive and commonly used technique is the elbow method, which helps visualize the trade-off between model complexity and goodness-of-fit. This method involves plotting a performance metric – such as the negative log-likelihood. As more clusters are added, the model's likelihood improves, but the gain diminishes after a certain point. The *elbow* refers to the inflection point on the curve where the rate of improvement begins to slow down. This point suggests the optimal number of components: a balance between having enough complexity to model the data accurately without unnecessary overfitting. Although the elbow method provides a visual and intuitive aid, it can be somewhat subjective, especially if the curve does not have a well-defined inflection point. Therefore, it is often used in conjunction with quantitative criteria like BIC/AIC and qualitative insights such as domain knowledge or visual inspection of cluster separation.

14.3.4 Convergence of EM algorithm

Altought EM algorithm is very popular and successful, it does not guarantee global convergence in general. It is only a continuous improvement scheme. In other words, after every iteration the fitted parameters improve the likelihood. This can be formalized as follows.

Theorem 14.3 (Convergence of the EM Algorithm)**.** *Let $\theta^{(t)}$ be the parameter estimates at iteration t of the EM algorithm for a Gaussian Mixture Model, and let $\log P(X \mid \theta)$ denote the log-likelihood of the observed data. Then, the EM algorithm produces a sequence $\{\theta^{(t)}\}_{t=1}^{\infty}$ such that*

$$\log P(X \mid \theta^{(t+1)}) \geq \log P(X \mid \theta^{(t)}),$$

for all $t \geq 0$. Furthermore, the sequence $\{\log P(X \mid \theta^{(t)})\}$ converges to a finite value and the sequence $\{\theta^{(t)}\}$ converges to a stationary point of the logarithmic likelihood function.

Proof. At each iteration of the EM algorithm, we define the *Q-function* as

$$Q(\theta \mid \theta^{(t)}) := \mathbb{E}_{Z|X,\theta^{(t)}} \left[\log P(X, Z \mid \theta) \right],$$

where the expectation is taken with respect to the current posterior distribution of the latent variables Z given the data X and current parameters $\theta^{(t)}$.

The EM algorithm proceeds as follows:

- **E-step:** Compute the Q-function $Q(\theta \mid \theta^{(t)})$.
- **M-step:** Maximize the Q-function to obtain the next parameters:

$$\theta^{(t+1)} = \arg\max_{\theta} Q(\theta \mid \theta^{(t)}).$$

By Jensen's inequality and the definition of the Q-function, it follows that

$$\begin{aligned}
\log P(X \mid \theta^{(t+1)}) &= \log \sum_Z P(Z|X,\theta^{(t)}) \frac{P(X,Z|\theta^{(t+1)})}{P(Z|X,\theta^{(t)})} \\
&\overset{\text{Jensen}}{\geq} \sum_Z P(Z|X,\theta^{(t)}) \log \left(\frac{P(X,Z|\theta^{(t+1)})}{P(Z|X,\theta^{(t)})} \right) \\
&= Q(\theta^{(t+1)} \mid \theta^{(t)}) + H(P(Z|X,\theta^{(t)})) \\
&\overset{\text{M-step}}{\geq} Q(\theta^{(t)} \mid \theta^{(t)}) + H(P(Z|X,\theta^{(t)})) \\
&= \log P(X \mid \theta^{(t)}),
\end{aligned}$$

where H is the entropy. This shows that the observed-data log-likelihood is non-decreasing at each iteration.

Since the log-likelihood is bounded above for finite data sets and fixed non-degenerate model structure, the sequence $\log P(X \mid \theta^{(t)})$ converges. However, because the log-likelihood surface is generally non-convex, the EM algorithm may converge to a local maximum or a saddle point, rather than the global maximum. This completes the proof. □

Index

Bibliography

[AV07] David Arthur and Sergei Vassilvitskii. "K-means++ the advantages of careful seeding". In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. 2007, pp. 1027–1035.

[Bar+20] Peter L Bartlett et al. "Benign overfitting in linear regression". In: *Proceedings of the National Academy of Sciences* 117.48 (2020), pp. 30063–30070.

[Bis+95] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[BN06] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[Bou+22] Olivier J Bousquet et al. "Monotone Learning". In: *Proceedings of Thirty Fifth Conference on Learning Theory*. Ed. by Po-Ling Loh and Maxim Raginsky. Vol. 178. Proceedings of Machine Learning Research. PMLR, July 2022, pp. 842–866. URL: https://proceedings.mlr.press/v178/bousquet22a.html.

[Cyb89] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7 (2011).

[Dur19] Rick Durrett. *Probability: theory and examples*. Vol. 49. Cambridge University Press, 2019.

[GB10] Xavier Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Journal of Machine Learning Research - Proceedings Track* 9 (Jan. 2010), pp. 249–256.

[He+15] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *IEEE International Conference on Computer Vision (ICCV 2015)* 1502 (Feb. 2015). DOI: 10.1109/ICCV.2015.123.

[Hin11] Geoff Hinton. 2011. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[JT18] Ziwei Ji and Matus Telgarsky. "Risk and parameter convergence of logistic regression". In: (Mar. 2018).

[KB17] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2017). arXiv: 1412.6980 [cs.LG].

[Kow73] Charles J. Kowalski. "Non-Normal Bivariate Distributions with Normal Marginals". In: *The American Statistician* 27.3 (1973), pp. 103–106.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[NJ01] Andrew Ng and Michael Jordan. "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes". In: *Advances in neural information processing systems* 14 (2001).

[Ros14] Sheldon M Ross. *A first course in probability*. Pearson, 2014.

[RS00] Sam T Roweis and Lawrence K Saul. "Nonlinear dimensionality reduction by locally linear embedding". In: *science* 290.5500 (2000), pp. 2323–2326.

[SC08] Ingo Steinwart and Andreas Christmann. *Support vector machines*. eng. Information science and statistics. New York, NY: Springer, 2008. ISBN: 978-0-387-77241-7 and 978-1-4899-8963-5 and 978-6-611-92704-2 and 978-0-387-77242-4.

[Sey21] Sadat Seyedmorteza. *Linear Regression*. May 2021. URL: https://las.inf.ethz.ch/courses/introml-s21/slides/introml-03-linreg-notes.pdf.

[SHS17] Daniel Soudry, Elad Hoffer, and Nathan Srebro. "The Implicit Bias of Gradient Descent on Separable Data". In: *Journal of Machine Learning Research* 19 (Oct. 2017).

[Sri+14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[TB20] Alexander Tsigler and Peter L Bartlett. "Benign overfitting in ridge regression". In: *arXiv preprint arXiv:2009.14286* (2020).

[TB97] Lloyd N. Trefethen and David Bau. *Numerical linear algebra*. SIAM, 1997.

[Vin+15] Oriol Vinyals et al. "Show and Tell: A Neural Image Caption Generator". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[Wik22] Wikipedia. *Big O notation*. Feb. 2022. URL: https://en.wikipedia.org/wiki/Big_O_notation.

[Wil91] David Williams. *Probability with Martingales*. Cambridge mathematical textbooks. Cambridge University Press, 1991.

[YGL24] Taesun Yeom, Chanhoe Gu, and Minhyeok Lee. "DuDGAN: Improving Class-Conditional GANs via Dual-Diffusion". In: *IEEE Access* 12 (2024), pp. 39651–39661. DOI: 10.1109/ACCESS.2024.3372996.