4 Machine Learning Basics

Konstantinos Chatzilygeroudis, Ioannis Hatzilygeroudis and Isidoros Perikos

Machine learning (ML) [Michalski et al. 2013] concerns the development and assessment of algorithms that enable computer systems to learn by trial-and-error, that is, to improve with more data their performance with respect to some task without being explicitly programmed for it. Although the term *machine learning* was coined in 1959 by Arthur Samuel [Samuel 1959], Tom Mitchell [Mitchell 1997] provided a more formal definition: "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."

ML has be applied to many real-world problems or tasks, like medical diagnosis, robotics, recommendation systems, facial recognition, stock prices prediction, and sentiment analysis, with great success.

We can divide ML algorithms into three main categories (see Figure 4.1): (1) *supervised learning* (SL) [Russell and Norvig 2016], (2) *reinforcement learning* (RL) [Sutton and Barto 1998], and (3) *unsupervised learning* (UL). In SL, the system is presented with labeled samples (i.e., inputs with desired outputs given by an *oracle*) and the task is to learn a mapping (e.g., a function) from the input space to the output space. In RL, the agent is given rewards (or punishments) as a feedback to its actions (and current state) in a possibly dynamic environment. In other words, the agent receives reinforcement signals when the actions it takes help toward solving the desired task(s). In UL, no labels or reward signals are given to the system and the system has to discover the underlying or hidden structure of the data (e.g., clustering). In this chapter, we will only consider SL and UL, and do not talk about RL as it is not commonly used in real-world engineering problems.

SL algorithms can be categorized into two main categories: (1) *regression*, and (2) *classification* algorithms. Regression algorithms attempt to estimate the mapping from the input variables to numerical or continuous output variables. On the contrary, classification algorithms attempt to estimate the mapping from the

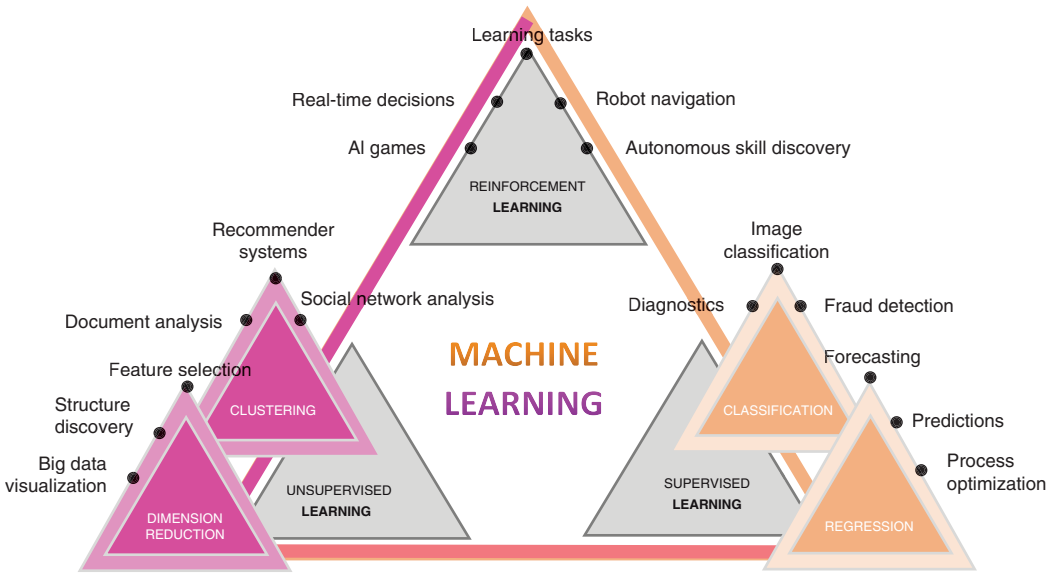

Figure 4.1 Machine learning taxonomy. Image inspired by http://www.cognub.com/index.php/cognitive-platform.

input variables to discrete or categorical output variables. Regression usually refers to predicting real-valued continuous parameters, whereas classification concerns assigning a label, representing a class, to an input sample. More formally, we assume that we have access to a dataset of the form $\boldsymbol{D} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, where N is the number of samples, $\boldsymbol{x}_i$ are the input variables (also called *features*) (in vector form), $\boldsymbol{y}_i = f(\boldsymbol{x}_i)$, and $f(\cdot)$ is the mapping of interest. The goal of regression and classification algorithms is to model this mapping f in an attempt to not only to minimize the error from the observed samples, but also to generalize to unseen data-points.

UL usually refers to either *clustering* or *dimensionality reduction*. In clustering, the task is to group a set of points in such a way that points in the same group (called a *cluster*) are more similar (given some *metric*) to each other than to those in other clusters. Dimensionality reduction is the process of reducing the number of variables under consideration by obtaining a set of principal variables that can describe adequately the original data. More formally, we assume that we have access to a dataset of the form $\boldsymbol{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, and the goal is to learn a mapping $f : \boldsymbol{X} \rightarrow \boldsymbol{Y}$, where $\boldsymbol{Y}$ is problem dependent. For example, in clustering f can be seen as a function that assigns to each data point a cluster (that is automatically generated from $\boldsymbol{D}$). In this chapter, we will only consider clustering algorithms, and do not talk about dimensionality reduction algorithms.

In ML, *models* play a crucial role as we can find them in most algorithms regardless of the category they fit in. A model in ML can be any mathematical representation of a real-world process. There exist two main classes of models (that we use both for regression and classification): (a) *parametric* and (b) *non-parametric* ones. Parametric models assume some finite set of parameters, θ. Given these parameters, future predictions, $f(\boldsymbol{x})$, are independent of the observed data, $\boldsymbol{Y}$. In other words, $P(f(\boldsymbol{x})|\theta, \boldsymbol{Y}) = P(f(\boldsymbol{x})|\theta)$ and therefore we say that *the dataset is diffused in the parameters* θ. Perhaps the most widely used parametric model is the liner model $f(\boldsymbol{x}) = \boldsymbol{x}^T\theta$ (see Section 4.2.1.1). Non-parametric models assume that the data distribution cannot be defined in terms of a finite set of parameters, but it can be defined by assuming an infinite dimensional θ. Usually we think of θ as a function. The amount of information that θ can capture about the data $\boldsymbol{Y}$ can grow as the amount of data grows. One of the most successful non-parametric models in ML are Gaussian processes (GPs; see Section 4.2.1.3).

A similar taxonomy that is common in ML is the following: *instance-based* and *model-based* algorithms. In instance-based methods (also called memory-based or lazy learning methods), prediction for each new sample is based on the whole dataset, stored in the memory. So, the prediction "model" consists of all the sample (instances) in the dataset. In model-based learning, a prediction "model" is extracted from the samples in the dataset and this model is used afterward for each new sample parameter value or class prediction. This is very similar to the taxonomy of models (parametric and non-parametric) as in most of the cases instance-based algorithms use non-parametric models and model-based algorithms use parametric models.

In the next section (Section 4.1), we will provide a brief summary of the *probability theory* that is crucial for understanding the concepts of ML algorithms. Next we will discuss about some of the most successful and common SL algorithms (Section 4.2): linear regression (LR), GPs, neural networks (NNs), naïve Bayes (NB), support vector machines (SVMs), and decision trees (DTs). Finally, we will see two algorithms for UL (Section 4.3), namely the K-means algorithm and Gaussian mixture models (GMMs).

4.1 Probability Primer

Probability measures the likelihood of an event happening and concerns, loosely speaking, the study of uncertainty. Probability can actually be seen as the fraction of times an event occurs, or how likely is an event to happen. It measures with a number between 0 and 1 the degree of certainty that a specific event will happen, where 0 indicates impossibility and 1 complete certainty. The higher the probability of an event, the more likely it is that the event will occur. Perhaps the simplest

example is the tossing of a fair coin. When tossing a coin there exist two outcomes: "heads" and "tails." The probability of "heads" equals the probability of "tails": we say that the two outcomes are equally probable. Of course, in our toy example there are no other outcomes possible, and thus the probability of either outcome is $\frac{1}{2} = 0.5$ (or 50%).

In order to quantify uncertainty and formulate the probability theory, we need the idea of a *random variable*. A random variable is a variable whose possible values are outcomes of a random phenomenon. In essence, it is a function that maps the outcomes of this random phenomenon to a set of properties that we can manipulate. Going one step further, we would also like to have a function that quantifies the probability that a set of outcomes will occur. We call this function a *probability distribution* and it will play a crucial role in understanding other concepts in probability theory.

4.1.1 Probability Theory Formalization

Modern probability theory is based on a set of axioms proposed by Kolmogorov [Grinstead and Snell 2012], which introduce three main concepts:

- The *sample space*, $\mathbf{\Omega}$, is the set of all possible outcomes of the experiment. For example, imagine tossing a fair coin two consecutive times: the sample space is defined as {"*heads, heads*," "*heads, tails*," "*tails, heads*," "*tails, tails*"}.
- The *event space*, $\mathcal{A}$, is the space of potential results of the random experiment. Usually, we obtain the event space by the collection of the subsets of $\mathbf{\Omega}$.
- For each event $A \in \mathcal{A}$, we define the *probability of this event* as a real number, $P(A) \in [0, 1]$, that measures the degree of belief or the relevant frequency of an event to the total number of events that can occur.

The total probability over all outcomes in the sample space must be 1. In other words, we need to have $P(\mathbf{\Omega}) = 1$. We define the *random variable*, $\mathcal{X}$, as the mapping $\mathcal{X} : \mathbf{\Omega} \to \Gamma$, where Γ is a quantity of interest. To better understand this concept, let's consider the following example:

Example 4.1 **Random variable**

Consider that we have a bag that contains red and yellow marbles. We play a game where we draw randomly (with replacement[1]) two marbles from the bag. It is easy to verify that the sample space, Ω, is: $\{(R, R), (R, Y), (Y, R), (Y, Y)\}$. We are now interested in measuring how many times a yellow marble appears in this game. In essence, we would like to investigate the mapping $\mathcal{X} : \mathbf{\Omega} \to \Gamma$, where $\Gamma = \{0, 1, 2\}$ as a yellow marble can appear 0, 1, or 2 times. This mapping is what we call a random variable.

1. In other words, once we draw something, we put it back in the bag.

4.1.2 Discrete and Continuous Probabilities

When the quantity of interest, Γ, is discrete, then we say that we are handling *discrete* probabilities. On the other hand, when Γ is continuous (i.e., $\Gamma \in \mathbb{R}$), we have *continuous* probabilities. In the discrete case, we define the probability that a random variable, $\mathcal{X}$, takes the value x as:

$$p(\mathcal{X} = x) = \frac{n_{\mathcal{X}=x}}{N_{\mathcal{X}}} \tag{4.1}$$

where $N_{\mathcal{X}}$ is the total number of values it can get.

In many cases, we will be handling multiple random variables at the same time (that possibly interact with each other). We define the *joint probability* that a random variable, $\mathcal{X}$, takes a specific value, x, and a second random variable, $\mathcal{Y}$, takes the value y as:

$$p(\mathcal{X} = x, \mathcal{Y} = y) = \frac{n_{\mathcal{X}=x,\mathcal{Y}=y}}{N_{\mathcal{X}} N_{\mathcal{Y}}} \tag{4.2}$$

In other words, the joint probability defines the probability of the intersection of the two events. We can write $p(x, y)$ instead of $p(\mathcal{X} = x, \mathcal{Y} = y)$. Similarly, we can write the *marginal probability* that $\mathcal{X}$ takes the value x irrespective of the value of the random variable $\mathcal{Y}$ as $p(x)$. Finally, when considering the *conditional probability* that $\mathcal{X}$ takes the value x given that $\mathcal{Y} = y$, we write $p(x|y)$.

4.1.2.1 Probability Density and Cumulative Distribution Function

Definition 4.1 A function $f \in \mathbb{R}^D \rightarrow \mathbb{R}$ is a *probability density function* (pdf) if:

- $\forall x \in \mathbb{R}^D : f(x) \geq 0$,
- *Its integral exists and*

$$\int_{\mathbb{R}^D} f(x)dx = 1 \tag{4.3}$$

In the discrete case, the pdf is called *probability mass function* and the integral is replaced with a sum. One can notice that the pdf can be any function f that satisfies Definition 4.1. We associate a random variable $\mathcal{X}$ with this function by:

$$p(a \leq \mathcal{X} \leq b) = \int_a^b f(x)dx \tag{4.4}$$

where $a, b \in \mathbb{R}$, and $x \in \mathbb{R}$ are outcomes of the continuous random variable $\mathcal{X}$. It might not be clear at first sight that for a continuous variable $p(\mathcal{X} = x)$ is zero. This is easy to verify from Equation 4.4 and contradicts the discrete case.

Definition 4.2 A *cumulative distribution function (cdf)* of a multivariate real-valued random variable $\mathcal{X}$ is:

$$F(\boldsymbol{x}) = p(\mathcal{X}_1 \leq x_1, \ldots, \mathcal{X}_D \leq x_D) \tag{4.5}$$

where $\boldsymbol{x} \in \mathbb{R}^D$.

The cdf basically represents the probability that the random variable $\mathcal{X}_i$ takes the value smaller than or equal to x_i.

4.1.3 Properties of Probabilities

Once we have defined our random variables, we can start manipulating them. Assuming that we have only two random variables, $\mathcal{X}$ and $\mathcal{Y}$, we can define the *sum rule* as follows:

$$p(\boldsymbol{x}) = \int_{\mathbb{Y}} p(\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{y} \tag{4.6}$$

where with $\mathbb{Y}$ we denote the possible states of the target space of the random variable $\mathcal{Y}$. From now on, we denote with bold symbols the values of multivariate random variables.

In the discrete case, the integral is replaced with a sum. We can exchange $\boldsymbol{x}$ and $\boldsymbol{y}$ in Equation 4.6: the sum rule holds for every random variable. With the sum rule, we integrate out the set of states of the variable $\mathcal{Y}$. This is why the sum rule is also known as the *marginalization property*. In other words, the sum rule relates the joint distribution, $p(\boldsymbol{x}, \boldsymbol{y})$, to a marginal distribution. We can, of course, extend the sum rule to more than two random variables.

We define the *product rule*, that relates the joint distribution to the conditional distribution, as follows:

$$p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x}) = p(\boldsymbol{x}|\boldsymbol{y})p(\boldsymbol{y}) \tag{4.7}$$

What the product rule really tells us is that every joint distribution of two random variables can be factorized (i.e., written as a product) of two other distributions.

In ML, we are often interested in making inferences about the value of unobserved random variables given other variables that we can observe. One of the most helpful and widely used theorems to do so is the *Bayes' theorem*:

$$\underbrace{p(\boldsymbol{x}|\boldsymbol{y})}_{\text{conditional}} = \frac{\overbrace{p(\boldsymbol{y}|\boldsymbol{x})}^{\text{likelihood}} \overbrace{p(\boldsymbol{x})}^{\text{prior}}}{\underbrace{p(\boldsymbol{y})}_{\text{evidence}}} \tag{4.8}$$

where we assume the following:

- We cannot observe $\mathcal{X}$,
- We have a prior over $\mathcal{X}$ (i.e., $p(\boldsymbol{x})$),
- We can observe $\mathcal{Y}$,
- We know the relationship between $\mathcal{X}$ and $\mathcal{Y}$ (i.e., $p(\boldsymbol{y}|\boldsymbol{x})$), that is the likelihood of $\mathcal{Y}$ given $\mathcal{X}$.

It is easy to see that Equation 4.8 can be directly derived from the product rule (Equation 4.7).

Example 4.2 **Bayes' theorem example**

To better understand this theorem, let's consider a simple example. Let's assume that 100% of the people that have pancreatic cancer, have a certain symptom, that is, $p(\boldsymbol{y}|\boldsymbol{x}) = 1$ (likelihood). We also know that there is a one in 100,000 chance of someone having pancreatic cancer, that is, $p(\boldsymbol{x}) = 0.00001$ (prior), and a random person can have the symptom with a probability of one in 10,000 (independently of having pancreatic cancer or not), that is, $p(\boldsymbol{y}) = 0.0001$ (evidence). Using Bayes' theorem, we have $p(\boldsymbol{x}|\boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x})}{\boldsymbol{y}} = \frac{1 \cdot 0.00001}{0.0001} = 0.1$. Thus, we can conclude that even if a person has the symptom and 100% of the cases that have pancreatic cancer have this symptom, *the probability of having pancreatic cancer is only* 10%.

Definition 4.3 Two random variables $\mathcal{X}, \mathcal{Y}$ are *statistically independent* if and only if:

$$p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x})p(\boldsymbol{y}) \tag{4.9}$$

This of course also implies that:

$$\begin{aligned} p(\boldsymbol{x}|\boldsymbol{y}) &= p(\boldsymbol{x}) \\ p(\boldsymbol{y}|\boldsymbol{x}) &= p(\boldsymbol{y}) \end{aligned} \tag{4.10}$$

Definition 4.4 Two random variables $\mathcal{X}, \mathcal{Y}$ are *conditionally independent* given $\mathcal{Z}$ if and only if:

$$p(\boldsymbol{x}, \boldsymbol{y}|\boldsymbol{z}) = p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{y}|\boldsymbol{z}), \text{ for all } \boldsymbol{z} \tag{4.11}$$

4.1.4 Gaussian Distribution

Perhaps the most widely used probability distribution is the Gaussian distribution. It is the most studied distribution, and it is also referred to as *normal distribution*. In the univariate case, the Gaussian distribution is described by a mean μ and a variance σ^2. Its pdf is given by (Figure 4.2[a]):

$$\mathcal{N}(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{4.12}$$

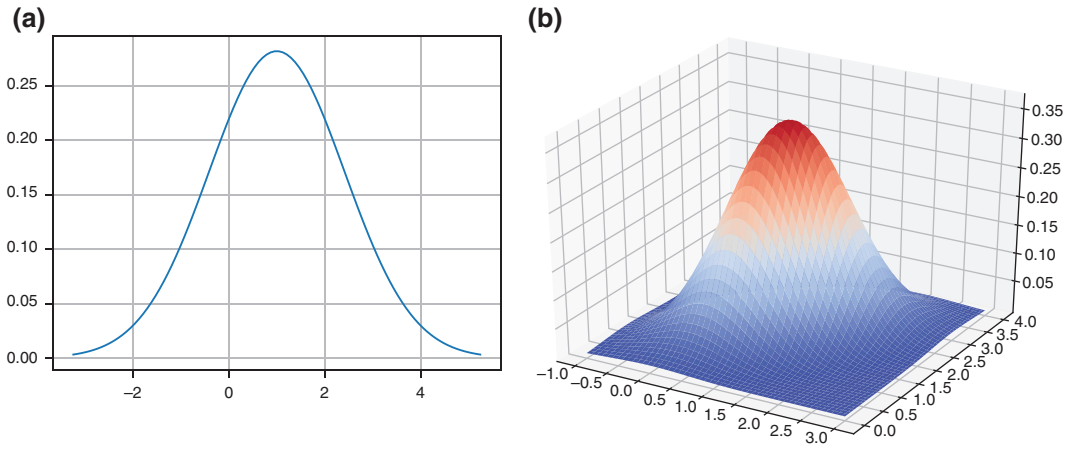

Figure 4.2 Examples of Gaussian distributions.
(a) Univariate Gaussian with $\mu = 1$ and $\sigma^2 = 2$. (b) Multivariate Gaussian with $\boldsymbol{\mu} = [1, 2]^T$ and $\boldsymbol{\Sigma} = \begin{bmatrix} 0.5 & 0.25 \\ 0.25 & 0.5 \end{bmatrix}$.

For the multi-variate case, the Gaussian distribution is defined by a mean vector $\boldsymbol{\mu} \in \mathbb{R}^D$ and a covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{D \times D}$. Its pdf is given by (Figure 4.2[b]):

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}\boldsymbol{\Sigma}^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})\right) \tag{4.13}$$

4.2 Supervised Learning

In this section, we talk about SL. As we have seen in the beginning of this chapter, SL algorithms can be categorized into two main categories: (1) *regression*, and (2) *classification*.

4.2.1 Regression

As we already stated, regression algorithms try to find the mapping $f : \boldsymbol{x} \rightarrow \boldsymbol{y}$ from the input variables $\boldsymbol{x} \in \mathbb{R}^D$ to the numerical or continuous output variables $\boldsymbol{y} \in \mathbb{R}^E$. More formally, we assume access to a dataset of the form: $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, where N is the number of samples, $\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \varepsilon$, and ε is an independent and identically distributed (IID) random variable that describes the measurement noise and potentially unmodeled processes (which we will not consider in this section). Throughout the section, we assume a zero mean Gaussian noise.

The goal of a regression algorithm is not only to minimize the error from the observed samples, but also to generalize to unseen datapoints. Figure 4.3 shows a typical example of a regression problem. The function we are trying to model is

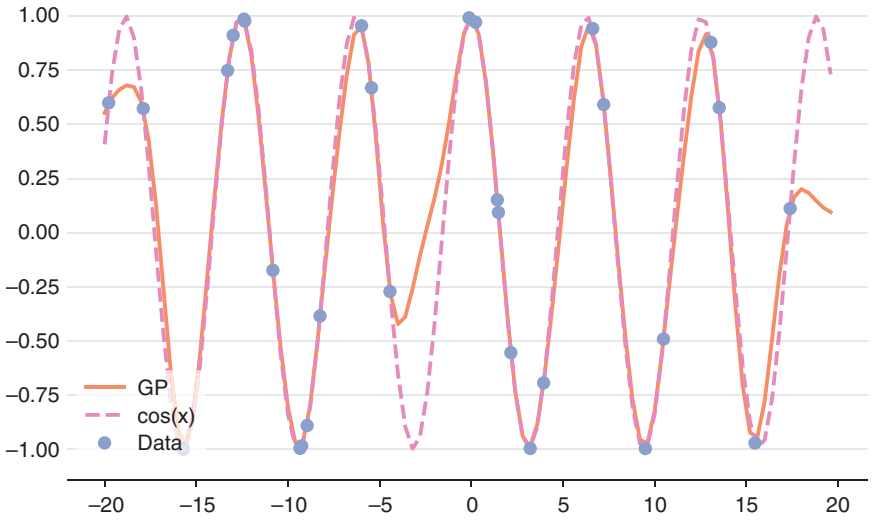

Figure 4.3 Regression example.

$f(x) = \cos(x)$ for $x \in [-20, 20]$, and we are given 30 data points. The ground truth is depicted with the dotted line and the solid line is a fit with GP regression that we will discuss in Section 4.2.1.3.

4.2.1.1 Linear Regression

One of the simplest, but widely used type of regression, is LR. LR falls into the parametric models category, and we assume that $f(\boldsymbol{x}) = \boldsymbol{x}^T\theta + \varepsilon$, that is, the output variables are just a linear combination of the input variables plus some noise. Because of this observation noise, we adopt a probabilistic approach and explicitly model the noise using a likelihood function. In the *stochastic linear regression* we seek to optimize:

$$\theta^* = \text{argmax}_\theta p(\boldsymbol{Y}|\boldsymbol{X}, \theta) \tag{4.14}$$

where $\boldsymbol{X}$ and $\boldsymbol{Y}$ define the dataset. In other words, we seek to find the parameters θ^* that maximize the likelihood of the parameterized function given the observed data. The likelihood is given by:

$$\begin{aligned} p(\boldsymbol{Y}|\boldsymbol{X}, \theta) &= p(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N|\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \theta) \\ &= \prod_{i=1}^{N} p(\boldsymbol{y}_i|\boldsymbol{x}_i, \theta) \end{aligned} \tag{4.15}$$

The likelihood factors, $p(\boldsymbol{y}_i|\boldsymbol{x}_i, \theta)$, are Gaussian because of the Gaussian noise assumption that we have made. As such, the likelihood function is a product of

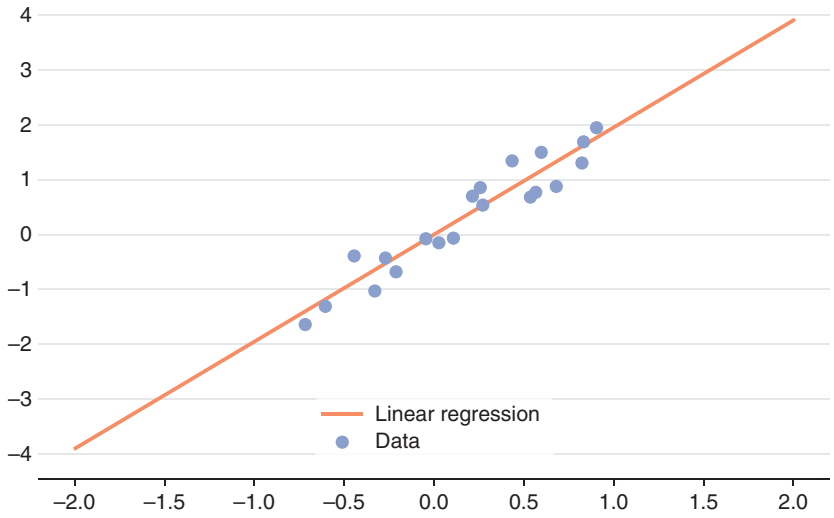

Figure 4.4 Linear regression example.

Gaussians. For ease of computation, but also to avoid numerical issues[2], we take the logarithm of Equation 4.14 and we seek to minimize the negative log-likelihood:

$$J(\theta) = -\log p(\boldsymbol{Y}|\boldsymbol{X}, \theta) = -\log \prod_{i=1}^{N} p(\boldsymbol{y}_i|\boldsymbol{x}_i, \theta) = -\sum_{i=1}^{N} \log p(\boldsymbol{y}_i|\boldsymbol{x}_i, \theta) \tag{4.16}$$

After observing that in our particular case, $\log p(\boldsymbol{y}_i|\boldsymbol{x}_i, \theta) = -\frac{1}{2\sigma^2}(\boldsymbol{y}_i - \boldsymbol{x}_i^T\theta)^2$, and some mathematical computations[3], we can define our final objective function as:

$$J(\theta) = \frac{1}{2\sigma^2}||\boldsymbol{Y} - \boldsymbol{X}\theta||^2 \tag{4.17}$$

where $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]^T$ and $\boldsymbol{Y} = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N]^T$. From this point, we continue by computing the analytic derivatives of J with respect to θ and we have two options[4]:

- Compute the analytical solution that satisfies $\frac{dJ}{d\theta} = 0$: in that case, the optimum parameters are $\theta^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y}$,
- Follow the gradient with any gradient-based optimizer (e.g., L-BFGS [Liu and Nocedal 1989] or simple gradient-descent) in order to minimize J.

To understand a bit better how LR works, let's consider the following example:

2. When implementing linear regression on a computer program.

3. We have ignored the constant terms that appear in the likelihood computations. Please refer to Deisenroth et al. [2019] for further details.

4. We omit the details for clarity and lack of space.

Example 4.3 **Linear regression example**

In this example, we use LR to fit some observed data. Let's assume the following dataset:

$$\boldsymbol{X} = \begin{bmatrix} 0.680375 \\ -0.211234 \\ 0.566198 \\ 0.59688 \\ 0.823295 \\ -0.604897 \\ -0.329554 \\ 0.536459 \\ -0.444451 \\ 0.10794 \\ -0.0452059 \\ 0.257742 \\ -0.270431 \\ 0.0268018 \\ 0.904459 \\ 0.83239 \\ 0.271423 \\ 0.434594 \\ -0.716795 \\ 0.213938 \end{bmatrix} \rightarrow \boldsymbol{Y} = \begin{bmatrix} 0.877051 \\ -0.679582 \\ 0.769628 \\ 1.49794 \\ 1.30327 \\ -1.30885 \\ -1.02932 \\ 0.681727 \\ -0.389977 \\ -0.0658633 \\ -0.0774794 \\ 0.854596 \\ -0.428222 \\ -0.150365 \\ 1.94647 \\ 1.68907 \\ 0.53643 \\ 1.34196 \\ -1.64107 \\ 0.699233 \end{bmatrix} \tag{4.18}$$

This dataset was generated by $\boldsymbol{x}_i = 2\boldsymbol{y_i} + \varepsilon$ (where ε is a small noise). In Figure 4.4, we can see the fitted curved found by LR (Equation 4.17).

Estimating the variance So far we have assumed a fixed and known σ^2. To estimate σ^2 from the observed data, we re-write Equation 4.16 to include it along with θ:

$$J(\theta, \sigma^2) = -\log p(\boldsymbol{Y}|\boldsymbol{X}, \theta, \sigma^2) = -\sum_{i=1}^{N} \log p(\boldsymbol{y}_i|\boldsymbol{x}_i, \theta, \sigma^2) \tag{4.19}$$

Taking the derivative of J with respect to σ^2 and setting it to zero (requiring $\sigma^2 > 0$), we get the following analytical solution: $\sigma^{2*} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{y}_i - \boldsymbol{x}_i^T\theta)^2$.

4.2.1.2 Neural Networks

NNs are currently the most widely used models in ML due to their successes over the last decades [LeCun et al. 2015, Schmidhuber 2015]. They fall into the parametric models class.

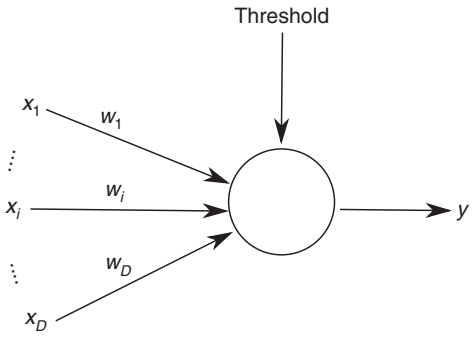

Figure 4.5 Perceptron.

What is a NN? We will start by examining the "ancestor" of an NN: the *perceptron* (Figure 4.5). Interestingly, perceptrons were first developed in the 1950s–1960s [Rosenblatt 1961], but still capture the main ideas behind NNs. A perceptron is a neuron that takes binary inputs $x_i \in \{0,1\}$ and produces a single binary output y. Each input is connected to the neuron by a weight, w_i, and the output $y \in \{0,1\}$ is computed as follows:

$$y = \begin{cases} 0, & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1, & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases} \tag{4.20}$$

Let's consider a simple example to better understand how a perceptron works:

Example 4.4 **Perceptron example**

Imagine that you are a student and your school is organizing a game night. You like games and thus you are considering going to this event. You might want to decide by weighting different factors, for example:

- Is your best friend coming?
- Is your favorite game (e.g., Dungeons and Dragons [D&D]) going to be part of the event?
- Is your worst game (e.g., poker) going to be excluded from the event?

We can represent these three factors by corresponding binary variables x_0, x_1, and x_2. For instance, we can have $x_0 = 0$ if your best friend is not coming and $x_0 = 1$ if he is coming. Similarly, we define x_1 and x_2. By varying the weights and the threshold, we can get different models of decision-making, that is, values for y. For example, if we set $w_0 = 5$, $w_1 = 2$, $w_1 = 1$, and threshold $= 3$, then if your best friend comes you will definitely go, whereas we need both your favorite game to be part and your worst game to be absent to attend the event, if your best friend is not

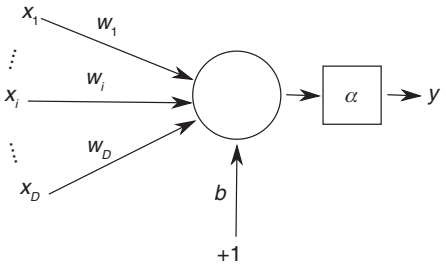

Figure 4.6 Neuron with bias and generic activation function.

coming. *Try different variations of the weights and the threshold to see what you get in this simple setting.*

One can easily observe that the perceptron has its limitations. For example, let's try to find weights for the following function (the XOR function):

$$y = f(x_0, x_1) = \begin{cases} 0, & \text{if } x_0 = x_1 \\ 1, & \text{if } x_0 \neq x_1 \end{cases} \tag{4.21}$$

Before moving on, please spend some time to try to find weights and threshold values that can represent the XOR function. No matter how hard you try, it is not possible to find weights and threshold values to properly model this function. This is because a perceptron can only separate linearly separable problems. Since the XOR function is not linearly separable, it really is impossible for a single hyperplane to separate it.

From the 1960s, things have improved a lot and now the modern neurons have a similar form, but allow for more flexibility (Figure 4.6). The main differences from the perceptron are: (a) the inputs can now be any real-valued number[5], that is, $\boldsymbol{x} \in \mathbb{R}^D$, (b) a bias unit is included that does not depend on the inputs, and (c) the step function (Equation 4.20) is replaced by any generic function that we call an *activation* function, α (we also do not need the threshold value anymore). In short, the output of the neuron is defined as follows:

$$y = \alpha\left(\sum_i w_i x_i + b\right) \tag{4.22}$$

Activation functions Some widely used activation functions are the following (see Figure 4.7):

5. This is of course application-dependent.

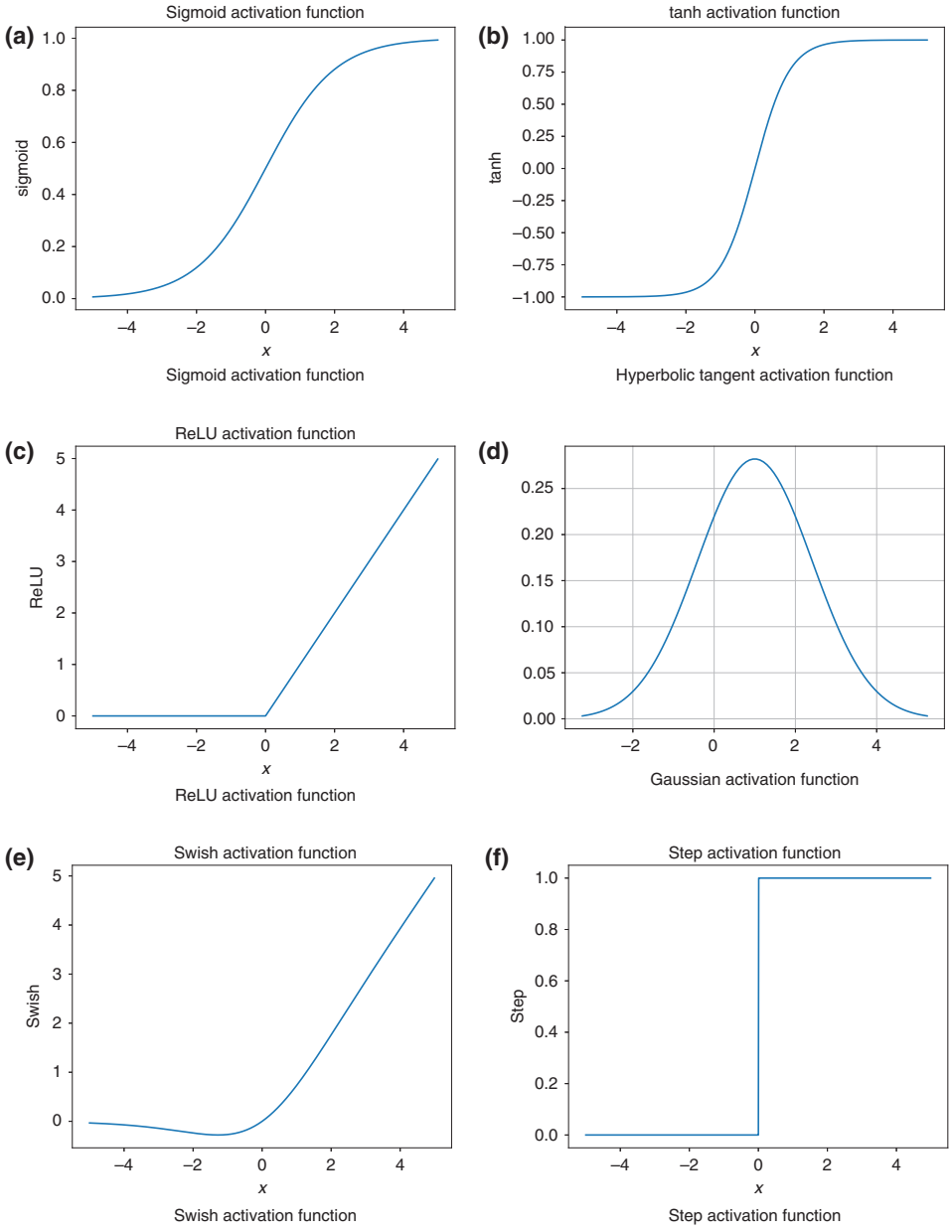

Figure 4.7 Activations functions for neural networks.
(a) Sigmoid activation function. (b) Hyperbolic tangent activation function. (c) ReLU activation function. (d) Gaussian activation function. (e) Swish activation function. (f) Step activation function.

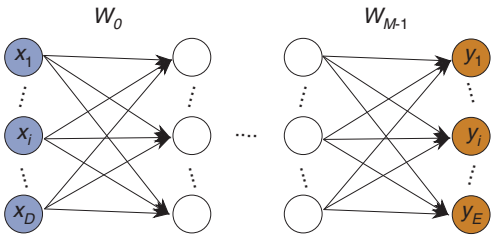

Figure 4.8 Feedforward neural networks.

- Sigmoid: $\alpha(x) = \frac{1}{1+e^{-x}}$ (Figure 4.7[a]),
- Hyperbolic tangent: $\alpha(x) = \tanh(x)$ (Figure 4.7[b]),
- ReLU: $\alpha(x) = \max(0, x)$ (Figure 4.7[c]),
- Gaussian: $\alpha(x) = e^{-x^2}$ (Figure 4.7[d]),
- Swish: $\alpha(x) = x \cdot \text{sigmoid}(x)$ (Figure 4.7[e]),
- Step: $\alpha(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if otherwise} \end{cases}$ (Figure 4.7[f]).

Even with these additions, a single neuron cannot represent any arbitrary function. To make the representation capabilities even bigger, we do exactly what the name of this section implies: we create a network of neurons, that is, we create a *neural network*.

Formally, an NN is a graph (of any form) where neurons are interconnected with weights, biases, and possibly activation functions. In this section, we will focus only on acyclic graphs where neurons are organized in sequential layers. In each layer the output is the weighted sum of the output of the previous layer passed through an activation function (Figure 4.8). This is the most common type of NNs and they are usually referred to as *feedforward neural networks*.

Formally, if we have an input variable $\boldsymbol{x} \in \mathbb{R}^D$ to the NN, an output variable $\boldsymbol{y} \in \mathbb{R}^E$ and M layers, we can define a feedforward NN as follows:

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{g}_M \\ \boldsymbol{g}_0 &= \boldsymbol{x} \\ \boldsymbol{g}_i &= \alpha_{i-1}(\boldsymbol{W}_{i-1}\boldsymbol{g}_{i-1} + \boldsymbol{b}_{i-1}) \end{aligned} \tag{4.23}$$

Now that we have defined what feedforward NNs are, how can we use them to perform regression?

Let $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and $\boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N\}$ be the observed dataset like in Section 4.2.1: i.e., $\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \varepsilon$. Let's also define $\hat{f}(\boldsymbol{x})$ as the output of our NN when given an input $\boldsymbol{x}$. Assuming that the network topology and activations are fixed, it is easy to observe that the shape of $\hat{f}$ depends only on the weights and biases. Defining $\theta = [\boldsymbol{W}_0, \boldsymbol{b}_0, \ldots, \boldsymbol{W}_{M-1}, \boldsymbol{b}_{M-1}]$, we can write the output of the feedforward NN as $\hat{f}(\boldsymbol{x}|\theta)$. For regression tasks, we usually seek to minimize the *mean squared error* (MSE):

$$J(\theta) = \frac{\sum_{i=1}^{N}(\boldsymbol{y}_i - \hat{f}(\boldsymbol{x}_i|\theta))^2}{N} \tag{4.24}$$

MSE is just one example of a loss function suited for regression; we can also use the sum of the squared error or the mean absolute error. We can of course create our own loss function that is suited to our specific case.

When trying to compute the derivative of J with respect to the parameters θ, we can observe the following: *The layer sequencing (see Equation 4.23) creates a composition of functions.* This makes it possible to apply the *chain rule* starting from the error in the last layer and propagating it back to the inputs. This procedure is widely known as *back-propagation* [Rumelhart et al. 1986, LeCun et al. 2015, Schmidhuber 2015]. More formally we can compute the partial derivative of the weights and biases of layer i with respect to J as follows:

$$\frac{\partial J}{\partial(\boldsymbol{W}_i, \boldsymbol{b}_i)} = \frac{\partial J}{\partial \boldsymbol{g}_M}\frac{\partial \boldsymbol{g}_M}{\partial \boldsymbol{g}_{M-1}} \cdots \frac{\partial \boldsymbol{g}_i}{\partial(\boldsymbol{W}_i, \boldsymbol{b}_i)} \tag{4.25}$$

In order to better understand how back-propagation works, let's consider the following example:

Example 4.5 **Back-propagation example**

Let's assume that we want to approximate the function $f_{ex}(\boldsymbol{x}) = \sin(x_0)\cos(x_1)$, where $\boldsymbol{x} = [x_0, x_1]^T \in \mathbb{R}^2$. For this we build a feedforward NN with one hidden layer of two units and tanh as the activation function (Figure 4.9). We have in total six weights and three biases.

The shape of the weights and biases is as follows:

$$\boldsymbol{W}_0 = \begin{bmatrix} w_{00}, & w_{01} \\ w_{02}, & w_{03} \end{bmatrix}, \ \boldsymbol{b}_0 = [b_{00}, \ b_{01}]^T$$
$$\boldsymbol{W}_1 = [w_{10}, \ w_{11}]^T, \ \boldsymbol{b}_1 = [b_{10}]$$

We want to compute the partial derivatives of the MSE when given the observation $\{\boldsymbol{x}_+ = [x_0, x_1]^T, f_{ex}(\boldsymbol{x}_+) = y_+\}$. We first compute the forward pass to find the error:

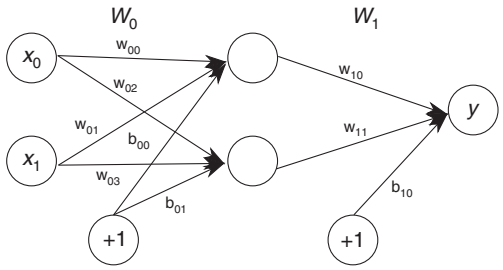

Figure 4.9 Neural network example.

$$
\begin{aligned}
\boldsymbol{g}_1 &= \tanh(\boldsymbol{W}_0\boldsymbol{x}_+ + \boldsymbol{b}_0) = \begin{bmatrix} g_{10} \\ g_{11} \end{bmatrix} \\
&= \begin{bmatrix} \tanh(w_{00}x_0 + w_{01}x_1 + b_{00}) \\ \tanh(w_{02}x_0 + w_{03}x_1 + b_{01}) \end{bmatrix} \\
\boldsymbol{g}_2 &= \tanh(\boldsymbol{W}_1\boldsymbol{g}_1 + \boldsymbol{b}_1) = [\tanh(w_{10}g_{10} + w_{11}g_{11} + b_{10})] \\
J &= ||y_+ - \boldsymbol{g}_2||^2
\end{aligned}
$$

Then we continue to the partial derivatives of the last layer:

$$
\begin{aligned}
\frac{\partial J}{\partial w_{10}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial \tanh}\frac{\partial \tanh}{\partial w_{10}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)\times g_{10} \\
\frac{\partial J}{\partial w_{11}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial \tanh}\frac{\partial \tanh}{\partial w_{11}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)\times g_{11} \\
\frac{\partial J}{\partial b_{10}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial \tanh}\frac{\partial \tanh}{\partial b_{10}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)\times 1
\end{aligned}
$$

Lastly, we continue to the partial derivatives of the first layer:

$$
\begin{aligned}
\frac{\partial J}{\partial w_{00}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial g_{10}}\frac{\partial g_{10}}{\partial \tanh}\frac{\partial \tanh}{\partial w_{00}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)w_{10}\times(1-g_{10}^2)\times x_0 \\
\frac{\partial J}{\partial w_{01}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial g_{10}}\frac{\partial g_{10}}{\partial \tanh}\frac{\partial \tanh}{\partial w_{01}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)w_{10}\times(1-g_{10}^2)\times x_1 \\
\frac{\partial J}{\partial b_{00}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial g_{10}}\frac{\partial g_{10}}{\partial \tanh}\frac{\partial \tanh}{\partial b_{00}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)w_{10}\times(1-g_{10}^2)\times 1 \\
\frac{\partial J}{\partial w_{02}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial g_{11}}\frac{\partial g_{11}}{\partial \tanh}\frac{\partial \tanh}{\partial w_{02}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)w_{11}\times(1-g_{11}^2)\times x_0 \\
\frac{\partial J}{\partial w_{03}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial g_{11}}\frac{\partial g_{11}}{\partial \tanh}\frac{\partial \tanh}{\partial w_{03}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)w_{11}\times(1-g_{11}^2)\times x_1 \\
\frac{\partial J}{\partial b_{01}} &= \frac{\partial J}{\partial \boldsymbol{g}_2}\frac{\partial \boldsymbol{g}_2}{\partial g_{11}}\frac{\partial g_{11}}{\partial \tanh}\frac{\partial \tanh}{\partial b_{01}} = -2\cdot(y_+ - \boldsymbol{g}_2)\times(1-\boldsymbol{g}_2^2)w_{11}\times(1-g_{11}^2)\times 1
\end{aligned}
$$

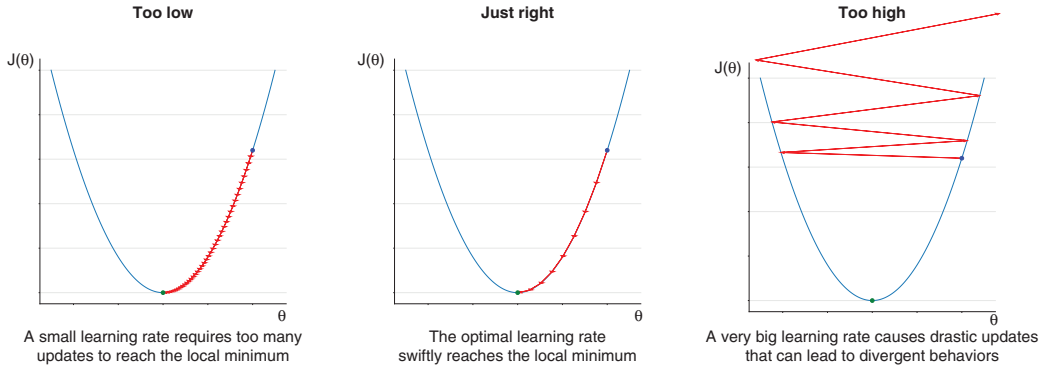

Figure 4.10 Choosing the best learning rate.

This is of course tedious to do by hand for big networks, and thus the usage of an NN library is recommended: pytorch [Paszke et al. 2017], tensorflow [Abadi et al. 2015], weka [Witten et al. 2016], and tiny-dnn[6] are a few examples of open-source libraries.

Gradient descent In order to optimize the loss function J and since we can compute the gradient using back-propagation, we use gradient descent. Gradient descent works as follows:

$$\theta_{n+1} = \theta_n + \eta \frac{\partial J}{\partial \theta} \tag{4.26}$$

where η is the so-called *learning rate* and n is the number of gradient optimization steps. Choosing the correct learning rate is crucial as a very small one can lead to very slow convergence, whereas a very big one can lead to complete divergence (Figure 4.10). For this reason, several approaches have been proposed to adapt the learning rate online (e.g., ADADELTA [Zeiler 2012]). To make the effect of choosing the learning rate less important, the aspect of momentum has been introduced in many cases. In practice, instead of the previous update, we use the following:

$$\begin{aligned} \boldsymbol{G}_{n+1} &= \beta \boldsymbol{G}_n + (1-\beta)\frac{\partial J}{\partial \theta} \\ \theta_{n+1} &= \theta_n + \eta \boldsymbol{G}_{n+1} \end{aligned} \tag{4.27}$$

In essence, the gradient update takes information from the previous steps, instead of relying only on the current estimate and thus avoiding big fluctuations.

Training with big datasets Usually we have very big datasets to train an NN, that is, $N \gg 10{,}000$. In this case, computing the forward and backward passes can be very time-consuming. For this reason, we usually split the dataset into *batches*: in

6. https://github.com/tiny-dnn/tiny-dnn

essence, a batch is subset of the whole dataset. Usually, the size of each batch is relatively small (i.e., $16 - 64$). Working with batches instead of the whole dataset, creates two additional questions/issues: (a) how to generate these batches, and (b) how to use gradient descent when we have access to just an approximation of the gradient[7]. For the first issue, the solution is rather simple: either we draw randomly (with replacement) data points from the whole dataset or we first shuffle the dataset and then we select sequential chunks of data until the whole dataset is used (and then we go back to the start). For the second issue, more advanced and sophisticated optimization algorithms, like Adam [Kingma and Ba 2014], are devised for solving this *stochastic gradient descent* problem.

4.2.1.3 Gaussian Processes

So far we have seen parametric models, which have limited capacity (defined by their structure and number of parameters). On the contrary, non-parametric models have potentially infinite capacity (usually limited by practical implementations). One of the most successful non-parametric models for ML is GP. A GP is an extension of multivariate Gaussian distribution to an infinite-dimension stochastic process for which any finite combination of dimensions will be a Gaussian distribution [Rasmussen and Williams 2006]. More precisely, it is a distribution over functions, completely specified by its mean function, $m(\cdot)$ and covariance function, $k(\cdot,\cdot)$ and we write:

$$\hat{f}(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x},\boldsymbol{x})) \tag{4.28}$$

GPs fall into the non-parametric class of models. Assuming $\boldsymbol{X} = \{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N\}$, $\boldsymbol{Y} = \{\boldsymbol{y}_1,\ldots,\boldsymbol{y}_N\}$ is a set of observations, like in Section 4.2.1, we can query the GP at a new input point $\boldsymbol{x}_+$ as follows:

$$\begin{aligned} p(\hat{f}|\boldsymbol{X},\boldsymbol{Y},\boldsymbol{x}_+) &= \mathcal{N}(\mu(\boldsymbol{x}_+), \sigma^2(\boldsymbol{x}_+)) \\ \mu(\boldsymbol{x}_+) &= \boldsymbol{k}^T K^{-1}\boldsymbol{Y} \\ \sigma^2(\boldsymbol{x}_+) &= k(\boldsymbol{x}_+,\boldsymbol{x}_+) - \boldsymbol{k}^T K^{-1}\boldsymbol{K} \end{aligned} \tag{4.29}$$

where the kernel vector is defined as $\boldsymbol{k} = [k(\boldsymbol{x}_1,\boldsymbol{x}_+),\ldots,k(\boldsymbol{x}_N,\boldsymbol{x}_+)]$ and the kernel matrix is computed as follows:

$$K = \begin{bmatrix} k(\boldsymbol{x}_1,\boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_1,\boldsymbol{x}_N) \\ \vdots & \ddots & \\ k(\boldsymbol{x}_N,\boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_N,\boldsymbol{x}_N) \end{bmatrix} + \sigma_n^2\boldsymbol{I} \tag{4.30}$$

where σ_n is the noise variance.

7. Because of the batches, we do not have accurate values of the gradient.

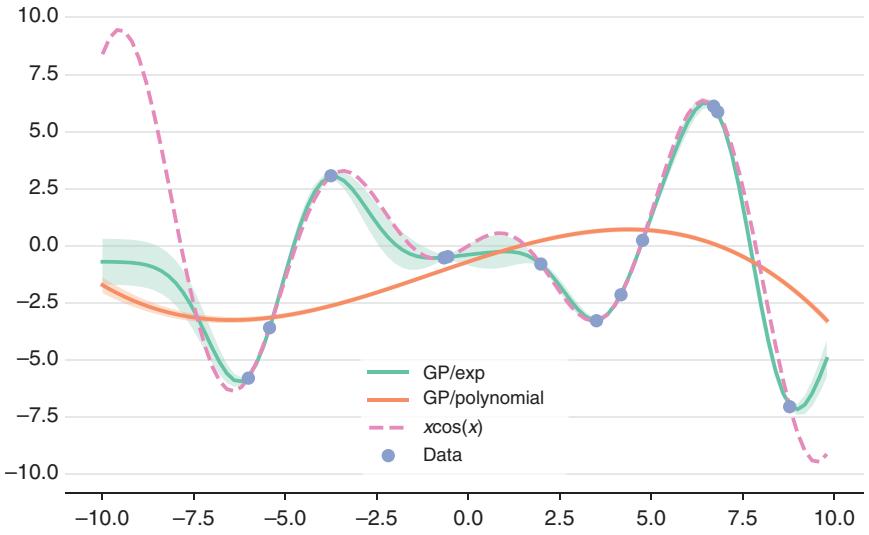

Figure 4.11 Gaussian process regression with different kernels.

The covariance function or *kernel*, k, is a measure of similarity between input points. The most widely used kernels are the following:

- Squared exponential kernel: $k(\boldsymbol{x}, \boldsymbol{x}') = \alpha \exp\left(\frac{||\boldsymbol{x}-\boldsymbol{x}'||^2}{2l^2}\right)$
- Matérn-class kernel: $k(\boldsymbol{x}, \boldsymbol{x}') = \alpha \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\frac{||\boldsymbol{x}-\boldsymbol{x}'||}{l}\right)^{\nu} K_\nu \left(\sqrt{2\nu}\frac{||\boldsymbol{x}-\boldsymbol{x}'||}{l}\right)$ where ν is usually either $\frac{3}{2}$ or $\frac{5}{2}$ and K_ν is a modified Bessel function [Rasmussen and Williams 2006].
- Rational quadratic kernel: $k(\boldsymbol{x}, \boldsymbol{x}') = \alpha\left(\frac{||\boldsymbol{x}-\boldsymbol{x}'||^2}{2al^2}\right)^{-a}$

Of course, the choice of a kernel will affect the shape of the predictions of the GP. In Figure 4.11, we show an example where we fit two GPs with different kernels: one with the squared exponential kernel and one with a third-order polynomial kernel[8]; we can see that the fitted functions are very different in each case.

Interestingly, GPs provide an analytical solution to the regression problem. However, when defining our kernel there are a few parameters that we can tune and greatly affect the shape of the GP. For example, in Figure 4.12, we see how a GP with the squared exponential kernel completely changes shape when we change the l parameter of the kernel. In order to identify the optimal hyperparameters, we need to define a loss function and minimize it. We have a lot of options, but the most common and successful ones are the leave-one-out-cross-validation, and the

8. Polynomial kernel: $k(\boldsymbol{x}, \boldsymbol{x}') = \alpha[\boldsymbol{x}, \boldsymbol{x}^2, \boldsymbol{x}^3]^T[\boldsymbol{x}', \boldsymbol{x}'^2, \boldsymbol{x}'^3]$.

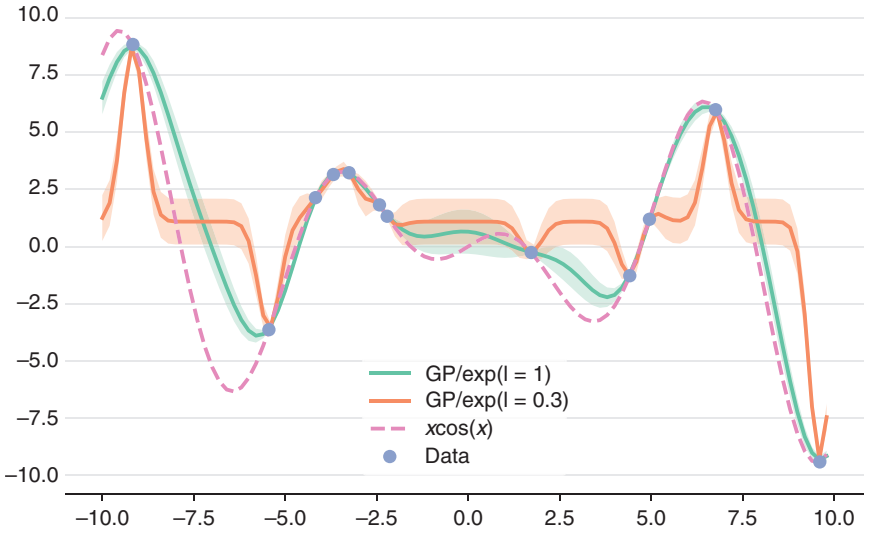

Figure 4.12 Gaussian process regression with the same kernel, but different hyperparameters.

likelihood [Rasmussen and Williams 2006]. We will focus here on the maximum likelihood estimation (MLE), where we define the *log marginal likelihood* and we maximize it:

$$\log p(\boldsymbol{Y}|\boldsymbol{X}, \theta_{\text{kernel}}) = -\frac{1}{2}\boldsymbol{Y}^T K^{-1}\boldsymbol{Y} - \frac{1}{2}\log|K| - \frac{N}{2}\log 2\pi \tag{4.31}$$

where θ_{kernel} are the hyperparameters of the kernel: the kernel matrix K depends on them.

The three terms of the marginal likelihood have interpretable roles: (a) $-\boldsymbol{Y}^T K^{-1}\boldsymbol{Y}/2$ is the only term involving the observed targets and thus the data-fit, (b) $-\log|K|/2$ is the complexity penalty depending only on the covariance function and the inputs, and (c) $N\log(2\pi)/2$ is a normalization constant.

To set the hyperparameters by maximizing the marginal likelihood, we seek the partial derivatives of the marginal likelihood with respect to the hyperparameters:

$$\begin{aligned}\frac{\partial \log p(\boldsymbol{Y}|\boldsymbol{X}, \theta_{\text{kernel}})}{\partial \theta_i} &= \frac{1}{2}\boldsymbol{Y}^T K^{-1}\frac{\partial K}{\partial \theta_i}K^{-1}\boldsymbol{Y} - \frac{1}{2}\text{tr}(K^{-1}\frac{\partial K}{\partial \theta_i}) \\ &= -\frac{1}{2}\text{tr}\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^T - K^{-1})\frac{\partial K}{\partial \theta_i}\right), \quad \text{where } \boldsymbol{\alpha} = K^{-1}\boldsymbol{Y}\end{aligned} \tag{4.32}$$

In order to quickly experiment with GPs, using an optimized library is recommended. For example, one can use *limbo* [Cully et al. 2018] in C++, and *GPy*[9] in Python.

9. http://github.com/SheffieldML/GPy

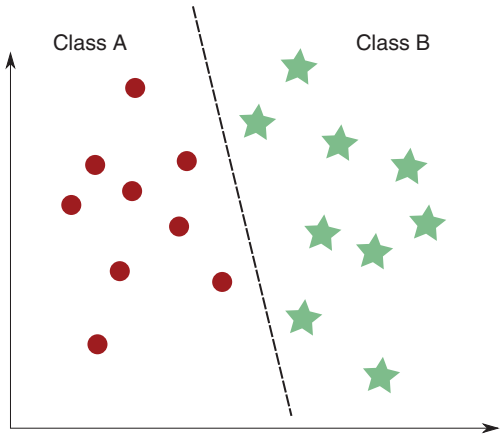

Figure 4.13 Classification example.

4.2.2 Classification

As we have already stated, classification algorithms try to find the mapping $f : \boldsymbol{x} \rightarrow \boldsymbol{y}$ from the input variables $\boldsymbol{x} \in \mathbb{R}^D$ to the discrete or categorical output variables $\boldsymbol{y}$. More formally, we assume access to a dataset of the form: $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, where N is the number of samples, and $\boldsymbol{y}_i = f(\boldsymbol{x}_i)$. We usually encounter two different forms of the output variables: (a) $\boldsymbol{y}_i \in \mathbb{Z}$ where each integer number defines a class, and (b) $\boldsymbol{y}_i$ is a one-hot vector of length K, where K is the number of classes. A one-hot vector is a vector with all zeroes except from one element.

The goal of a classification algorithm, like in regression, is not only to minimize the error from the observed samples, but also to generalize to unseen datapoints. Figure 4.13 shows an example of classification where using the observed samples we can split the space with a line that separates class A from class B.

4.2.2.1 Naïve Bayes

NB is a category of techniques for constructing classifiers [Maron 1961]. There exist many variations on how to learn such classifiers, but NB classifiers make a common assumption: *the value of a particular feature is independent of the value of any other feature, given the class variable*. More formally, NB is a probabilistic model that assigns probabilities to each possible class k as follows:

$$p(C_k|\boldsymbol{x} = \{x_1, \ldots, x_D\}) \tag{4.33}$$

Note that $\boldsymbol{x} \in \mathbb{R}^D$ is one sample, and we use the notation x_i to denote the ith element of the vector $\boldsymbol{x}$. Using Bayes' theorem (Equation 4.8), we can re-write this conditional as:

$$p(C_k|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|C_k)p(C_k)}{p(\boldsymbol{x})} \tag{4.34}$$

In practice, we only care about the numerator of this fraction, as the denominator does not depend on C and the x_i are given: hence, the denominator is just a constant. The numerator, now, is equivalent to the joint probability (see Equations 4.2 and 4.7):

$$\begin{aligned} p(\boldsymbol{x}|C_k)p(C_k) &= p(x_0, \ldots, x_D, C_k) \\ &= p(x_1|x_2, \ldots, x_D, C_k)p(x_2, \ldots, x_D, C_k) \\ &= p(x_1|x_2, \ldots, x_D, C_k)p(x_2|x_3, \ldots, x_D, C_k)p(x_3, \ldots, x_D, C_k) \\ &= \ldots \\ &= p(x_1|x_2, \ldots, x_D, C_k) \cdots p(x_{D-1}|x_D, C_k)p(x_D|C_k)p(C_k) \end{aligned} \tag{4.35}$$

In NB as already discussed we assume that each feature is independent of the value of any other feature given the class variable. In mathematical terms, this is equivalent to:

$$p(x_i|x_{i+1}, \ldots, x_D, C_k) \approx p(x_i|C_k) \tag{4.36}$$

Note that we used $\approx$ instead of $=$ as this is only an approximation of the actual conditional, and really holds only in special cases. Nevertheless, all NB classifiers make this assumption to simplify the mathematical computations. Thus, with this approximation we can now write the conditional in Equation 4.34 as:

$$p(C_k|\boldsymbol{x}) = \frac{1}{Z}p(C_k)\prod_{i=1}^{D} p(x_i|C_k) \tag{4.37}$$

where $Z = p(\boldsymbol{x}) = \sum_k p(C_k)p(\boldsymbol{x}|C_k)$ is a normalization constant that depends only on $\boldsymbol{x}$ and on prior probabilities.

Once we have the approximation of the conditional in Equation 4.37, we can construct a classifier as follows:

$$\boldsymbol{y} = \hat{f}(\boldsymbol{x}^+) = \max_{k \in \{1 \ldots K\}} p(C_k)\prod_{i=1}^{D} p(x_i^+|C_k) \tag{4.38}$$

Practical considerations In practice, when given a dataset $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\} \rightarrow \boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N\}$, where N is the number of samples, and $\boldsymbol{y}_i \in C_k$ for $k \in \{1 \ldots K\}$, we need to compute two set of values: (1) $p(C_k)$, and (2) $p(x_i|C_k)$ for $i \in \{1 \ldots D\}$, $k \in \{1 \ldots K\}$. These values will enable us to construct the classifier given by Equation 4.38. $p(C_k)$ can be very easily computed as $p(C_k) = \frac{N_{C_k}}{N}$, where N_{C_k} is the number of times that class C_k appears in the dataset. Another way is to have some prior information about the probabilities of the classes: for example, if the two classes are *male* and *female*, then from the general population we know that $p(\text{male}) \approx 0.5$ and $p(\text{female}) \approx 0.5$. On the contrary, calculating $p(x_i|C_k)$ is a bit trickier. If $\boldsymbol{x}$ is discrete or categorical, then $p(x_i = v|C_k) = \frac{N^{vi}_{C_k}}{N_{C_k}}$, where $N^{vi}_{C_k}$ is the number of times the class C_k appears in dataset when $x_i = v$. When $\boldsymbol{x}$ is continuous, we need to somehow approximate $p(x_i|C_k)$ from the available datapoints. To do so, we can choose any probability distribution: the most popular choice is the Gaussian distribution. Interestingly, we can approximate $p(x_i|C_k)$ even when $\boldsymbol{x}$ is discrete: multinomial and Bernoulli distributions are the most popular in this case. To better understand the concepts behind the NB classifiers, let's see an example usage. Before we start calculations, it is worth reminding of the need independence of the input variables. NB results are valid if the input variables are statistically independent, given the class variable. In case this does not hold, the results may be not valid. There exist formal methods for testing statistical independence (see Chapter 2, Section 2.7). In this particular example, we consider that input variables are statistically independent, although one might have objections on whether "outlook" and "temperature" are independent.

Example 4.6 **Naïve Bayes classification**

In this example, we use the "Play Tennis" dataset (see Table 4.1). The dataset consists of 14 samples ($N = 14$), where both the input variables, $\boldsymbol{x}$, and the output variable, $\boldsymbol{y}$, are discrete. In particular: $x_1 = \{\text{Sunny, Overcast, Rain}\}$, $x_2 = \{\text{Hot, Mild, Cool}\}$, $x_3 = \{\text{High, Normal}\}$, $x_4 = \{\text{Weak, Strong}\}$, and $y = \{\text{Yes, No}\}$. In this dataset, there exist two labels, "Yes" and "No," that determine whether we are allowed to play tennis or not depending on the input variables. This means that $K = 2$. We begin by computing the probabilities $p(C_k)$:

$$p(C_1) = p(y = \text{Yes}) = \frac{N_{C_1}}{N} = \frac{9}{14} \approx 0.64$$

$$p(C_2) = p(y = \text{No}) = \frac{5}{14} = 1 - p(C_1) \approx 0.36$$

Table 4.1 **Play tennis dataset**

#	Outlook	Temperature	Humidity	Wind	Play tennis
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

We then compute the probabilities $p(x_i = v|C_k)$:

$$p(x_1 = \text{Sunny} \mid C_1) = \frac{N_{C_k}^{vi}}{N_{C_k}} = \frac{2}{9} \approx 0.22$$
$$p(x_1 = \text{Sunny} \mid C_2) = \frac{3}{5} \approx 0.6$$
$$p(x_1 = \text{Overcast} \mid C_1) = \frac{4}{9} \approx 0.44$$
$$p(x_1 = \text{Overcast} \mid C_2) = \frac{0}{5} = 0$$

$$p(x_1 = \text{Rain} \mid C_1) = \frac{3}{9} \approx 0.33$$
$$p(x_1 = \text{Rain} \mid C_2) = \frac{2}{5} = 0.4$$
$$p(x_2 = \text{Hot} \mid C_1) = \frac{2}{9} \approx 0.22$$
$$p(x_2 = \text{Hot} \mid C_2) = \frac{2}{5} = 0.4$$
$$p(x_2 = \text{Mild} \mid C_1) = \frac{4}{9} \approx 0.44$$
$$p(x_2 = \text{Mild} \mid C_2) = \frac{2}{5} = 0.4$$
$$p(x_2 = \text{Cool} \mid C_1) = \frac{3}{9} \approx 0.33$$
$$p(x_2 = \text{Cool} \mid C_2) = \frac{1}{5} = 0.2$$

$$
\begin{aligned}
p(x_3 = \text{High} \mid C_1) &= \frac{3}{9} \approx 0.33 \\
p(x_3 = \text{High} \mid C_2) &= \frac{4}{5} = 0.8 \\
p(x_3 = \text{Normal} \mid C_1) &= \frac{6}{9} \approx 0.66 \\
p(x_3 = \text{Normal} \mid C_2) &= \frac{1}{5} = 0.2
\end{aligned}
$$

$$
\begin{aligned}
p(x_4 = \text{Weak} \mid C_1) &= \frac{6}{9} \approx 0.66 \\
p(x_4 = \text{Weak} \mid C_2) &= \frac{2}{5} = 0.4 \\
p(x_4 = \text{Strong} \mid C_1) &= \frac{3}{9} \approx 0.33 \\
p(x_4 = \text{Strong} \mid C_2) &= \frac{3}{5} \approx 0.6
\end{aligned}
$$

Now that we have computed all these probabilities, we can classify new samples. Let's assume that we have the following: $\boldsymbol{x}_+ = [\text{Sunny, Cool, High, Strong}]$. We should optimize Equation 4.38 to see which class this new sample belongs to. We need to compute the following values:

$$
\begin{aligned}
v_{C_1} &= p(C_1) \prod_{i=1}^{D} p(x_i^+ | C_1) \\
&= p(C_1) p(x_1 = \text{Sunny} | C_1) p(x_2 = \text{Cool} | C_1) p(x_3 = \text{High} | C_1) p(x_4 = \text{Strong} | C_1) \\
&= 0.64 \times 0.22 \times 0.33 \times 0.33 \times 0.33 \approx 0.00506 \\
v_{C_2} &= p(C_2) \prod_{i=1}^{D} p(x_i^+ | C_2) \\
&= p(C_2) p(x_1 = \text{Sunny} | C_2) p(x_2 = \text{Cool} | C_2) p(x_3 = \text{High} | C_2) p(x_4 = \text{Strong} | C_2) \\
&= 0.36 \times 0.6 \times 0.2 \times 0.8 \times 0.6 \approx 0.02074
\end{aligned}
$$

Finally we can also compute the proper probabilities (although not needed since the denominator is constant):

$$
\begin{aligned}
p(C_1 | \boldsymbol{x}_+) &= \frac{v_{C_1}}{v_{C_1} + v_{C_2}} = \frac{0.00506}{0.00506 + 0.02074} \approx 0.196 \\
p(C_2 | \boldsymbol{x}_+) &= \frac{v_{C_2}}{v_{C_1} + v_{C_2}} = \frac{0.02074}{0.00506 + 0.02074} \approx 0.804
\end{aligned}
$$

Since $p(C_2 | \boldsymbol{x}_+) > p(C_1 | \boldsymbol{x}_+)$, we classify $\boldsymbol{x}_+$ as being in the class C_2 and thus the conditions are not suitable for playing tennis.

4.2.2.2 Support Vector Machines

SVMs [Guyon et al. 1993, Cristianini and Shawe-Taylor 2000, Hsu et al. 2003] is a supervised ML algorithm which can be used for both classification or regression challenges[10]. However, it is mostly used in *binary classification* problems, and thus we will present this version. By binary classification, we mean classification problems where only two possible classes are available and each input can correspond only to one class.

The objective of SVMs is to find a hyperplane in an N_f-dimensional space (where N_f is the number of features) that distinctly classifies the data points in two classes. To do so, there exist many possible hyperplanes that we can choose. Our objective is to find a plane that has the maximum margin, that is, the maximum distance between data points of both classes to the hyperplane. Maximizing the margin distance provides some stronger confidence that our classifier will generalize to new data points.

To begin with, let's consider that we have a dataset $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, $\boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N\}$, where $\boldsymbol{x}_i \in \mathbb{R}^D$ and $\boldsymbol{y} \in \{-1, 1\}$: a value of $+1$ means that it is classified in the first class and a value of -1 means that it is in second class.

Hard-margin SVMs If the data points are linearly separable, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible. The region bounded by these two hyperplanes is called the *margin*, and the maximum-margin hyperplane is the hyperplane that lies halfway between them. We can describe these hyperplanes by: $\boldsymbol{w}^T\boldsymbol{x} - b = 1$ for the first class and $\boldsymbol{w}^T\boldsymbol{x} - b = -1$ for the second class. According to geometry (Figure 4.14), the distance between these two hyperplanes is $\frac{2}{\|\boldsymbol{w}\|}$, so to maximize the distance between the planes we want to minimize $\|\boldsymbol{w}\|$. We also want to prevent data points from falling into the margin, and thus we add the following constraints: $\boldsymbol{y}_i(\boldsymbol{w}^T\boldsymbol{x}_i - b) \geq 1$, for all $1 \leq i \leq N$. More formally, we have the following optimization problem:

$$\min_{\boldsymbol{w},b} \|\boldsymbol{w}\|^2 \tag{4.39a}$$

$$\text{subject to } \boldsymbol{y}_i(\boldsymbol{w}^T\boldsymbol{x}_i - b) \geq 1,$$

$$i = 1, \ldots N \tag{4.39b}$$

We can optimize this problem with any off-the-shelf quadratic programming (QP) solver, as the objective is of quadratic form with linear constraints with respect

10. When used in regression problems, we use the name *support vector regression*.

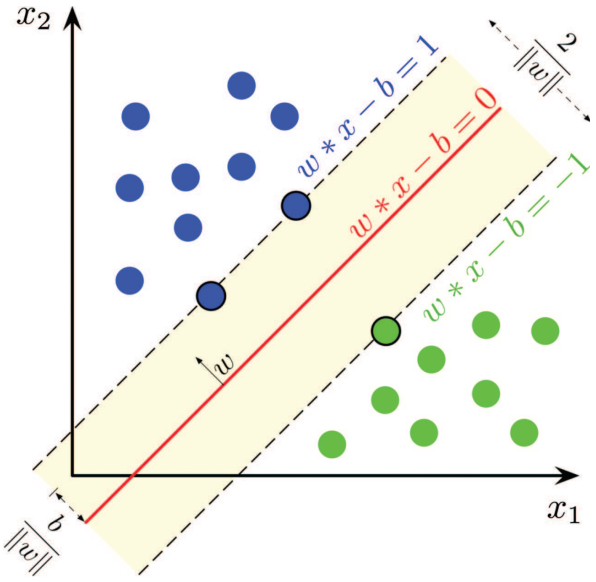

Figure 4.14 Support vector machines for binary classification. Image source: https://en.wikipedia.org/wiki/File:SVM_margin.png.

to $\boldsymbol{w}$ and b. A few examples of available QP solvers are: qpOASES[11], osqp [Stellato et al. 2017][12], GUROBI[13], and MOSEK[14]. Once we find the best $\boldsymbol{w}$ and b, our classifier can be defined as $\hat{f}(\boldsymbol{x}) = \text{sgn}(\boldsymbol{w}^T\boldsymbol{x} - b)$, where sgn is the *sign* function.

Soft-margin SVMs If the data points are not exactly linearly separable (e.g., because of noise), we change a bit the optimization problem and we have the following:

$$\min_{\boldsymbol{w},b} \quad \frac{1}{N}\sum_{i=1}^{N}\max\big(0, 1 - \boldsymbol{y}_i(\boldsymbol{w}^T\boldsymbol{x}_i - b)\big) + \lambda\|\boldsymbol{w}\|^2 \tag{4.40}$$

where λ handles the trade-off between increasing the margin size and ensuring that the $\boldsymbol{x}_i$ lie on the correct side of the margin[15]. We call this soft-margin because the hard constraints in Equation 4.39 are put in the objective function, and thus become *soft*. For sufficiently small values of λ, the second term in the loss function

11. https://github.com/coin-or/qpOASES
12. https://osqp.org/
13. http://www.gurobi.com
14. https://docs.mosek.com/9.0/toolbox/index.html
15. This is just one formulation of the soft-margin SVMs. Please refer to Hsu et al. [2003] for more information.

will become negligible, hence, it will behave similar to the hard-margin SVM, but will still learn if a classification rule is viable or not. We can still optimize this objective with any of-the-shelf QP solver.

Non-linear classification So far, the classifier that we have defined is a linear one: in essence, we can only use it if the data points are (more or less) linearly separable. To make an SVM classifier suitable for non-linear classification, instead of performing the classification in the original space, we transform the input space using the kernel trick [Boser et al. 1992] and perform the classification in this transformed space. This allows us to perform linear classification in the transformed space that can potentially be non-linear in the original space. In essence, we define the following optimization:

$$\min_{\boldsymbol{w},b} \quad \frac{1}{N}\sum_{i=1}^{N}\max\big(0, 1-\boldsymbol{y}_i(\boldsymbol{w}^T\phi(\boldsymbol{x}_i)-b)\big)+\lambda\|\boldsymbol{w}\|^2 \tag{4.41}$$

Here training vectors $\boldsymbol{x}_i$ are mapped into a feature space (possibly higher dimensional) by the function ϕ. SVM finds a linear separating hyperplane with the maximal margin in this feature space. Furthermore, we define the kernel function as $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j)$. We can use any kernel function, but the most widely used ones are:

- Linear: $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T\boldsymbol{x}_j$
- Polynomial: $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma\boldsymbol{x}_i^T\boldsymbol{x}_j + r)^d, \gamma > 0$
- Radial basis function (RBF): $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2), \gamma > 0$
- Sigmoid: $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(\gamma\boldsymbol{x}_i^T\boldsymbol{x}_j + r)$

where γ, r, d are parameters of the kernels.

4.2.2.3 Decision Trees

In this category of algorithms, the produced model is a tree, called a *decision tree* (DT) [Rokach and Maimon 2005]. A DT is a directed tree having the following characteristics:

- There is a node, called a root, which has no incoming branches
- Each internal node has exactly one incoming branch, two or more outgoing branches and is labeled with the name of a feature x_i
- Each leaf node (or decision node) is labeled with the name of a class C_k
- Each branch, descending from an internal node, is usually labeled with one of the possible values of the node's feature, in case it has nominal values, or a value range, in case it is numeric.

A DT is an effort to divide search space into orthogonal regions, following a "divide and conquer" strategy. In other words, it tries to distribute the samples of the (training) dataset into subsets that contain points of different classes. A DT is produced from a given (training) dataset in such a way that it can generalize. In most cases, a top-down process is followed. This means that initial dataset is split in two or more subsets, based on one of the input features, in such a way that the subsets are as homogeneous as possible, that is, their samples belong to as less different classes as possible. The target is to end up to sets with samples of the same class. So, the process iterates until this is achieved or some other criterion is met. The leaves typically include samples of the same class, except if for generalization reasons, the process stops before it is achieved, and the dominant classes are chosen as leaf labels.

In the process of creating a DT, there are some important notions, like the *splitting features*, the *splitting predicates*, the *splitting criterion*, and the *stopping criterion*. The splitting features are those that are selected as labels of the nodes in the DT. Notice that may not all features of a dataset are selected as nodes in the DT. Thus, different subsets of features may lead to different DTs. Also, the features selection order plays an important role in the quality of the DT. In line with this an important issue is which feature will be selected to be the root of the DT.

The selection of features is based on a splitting criterion. A splitting criterion is a measure of the quality of a feature determining whether it is going to be used as a splitting feature at some node or not. There are various splitting criteria, like Information Gain, Gini Index, DKM, Gain Ratio, and so on [Rokach and Maimon 2005]. When a feature is selected it becomes the label of the current node. Each node is a decision point (i.e., we perform some test), where branches go out of it depending on the outcome of the decision. Those branches represent possible outputs of the test and split the set corresponding to the node into as many subsets as its branches. Each instance (example) of the dataset follows one of the branches towards the corresponding child nodes. The test may be an equality test (e.g., between a value of the parent node feature and the corresponding value in the example) or an inequality test (e.g., whether the corresponding value in the example is within a range of values of the parent node feature) or something more complicated. Each branch is labeled with what is called a splitting predicate, which denotes the corresponding test output. When a splitting predicate involves only the corresponding feature, we have the case of a *univariate split*. If it involves more than one feature, it is called a *multivariate split*.

The stopping criterion concerns the point where a DT algorithm stops. It is reasonable to expect that the algorithm stops when all training data are classified by

Algorithm 4.1 **Basic pseudocode for decision trees**

CreateDT($\boldsymbol{X}, \boldsymbol{Y}$)
$T \leftarrow \emptyset$
Determine the best splitting feature, x_i
Add a node to T and label it with the best splitting feature
Add to T one branch per splitting predicate v_i^d of x_i
for $d = 1 \rightarrow V$ {V is the number of values that x_i can take}
  $(\boldsymbol{X}', \boldsymbol{Y}') \leftarrow$ ApplyPredicate($x_i = v_i^d$)
  if stopping point reached for this path
    $T' \leftarrow$ leaf node with appropriate class
  else
    $T' \leftarrow$ CreateDT($\boldsymbol{X}', \boldsymbol{Y}'$)
  endif
Add T' to T
endfor
return T

the tree. However, this is not usually the case. For example, the algorithm may stop earlier to avoid the creation of big trees or to avoid the *overfitting* phenomenon. So, the decision about when to stop concerns a balance between accuracy, algorithm performance, and the generalization capability of the tree.

A high-level pseudocode for a DT is described in Algorithm 4.1. Combinations of different splitting and stopping criteria create a variety of DT algorithms. Some of well-known algorithms are: ID3 [Quinlan 1986], C4.5 [Quinlan 2014], CART [Breiman et al. 1984], and so on.

ID3 Algorithm The ID3 (Iterative Dichotomizer 3) [Quinlan 1986] requires that all the features have nominal (or categorical) values. The splitting predicates are the values of the features and test equality. Its splitting criterion is *information gain*. Information gain is based on the notion of *entropy* [Quinlan 1987]. Entropy denotes the degree of non-homogeneity of a dataset. If all samples in a dataset belong to the same class, then entropy equals zero. For discrete values that ID3 handles, the entropy of a random value $\mathcal{C}$ is defined as follows: $E(\mathcal{C}) = -\sum_{i=1}^{K} p(C_i)\log p(C_i)$, where $p(C_i)$ is the probability of $\mathcal{C}$ taking the value C_i and K is the number of values that $\mathcal{C}$ can take. In our setup, we assume that we represent the possible classes by a random variable and we use the following notation $E(\boldsymbol{X}, \boldsymbol{Y}) = -\sum_{i=1}^{K} p(C_i)\log p(C_i)$ to denote the entropy of this random variable given a dataset $(\boldsymbol{X}, \boldsymbol{Y})$, where $p(C_i) = \frac{N_{\boldsymbol{Y}}^i}{N_{\boldsymbol{X}}}$, $N_{\boldsymbol{X}}$ is the number of elements in $\boldsymbol{X}$, and $N_{\boldsymbol{Y}}^i$ is the number of times the class value

C_i appears in the dataset $(\boldsymbol{X}, \boldsymbol{Y})$. We also define the information gain of a dataset as follows:

$$IG(\boldsymbol{X}, \boldsymbol{Y}, x_i) = E(\boldsymbol{X}, \boldsymbol{Y}) - \sum_{d=1}^{V} \frac{N_{v_i^d}}{N_{\boldsymbol{X}}} E(\boldsymbol{X}_{-v_i^d}, \boldsymbol{Y}_{-v_i^d}) \tag{4.42}$$

where $N_{v_i^d}$ is the number of times that x_i takes the value v_i^d in the dataset, V is the number of values that x_i can take, and $(\boldsymbol{X}_{-v_i^d}, \boldsymbol{Y}_{-v_i^d})$ is the dataset $(\boldsymbol{X}, \boldsymbol{Y})$ without the samples that $x_i = v_i^d$.

Example 4.7 **ID3 example**

In this example, we use again the "Play Tennis" dataset like in the NB example. Let's remember that:

$$p(C_1) = p(y = \text{Yes}) = \frac{9}{14} \approx 0.64$$

$$p(C_2) = p(y = \text{No}) = \frac{5}{14} \approx 0.36$$

In order to decide which feature to split along, we have to compute the information gains:

$$\begin{aligned} IG(\boldsymbol{X}, \boldsymbol{Y}, x_1) &= E(\boldsymbol{X}, \boldsymbol{Y}) - \sum_{d=1}^{V} \frac{N_{v_1^d}}{N_{\boldsymbol{X}}} E(\boldsymbol{X}_{-v_1^d}, \boldsymbol{Y}_{-v_1^d}) \\ &= E(\boldsymbol{X}, \boldsymbol{Y}) - \left(\frac{N_{x_1=\text{Sunny}}}{N} E(\boldsymbol{X}_{-x_1=\text{Sunny}}, \boldsymbol{Y}_{-x_1=\text{Sunny}}) \right. \\ &\quad + \frac{N_{x_1=\text{Overcast}}}{N} E(\boldsymbol{X}_{-x_1=\text{Overcast}}, \boldsymbol{Y}_{-x_1=\text{Overcast}}) \\ &\quad \left. + \frac{N_{x_1=\text{Rain}}}{N} E(\boldsymbol{X}_{-x_1=\text{Rain}}, \boldsymbol{Y}_{-x_1=\text{Rain}}) \right) \end{aligned}$$

By doing the calculations, we arrive at:

$$\begin{aligned} IG(\boldsymbol{X}, \boldsymbol{Y}, x_1) &= 0.074 \\ IG(\boldsymbol{X}, \boldsymbol{Y}, x_2) &= 0.0087 \\ IG(\boldsymbol{X}, \boldsymbol{Y}, x_3) &= 0.04565 \\ IG(\boldsymbol{X}, \boldsymbol{Y}, x_4) &= 0.0144 \end{aligned}$$

Comparing the information gains, it is clear that x_1 (named "Outlook," see Table 4.1) has the largest information gain and is selected as the root of the tree. After that, the dataset $(\boldsymbol{X}, \boldsymbol{Y})$ is split into three subsets[16], based on equality test

16. These are three because x_1 can take three values.

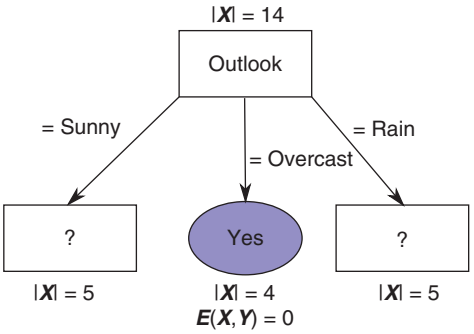

Figure 4.15 Decision tree for Play Tennis dataset at first stage with ID3.

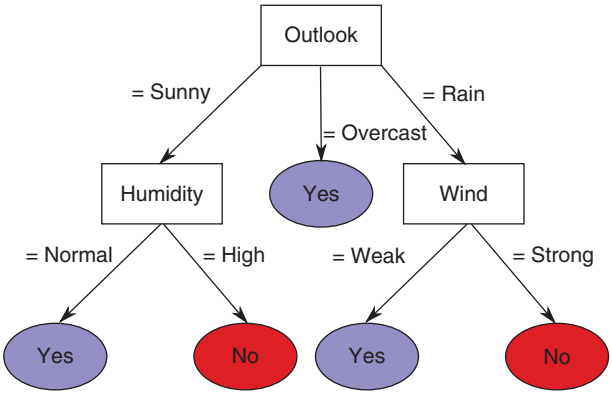

Figure 4.16 Final decision tree for the Play Tennis dataset with ID3.

in relation to the three possible values of "Outlook," which become the splitting predicates of the root branches (see Figure 4.15). The same process is followed for each of the two remaining internal nodes and their subsets and finally the tree of Figure 4.16 is produced.

We can notice that one of the features (x_2—"Temperature") is not necessary for constructing the "Play Tennis" classification (tree) model. Now, the model can be used for classifying a new example, not included in the training set. Let's assume that we want to classify the same point as we did in the NB example: $\boldsymbol{x}_+ = [\text{Sunny, Cool, High, Strong}]$. If we put it on the root of the tree and pass it through the tree, we end up classifying it in the second class (C_2—"No"). Thus, the conditions are not ideal to play tennis. We see that we arrived at the same conclusion of the NB classifier.

C4.5 Algorithm One of the problems of ID3 is that it creates problems in cases with variables that can take many values (e.g., "month_day" may take 30–31 values).

ID3 moves them to the root, because entropy gets low, hence information gain gets large, and the produced tree becomes very wide, which is not desirable. To remedy this, C4.5 [Quinlan 2014] introduces a different splitting criterion, called the *Gain Ratio*, which is defined as follows:

$$GR(\boldsymbol{X}, \boldsymbol{Y}, x_i) = \frac{IG(\boldsymbol{X}, \boldsymbol{Y}, x_i)}{\text{SplitInfo}(\boldsymbol{X}, \boldsymbol{Y}, x_i)}$$

$$\text{SplitInfo}(\boldsymbol{X}, \boldsymbol{Y}, x_i) = -\sum_{d=1}^{V} \frac{N_{v_i^d}}{N_{\boldsymbol{X}}} \log\left(\frac{N_{v_i^d}}{N_{\boldsymbol{X}}}\right) \tag{4.43}$$

where $N_{v_i^d}$ is the number of times that x_i takes the value v_i^d in the dataset $(\boldsymbol{X}, \boldsymbol{Y})$, V is the number of values that x_i can take, and $N_{\boldsymbol{X}}$ is the number of samples in $\boldsymbol{X}$. The main idea is to penalize the big information gain that variables with many values have, and thus, try to balance between selecting features with big information gain and features with a smaller number of possible values.

Tree pruning In order to avoid overfitting and intractability, most DT approaches perform some kind of *tree pruning*. In essence, tree pruning reduces the size of DTs by removing sections of the tree that provide little power to classify instances [Quinlan 2014]. We briefly discuss two of them:

- *Reduced error pruning*: In this approach, the available data is separated into a *training set* and a *validation set*. A DT is produced based solely on the training set. We start removing nodes in a bottom-up manner. Every node is candidate for pruning. Pruning a node means pruning the subtree having it as a root and making it a leaf, labeled with the class of most of the examples associated with the node. A node is finally pruned if the resulted tree is better than previous one, in terms of accuracy (the number of correctly predicted examples) in the validation set. Pruning continuously until no better tree is produced.
- *Rule post-pruning*: In this approach, a DT is produced based solely on the training set. Then, it converts the tree into an equivalent set of production rules [Grosan and Abraham 2011], by creating one rule for each path from the root to a leaf. Afterwards, it prunes conditions of rules one-by-one making them more general, and evaluates the system, in terms of accuracy, after each condition removal. If a condition pruning results in better accuracy, it remains, otherwise it's put back. The process stops if no further improvement is achieved and the rules are ordered based on the estimated accuracy.

4.2.2.4 Neural Networks

In Section 4.2.1.2 we defined what feedforward NNs are and saw how to use them to perform regression. In this section, we will investigate how we can perform binary and multi-class classification with feedforward NNs.

To perform classification with NNs, we only need to change the loss function that we use to optimize the weights of the NN. This in turn changes the interpretation of output of the NN, and thus we would also need to adapt the activation function of the last layer.

More formally, let's assume a dataset of the form $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\} \rightarrow \boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N\}$, where N is the number of samples, $\boldsymbol{y}_i \in C_k$ for $k \in \{1 \ldots K\}$ and K is the number of classes. We further assume that we have a feedforward NN of M layers, like in Section 4.2.1.2. We will handle the binary and multi-class classification differently.

In both cases, when dealing with discrete input variables, we need to transform the inputs appropriately in order for our NN to learn effectively. In particular, we usually encode each discrete input variable as a one-hot vector of length V (where V is the number of values of the variable), that is, a vector with all zeroes except from the row of the value that is selected.

Binary classification As we saw already, in binary classification we assume the output variable $\boldsymbol{y} \in \{0, 1\}$ or $\boldsymbol{y} \in \{-1, 1\}$. In NNs, we usually take the first case and assume that the output of the NN is the probability of the input $\boldsymbol{x}$ being classified as $\boldsymbol{y} = 1$. For this reason, we usually only use one output neuron, and its activation function is the sigmoid because $\text{sigmoid}(x) \in [0, 1]$. For the loss function we usually use the *cross entropy loss*:

$$H(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{y}_i \log(p(\boldsymbol{y}_i)) + (1 - \boldsymbol{y}_i) \log(1 - p(\boldsymbol{y}_i)) \tag{4.44}$$

where we write the output of the NN $\hat{f}(\boldsymbol{x}|\theta)$ as $p(\boldsymbol{y}_i)$ to emphasize that the output is interpreted as a probability. So, this is it! Now you can use back-propagation to adapt the weights of the NN and perform binary classification with NNs. Once a new input, $\boldsymbol{x}_+$, is given we pass it through the network to get $\hat{f}(\boldsymbol{x}_+|\theta) = p(\boldsymbol{y}_+)$. If $p(\boldsymbol{y}_+) \geq 0.5$, $\boldsymbol{x}_+$ is classified as +1, and as 0 otherwise.

Example 4.8 **Neural network binary classification**

In this example, we use again the "Play Tennis" dataset (see Table 4.1) like in the previous examples. We remind that the dataset consists of 14 samples ($N = 14$), where both the input variables, $\boldsymbol{x}$, and the output variable, $\boldsymbol{y}$, are discrete. In particular:

Table 4.2 **Play tennis dataset transformed for NN classification**

#	Outlook	Temperature	Humidity	Wind	Play tennis
1	1 0 0	1 0 0	1 0	1 0	0
2	1 0 0	1 0 0	1 0	0 1	0
3	0 1 0	1 0 0	1 0	1 0	1
4	0 0 1	0 1 0	1 0	1 0	1
5	0 0 1	0 0 1	0 1	1 0	1
6	0 0 1	0 0 1	0 1	0 1	0
7	0 1 0	0 0 1	0 1	0 1	1
8	1 0 0	0 1 0	1 0	1 0	0
9	1 0 0	0 0 1	0 1	1 0	1
10	0 0 1	0 1 0	0 1	1 0	1
11	1 0 0	0 1 0	0 1	0 1	1
12	0 1 0	0 1 0	1 0	0 1	1
13	0 1 0	1 0 0	0 1	1 0	1
14	0 0 1	0 1 0	1 0	0 1	0

$x_1 = \{\text{Sunny, Overcast, Rain}\}$, $x_2 = \{\text{Hot, Mild, Cool}\}$, $x_3 = \{\text{High, Normal}\}$, $x_4 = \{\text{Weak, Strong}\}$, and $y = \{\text{Yes, No}\}$.

As discussed earlier, since we are performing binary classification, we use only one output neuron with a sigmoid activation function that represents the probability that the input is classified as "Yes." Thus, we set $\boldsymbol{y}_i = 1$ for every sample i that is classified as "Yes," and $\boldsymbol{y}_i = 0$ otherwise. Moreover, we need to transform the discrete input variables to one-hot vectors for our NN to learn effectively. In Table 4.2 we find the transformed dataset. It is easy to verify that the input to the NN is a 10-D binary vector. To learn with this dataset, we devise a very simple feedforward NN with one hidden layer with the ReLU activation function and two hidden neurons (we remind the reader that the activation function of the output layer is the sigmoid).

After training this network with this dataset for ∼1,000,000 iterations using simple gradient descent, we get the a model that has 100% accuracy on the training dataset. Now, if we want to query the network with a new sample, for example, $\boldsymbol{x}_+ = [\text{Sunny, Cool, High, Strong}]$, we need to first transform it to the proper format. The transformed input is $\boldsymbol{x}'_+ = [1, 0, 0, \quad 0, 0, 1, \quad 1, 0, \quad 0, 1]$. Passing this input through the network we get a value that is very close to zero, and thus, we classify it as "No" matching the result of the previous examples.

Multi-class classification In multi-class ($K > 2$) classification, we proceed within the same lines of thinking, and use one output neuron per class. The kth output

neuron will correspond to the probability that the input, $\boldsymbol{x}$, is classified in the kth class. Using this intuition, we devise a dataset so that $\boldsymbol{y}_i$ is a one-hot vector of length K, where it is all zeroes except from row of the class that this sample corresponds to. Since we are dealing with probabilities and we assume that each label can only correspond to one class, we want that $\sum_{k=1}^{K} p(\boldsymbol{y} = C_k|\boldsymbol{x}) = 1$. To ensure this, we usually use the *softmax* activation function in the last layer of the NN:

$$\text{softmax}(j) = \frac{e^{p(\boldsymbol{y}=C_j|\boldsymbol{x})}}{\sum_{k=1}^{K} e^{p(\boldsymbol{y}=C_k|\boldsymbol{x})}}, \quad \text{for } j = 1, \ldots, K \tag{4.45}$$

Practically, the input to the softmax activation function is the whole output layer, as a vector of length K, and the output is a vector of the same length with the values at each row, j, computed with Equation 4.45. Lastly, we need to adapt the cross-entropy loss function for the *multi-class* case:

$$H(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{y}_i \log(p(\boldsymbol{y}_i|\boldsymbol{x}_i)) \tag{4.46}$$

where $p(\boldsymbol{y}_i|\boldsymbol{x}_i)$ is the predicted probability that $\boldsymbol{x}_i$ belongs to the class $\boldsymbol{y}_i$. Using this machinery, enough data, and computation time, you should be able to achieve state-of-the-art multi-class classification results even on demanding datasets.

4.3 Unsupervised Learning: Clustering

In this section, we will discuss about two UL approaches and show how to use them to perform clustering. We remind the reader that we assume that we have access to a dataset of the form $\boldsymbol{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, and the goal is to learn a mapping $f : \boldsymbol{X} \rightarrow \boldsymbol{Y}$, where f can be seen as a function that assigns to each data point a cluster. Although, there exist ways to automatically discover the number of clusters (e.g., the Bayesian information criterion [BIC] [Neath and Cavanaugh 2012]), in this section we will assume the number of clusters K to be fixed.

4.3.1 K-means

K-means clustering [Lloyd 1982] is a method for clustering N observations into K clusters. The main objective of K-means is to assign each sample to the cluster with the nearest mean. This results in a partition of the space that is very similar to a Voronoi tessellation of the space [Aurenhammer 1991]. The overall problem is NP-hard, but we will see a heuristic but efficient algorithm that can converge very quickly to a local optimum.

K-means is usually implemented as an iterative algorithm that tries to make the inter-cluster data points as similar as possible while also keeping the clusters

as different as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the points that belong to that cluster) is minimized. By similar and different, we mean close and far respectively using an appropriate metric (e.g., Euclidean distance).

The K-means algorithm operates as follows:

1. Select the number of clusters K
2. Initialize the cluster centroids
3. Compute the distances between each data point and all centroids
4. Assign each point to the closest centroid
5. Compute the new centroids by averaging all the points that belong to them
6. Go back to 3, until there is no change in the centroids

This procedure is very close to the expectation–maximization (EM) algorithm [Dempster et al. 1977] that we will see in the next subsection. In practice, this procedure will converge to the closest local minimum, and thus we usually execute several K-means runs and take the best one, that is, the one that produces the smallest sum of distances.

In Figure 4.17, an example run of K-means is shown. Here $\boldsymbol{x} \in \mathbb{R}^2$ and we generate 1,300 random points in such a way that there exist three main clusters, but also considerable noise. We show the computed centroids with stars and color the points in each cluster with different colors. Of course, the goal is to cluster the data in $K = 3$ clusters. We see how K-means moves the centroids in such a way that in the end it discovers the underlying structure of the data.

4.3.2 Gaussian Mixture Models

A GMM is a probability distribution. Unlike unimodal distributions, like the Gaussian, that model a single peak, GMMs can model distributions with multiple peaks. In essence, a GMM is a sum of several Gaussians together. If we have a sufficient number of Gaussians, and adjust their parameters, almost any continuous density can be approximated.

Since a GMM is a sum of probability distributions, we can use it as a clustering algorithm by comparing the probabilities of a new data point belonging to each of the sub-distributions. More formally, if we use K Gaussians, a GMM is defined as:

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} \boldsymbol{\pi}_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{4.47}$$

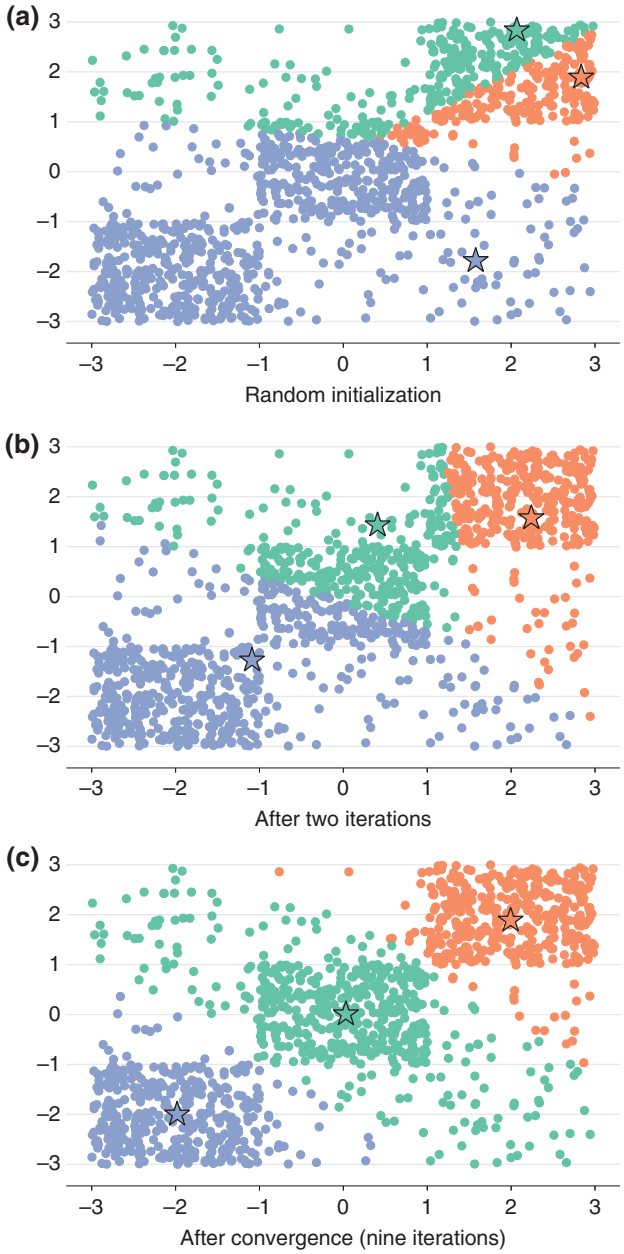

Figure 4.17 K-means example.

where $\boldsymbol{\pi}_k \in \mathbb{R}$, $\boldsymbol{\mu}, \boldsymbol{x} \in \mathbb{R}^D$, and $\boldsymbol{\Sigma} \in \mathbb{R}^{D \times D}$. As a quick reminder, the pdf of a Gaussian distribution is defined as (also defined in Equation 4.13):

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}\boldsymbol{\Sigma}^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right) \tag{4.48}$$

Assuming access to a dataset of the form $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ we perform again maximum likelihood optimization to find the best parameters $\boldsymbol{\pi}_k$, $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ for $k = 1, \ldots, K$. In this case, we will use the EM algorithm [Dempster et al. 1977] to do so. This is similar to the K-means procedure and is an iterative algorithm with two steps per iteration: the expectation (E) step and the maximization (M) step. The update of the GMM parameters from an E-step followed by an M-step is guaranteed to increase the log-likelihood function:

$$\log p(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{i=1}^{N} \log\left(\sum_{k=1}^{K} \boldsymbol{\pi}_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right) \tag{4.49}$$

The procedure of the EM algorithm for optimizing a GMM is a follows:

1. Initialize $\boldsymbol{\pi}_k$, $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$; usually K-means (Section 4.3.1) can give a good initial estimate
2. Perform the *E-step*: Evaluate the responsibilities using the current parameter values

$$\gamma(z_{ik}) = \frac{\boldsymbol{\pi}_k \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \boldsymbol{\pi}_j \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

3. Perform the *M-step*: Re-estimate the parameters using the new responsibilities

$$\boldsymbol{\mu}_k^{\text{new}} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(z_{ik}) \boldsymbol{x}_i$$

$$\boldsymbol{\Sigma}_k^{\text{new}} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(z_{ik})(\boldsymbol{x}_i - \boldsymbol{\mu}_k^{\text{new}})(\boldsymbol{x}_i - \boldsymbol{\mu}_k^{\text{new}})^T$$

$$\boldsymbol{\pi}_k^{\text{new}} = \frac{N_k}{N}$$

where $N_k = \sum_{i=1}^{N} \gamma(z_{ik})$

4. Evaluate the log-likelihood given by Equation 4.49 and check convergence
5. Go back to 2 until convergence

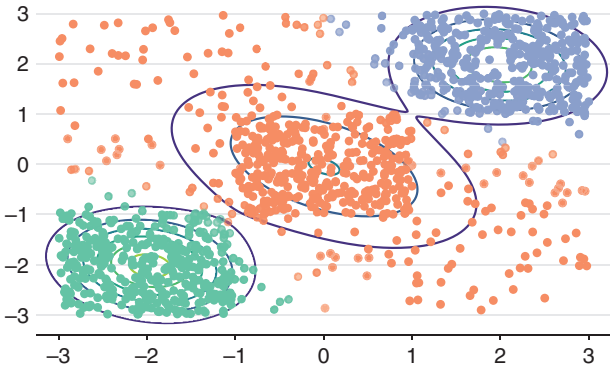

Figure 4.18 Gaussian mixture model example.

Once the GMM is computed, we have K clusters where the center of each cluster is $\boldsymbol{\mu}_k$ and each new data point $\boldsymbol{x}_+$ is associated to a cluster using the following rule:

$$\begin{aligned} C_{\boldsymbol{x}_+} &= \max_{k \in \{1 \ldots K\}} \gamma(z_{+k}) \\ &= \max_{k \in \{1 \ldots K\}} \frac{\boldsymbol{\pi}_k \mathcal{N}(\boldsymbol{x}_+ | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \boldsymbol{\pi}_j \mathcal{N}(\boldsymbol{x}_+ | \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \end{aligned} \tag{4.50}$$

In Figure 4.18, we see the result of training a GMM with three sub-models ($K = 3$) using EM on the same dataset as the one for K-means. We see that GMMs are also capable of discovering the underlying structure of the data, and place the means of the sub-models (Gaussians) in a similar way to the centroids of K-means. Moreover, GMMs gives us also the actual probability of a given input belonging to a certain cluster. In Figure 4.18 we set the α value of each point proportionally to the value of Equation 4.50. In essence, for points that are more transparent, we are less certain in which class they belong to. Essentially, GMMs let you determine the structure of the data (see the contour lines in Figure 4.18), without necessarily associating each sample with a cluster. One can think of GMMs as generalizing K-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

4.4 Practical Aspects

So far we have seen mostly theoretical aspects of several ML algorithms, but when applying these algorithms on real data a few considerations need to be taken into account. Overall, when applying ML algorithms for SL on real data, we usually follow the pipeline illustrated in Figure 4.19. We begin by formalizing the problem and performing some pre-processing on the data (Section 4.4.1). Then, we select

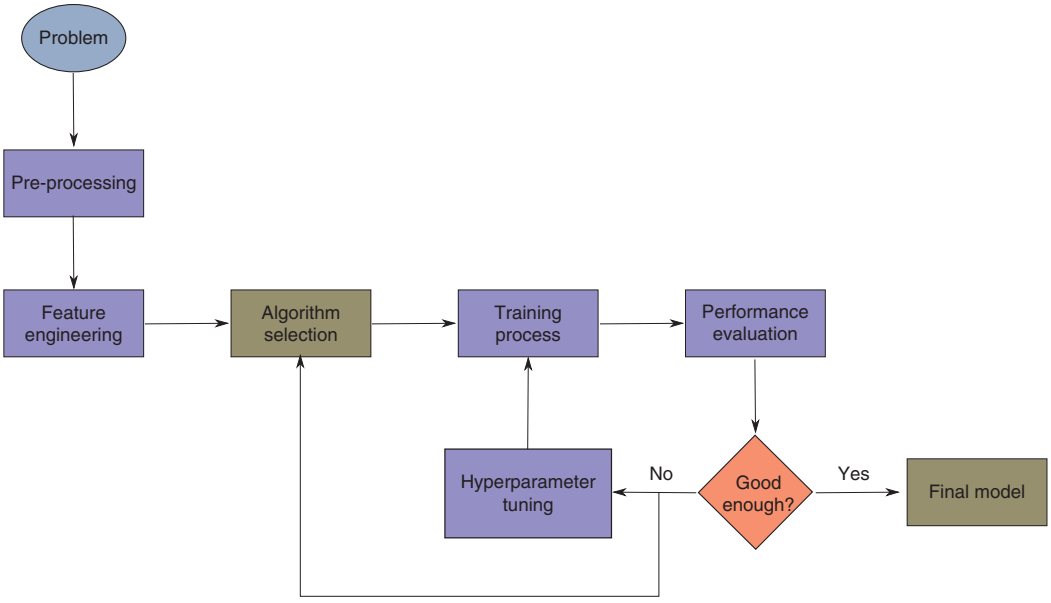

Figure 4.19 Machine learning pipeline for supervised learning.

or learn the features that are suitable to be used (Section 4.4.2) and we choose the learning algorithm. Then, we enter a loop where we train our algorithm, evaluate it (Section 4.4.4), and perform model selection (or hyperparameter optimization) (Section 4.4.5) in order to find the best hyperparameters (and/or algorithm) to perform the task at hand. Once we are satisfied with the result, we return the learned model.

4.4.1 Data Preprocessing

When gathering data from real processes, it may often not be in a suitable format for ML algorithms to operate on. Apart from re-structuring the data to make it in proper form (e.g., removing NULL values, filling missing values, etc.), some ML algorithms work better if the data is *normalized* or *standardized*. Let's assume access to a dataset $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, where N is the number of samples, and $\boldsymbol{x}_i \in \mathbb{R}^D$. We say that we normalize the dataset when:

$$\boldsymbol{x}_i = \frac{\boldsymbol{x}_i - \min(\boldsymbol{X})}{\max(\boldsymbol{X}) - \min(\boldsymbol{X})} \tag{4.51}$$

In essence, *normalization* scales all the points in the dataset to $[0, 1]$, while retaining their proportional range to each other. On the other hand, we say that we standardize the dataset when:

$$\begin{aligned} \boldsymbol{x}_i &= \frac{\boldsymbol{x}_i - \boldsymbol{\mu}_{\boldsymbol{X}}}{\boldsymbol{\sigma}_{\boldsymbol{X}}} \\ \boldsymbol{\mu}_{\boldsymbol{X}} &= \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i \\ \boldsymbol{\sigma}_{\boldsymbol{X}} \in \mathbb{R}^D, \text{ with } \boldsymbol{\sigma}_{\boldsymbol{X}}^j &= \frac{1}{N}\sum_{i=1}^{N} (\boldsymbol{x}_i^j - \boldsymbol{\mu}_{\boldsymbol{X}}^j)^2 \end{aligned} \tag{4.52}$$

where $\boldsymbol{x}_i^j$ is the jth element of the ith sample in the dataset. In essence, *standardization* transforms the data in order to have a mean of zero and standard deviation (SD) of 1. In other words, they will look like they were drawn from a standard normal distribution to the ML algorithm.

4.4.2 Feature Engineering

In ML, a feature is an individual measurable property or characteristic of a phenomenon being observed. Features are the most important ingredient of ML. In our formalization, features are the input variables to the algorithm, $\boldsymbol{x}_i$.

Feature engineering is the process of using domain and general knowledge to create features that will assist an ML algorithm. It is a mostly manual process and is one of the most important arts in ML. The need for manual feature engineering can be obviated by automated feature learning. Feature selection [Guyon and Elisseeff 2003] is the process of selecting a subset of the original feature set, that includes the most important and relevant features. It is performed before the learning method is executed (Figure 4.19).

To identify the most important and relevant features, UL algorithms are usually applied (Section 4.3). The most common and effective way of performing automated feature engineering is *dimensionality reduction*. A few of the most widely used algorithms for dimensionality reduction are principle component analysis (PCA) [Pearson 1901, Hotelling 1933], non-negative matrix factorization (NNMF) [Sra and Dhillon 2006, Tandon and Sra 2010], linear discriminant analysis (LDA), and NN autoencoders [Hinton and Salakhutdinov 2006].

4.4.3 Overfitting

In ML, parametric models, like for example, NNs (Section 4.2.1.2), can suffer from overfitting. Overfitting is a problem when the regression function fits the training data "too well," but does not generalize to unseen test data (see Figure 4.20). Overfitting typically occurs if the underlying model (or its parameterization) is overly flexible and expressive. Underfitting is when the regression function fits the training data very badly (see Figure 4.20). Underfitting usually occurs because

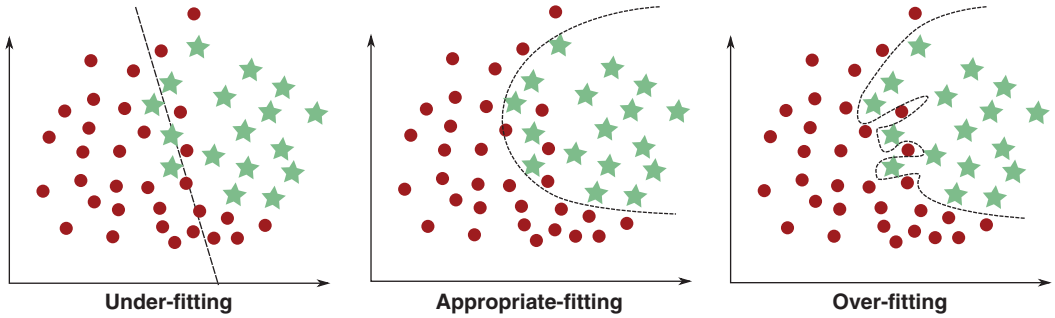

Figure 4.20 Over- and under-fitting in machine learning.

of premature convergence of the optimization process or because the underlying model is not expressive enough. In NN learning, we usually add a regularization term in the loss function [Girosi et al. 1995] or use a technique named dropout [Srivastava et al. 2014] to prevent overfitting, and choose the appropriate architecture and optimization hyperparameters to prevent underfitting.

On the contrary, non-parametric models, and especially the Bayesian approaches (e.g., Section 4.2.1.3), tend to be less prone to overfitting as their predictions do not focus on a single point estimate set of parameters, but average over all plausible sets of parameters.

4.4.4 Evaluation Metrics for Supervised Algorithms

Once we have trained our supervised ML algorithm, we can use a few metrics to evaluate its prediction capabilities. We devise different metrics for regression and classification tasks. Nevertheless, in both cases one of the most common approaches is to split the dataset into *training* and *test* datasets. Usually, 70% of the samples (randomly selected) are used for the training dataset and 30% for the test one. The idea here is to use the training dataset for training the model and use the test dataset for evaluating the learned model. In the following, we formulate the metrics using notation as if we were using the whole dataset, but usually they are applied on the test dataset.

4.4.4.1 Regression Algorithms

We remind to the reader that regression algorithms try to find the mapping $f : \boldsymbol{x} \rightarrow \boldsymbol{y}$ from the input variables $\boldsymbol{x} \in \mathbb{R}^D$ to the numerical or continuous output variables $\boldsymbol{y} \in \mathbb{R}^E$. Moreover, we assume access to a dataset of the form: $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, where N is the number of samples, $\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \varepsilon$, and ε is some noise.

Mean squared error Probably the most common metric for evaluating the performance of a regression algorithm is the MSE (Section 4.2.1.2):

$$\text{MSE} = \frac{\sum_{i=1}^{N}(\boldsymbol{y}_i - \hat{f}(\boldsymbol{x}_i))^2}{N} \tag{4.53}$$

where $\hat{f}$ is the prediction of our learned model.

Normalized mean squared error Another similar metric is the normalized mean squared error (NMSE):

$$\text{NMSE} = \frac{MSE}{\text{var}(\boldsymbol{Y})} = \frac{\frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{y}_i - \hat{f}(\boldsymbol{x}_i))^2}{\frac{1}{N-1}\sum_{i=1}^{N}(\boldsymbol{y}_i - \boldsymbol{\mu}_{\boldsymbol{Y}})^2} \tag{4.54}$$

where $\hat{f}$ is the prediction of our learned model, and $\boldsymbol{\mu}_{\boldsymbol{Y}}$ is the mean of the observed values.

Negative log prediction probability In cases where our model is probabilistic, that is, the output is not a single point estimate, but a distribution, a metric that we can use is the negative log prediction probability (NLPP):

$$\text{NLPP} = -\sum_{i=1}^{N} \log \tilde{f}(\boldsymbol{y}_i|\boldsymbol{x}_i) \tag{4.55}$$

where $\tilde{f}$ is the prediction distribution of our learned model. We interpret this as the probability that the prediction of $\boldsymbol{x}_i$ is $\boldsymbol{y}_i$. For example, if our model's output distribution is a Gaussian with diagonal covariance, that is, $\tilde{f}(\boldsymbol{y}_i|\boldsymbol{x}_i) = \mathcal{N}(\boldsymbol{\mu}(\boldsymbol{x}_i), \boldsymbol{\Sigma}(\boldsymbol{x}_i))$, then we can define the NLPP as follows:

$$\text{NLPP} = \sum_{i=1}^{N}(\boldsymbol{\mu}(\boldsymbol{x}_i) - \boldsymbol{y}_i)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_i)(\boldsymbol{\mu}(\boldsymbol{x}_i) - \boldsymbol{y}_i) + \log\det\boldsymbol{\Sigma}(\boldsymbol{x}_i) \tag{4.56}$$

4.4.4.2 Classification Algorithms

We remind the reader that classification algorithms try to find the mapping $f : \boldsymbol{x} \rightarrow \boldsymbol{y}$ from the input variables $\boldsymbol{x} \in \mathbb{R}^D$ to the discrete or categorical output variables $\boldsymbol{y}$. Moreover, we assume access to a dataset of the form: $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, where N is the number of samples, and $\boldsymbol{y}_i = f(\boldsymbol{x}_i)$.

Confusion matrix One of the most common ways to evaluate (but also visualize) the result of a learned classifier is the confusion matrix. The confusion matrix is a table where rows represent the predicted classes and the columns the actual

Table 4.3 **Example of a confusion matrix.**

Confusion matrix	Actual positive	Actual negative
Predicted positive	4	1
Predicted negative	2	3

classes. We fill each cell with the number of samples that have the classifier predict the row class, but the actual class is given by the column. For example, let's assume that we have dataset consisting of 10 samples and the possible classes are two: positive and negative. A possible confusion matrix is shown in Table 4.3.

From this table, we can see that four samples were correctly predicted as positives, whereas one of them was predicted to be positive when in reality it was negative. Similarly, we see that three samples were correctly predicted as negatives, whereas two of them were predicted to be negative when they actually were positive. We can also get some interesting values:

- True positive (TP): The number of samples that were correctly classified as positive,
- True negative (TN): The number of samples that were correctly classified as negative,
- False positive (FP): The number of samples that were wrongly classified as positive,
- False negative (FN): The number of samples that were wrongly classified as negative.

These values, as defined here, make sense only in binary classification, but there are ways to compute them for the multi-class case. One way of doing so, is to compute these metrics per class and treat the results each time as a binary classification process where the goal is to find whether a sample belongs to one class or not. Using these values allows us to define a few other metrics for evaluating classifiers:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.57}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{4.58}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{4.59}$$

$$\text{specificity} = \frac{TN}{TN + FP} \tag{4.60}$$

$$\text{F1-score} = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4.61}$$

4.4.5 Model Selection

Many ML algorithms have a few hyperparameters that the user needs to choose. Sometimes we can choose them empirically using some high-level intuitions about the task at hand or observing the behavior of the system. Nevertheless, a few more principled ways of selecting the hyperparameters exist. In most cases, we assume that the process of identifying the best hyperparameters is an optimization process where we define an evaluation metric (e.g., one of the many we introduced in Section 4.4.4) and choose a suitable optimizer. In some cases, like for example GPs (see Section 4.2.1.3), we can have an analytical formulation that we can optimize for the hyperparameters, but in the general case this is not possible and thus we need to find alternative optimizers.

4.4.5.1 Grid Search

When the hyperparameters are discrete or low-dimensional (i.e., less than 5–6D), then performing a grid search over the hyperparameter space usually yields the best results. In essence, we extensively evaluate the whole hyperparameter space and choose the parameters that perform the best according to our metric. When the hyperparameters are continuous, we have to discretize them in order to be able to use a grid search. The main disadvantages of a grid search for hyperparameter optimization are: (a) it cannot scale to high-dimensional continuous spaces, and (b) many training processes must be performed as there is not an intelligent way to identify when to stop.

4.4.5.2 Bayesian Optimization

Another popular approach for performing hyperparameter optimization (also referred to as automated ML) is Bayesian optimization (BO) [Brochu et al. 2010]. BO is a model-based, black-box optimization algorithm that is tailored for very expensive objective functions. Like all model-based optimization algorithms, BO creates a model of the objective function with a regression method (usually GPs), uses this model to select the next point to acquire, then updates the model, and so on. The main disadvantage of BO is that it cannot scale to high-dimensional continuous spaces, but it has been shown that when combined with intelligent feature selection it can work very well in practice [Feurer et al. 2015].

4.5 Summary and Links

This chapter provided a brief overview of the most important ML algorithms. We attempted to provide a wide spectrum of ML algorithms, but also give as much as possible the mathematical foundations of each approach. We began with a brief summary of probability theory that is important for understanding many of the concepts in ML. We discussed several SL algorithms: both for regression and classification. We also highlighted that a few approaches (like NNs, GPs, and SVMs) can be used both for classification and regression tasks. We also analyzed two main algorithms for clustering, K-means and GMMs, which are able to automatically discover the underlying structure of the data. Lastly, we gave a brief overview of what a real data ML pipeline would look like.

We believe that this chapter can serve both as an introduction to ML, but also as a reference, as we tried to have an educational perspective as well as provide all the mathematical foundations and formulas of the algorithms.

The remainder of the book contains case studies (CSs) that use different styles of ML: CS1 (Chapter 5) looks at predicting hotel occupancy in a European city, CS2 (Chapter 6) classifies signals to support brain–computer interaction, CS4 (Chapter 8) uses GMMs to recognize gestures, CS5 (Chapter 9) to model personal contexts and includes a detailed discussion of understanding the link from sensing to matching, and CS7 (Chapter 11) reports experiments using different ML models for detecting driver cognitive load. CS3 (Chapter 7) and CS6 (Chapter 10) use different sequencing models for touchscreen text entry and general adaptive touch interfaces.

4.6 Follow-up Questions

Although we have provided a wide range of ML models/algorithms and their formulations, there exists no better way to learn than implementing the algorithms/models yourself and experimenting with data. Here are a few questions/tasks to guide your self-practice:

1. Try to implement a simple library in the language of your choice for feedforward NNs. Make sure that you have implemented back-propagation correctly. How important is the optimization procedure? Try to compare between vanilla gradient descent and more sophisticated schemes (e.g., Adam).
2. Can you create examples of ML models that under- or over-fit some data? What are the factors that make this happen?
3. Can you compare the performance of different type of classification models? For example, NNs and SVMs? Start from a binary classification example

and move to multi-class classification problems. How can we use SVMs for multi-class classification?

4. Can you make a feedforward NN model output a Gaussian distribution per query point instead of a single value? How does it compare to Bayesian methods (e.g., GPs)?
5. In Example 4.8 we used a very simple NN architecture. Can you use either grid search or BO to identify the optimal NN structure? How does it compare with our original design? Why is this?
6. Can you identify the importance of data pre-processing? Try inserting in the data some outlier values (very big positive and/or negative data). Is normalization or standardization more effective? Is data pre-processing necessary for Bayesian methods (e.g., GPs)?

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. DOI: https://www.tensorflow.org/. Software available from tensorflow.org.

F. Aurenhammer. 1991. Voronoi diagrams – A survey of a fundamental geometric data structure. *ACM Comput. Surv.* 23, 3, 345–405. DOI: https://doi.org/10.1145/116873.116880.

B. E. Boser, I. M. Guyon, and V. N. Vapnik. 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. ACM, 144–152. DOI: https://doi.org/10.1145/130385.130401.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA, 432.

E. Brochu, V. M. Cora, and N. De Freitas. 2010. *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*. arXiv preprint arXiv:1012.2599.

N. Cristianini and J. Shawe-Taylor. 2000. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press. DOI: https://doi.org/10.1017/CBO9780511801389.

A. Cully, K. Chatzilygeroudis, F. Allocati, and J.-B. Mouret. 2018. Limbo: A flexible high-performance library for Gaussian processes modeling and data-efficient optimization. *J. Open Source Softw.* 3, 26, 545. DOI: https://doi.org/10.21105/joss.00545.

M. P. Deisenroth, A. A. Faisal, and C. Soon Ong. 2019. *Mathematics for Machine Learning*. DOI: https://mml-book.github.io/.

A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *J. R. Stat. Soc. Series B Methodol.* 39, 1, 1–22. DOI: https://doi.org/10.1111/j.2517-6161.1977.tb01600.x.

M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. 2015. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, 2962–2970.

F. Girosi, M. Jones, and T. Poggio. 1995. Regularization theory and neural networks architectures. *Neural Comput.* 7, 2, 219–269. DOI: https://doi.org/10.1162/neco.1995.7.2.219.

C. M. Grinstead and J. L. Snell. 2012. *Introduction to Probability*. American Mathematical Society.

C. Grosan and A. Abraham. 2011. Rule-based expert systems. In *Intelligent Systems*. Springer, 149–185. DOI: https://doi.org/10.1007/978-3-642-21004-4_7.

I. Guyon and A. Elisseeff. Mar. 2003. An introduction to variable and feature selection. *J. Mach. Learn. Res.* 3, 1157–1182.

I. Guyon, B. Boser, and V. Vapnik. 1993. Automatic capacity tuning of very large VC-dimension classifiers. In *Advances in Neural Information Processing Systems*, 147–155. DOI: https://doi.org/10.1162/153244303322753616.

G. E. Hinton and R. R. Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786, 504–507. DOI: https://doi.org/10.1126/science.1127647.

H. Hotelling. 1933. Analysis of a complex of statistical variables into principal components. *J. Educ. Psychol.* 24, 6, 417. DOI: https://doi.org/10.1037/h0071325.

C.-W. Hsu, C.-C. Chang, and C.-J. Lin. 2003. *A Practical Guide to Support Vector Classification*.

D. P. Kingma and J. Ba. 2014. *Adam: A Method for Stochastic Optimization*. arXiv preprint arXiv:1412.6980.

Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep learning. *Nature* 521, 7553, 436–444. DOI: https://doi.org/10.1038/nature14539.

D. C. Liu and J. Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Math. Program.* 45, 1–3, 503–528. DOI: https://doi.org/10.1007/bf01589116.

S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inform. Theor.* 28, 2, 129–137. DOI: https://doi.org/10.1109/tit.1982.1056489.

M. E. Maron. 1961. Automatic indexing: An experimental inquiry. *J. ACM* 8, 3, 404–417. DOI: https://doi.org/10.1145/321075.321084.

R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. 2013. *Machine Learning: An Artificial Intelligence Approach*. Springer Science & Business Media. DOI: https://doi.org/10.1007/978-3-662-12405-5.

T. Mitchell. 1997. *Machine Learning*. McGraw-Hill Higher Education, New York.

A. A. Neath and J. E. Cavanaugh. Mar. 2012. The Bayesian information criterion: Background, derivation, and applications. *WIREs Comput. Stat.* 4, 2, 199–203. ISSN: 1939-5108. DOI: https://doi.org/10.1002/wics.199.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. 2017. Automatic differentiation in pytorch. In *NIPS-W*.

K. Pearson. 1901. On lines and planes of closest fit to systems of points in space. *Lond. Edinb. Dublin Philos. Mag. J. Sci.* 2, 11, 559–572. DOI: https://doi.org/10.1080/14786440109462720.

J. R. Quinlan. 1986. Induction of decision trees. *Mach. Learn.* 1, 1, 81–106. DOI: https://doi.org/10.1007/bf00116251.

J. R. Quinlan. 1987. Simplifying decision trees. *Int. J. Man Mach. Stud.* 27, 3, 221–234. DOI: https://doi.org/10.1016/s0020-7373(87)80053-6.

J. R. Quinlan. 2014. *C4.5: Programs for Machine Learning*. Elsevier.

C. E. Rasmussen and C. K. Williams. 2006. *Gaussian Processes for Machine Learning, Vol. 1*. MIT Press, Cambridge.

L. Rokach and O. Maimon. 2005. Decision trees. In *Data Mining and Knowledge Discovery Handbook*. Springer, 165–192. DOI: https://doi.org/10.1007/0-387-25465-X_9.

F. Rosenblatt. 1961. *Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms*. Technical Report, Cornell Aeronautical Lab Inc, Buffalo, NY.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. 1986. Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 1, 316–362.

S. J. Russell and P. Norvig. 2016. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, Malaysia.

A. L. Samuel. Jul. 1959. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.* 3, 3, 210–229. ISSN: 0018-8646. DOI: http://dx.doi.org/10.1147/rd.33.0210.

J. Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Netw.* 61, 85–117. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

S. Sra and I. S. Dhillon. 2006. Generalized nonnegative matrix approximations with Bregman divergences. In *Advances in Neural Information Processing Systems*. 283–290.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1, 1929–1958.

B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. Nov. 2017. *OSQP: An Operator Splitting Solver for Quadratic Programs*. ArXiv e-prints.

R. S. Sutton and A. G. Barto. 1998. *Reinforcement Learning: An Introduction, Vol. 1*. MIT Press, Cambridge.

R. Tandon and S. Sra. 2010. *Sparse Nonnegative Matrix Approximation: New Formulations and Algorithms*.

I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. 2016. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

M. D. Zeiler. 2012. *ADADELTA: An Adaptive Learning Rate Method*. arXiv preprint arXiv: 1212.5701.