Imperial College London

MATH60026/MATH70026

Methods for Data Science

Lecture notes

Spring Term 2025

Written by Prof Mauricio Barahona and Dr Barbara Bravi
Chapters 7, 8 based on material by Dr Kevin Webster
Editing contributions and integrations by Dr Hardik Rajpal, Dr Philipp Thomas, Dr Sahil Loomba, and Dr Florian Song

Contents

Introduction & Preliminaries

1. Introduction: what is Data Science?

Data science, also known as data-driven science, is an interdisciplinary field about scientific methods, processes, and systems to extract knowledge or insights from data in various forms, either structured or unstructured.[1]

Data science is a 'concept to unify statistics, data analysis and their related methods' in order to 'understand and analyse actual phenomena' with data. It employs techniques and theories drawn from many fields within the broad areas of mathematics, statistics, information science, and computer science, in particular from the subdomains of machine learning, classification, cluster analysis, data mining, databases, and visualisation.[2]

The ability to take data, to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it, is going to be a hugely important skill in the next decades, not only at the professional level but even at the educational level for elementary school kids, for high school kids, for college kids. Because now we really do have essentially free and ubiquitous data. So the complimentary scarce factor is the ability to understand that data and extract value from it.[3]

Figure 0.1. There's a lot of hype around Data Science.

[1] Source: `https://en.wikipedia.org/wiki/Data_science`
[2] See footnote 1.
[3] Source: 'Hal Varian on how the Web challenges managers', McKinsey & Company

There *is* something new in "data science". It's not a science. It's more of a process; a merging of firstly, skills in applied mathematics, computer science, statistics, visualisation and communication, with secondly, rich data sets and domain knowledge. In practice, it's usually done in teams. No one has deep knowledge of statistics, mathematics, computer science, visualisation together with detailed knowledge of the data source, industry as well as its interesting questions.

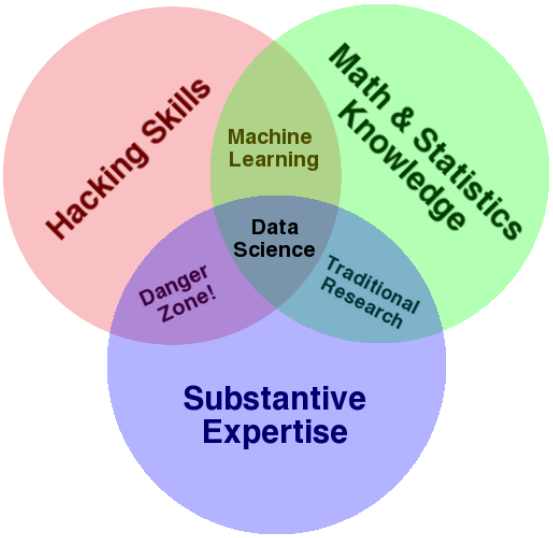


Figure 0.2. Venn diagram: Where does data science fit?

There's no academic field called 'data science'. After all, scientists have always used data. Statisticians theorise about data, and analyse it too. Very few academics are 'professors of data science'. Cathy O'Neil's book 'Doing Data Science' says that an academic data scientist might be defined as: 'a scientist, trained in anything from social science to biology, who works with large amounts of data, and must grapple with computational problems posed by the structure, size, messiness, and the complexity and nature of the data, while simultaneously solving a real-world problem.'[4]

There are a lot of jobs as data scientists. What do these people do? Fundamentally, a data scientist is someone who can extract meaning from data and communicate the results clearly. Data science at a company typically includes:

- data collection, storage and management
- who has access to what data
- how will the data be used, how does it add value
- privacy considerations
- analysis of data
- communicating the results

2. Case studies: successes of data science

Moneyball! Data in baseball outperforms standard predictors of who will be successful. There's a book, and a movie with Brad Pitt. Other big success areas are:

[4] O'Neil, Cathy, and Rachel Schutt. *Doing data science: Straight talk from the frontline.* 'O'Reilly Media, Inc.', 2013.

- Spam filters
- Fraud detection
- Election predictions (Nate Silver, `https://fivethirtyeight.com`)
- Predictions of crime hotspots
- Text analysis: twitter, blogs
- Targeted advertising
- Song, movie and product recommendations
- Forecasting (e.g. energy demand prediction, finance)
- Imputing missing data (e.g. netflix recommendations)
- Detecting anomalies (e.g. security, fraud, virus mutations)
- Classifying (e.g. credit risk assessment, cancer diagnosis)
- Ranking (e.g. Google search, personalization)
- Summarizing (e.g. news zeitgeist, social media sentiment)
- Decision making (e.g. AI, robotics, compiler tuning, trading)

Examples of specific questions that one could ask:

- Predict whether a heart attack patient will have a second heart attack, using demographic, diet and clinical data.
- Predict the price of a stock in 6 months, using company performance measures and economic data (but don't invest your money by your answers...).
- Identify the numbers in a handwritten ZIP code, from a digital image.
- Estimate the amount of glucose in the blood of a diabetic person from the infrared absorption spectrum of that persons blood.
- Identify the risk factors for prostate cancer, based on clinical and demographic data.
- Predict which flu strain will be successful next year based on its DNA sequence and relationships to other flu strains.

3. The process of data science

In most cases, doing data science will consist of the following pipeline:

- The science: what's a good question and what's the data set you will use to answer it?
- What is the input and what are you trying to determine?
- Data handling, cleaning and exploration: using computer programming and as well as knowing things
- The maths and the main analysis:
  - statistics, computing: what do you compute, why, and what does it mean?
  - machine learning: supervised, unsupervised..
  - networks: characteristics, analysis, comparisons
- Communication: words, visualisations, summaries to tell the story

In this course, you will be exposed mainly to the mathematical part of this workflow, although some information on the other aspects here will be mentioned in passing, too.

4. Aims and outcomes of this course

As mentioned above, this course's main aim is to equip you with the necessary skills to understand and work with the mathematical side of data science. To this end, you will learn about:

- The mathematicalconcepts underpinning methods in learning from data
- The process of conceptualisation of the analysis
- The process of explanation of the analysis
- The mathematical justification of the analysis
- Getting a good exposure to current methods and their use in practice in dealing with (realistic) data
- What the tools are, how they work mathematically, and how to analyse a data set and clearly communicate the results

5. Exploratory data analysis

In the pipeline described previously, one of the very first steps after determining the main question and data necessary to answer it, is conducting some *exploratory data analysis* (EDA). Whilst this course will not go in-depth on how to do this, we will give an overview of what it entails.

You will not have perfect data. The goal is to transform this into something tractable. This is why EDA is needed, it helps you to:

- Find mistakes! (negative heights? copies of columns? unreasonable values? missing values?)
- Get an idea of whether the signal you want is actually in your data (if so, where? Maybe your problem is simple, so just one predictor works. Find the direction and size of relationships between predictors and outcome.)
- Check the existence of statistical relationships between predictors
- Select your methods and metrics based on the type of task and analysis you want to carry out

Some data will be rubbish. EDA is important to sift out the good from the bad!

6. Some preliminary definitions

We now give some definitions of terms used throughout the course.

6.1. The different types of data. Data in its very simplest form usually consists of a big spreadsheet, a table, or a data frame full of numbers and categories. Typically, there is one row per observation, and one column per variable, i.e. all the predictors and the outcome(s). Reading 37 columns and 117,000 rows of a spreadsheet is not helpful. We need to summarise and visualise the data set in lots of ways to explore it, choose our method, find mistakes, etc. Broadly speaking, there are two classes of data, that you will find:

6.1.1. *Categorical data.* A *categorical* (or *discrete*) variable takes on one of a limited number of values (usually fixed), such as classes, or memberships. This means that all observations in the data will fall into k classes, for example. In essence, this leads to a discrete space of values. Some examples include: team name, which county someone lives in, breed of a dog, modules (mathematics, physics, computer science). The distribution is really just the counts or frequencies of the different values. Predicting a categorical variable is called *classification*: we want to classify a new point into the right category.

6.1.2. *Quantitative data.* A *quantitative* (or *continuous*) variable is numerical, and represents a measurable quantity. This is usually defined on a continuous space. Some examples are: height, salary, stock price, profit, speed. In some cases, it's important that your data is ordered and can be ranked. This means that you have *ordinal* data, e.g. number of bedrooms, number of players, number of people in a city (nearly continuous for practical purposes). The distribution of quantitative data usually has a mean, variance, skewness etc. Predicting a quantitative variable is called *regression*.

6.1.3. *Other, more specific types of data.* Note that aside from the two main categories above, there exist some more data types that are less common.

For example, *time series* data, which can be seen as a special case of ordinal quantitative data, as is usually also numerical. However, in this case, the order usually follows real-life timestamps, where data is collected at regular (or irregular) intervals over some amount of time. Some examples are: average number of home sales for many years, stock prices.

Sometimes you will also encounter simple *text* data. In this case the data contains words, sentences and sometimes even whole books. Usually, the first step will be to convert this data into something more tangible for the computer, i.e. numbers. Examples can be found in modern-day translation (such as Google Translate) and more generally, the field of *Natural Language Processing*.

6.2. Predictors vs outcomes. The first step to exploring the data is to determine and declare which variables are *predictors* (sometimes also known as *descriptors*) and which are *outcomes*. These can sometimes also be called *input/output*, respectively. The game of this course is to establish relationships between these two spaces: be aware that this is a decision that you make.

6.2.1. *Predictor variables.* Examples of predictor variables are:

- numbers
- continuous measurements like height, price, profit, concentration
- discrete measurements (integers): number of people, counts
- text: tweets, emails, patient records
- images: handwriting, medical images, photos
- even DNA sequences..
- combinations of these things

Be aware that randomness happens! It happens both in the process of getting the data, and in the form of noise in the actual observations. Simply declaring certain variables of data as predictors should not rule out the possibility that they are noisy, much like the outcome variables.

6.2.2. *Outcome variables.* This is the piece of the data set that we will aim to model and predict. What might your outcome (or "label") variables look like?

- Continuous data: in this case the problem is *regression*.

- Categorical data: in this case the problem is *classification*.

6.3. Learning from data. Inputs belong to an input space: $x^{(i)} \in X$. For example, if you have p different continuous features (height, weight, grade on exam, peak exhaled air flow, blood hemoglobin...) for each data point, then $x^{(i)} \in \mathbb{R}^p$. In this case, our data is *multivariate*. $x^{(i)}$ could also contain some other data that are not real numbers (i.e., categorical): county of residence, gender, company name...

Outputs $y^{(i)}$ as well can be continuous or categorical: some number or feature or group, some knowledge about the data point. Fundamentally, supervised machine learning is about finding ways to link the predictors $x^{(i)}$ to the outcomes $y^{(i)}$ so that we can make new predictions, i.e. generalise to unseen data. Indeed, there are two big, main classes of problems/algorithms:

6.3.1. *Supervised learning.* This is the case where there is a known outcome variable: the truth. For example:

- emails where we we know which ones are spam
- patients for whom we know their eventual outcome (heart attack or not)
- patients where we know their glucose level
- stocks and their prices 6 months later
- tweets and their author (election team or Donald himself)

We want to be able to predict that outcome for new observations of the input (predictor) variables.

The first part of this course will exclusively consist of methods that perform supervised learning. The latter few chapters will deal with it's counterpart:

6.3.2. *Unsupervised learning.* In this case, there is not a known outcome variable that we want to predict. Instead, we want to understand how the data are intrinsically organised, clustered, related. This is something that we want to extract from the data. For example:

- use diverse measurements (blood, tumour shapes, gene expression) to identify similar types of breast cancers
- explore differences in the bacterial composition inside the guts of healthy vs unhealthy individuals
- gain intuition for very high-dimensional data; make it more tractable

Supervised learning

Linear regression

1. Basics: setting up the problem

Linear regression is a form of supervised learning, which we have already introduced in the previous chapter. Here, we give a slightly more formal description of the process that supervised learning entails. We start with a brief summary of the typical workflow starting with some data.

The first step, as we said, is EDA (see Chapter 0). Then we need to decide on the task as well as the variables to use in the analysis as predictors and outcomes. Formally, this will lead to a declaration of the *predictor space* X and the *outcome space* Y. Both can be either continuous (i.e. quantitative) or discrete (e.g., categorical). Then we define the samples that we use for the *training* task, i.e. $x^{(i)} \in X$, as well as $y^{(i)} \in Y$ (the *training set*). At this point, we will also have to keep some samples aside for *validation*, but this will be explained in depth at a later point.

For a given task and training set, the goal is to find a function $y = f(x)$ relating *predictor* variables x to *outcome* variable y by defining a *loss function* $L\big(f(x), y\big)$. The loss function can sometimes be thought of as a function that computes the error between the model's estimates and the ground truth: f and L are thus intimately coupled. The data of *training set* is a set of N input-output pairs, or data points:

$$x^{(i)}, y^{(i)}, \quad i = 1, \ldots, N$$

Note that neither $x^{(i)}$ nor $y^{(i)}$ have to be uni-variate, as they can both have multiple features. The goal is to be able to predict y^{in} from a new observation x^{in}. Somehow we should be able to take uncertainty into account.

Schematically, the basic process for supervised learning is as following:

(1) Start with data $(x^{(i)}, y^{(i)}), \quad i = 1, \ldots, N$.

(2) Decide: Classification or regression?

(3) Define a *loss function* $L(y, f(x))$

(4) Minimise the *mean sample loss*: $E(L) = \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, f(x^{(i)}))$

(5) Try to also achieve a reasonable *expected test loss*:

$$E\big(L(y^{\text{in}}, f(x^{\text{in}}))\big)$$

In regression, the mean sample loss is typically the mean squared error:

$$L_{MSE}\big(y, f(x)\big) = \frac{1}{N}\sum_{i=1}^{N}\big(y^{(i)} - f(x^{(i)})\big)^2 \tag{1.1}$$

In general, the solution for f will be the result of an *optimisation*, minimising the in-sample loss (error) L. That is, we are aiming to minimise the following:

$$\mathbb{E}\Big[L\big(f(\{x^{(i)}\}), \{y^{(i)}\}\big)\Big] \text{ for } \{(x^{(i)}, y^{(i)}\}_{i\in\text{training}}$$

However, we also want to avoid *over-fitting*, i.e. obtaining a model that fits the training data perfectly, but will not perform well on unseen data. This means that we need to ensure that the expected out-of-sample loss

$$\mathbb{E}\Big[L\big(f(\{x^{(k)}\}), \{y^{(k)}\}\big)\Big] \text{ for } \{(x^{(k)}, y^{(k)})\}_{k\in\text{test}}$$

is also small. This a very important property is also known as *generalisability* - we will discuss several strategies to avoid overfitting and improve generalisability in this course.

When the model has good generalisation power, we have that $f(x_{\text{unknown}})$ is be a good predictor for *totally unseen* y_{unknown} for which we have no ground truth.

2. The Solution of Linear Regression: the Least Squares method

In the case of linear regression, we assume that our observations follow a **linear** relationship with respect to the input variables. This assumption could be based on some prior knowledge about the data (e.g. based on a known model with a basis in physics, biology, economics, etc); it could follow from our exploratory data analysis; or it could be dictated by convenience.

For simplicity of notation, we will restrict ourselves to the case where the observable variable is univariate and real: $y^{(i)} \in \mathbb{R}$.

The data will look like:

$$\big\{x_1^{(i)}, x_2^{(i)}, \ldots, x_p^{(i)}, \quad y^{(i)}\big\}_{i=1}^{N} \tag{1.2}$$

We then write the following linear model for our observations:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p =: f_{\text{LR}}(\mathbf{x}, \boldsymbol{\beta}), \tag{1.3}$$

where the vector $\boldsymbol{\beta}$

$$\boldsymbol{\beta} = \underbrace{\begin{pmatrix}\beta_0\\ \vdots \\ \beta_p\end{pmatrix}}_{(p+1)\times 1} \in \mathbb{R}^{(p+1)}$$

contains the $(p+1)$ *parameters* of the model, which need to be estimated (*learnt*) from the training data.

For every data point, we will then have a predicted value

$$\hat{y}^{(i)} = f_{\text{LR}}(\mathbf{x}^{(i)}, \boldsymbol{\beta}) = \beta_0 + \beta_1 x_1^{(i)} + \cdots + \beta_p x_p^{(i)}, \qquad i = 1, \ldots, N.$$

Ideally, given the data (1.2), we would want to find the values of the parameters $\boldsymbol{\beta}$ such that the prediction $\hat{y}^{(i)}$ is equal to the observation $y^{(i)}$ for every single data point:

$$\left\{\begin{array}{l} \hat{y}^{(1)} = \beta_0 + \beta_1 x_1^{(1)} + \cdots + \beta_p x_p^{(1)} \stackrel{?}{=} y^{(1)} \\ \hat{y}^{(2)} = \beta_0 + \beta_1 x_1^{(2)} + \cdots + \beta_p x_p^{(2)} \stackrel{?}{=} y^{(2)} \\ \quad \vdots \\ \hat{y}^{(N)} = \beta_0 + \beta_1 x_1^{(N)} + \cdots + \beta_p x_p^{(N)} \stackrel{?}{=} y^{(N)} \end{array}\right\} \tag{1.4}$$

But the system of linear equations (1.4) in the parameters is usually over-determined ($N \gg (p+1)$, more data points, hence equations, than unknowns β_i), which means that unless we have a very special case where all the data points are collinear, there will *not* be one solution for $\beta_0, \ldots, \beta_p$ that satisfies each equation in the system with the $\stackrel{?}{=}$ in (1.4).

What to do? The solution comes from an associated problem, the *Least Squares* (LS) problem, which has many nice and interesting properties. Essentially, it gives a set of parameters such that our predictions are *close* to the observations in a precise sense.

To see how this problem is solved in generality, we rewrite (1.4) in matrix-vector form. Let us organise our given data into a matrix $\mathbf{X}$ containing the descriptor variables and a vector $\mathbf{y}$ containing the observed variables:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_p^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_p^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & \cdots & x_p^{(N)} \end{bmatrix}_{N \times (p+1)} \qquad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{bmatrix}_{N \times 1}$$

It is then clear that the $N \times 1$ vector of predicted values $\hat{\mathbf{y}}$ is given by:

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}.$$

Since we want the parameter values such that prediction and observations are close, we need to quantify the *error* of our model:

$$\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - \mathbf{X}\boldsymbol{\beta}.$$

The meaning of the error is clear in Figure 1.1. The vector $\mathbf{e}_{N\times 1}$ contains all the deviations between the predicted values and the observed outcomes.

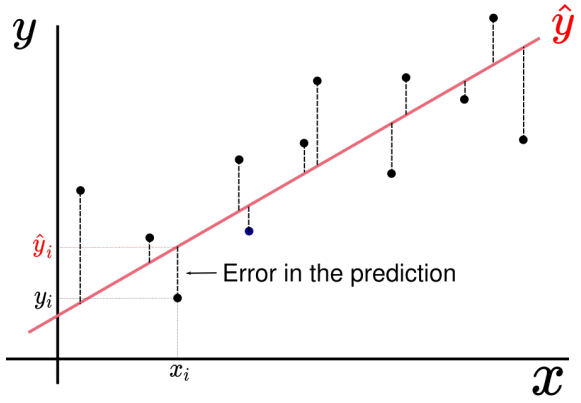


Figure 1.1. Simple linear regression task with one predictor and one outcome variable.

The LS method finds a solution for the values of $\boldsymbol{\beta}$ that minimises the *Mean Squared Error* (MSE) (1.1). In our matrix-vector notation, the MSE can be written compactly as:

$$L\big(f_{\mathrm{LR}}(\mathbf{x}), y\big) = \frac{1}{N}\mathbf{e}^T\mathbf{e} = \frac{\|\mathbf{e}\|^2}{N}.$$

Key message: The parameters $\boldsymbol{\beta}$ of the linear regression model are obtained by solving the **Least Squares optimisation problem**:

$$\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2$$

i.e., given $\mathbf{X}$ and $\mathbf{y}$, find $\boldsymbol{\beta}$ that minimises the MSE loss function:

$$L(\boldsymbol{\beta}) = \frac{1}{N}\|\mathbf{e}\|^2 = \frac{1}{N}\big[(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\big]. \tag{1.5}$$

Optimisations can be (and in most interesting cases are) very complicated. In fact, you will spend this course optimising difficult loss functions. But the LS optimisation for this linear case is, unsurprisingly, easy.

We will do two things about this easy optimisation: (1) in this section, we will first solve it explicitly, so that you gain some insight into notation that is used widely in the literature (and throughout the course); (2) in a later section, we will show how it can be solved numerically in a systematic manner.

2.1. Explicit, analytical minimisation: We have to minimise the loss function L (1.5) in the parameter space of the β_i's, i.e., the loss function is a multivariable real function of $\boldsymbol{\beta}$. To this end, we find the value $\boldsymbol{\beta}^\star$ where the gradient vanishes:

$$\left.\frac{dL}{d\boldsymbol{\beta}}\right|_{\boldsymbol{\beta}^\star} = \nabla_{\boldsymbol{\beta}} L\big|_{\boldsymbol{\beta}^\star} = \mathbf{0}$$

Expand the loss function:

$$L(\boldsymbol{\beta}) = \frac{1}{N}\big[\mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta}\big]$$

Quick aside: Check you understand these facts and notation as they are widely used in ML algorithms.

$$\nabla_{\boldsymbol{\beta}}(\boldsymbol{\alpha}^T\boldsymbol{\beta}) = \nabla_{\boldsymbol{\beta}}(\alpha_1\beta_1 + \ldots + \alpha_p\beta_p) = \begin{pmatrix}\alpha_1\\ \vdots \\ \alpha_p\end{pmatrix} = \boldsymbol{\alpha}$$

$$\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\boldsymbol{\alpha}) = \boldsymbol{\alpha}$$

$$\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta}) = \mathbf{A}\boldsymbol{\beta} + \mathbf{A}^T\boldsymbol{\beta} = (\mathbf{A} + \mathbf{A}^T)\boldsymbol{\beta}$$

Back to the loss function, it is then easy to check that:

$$\begin{aligned}\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) &= \frac{1}{N}\Big[-\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{y} + \big(\mathbf{X}^T\mathbf{X} + (\mathbf{X}^T\mathbf{X})^T\big)\boldsymbol{\beta}\Big]\\ &= -\frac{2}{N}\Big[\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{X}\boldsymbol{\beta}\Big]\end{aligned}$$

In order to achieve $\nabla_{\boldsymbol{\beta}} L|_{\boldsymbol{\beta}^\star} = \mathbf{0}$, we arrive at:

$$\mathbf{X}^T\mathbf{y} = (\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}^\star$$

This is called the *Normal Equation*, a name that becomes obvious when rewritten as:

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^\star) = 0,$$

i.e., the error is normal to $\mathbf{X}$. Given the definition of $\mathbf{X}$, we have that $\mathbf{X}^T\mathbf{X}$ is invertible. Therefore we can write explicitly:

$$\boldsymbol{\beta}^\star = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This is our LS solution.

To ensure that the solution $\boldsymbol{\beta}^\star$ is a minimum, we have to check that the Hessian matrix H evaluated at the optimum is positive definite, where the elements of the matrix are given by:

$$H(L)_{ij} = \frac{\partial^2 L}{\partial\beta_i\partial\beta_j}$$

Hence the Hessian matrix can be written compactly as:

$$H(L) = \nabla_{\boldsymbol{\beta}}(\nabla_{\boldsymbol{\beta}}L)$$

Remember that in the particular case of linear regression, we have:

$$L = \frac{1}{N}\left[(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right]$$

and:

$$\nabla_{\boldsymbol{\beta}}L = -\frac{2}{N}\Big[\mathbf{X}^T\mathbf{y} - (\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}\Big]$$

Using the aside from above:

$$H = \frac{2}{N}\nabla_{\boldsymbol{\beta}}\Big((\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}\Big)$$

Quick Aside: Once again, a little aside will help:

$$\nabla_{\boldsymbol{\beta}}\left(\overrightarrow{f}(\boldsymbol{\beta})\right) = \begin{pmatrix}\nabla_{\boldsymbol{\beta}}(f_1) & \dots & \nabla_{\boldsymbol{\beta}}(f_m)\end{pmatrix},$$

$$\text{where } \overrightarrow{f} = \begin{pmatrix} f_1 \\ \vdots \\ f_m \end{pmatrix}$$

$$\nabla_{\boldsymbol{\beta}}(\mathbf{A}\boldsymbol{\beta}) = \nabla_{\boldsymbol{\beta}}\begin{bmatrix}\mathbf{a}_1^T\boldsymbol{\beta} \\ \vdots \\ \mathbf{a}_m^T\boldsymbol{\beta}\end{bmatrix}$$

$$= \begin{bmatrix}\nabla_{\boldsymbol{\beta}}(\mathbf{a}_1^T\boldsymbol{\beta}) & \dots & \nabla_{\boldsymbol{\beta}}(\mathbf{a}_m^T\boldsymbol{\beta})\end{bmatrix}$$

$$\text{where } A = \begin{bmatrix}\mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_m^T\end{bmatrix}$$

Remember that from a previous aside:

$$\nabla_{\boldsymbol{\beta}}(\boldsymbol{\alpha}^T\boldsymbol{\beta}) = \boldsymbol{\alpha}$$

Therefore the above equation then becomes:

$$\nabla_{\boldsymbol{\beta}}(\mathbf{A}\boldsymbol{\beta}) = \begin{bmatrix}\mathbf{a}_1 & \dots & \mathbf{a}_m\end{bmatrix} = \mathbf{A}^T$$

Now back to the Hessian:

$$H = \frac{2}{N}\nabla_{\boldsymbol{\beta}}\Big((\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}\Big) = \frac{2}{N}(\mathbf{X}^T\mathbf{X})^T = \frac{2}{N}(\mathbf{X}^T\mathbf{X})$$

We have thus concluded that H does not depend on $\boldsymbol{\beta}$. Since $\mathbf{X}$ only contains data (i.e. it is constant), H also remains the same everywhere in parameter space and gives information on the local curvature of the function minimised. It is obvious that H is positive definite, which is defined to be the case iff $\mathbf{z}^T H\mathbf{z} > 0 \quad \forall \mathbf{z} \neq 0$. This works almost by inspection, but for completeness we show that $\mathbf{X}^T\mathbf{X}$ is positive definite:

$$\begin{aligned}\mathbf{z}^T\mathbf{X}^T\mathbf{X}\mathbf{z} &= (\mathbf{X}\mathbf{z})^T(\mathbf{X}\mathbf{z}) \\ &= \|\mathbf{X}\mathbf{z}\|^2 > 0 \text{ if } \mathbf{z} \neq 0.\end{aligned}$$

This shows that MSE is a convex quadratic function in the space of parameters. (Basically, the MSE is a multidimensional parabola.)

Key message: The solution to the Least-Squares (LS) problem is:

$$\boldsymbol{\beta}^\star = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{X}^+\mathbf{y},$$

where $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T =: \mathbf{X}^+$ is the *Moore-Penrose pseudo-inverse* of $\mathbf{X}$.

Our original problem had the following form:

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{y}$$

where $\mathbf{X}$ is not invertible (over-determined system). Hence we **cannot** 'solve' the problem by inverting the $\mathbf{X}$ on the left:

$$\text{“}\cancel{\boldsymbol{\beta} = \mathbf{X}^{-1}\mathbf{y}}\text{”}.$$

The LS solution applies a pseudo-inversion:

$$\boldsymbol{\beta}^\star = \mathbf{X}^+\mathbf{y}.$$

There exist many computational tools that can compute Moore-Penrose pseudo-inverse $\mathbf{X}^+$. The name pseudo-inverse is not accidental, since $\mathbf{X}^+$ has many of the properties of the inverse, if the inverse does not exist. Here are some properties of the pseudo-inverse with the equivalent properties of some invertible $\mathbf{A}$:

$$\mathbf{X}^+\mathbf{X}\mathbf{X}^+ = \mathbf{X}^+ \longleftrightarrow \mathbf{A}^{-1}\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}$$

$$\mathbf{X}\mathbf{X}^+\mathbf{X} = \mathbf{X} \longleftrightarrow \mathbf{A}\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}.$$

To conclude: in the case of a linear regression, we are able to find an exact, analytical solution by computing the Normal Equation. This is usually not possible with other models. In most cases, we need to use numerical optimisation instead to arrive at a solution, by looking (numerically) for a minimum of the loss function. This only works absolutely reliably if the function is convex. We will come back to this point in Section 4 when we cover briefly the computational aspects of solving the LS optimisation problem.

3. Statistical interpretation of the Least-Squares solution

The optimisation perspective that we introduced above also has an intuitive statistical interpretation. As a rule of thumb, **optimisation of convex quadratics is usually related to (multivariate) Gaussian distributions.** This is the case here. As we will show briefly in a few lines below, the above optimisation is equivalent to maximising the likelihood of a Gaussian linear model.

Without loss of generality, let us take $p = 1$ for simplicity. Then, given some data

$$(x^{(i)}, y^{(i)}),\ i = 1, \dots, N$$

we expect that our outcome will be a random variable that can be expressed as a linear combination of the input and some Gaussian *i.i.d.* noise:

$$y^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + \varepsilon^{(i)}$$

This noise will be a particular instance $\varepsilon^{(i)}$ drawn i.i.d. (*independent and identically distributed*) from a Gaussian distribution with zero mean and variance σ^2:

$$\varepsilon \sim \mathcal{N}(0, \sigma^2)$$

and the different $\varepsilon^{(i)}$ are all independent.

Going back to your knowledge of Statistics, for this we can write down a *likelihood function* which is given by:

$$\forall i, \quad \mathrm{Lik}(y^{(i)} \mid \boldsymbol{\beta}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \frac{-\big(y^{(i)} - (\beta_0 + \beta_1 x_1^{(i)})\big)^2}{2\sigma^2}$$

Then, using independence, we can compute the total likelihood for the whole data set:

$$\underset{\text{tot}}{\mathrm{Lik}}(\mathbf{y} \mid \boldsymbol{\beta}) = \prod_{i=1}^{N} \mathrm{Lik}(y^{(i)} \mid \boldsymbol{\beta})$$

From this statistical perspective, we are interested in finding the $\boldsymbol{\beta}$ that maximises the total likelihood of the model. We make our life a bit easier by maximising the logarithm of the likelihood (which is equivalent to the original problem, since the logarithm is monotonically increasing, and gives greater numerical stability). Therefore consider the *log-likelihood*:

$$\begin{aligned}
\mathcal{L}_{\text{tot}} = \log\Big(\underset{\text{tot}}{\mathrm{Lik}}\Big) &= \sum_{i=1}^{N} \log\big(\mathrm{Lik}(y^{(i)} \mid \boldsymbol{\beta})\big) \\
&= C - \frac{1}{2\sigma^2} \sum_{i=1}^{N} \big(\underbrace{y^{(i)} - (\beta_0 + \beta_1 x_1^{(i)})}_{e^{(i)}}\big)^2 \\
&= C - \frac{1}{2\sigma^2} \mathbf{e}^T \mathbf{e} \\
&= C - \frac{N}{2\sigma^2} L_{\text{MSE}},
\end{aligned}$$

where C is constant.

Key message: In statistical terms, minimising the loss function (as in the previous section) means maximising the likelihood:

$$\underbrace{-\frac{d\mathcal{L}_{\text{tot}}}{d\boldsymbol{\beta}}}_{\text{maximum likelihood}} \longleftrightarrow \underbrace{\frac{dL_{\text{MSE}}}{d\boldsymbol{\beta}}}_{\text{minimal loss (MSE)}}$$

having assumed that the variability across data points is Gaussian distributed.

4. Numerical Optimisation with Gradient Descent

In the previous sections, we optimised the loss (or the likelihood) of the linear regression problem by minimising

$$L(\boldsymbol{\beta}) = \frac{1}{N}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$$

over the space of $\boldsymbol{\beta}$. We already showed that this function (a quadratic) is convex and has a unique minimum, given by the Normal Equation, and obtained using the pseudo-inverse of $\mathbf{X}$.

$$\boldsymbol{\beta}^\star = \mathbf{X}^+\mathbf{y}.$$

This formal solution is straightforward and easily done on paper. But in some cases, $\mathbf{X}$ may become huge and inverting big matrices may become computationally infeasible.

More importantly, there are many other problems (i.e., most other models of any interest beyond linear regression), for which the minimum of the loss function **cannot** be found analytically. That is to say, one would not be able to solve $\nabla_{\boldsymbol{\beta}} L|_{\boldsymbol{\beta}^\star} = 0$ explicitly. In other words, we will not be able to solve for the equivalent of the normal equations. In such cases, the optimisation needs to be done numerically.

We will use the least-squares problem to illustrate these ideas (although here we do have the analytical solution available and we would not need to follow the purely numerical route). We have shown that finding the optimal linear model, i.e. minimising the sum of squared errors, is in reality the minimisation of a function that is quadratic in the parameters $\boldsymbol{\beta}$. How is this done numerically without solving the normal equations?

The general type of methods through which this is done is called *gradient methods*. (We covered this in Calculus in Year 1 and you will have seen some of those in Numerical Analysis, think also of Newton-Raphson from high school.)

> **Key message:** Gradient methods rely on the simple idea: the gradient of a multivariate function gives the direction of maximum variation of the function. Hence, **following the gradient along an optimisation trajectory leads to a maximum of the function** (minimising is easily done by changing signs.)

The above idea would seem to be the solution to any problem but, of course, things are not that simple. The above approach will always take us to a *local* maximum (or minimum) but once we are at that point, the gradient is zero and we stop moving. However, we will not know if the maximum we found is the *global* maximum of the function unless we have extra conditions. **Only for convex functions we have guarantees that there is one global maximum that can be reached with gradient methods.** The LS problem is one such case since we have a quadratic function with positive definite Hessian everywhere, as shown above and exploited below.

> *Quick Aside.* Most problems you will see in the course, and those that you will find in real applications, tend to be *non-convex*, hence 'hard' to optimise. To complicate matters (but to make things interesting), convexity is elusive and some problems can be convexified by changes of coordinates or relaxations, suddenly making them 'easy'. You can read more about these issues in the book *Convex Optimisation* by Stephen Boyd (see 'Additional Reading' folder).

Numerical optimisation of the Least Squares problem: The loss function of the least squares problem can be rewritten in the following form:

$$L(\boldsymbol{\beta}) = L(\boldsymbol{\beta}^\star) + \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)^T \underbrace{H}_{\frac{2}{N}(\mathbf{X}^T\mathbf{X})} (\boldsymbol{\beta} - \boldsymbol{\beta}^\star) \tag{1.6}$$

This follows immediately from the definition of the loss function by 'completing the square':

$$\begin{aligned} L(\boldsymbol{\beta}) &= \frac{1}{N}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \\ &= \frac{1}{N}\left((\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^\star) - \mathbf{X}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)\right)^T\left((\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^\star) - \mathbf{X}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)\right) \\ &= L(\boldsymbol{\beta}^\star) - \frac{2}{N}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^\star)^T\mathbf{X}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star) + \frac{1}{N}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)^T\mathbf{X}^T\mathbf{X}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star) \\ &= L(\boldsymbol{\beta}^\star) + \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)^T\left(\frac{2}{N}\mathbf{X}^T\mathbf{X}\right)(\boldsymbol{\beta} - \boldsymbol{\beta}^\star), \end{aligned}$$

where we have used the normal equation condition:

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^\star) = 0$$

Clearly, this function has positive curvature at all points:

$$\nabla_{\boldsymbol{\beta}}(\nabla_{\boldsymbol{\beta}} L) = \frac{2}{N}\mathbf{X}^T\mathbf{X} = H$$

since the Hessian is positive definite. Hence the loss function $L(\boldsymbol{\beta})$ is convex in the space of parameters. Making use of the fact that if a function is convex over the whole space, the local minimum is equivalent to the global minimum, any algorithm that converges to a minimum will lead to the global minimum for this loss function.

To get some intuition, see Figure 1.2, where we show a sketch of this loss function, with lines indicating the level sets

$$s_k = \{\boldsymbol{\beta} \mid L(\boldsymbol{\beta}) = k\},$$

i.e., the sets of points where the loss function has a constant value. In the particular case of the convex loss function defined above, these ellipses are concentric and as we decrease k, we move closer to the optimal point $\boldsymbol{\beta}^\star$.

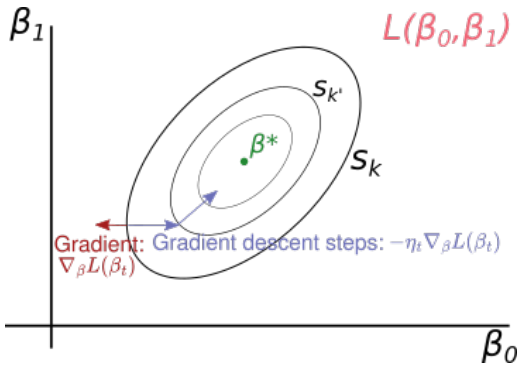


Figure 1.2. Finding $\boldsymbol{\beta}^\star$ is a convex optimisation problem in the case of linear regression. This sketch shows the optimal value of $\boldsymbol{\beta}^\star$ in green, as well as the level sets where the loss function is constant. The gradient of the loss at each point gives the direction of maximal positive variation, hence gradient descent, to find the minimum, follows the opposite of the gradient direction and makes a step size controlled by η_t at a certain iteration t.

Such a loss function can be globally minimised with gradient methods, which use the fact that $\nabla_{\boldsymbol{\beta}} L$ marks the direction of maximum change of L.

Key message: The algorithm for gradient-based optimisation of the LS problem is: iteratively follow the direction of $-\nabla_{\boldsymbol{\beta}} L$. The iteration $t+1$ will be of the form:

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \eta_t \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}_t)$$

where η_t is the step size, which can be adjusted at each step of the algorithm.

There are many algorithms that use the gradient, amongst which are: line search, back tracking or conjugate gradient. Each of these methods has different strategies (e.g., how to choose η_t) to speed up convergence to the minimum in a shorter number of steps, see again Stephen Boyd's book on *Convex optimization* if you are interested.

As an illustration of the importance of the non-convexity of the loss function, consider Figure 1.3. From the picture, it is clear that, if we use gradient methods naively, choosing an initial guess at random for the parameter set $\boldsymbol{\beta}$ will likely result in convergence to the green point $\boldsymbol{\beta}^\star$. However, in this case there is another (deeper) minimum (the global optimum) of the function that is more difficult to reach by using gradient-based trajectories in parameter space. Hence this problem becomes difficult for gradient-based methods to crack. In fact, most loss functions in more complicated models (beyond linear regression) are usually highly non-convex, with multiple local minima. We will therefore see other optimisation methods that attempt to alleviate this issue later in the course, but be warned that no perfect solution exist for this problem.

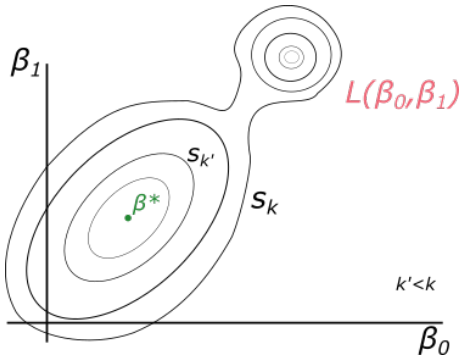


Figure 1.3. Cartoon of a non-convex loss function, with a second (deeper) well that contains the globally optimal solution that is hard to find using gradient descent methods.

5. Bias vs. variance

Here, we will describe what is sometimes known in Machine Learning as the *no-free-lunch* theorem, that is, the fact that an unbiased model will generally have high variance and, conversely, reducing the variance will increase the bias. Therefore, both aspects need to be balanced to achieve models whose predictions are as close as possible to the true value ('accurate') with as reduced variability as possible.

Quick Aside. Whilst this course will not go further into the statistical details of this problem, some pointers for interested students are as follows. For more in-depth derivations, see *Hastie, Tibshirani, Friedman, The Elements of statistical learning, chapter 3, sections 3.2, 3.3*; for the *no-free-lunch theorem*, see *Goodfellow, Bengio, Courville, Deep Learning*, chap. 5, section 5.2.1.

Here, we mention a few aspects related to this issue. In statistics, one calls an 'estimator' a rule to estimate a given quantity based on observed data. Let $\boldsymbol{\beta}$ be the true parameters

of the problem to be estimated and $\boldsymbol{\beta}^\star$ the LS estimate. First, we define two measures, the bias and the error covariance matrix of the estimator $\boldsymbol{\beta}^\star$:

$$\text{Bias: } \|\mathbb{E}[\boldsymbol{\beta}^\star] - \boldsymbol{\beta}\|$$
$$\text{Error Covariance Matrix: } \mathbb{E}\big[(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)^T\big]$$

Note that here $\mathbb{E}$ is meant over an ensemble of different data sets, each of them giving a different value of the loss and hence of $\boldsymbol{\beta}^\star$. These two measures, based on expectation values, allow one to assess the performance of a learning algorithm across many possible data sets.

Remember the model formulation for the linear regression:

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

We can then compute the expectation of our estimated parameters:

$$\begin{aligned}\mathbb{E}[\boldsymbol{\beta}^\star] &= \mathbb{E}[(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}] \\ &= \boldsymbol{\beta} + \mathbb{E}[(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{\varepsilon}] = \boldsymbol{\beta}\end{aligned}$$

This particular estimator ($\boldsymbol{\beta}^\star$) is *unbiased*, i.e. $\|\mathbb{E}[\boldsymbol{\beta}^\star] - \boldsymbol{\beta}\| = 0$. That is if we obtain an estimate for a new set of samples drawn from the same data source, then we will always expect to get the right $\boldsymbol{\beta}$ on average.

On average, we will get the correct estimate, but what about the expected error of our estimator? This can be estimated from the error covariance matrix (this is the matrix from which we can obtain the MSE of the *estimator* by taking its trace). It can be obtained easily from our expressions above as follows:

$$\mathbb{E}\big[(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)^T\big] = \mathbb{E}\big[(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\underbrace{\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^T}_{\mathbb{E}[\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^T]=\sigma^2\mathbf{I}}\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\big] = (\mathbf{X}^T\mathbf{X})^{-1}\sigma^2 \tag{1.7}$$

where $\mathbf{I}$ is the identity matrix. Note that σ^2 is the variance of the noise in the observations, which we can estimate from the data. In statistical terms, the estimator for σ^2 is:

$$\begin{aligned}\widehat{\sigma}^2 &= \frac{1}{N - (p+1)}\sum_{i=1}^{N}(y^{(i)} - \widehat{y}^{(i)})^2 \\ \mathbb{E}[\widehat{\sigma}^2] &= \sigma^2.\end{aligned}$$

where we use the symbol $\widehat{\cdot}$ to denote quantities estimated from data. The expected squared error of our estimated parameters depends on the empirical variance of the data *and* on the properties of $(\mathbf{X}^T\mathbf{X})^{-1}$. However, $(\mathbf{X}^T\mathbf{X})$ might be badly conditioned, hence non-invertible (it is close to losing full rank and has small singular values). **As a result, although LS is unbiased (good!), it can have high MSE in the estimated parameters (depending on properties of the data matrix X)**.

In general, for an estimator $\theta^\star$ of the true value θ, we always have that:

$$\begin{aligned}\mathbb{E}\big[(\theta - \theta^\star)^2\big] &= \mathbb{E}[\theta^2] + \mathbb{E}[\theta^{\star 2}] - 2\mathbb{E}[\theta\theta^\star] \\ &= \theta^2 + \mathrm{var}(\theta^\star) + \mathbb{E}(\theta^\star)^2 - 2\theta\mathbb{E}(\theta^\star) \\ &= \underbrace{\mathrm{var}(\theta^\star)}_{\substack{\text{variance of} \\ \text{the estimator}}} + \underbrace{\big[\theta - \mathbb{E}(\theta^\star)\big]^2}_{\text{Bias}^2}\end{aligned}$$

where we have used the fact that θ is the true value to be estimated, i.e. not a random variable.

Let $\boldsymbol{\beta}^\star_{\text{LS}}$ define the LS estimator such that $\hat{f}_{\text{LS}}(\mathbf{x}_0) = \mathbf{x}_0^T\boldsymbol{\beta}^\star_{\text{LS}}$, which is unbiased:

$$\mathbb{E}\big[\hat{f}(\mathbf{x}_0)\big] = \mathbf{x}_0^T\boldsymbol{\beta}$$

where $\boldsymbol{\beta}$ is the true value. Then we have:

Key message:

Expected squared error $= \mathbb{E}[(\hat{f} - y)^2] = \text{var}(\hat{f}) + \big(y - \mathbb{E}[\hat{f}]\big)^2 = \text{Variance} + \text{Bias}^2$.

The main issue is that by minimising the expected squared error we could face a compromise between variance and bias (bias-variance trade-off).

It is probably good to end with a visual representation of this discussion. Effectively, how good an estimator is depends on a combination of low bias and low variance. You can see an illustration of this idea in Figure 1.4, and how in some cases we might be better off with an estimator that has some bias (i.e., it is a bit *inaccurate*) but has reduced variability (i.e., it is more *reliable*).

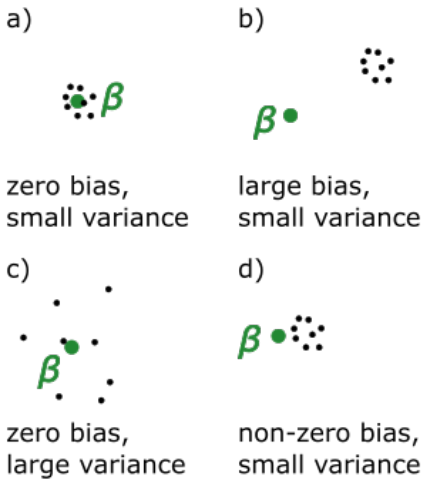


Figure 1.4. Trade-off between bias and variance. Each black dot represents an estimate $\boldsymbol{\beta}^\star$ of the true $\boldsymbol{\beta}$ (in green). In case a), we have an almost ideal case, with zero bias and small variance, which is rarely achievable in reality. Case b) shows the worse case scenario: having large bias and low variance will lead to a badly fit model. Case c) will be a somewhat good model, but with a large variance. Note that the least squares method has this issue. Finally, case d) shows the other side of the trade-off, where a small variance was achieved by introducing some small bias. In reality, one must often choose between c) and d).

Quick Aside. One could think that perhaps we could do better than LS. However, there is a general result (that follows from the Gauss-Markov theorem) that tells us that if we restrict ourselves to a *linear unbiased* estimator, we cannot do better than LS. We only sketch briefly the arguments here (see *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*, Chap. 3, for more details).

Let $\hat{\hat{f}}$ be another unbiased linear estimator different to LS. Recall that, according to the Gauss-Markov theorem, the least squares estimator is the best unbiased linear estimator (see *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*,

Chap. 3, section 3.2.2 for more details). Hence we know that:

$$\text{var}\left(\hat{f}_{\text{LS}}(\mathbf{x}_0)\right) \leq \text{var}\left(\hat{\tilde{f}}(\mathbf{x}_0)\right)$$

Since we have

$$\text{Expected squared error} = \mathbb{E}[(\hat{f} - y)^2] = \text{var}(\hat{f}) + \left(y - \mathbb{E}[\hat{f}]\right)^2$$

then it follows directly that LS has the lowest error of all linear unbiased estimators. Thus, if we want to stick with a linear model and we want it to be unbiased, we cannot do better than least-squares. On the other hand, as we saw above in (1.7), LS can have large variance.

6. Methods to reduce the variance of an estimator

Remaining within unbiased linear methods, no method can do better than LS, and the only way of improving the LS solution has to do with $\mathbf{X}^T\mathbf{X}$ (see (1.7)) to reduce directly the variance. The alternative is to go for methods that are no longer unbiased but might have lower variance (see Figure 1.4).

A second consideration here is that **increasing the interpretability of the models can be achieved through *sparse models*, which reduce the number of predictors** p by judiciously eliminating redundant descriptors.

From (1.7), it is clear that one of the sources of large variance is the inverse $(\mathbf{X}^T\mathbf{X})^{-1}$. This inverse can induce large values when $\mathbf{X}$ has a high *condition number*, i.e, when it is close to losing its full rank. In that case, $\|(\mathbf{X}^T\mathbf{X})^{-1}\| \gg 1$, and the variance (1.7) is large in particular directions of the associated vector spaces. (Of course, in the extreme case where some columns are linear combinations of others, then $\mathbf{X}$ does not have full rank and $(\mathbf{X}^T\mathbf{X})$ is not invertible. Another way of saying this is that its condition number is infinite.)

If two of the columns of $\mathbf{X}$ (i.e., the predictor variables), are nearly collinear then the condition number is large. This means that those two predictor variables are linearly related, i.e., one of them is 'redundant'. One solution is to eliminate predictor variables that are redundant, i.e., reduce the number of columns by eliminating such variables.

There is several ways of performing optimal subset selection, the two main ones being: 1. brute-force complete enumeration i.e., find the subset of k best parameters out of the p parameters by complete enumeration and benchmark of all possible choices of k out of p features; 2. Greedy sequential selection, i.e. following a sequential approach of adding/reducing descriptors until a criterion is met. The algorithms for sequential subset selection are beyond the scope of our course, we refer to *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*, Chap. 3, section 3.3 if you are interested.

In any case, selection of descriptors is a 'combinatorial optimisation problem'. Such problems tend to be 'hard' and, in many cases, can only be solved by complete enumeration, which can only be applied to small problems. For an alternative approach (very different in nature), which can scale to larger systems, we turn now to shrinkage methods.

6.1. Shrinkage methods. Shrinkage methods approach this problem by relinquishing the goal of an unbiased estimator. The key is to consider *modified* linear regressions by changing the loss function using heuristics that aim to induce sparsity in the models (i.e., by reducing the number of descriptors).

Shrinkage methods come with two main alternatives: Ridge regression and LASSO. Both methods entail a change in the loss function to be optimised, based on an heuristic

that transforms the problem from a combinatorial optimisation to a continuous optimisation. In other words, since selecting predictors is a combinatorial optimisation problem ('hard'), **shrinkage methods change the loss function to try and make the *size* of coefficients (that is, the learnt parameters) small.** The idea is to keep small the coefficients that cannot be robustly determined from data, hence **enforcing a continuous version of sparsity**. In doing so, **we lose the unbiased nature of the estimators, but we gain the potential benefit of lower variance**.

6.1.1. *Ridge regression.* As mentioned above, the idea is to mimic the elimination of descriptors in a continuous fashion. In practice, one weighs them low by introducing an altered loss function that penalises large values of $\|\boldsymbol{\beta}\|$ through the inclusion of a penalty (or regularisation) term:

$$L_{\text{RIDGE}}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2$$

where $\lambda > 0$ is the *penalty term* and $\|\boldsymbol{\beta}\|^2 = \sum_{i=1}^{p} |\beta_i|^2$. As before, we now seek to minimise this loss:

$$\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2 \quad \Leftrightarrow \quad \min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 \text{ subject to } \|\boldsymbol{\beta}\|^2 \leq t,$$

where the two minimisations are equivalent in the sense of duality (see aside).

> *Quick aside:* The dual equivalence of the minimisations can be obtained by considering the KKT conditions, which we do not cover in detail in this course. The KKT conditions are a generalisation of Lagrange multipliers, which you learnt for the case of *equality* constraints, to the case of *inequality* constraints, such as we have here. You can think of λ as the 'Lagrange multiplier' enforcing the constraint $\|\boldsymbol{\beta}\|^2 \leq t$. The full equivalence also follows more generally from the *strong duality* of this convex problem. This is out of the remit of this course but you can read more about it in Stephen Boyd's book on *Convex optimization*. Note that λ and t are inversely related; intuitively, the smaller we want to make $\|\boldsymbol{\beta}\|^2$ the larger our 'Lagrange multiplier' λ has to be to enforce the constraint.

This problem can be solved explicitly:

$$L_{\text{RIDGE}}(\boldsymbol{\beta}) = \mathbf{y}^T\mathbf{y} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta}$$

As per usual, taking the derivative:

$$\nabla_{\boldsymbol{\beta}} L_{\text{RIDGE}}(\boldsymbol{\beta}) = -2\mathbf{X}^T\mathbf{y} + 2(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta}$$

Setting the above to 0 gives:

$$\mathbf{X}^T\mathbf{y} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta}^\star$$

Finally, we arrive at our solution for ridge regression:

$$\boldsymbol{\beta}^\star_{\text{RIDGE}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

As above, one should check that the Hessian is positive definite, which will be left as an exercise to the reader. Note that the penalty is not applied to β_0, hence, the element of the identity matrix $\mathbf{I}$ corresponding to β_0 should be set to zero.

We now compute the bias of the estimator. To this end, we first compute the following (remember that $\mathbb{E}[\boldsymbol{\varepsilon}] = 0$):

$$\begin{aligned}\mathbb{E}\big[\boldsymbol{\beta}^\star_{\text{RIDGE}}\big] &= \mathbb{E}\Big[(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta} + (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\boldsymbol{\varepsilon}\Big] \\ &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}\end{aligned}$$

Quick aside: A reminder about diagonalisation and eigendecompositions. The eigendecomposition of $\mathbf{X}^T\mathbf{X}$

$$(\mathbf{X}^T\mathbf{X}) = \mathbf{V}\mathbf{D}\mathbf{V}^T$$

has the following properties:

$$\mathbf{V}\mathbf{V}^T = \mathbf{V}\mathbf{V}^{-1} = \mathbf{I}$$
$$(\mathbf{X}^T\mathbf{X})^{-1} = \mathbf{V}\mathbf{D}^{-1}\mathbf{V}^T,$$

where $\mathbf{V}$ contains the eigenvectors as columns, and $\mathbf{D}$ is a diagonal matrix with $\mathbf{D} = \text{diag}(d_i)$, where d_i are the corresponding eigenvalues. Some of the properties rely on the fact that $\mathbf{X}^T\mathbf{X}$ is symmetric.

Using this aside, we can now use the eigendecomposition to obtain:

$$\begin{aligned}\text{bias}(\boldsymbol{\beta}^\star_{\text{RIDGE}}) &= \mathbb{E}\big[\boldsymbol{\beta}^\star_{\text{RIDGE}}\big] - \boldsymbol{\beta} \\ &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta} - \boldsymbol{\beta} \\ &= \mathbf{V}\big[(\mathbf{D} + \lambda\mathbf{I})^{-1}\mathbf{D} - \mathbf{I}\big]\mathbf{V}^T\boldsymbol{\beta}\end{aligned}$$

Using the fact that $\mathbf{D}$ is diagonal (and $\mathbf{I}$ of course, too), we can write:

$$\begin{aligned}\mathcal{D} &= (\mathbf{D} + \lambda\mathbf{I})^{-1}\mathbf{D} - \mathbf{I} \\ \mathcal{D}_{ii} &= \Big[\frac{d_i}{d_i + \lambda} - 1\Big] \\ &= -\frac{\lambda}{d_i + \lambda} \\ \Rightarrow \mathcal{D} &= -\lambda(\mathbf{D} + \lambda\mathbf{I})^{-1}\end{aligned}$$

Finally, we can write:

$$\begin{aligned}\text{bias}(\boldsymbol{\beta}^\star_{\text{RIDGE}}) &= -\lambda\mathbf{V}\big[(\mathbf{D} + \lambda\mathbf{I})^{-1}\big]\mathbf{V}^T\boldsymbol{\beta} \\ &= -\lambda(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\boldsymbol{\beta}\end{aligned}$$

As expected, as $\lambda \to 0$, we have that bias $\to 0$, because we recover the LS solution from before. Furthermore, as $\lambda \to \infty$, bias $\to -\boldsymbol{\beta}$ which corresponds to about 100% error.

Now that we have obtained an expression for the bias, we turn towards the variance:

$$\begin{aligned}\text{var}(\boldsymbol{\beta}^\star_{\text{RIDGE}}) &= \mathbb{E}\Big[\big(\boldsymbol{\beta}_{\text{RIDGE}} - \mathbb{E}[\boldsymbol{\beta}^\star_{\text{RIDGE}}]\big)^2\Big] \\ &= \mathbb{E}\Big[(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\underbrace{\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}}_{\sigma^2\mathbf{I}}\mathbf{X}\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\Big] \\ &= \sigma^2(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^T\mathbf{X})\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\end{aligned}$$

Using the eigendecomposition of $\mathbf{X}^T\mathbf{X}$ from above:

$$= \sigma^2\mathbf{V}\Big[\underbrace{(\mathbf{D} + \lambda\mathbf{I})^{-1}\mathbf{D}(\mathbf{D} + \lambda\mathbf{I})^{-1}}_{\mathcal{P}}\Big]\mathbf{V}^T$$

By the diagonality of $\mathbf{D}$ and $\mathbf{I}$, we have:

$$\mathcal{P}_{ii} = \frac{d_i}{(d_i + \lambda)^2}$$

From this we can now draw the following conclusion: as $\lambda \to \infty$, $\mathcal{P}_{ii} \to 0$ (quadratically), thus reducing the variance.

In summary, increasing λ reduces the variance of the ridge estimator but increases its bias. Both trends can be visualised as in Figure 1.5.

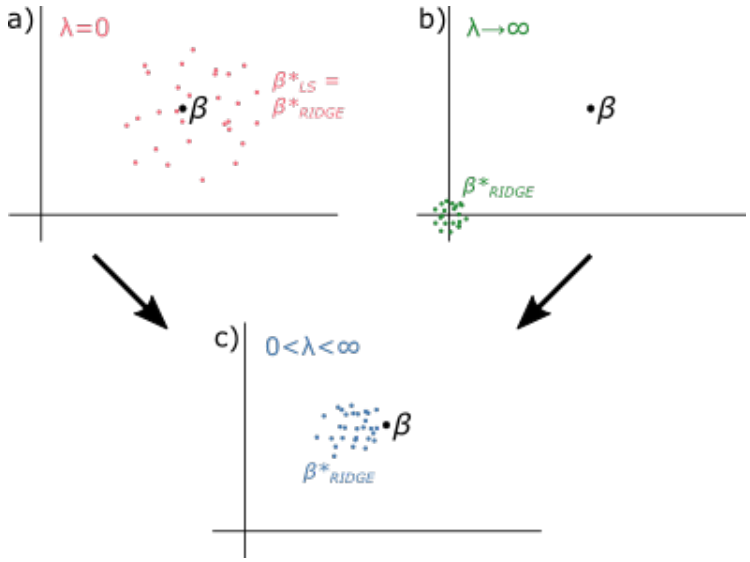


Figure 1.5. Bias and variance visualised for ridge regression. a) At $\lambda = 0$, we simply recover the least-squares estimate. b) As $\lambda \to \infty$, the bias tends towards $-\boldsymbol{\beta}$, resulting in the estimates being around 0. The variance decreases towards 0. c) As usual, the 'sweet spot' will be somewhere in-between.

6.1.2. *LASSO (Tibshirani).* In the LASSO (least absolute shrinkage and selection operator) method, as with the Ridge regression, we redefine the loss function to include a penalty term that tries to reduce the size of the parameter vector:

$$L_{\text{LASSO}}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_1$$

Here $\|\boldsymbol{\beta}\|_1$ denotes the *Taxicab norm* (also known as the *Manhattan norm*):

$$\|\boldsymbol{\beta}\|_1 = \sum_{i=1}^{p} |\beta_i|$$

Note that i starts from one, i.e. the penalty does not apply to the intercept β_0. Compared to the 2-norm used in the penalty term of the ridge regression cost function, the 1-norm used in the LASSO penalty term is closer to 'direct selection of parameters' (which would correspond to the 0-pseudonorm) but it makes the optimisation harder (we will not be able to do it by hand); in other words, the 1-norm is a better relaxation of the 0-pseudonorm but more difficult to deal with mathematically.

Once again, we aim to minimise the corresponding loss function:

$$\min_{\boldsymbol{\beta}} L_{\text{LASSO}}(\boldsymbol{\beta}) \Leftrightarrow \min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 \text{ subject to } \|\boldsymbol{\beta}\|_1 \leq t.$$

The problem is convex also and strong duality applies here too. However, as opposed to ridge regression, there exists no analytical solution for LASSO. However, since the problem is convex, it is possible to apply convex optimisation techniques[1] to find the global optimum computationally, an example being gradient descent methods with the Huber regularisation to make the loss function smoother, as you will see in the coding notebook.

In Figure 1.6, we show a quick visual overview of both ridge and LASSO regressions and how the penalty functions modify the results. The important thing to notice is that LASSO tends to concentrate the solution towards the axes of the parameter space, i.e., it

[1]See again the book *Convex Optimisation* by Stephen Boyd if you want to know more.

makes many parameters have *small values*, so it induces a stronger sparsity than ridge. You can see this in Figure 1.6 by looking at where the optimal points are located in each case. From this we can get an intuition of why LASSO tends to give sparse solutions.

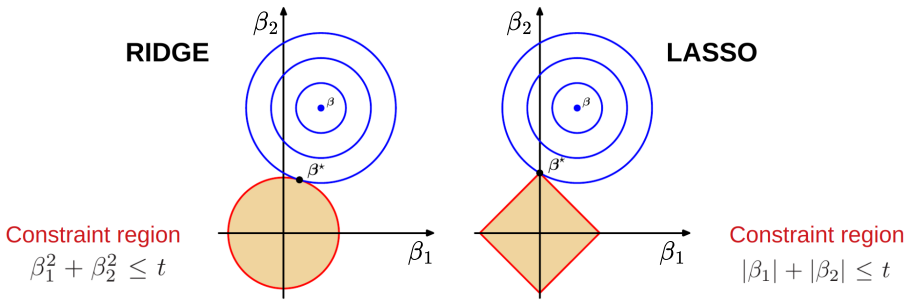


Figure 1.6. Figure showing the level curves of the unconstrained Least Squares solution (blue) and, in red, we the ridge constraint region (left) and the LASSO one (right). Figure adapted from Bishop, *Pattern Recognition and Machine Learning*, chapter 3.

6.2. Regularisation. The above optimisation approach indicates a general procedure to introduce penalty terms that induce sparsity. This penalty terms are also known as *regularisation terms* in optimisation, as they balance two conflicting terms of a cost function. In statistical and machine learning, ***regularisation* generally refers to terms that help fix the issue of overfitting by imposing a penalty on the parameters in the loss function, keeping in this way under control the parameter values learnt**.

Key message: General variations of shrinkage methods involve regularisation terms containing the l_q-norm:

$$L_q(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_q^q$$

where:

$$\|\boldsymbol{\beta}\|_q^q = \sum_{i=1}^{p} |\beta_i|^q$$

Ridge and LASSO are particular cases of this type of regularisation with respectively $q = 2$ and $q = 1$.

Figure 1.7 shows a visual overview of this approach. All of these different regularisation variants aim to control the model's sparsity. The figure summarises the overall trend in sparsity and the ease of optimisation. In general, the sparser we want things, the more difficult to optimise.

Note that $q = 0$ is the 'l_0'-pseudonorm case. In this case, we only have solutions where some of the parameters β_i are zero, which is equivalent to the selection of optimal subsets. This leads to truly sparse models (i.e., with zeros for many parameters); however it is difficult to optimise for sparsity in this $q = 0$ (combinatorial) case. The closer we get to this $q = 0$ case, the harder it is to optimise. The most important point is when we cross the boundary of $q = 1$ and enter the realm of 'non-convex' sets. At that point, convex optimisation is not applicable and we are not able to apply any of the powerful computational methods that rely on convexity.

To note, another direction also pursued in shrinkage methods is to combine norms in the penalty term. An example of this is the *elastic net* regularisation, which produces a

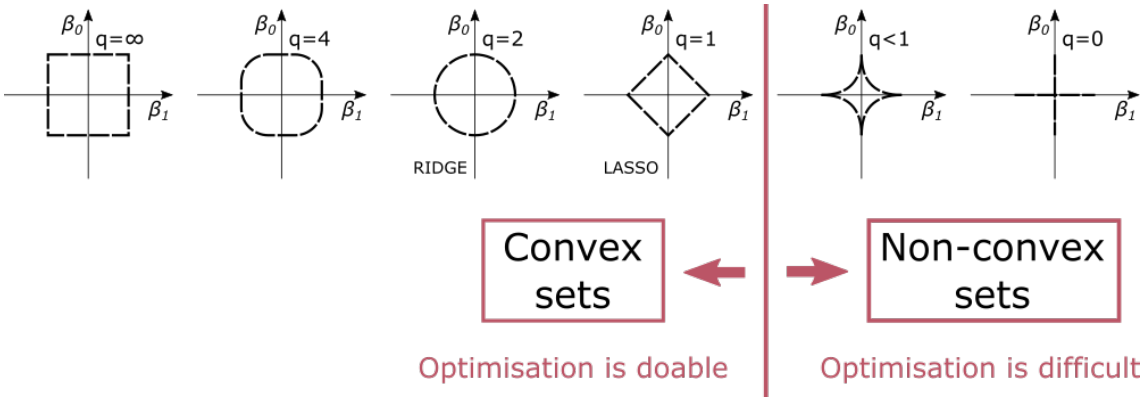


Figure 1.7. An overview of regularisation variants, i.e. $\lambda\|\boldsymbol{\beta}\|_q$ for a range of q.

convex combination of ridge and LASSO penalty terms:

$$L_{\mathrm{EN}}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\big[\alpha\|\boldsymbol{\beta}\|_1 + (1-\alpha)\|\boldsymbol{\beta}\|^2\big].$$

The elastic net is widely used in statistics to achieve sparsity.

6.3. Regularisation and overfitting. The regularisation parameter λ is an example of *hyperparameter*. We will see that more complex models, like neural networks, have several hyper-parameters. These parameters are called 'hyper' because typically they specify the overall structure of the model or the algorithm. These are parameters that are chosen and remain fixed during the optimisation that is done on the training set. For instance, in Linear Regression, when we optimise over β_i (the parameters of the model), we maintain fixed the value of λ (hyperparameter).

Hyperparameters are key to keeping under control over-fitting, the issue we mentioned at the beginning of this chapter. To avoid overfitting and enhance the robustness and generalisability of our models, it is customary to calibrate the influence of the hyperparameters on the results (a so-called 'hyperparameter search'), using a subset of the samples called the *validation set*. The model trained on the training set with a particular choice of hyperparameters is used to predict the outcomes on the validation set. This process is repeated with different values/choices of the hyperparameters. This scanning (or optimisation) over the hyperparameters fulfills a double function: finding the optimal combination of hyperparameters, and providing us with some reassurance that the model is robust and has the potential to generalise well. After this validation step, we will choose a model with an optimised set of hyperparameters with good performance and robustness. This model is then used on the *test data*, which has not been used *at all* in the optimisations of parameters or hyperparameters.

6.4. Practical tips. When then should we use ridge or Lasso shrinkage methods? A simple rule-of-thumb is the following:
1. If you don't have many data, always consider ridge regression to avoid overfitting. You can easily scan the model trained for different values of the penalties on a held-out validation set to check that you need ridge regularisation to control for overfitting.
2. If you want to achieve sparsity, use Lasso regression: it's an effective strategy to obtain more 'parsimonoius' models, i.e. models that explain the outcome by fewer predictors and are hence more interpretable.

Final aside: So far, all the methods discussed in this chapter have been linear. In mathematical terms, we have so far concerned ourselves with:

$$\hat{f}_{\text{lin}}(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\beta}^{\star}_{\text{LS, Ridge, LASSO, EN, ...}}$$

However, introducing non-linearity is not that far from this framework. As we will see in the following chapters, many methods become nonlinear by extending this inference framework to models expressed in terms of nonlinear functions, i.e., models in the following non-linear form:

$$\hat{f}_{\text{nonlin}}(\mathbf{x}) = \mathbf{h}_m^T(\mathbf{x})\boldsymbol{\beta}^{\star}_{\text{nonlin}}$$

An example of such non-linear functions are polynomials of the inputs up to a certain degree:

$$\left\{h_{m,1}, \ldots, h_{m,T}\right\} \text{ from } \left(x_1^{d_1} \quad x_2^{d_2} \quad \ldots \quad x_p^{d_p}\right)$$

In the case of polynomials, one does have to choose how large the degrees should be, i.e. determine a D such that $\sum d_i < D$. Sparsity is desirable here since the number of polynomial terms grows combinatorially with the number of descriptors p and the largest degree D. Furthermore, polynomial models tend to overfit for high degrees. Hence sparsity is desirable here. These polynomial models are called Wiener-Volterra models in the applied math and signal processing literatures.

The collection of nonlinear functions $\left\{h_{m,1}, \ldots, h_{m,T}\right\}$ is usually called the *dictionary* in the Computer Science literature. Other examples of non-linear function dictionaries include: $\log(x_i), \sin(x_i), \cos(x_i)$, (or more generally $\sin(kx_i)$), or wavelets. The choice of dictionary is dictated by knowledge about the data based on modelling assumptions, usually. Additional properties, such as orthogonality and completeness, are also desirable. Fourier analysis and the whole of the 19th century orthogonal polynomial literature are precursors for this area in Machine Learning.

K-Nearest Neighbours

All the models we saw in Chapter 1, either the strictly linear ones defined by

$$\hat{y} = \hat{f}_{\text{Lin}}(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\beta}^\star,$$

or the ones defined in terms of sets of nonlinear functions $\mathbf{h}(\mathbf{x})$ given by

$$\hat{y} = \hat{f}_{\text{Nonlin}}(\mathbf{x}) = \mathbf{h}(\mathbf{x})^T \boldsymbol{\beta}_h^\star$$

are *global* models. Indeed, each of those models is described by the vector of constant parameters $\boldsymbol{\beta}$, and it applies unchanged over the whole domain of the inputs, i.e., it spans the entire data space.

The alternative is to obtain *local* models, which lead to piece-wise descriptions of the data. In other words, we aim not for one model $\hat{f}$ for any value of the input, but for **a model that is different depending on the input $\mathbf{x}^{\text{in}}$**.

Local models have advantages and disadvantages, but they can be extremely powerful tools in data analysis. We will now introduce perhaps the simplest such model, k-Nearest Neighbours, which has wide applications.

1. The k-Nearest Neighbours (kNN) algorithm: continuous variables

Our aim is to construct a *local model*:

$$\hat{y} = \hat{f}_{\mathcal{N}(\mathbf{x}^{\text{in}})}(\mathbf{x}^{\text{in}}),$$

where $\mathbf{x}^{\text{in}}$ represents a new (unseen) data point (i.e. not in the data set used to train the model) for which we want to make a prediction. Note how, instead of applying a model globally to all the input data, the model $\hat{f}_{\mathcal{N}(\mathbf{x}^{\text{in}})}(\cdot)$ is dependent on the input.

In the case of kNN, we construct a set of models $\left\{\hat{f}_{\mathcal{N}(\mathbf{x}^{\text{in}})}\right\}$, i.e., a set of functions that return different subsets of the data depending on the input. The subsets are chosen based on a neighbourhood of the input $\mathbf{x}^{\text{in}}$ as defined by a chosen metric.

Let us consider continuous input variables $\mathbf{x}^{(i)} \in \mathbb{R}^p,\ y \in \mathbb{R},\ i = 1, \ldots, N$, and let us choose a distance metric in the space of inputs (in this case, $\mathbb{R}^p$), let's denote it by:

$$\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|.$$

Typically, we will choose the Euclidean distance as our metric.

The kNN method proceeds as follows: for any input $\mathbf{x}^{\text{in}}$, find the k closest samples in the training data set (with N^{training} data points) to $\mathbf{x}^{\text{in}}$ according to our chosen distance. This

subset of samples in the training set define the neighbourhood $\mathcal{N}(\mathbf{x}^{\text{in}})$. Then construct a prediction function for $\mathbf{x}^{\text{in}}$ based on the samples in $\mathcal{N}(\mathbf{x}^{\text{in}})$. The prediction function is usually very simple, i.e., a simple averaging. Figure 2.1a) shows an outline of these ideas.

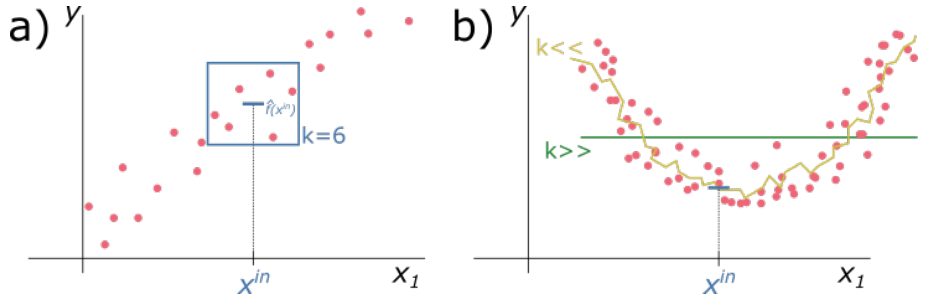


Figure 2.1. k-Nearest Neighbours. a) Given some training data (not necessarily linear like in this cartoon example!), a new data point $\mathbf{x}^{\text{in}}$, and a choice of k (in this case $k = 6$), the algorithm searches for the 6 nearest neighbours to $\mathbf{x}^{\text{in}}$. The prediction for $\mathbf{x}^{\text{in}}$ is then the average of the 6 points' outcome variables, marked by a blue bar. b) The same idea works for more complicated data. But care should be taken when choosing k: overfitting occurs for too small k (yellow line) whereas underfitting occurs for too large k (green line). Hence the parameter k needs to be optimised on the data by a hyperparametric search.

Algorithm: The kNN algorithm is implemented as follows. Given an input $\mathbf{x}^{\text{in}}$:

(1) Compute all distances between $\mathbf{x}^{\text{in}}$ and the samples in the training set:

$$\|\mathbf{x}^{\text{in}} - \mathbf{x}^{(i)}\| \qquad i = 1, \ldots, N^{\text{training}}$$

(2) Find the k nearest neighbours to $\mathbf{x}^{\text{in}}$ which define the neighbourhood:

$$\mathcal{N}_k(\mathbf{x}^{\text{in}})$$

Note that k is a choice, i.e., it is a hyperparameter to set via model selection.

(3) The simplest choice for predictor is just averaging over the values of the outcome variables for the samples in the neighbourhood:

$$\hat{f}_{\mathcal{N}(\mathbf{x}^{\text{in}})} = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}^{\text{in}})} y^{(i)}$$

The outcome of this algorithm depends on the choices made:

- **Distance metric and standardisation are very important**. Given p predictor variables $\mathbf{x} = (x_1, \ldots, x_p)$ with vastly different ranges, e.g. $1 < x_1 < 10^6$ and $10^{-10} < x_p < 10^{-5}$, it is clear that x_1 will overly dominate the distance measures. Hence, standardising all predictor variables to make them vary on a similar scale is key, unless the hypothesis states that some descriptors indeed are more important than others. By standardisation, we mean replacing every variable $x_m^{(i)}$, $m = 1, ..., p$, $i = 1, ..., N$, by a variable $z_m^{(i)}$ given by:

$$z_m^{(i)} = \frac{x_m^{(i)} - \mu_m}{\sigma_m} \quad \text{with} \quad \mu_m = \frac{1}{N} \sum_{i=1}^{N} x_m^{(i)}, \quad \sigma_m = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_m^{(i)} - \mu_m)^2}$$

where μ_m is the mean of the feature m over the sample (so the mean of $z_m^{(i)}$ is zero), and σ_m is its standard deviation over the sample (so the standard deviation of $z_m^{(i)}$ is one).

Similarly, the distance metric can serve to give more or less importance to the different variables.

- **The algorithm works better when the number of input variables, p, is relatively small and the variables are all relevant and not redundant.** In the presence of high-dimensional noisy inputs (i.e., with large p and many descriptors being unrelated to the output) the local prediction can be erratic. Hence, kNN is sensitive to the dimensionality of the data.
- As can be seen in Figure 2.1, the choice of k is crucial. Choosing k does allow us to scan the bias-variance trade-off: from the picture we can see, intuitively, that a small k will be highly accurate for a given input, but very variable across inputs, and thus not very generalisable (hence prone to overfitting), and vice versa. In other words, k acts here equivalently to the parameter λ that modulates the regularisation term in linear regression, and hence the model complexity. How to choose the parameter k is the object of the following section. For an expanded discussion of the bias-variance tradeoff in kNN, see also *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*, Chap. 2, section 2.3.2 and Chap. 7, section 7.3.1.

2. The general procedure of T-fold cross validation

Remark: Following on from the last point of the previous section, we now ask the question of how to choose k, which is a *hyperparameter*. But this section is much more general—it introduces **a procedure that is applied generally across statistical and machine learning methods, termed T-fold cross validation**. It aims to control for the complexity of the model **to achieve optimal generalisability**. Although this procedure is introduced here using kNN as an illustration, it is used widely for most other models in ML for the hyperparametric search.

To address the problem of the selection of k, we introduce T-fold cross validation. (Note that in the literature it is more often denoted k-fold cross validation, but in order to avoid confusion with our kNN, we use T here.)

The main idea of T-fold cross validation is to split our available data for training (for which we have full knowledge, i.e. predictors and observations) into a training set and a validation set. The model is learnt on the training set and checked against the validation set to see how well it performs. To enhance robustness, this split into training-validation is done T times, so that we reduce the risk of overfitting to the training data.

More specifically, first we split the data $\mathcal{S} = \{\mathbf{x}^{(i)}, y^{(i)}\}$ for $i = 1, \ldots, N^{\text{training}}$ into T equal subsets (or "folds") called $\mathcal{S}_t$, such that:

$$\mathcal{S} = \bigcup_{t=1}^{T} \mathcal{S}_t \text{ and } |\mathcal{S}_t| = \frac{|\mathcal{S}|}{T}, \text{ where } |\mathcal{S}| = N^{\text{training}}$$

Then, we set aside one of the subsets ($\mathcal{S}_t$, the validation subset) and train the model on the rest of the samples, i.e., on the complement set $\overline{\mathcal{S}_t} = \mathcal{S} - \mathcal{S}_t$. We learn a model $\hat{f}_{\overline{\mathcal{S}_t}}$, which we use to predict $\mathcal{S}_t$. We then compute an error measure for this prediction on $\mathcal{S}_t$ (in this case, the mean square error, MSE):

$$\text{MSE}_t = \frac{1}{|\mathcal{S}_t|} \sum_{i \in \mathcal{S}_t} \left[\hat{f}_{\overline{\mathcal{S}_t}}(\mathbf{x}^{(i)}) - y^{(i)} \right]^2$$

The same process is done in turn for each of the subsets of our split $\mathcal{S}_t, t = 1, \ldots, T$ in each case obtaining a different model from their corresponding complement. Finally, we compute the average MSE over all T 'folds' (or splits):

$$\langle \text{MSE} \rangle = \frac{1}{T} \sum_{t=1}^{T} \text{MSE}_t$$

This average MSE can be seen as a measure of how well the model predicts out-of-sample, i.e., on data unseen during the training.

The T-fold cross validation is applied to the data that we decide to use for *training* the method, that is, to learn its parameters having set the hyperparameters. **To test the model after having set the hyperparameters, we need to evaluate it on a portion of the data, of size $N^{\text{test}} = N - N^{\text{training}}$, which is not used in the training nor in the cross-validation procedure**. We call it *test set*, and it needs to be split from the data to use for training before any training or cross-validation step.

Example application to kNN. Coming back to the question of choosing k for the kNN model, this procedure can of course be done for a range of values of k, i.e., computing $\langle \text{MSE}(k) \rangle$. As a result, one might obtain a graph similar to Figure 2.2, from which one can obtain an optimal $k^\star$ (or a range of 'good' k values in more realistic scenarios).

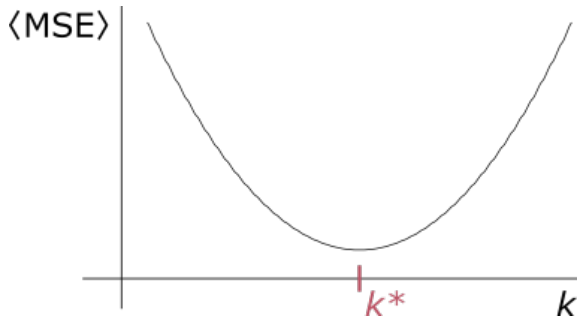


Figure 2.2. A schematic of the outcome of scanning k for a kNN model through T-fold cross-validation. This can help us choose an optimal $k^\star$. For small k, the fold-averaged MSE will be large, since the model will not be generalisable (refer to Figure 2.1) and thus will be performing badly when predicting out-of-sample. Similarly for large k, the model will be too generic and thus also performs badly. Therefore, there will be a k, i.e. a level of complexity of the model, which will be optimal. In practice, such nice behaviour is rarely observed but one can obtain good regions for the relevant parameter. Alternatively, one can find that certain parameters have little or no influence on the out-of-sample prediction, so they can be set to a value that does not increase unnecessarily the model complexity.

How to choose the number of folds T: Whilst cross-validation helps choose a k, what about choosing the number of splits T? It seems clear that the way in which we split the data into training and validation, and the balance between both sets, will affect the results of the cross-validation.

The extreme case is $T = N^{\text{training}}$, which is called *leave-one-out* cross-validation (LOO-CV). This tests how well each sample can be predicted from the rest of samples. However, this is computationally expensive and not necessary.

In practice, **the number of folds T is chosen based on the size of the data set and the computing power available** (larger T means more models need fitting). Commonly, $T = 5$ is used for a smaller data set and $T = 10$ for larger ones, so that one

does out-of-sample prediction on validation sets that contain, respectively, around 20 or 10% of the data.

3. k-Nearest Neighbours for discrete variables

3.1. kNN for discrete predictors. The kNN algorithm can also be applied to discrete data by extending the concepts described above. For example, let us suppose that our data is binary:

$$\mathbf{x}^{(i)} \in \{0,1\}^p, \quad y^{(i)} \in \mathbb{R}$$

Then, in order to use the kNN algorithm, **all we have to do is to define a distance measure on the input set of binary variables**, e.g. the Hamming distance:

$$d_H(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sum_{m=1}^{p} I(x_m^{(i)} \neq x_m^{(j)})$$

Here we have made use of a version of the *indicator function*, which in this case, can simply be defined as:

$$I(b) = \begin{cases} 1, & \text{if } b \text{ is true} \\ 0, & \text{if } b \text{ is false} \end{cases}$$

In other words, the Hamming distance is the number of positions at which the corresponding entries are different. The rest of the kNN algorithm is exactly the same: this shows that defining a distance measure is key to incorporating any type of predictor spaces.

3.2. k-Nearest Neighbours with discrete outcomes. The previous sections addressed a *regression problem* where the input variables were discrete but the outcomes were continuous variables. We concluded that this can be accommodated easily by changing to a distance metric in the space of discrete inputs.

We now show that kNN can also be applied to classification, i.e. when the outcomes are discrete variables belonging to Q classes:

$$y^{(i)} \in \mathcal{C}_q = \{c_1, \ldots, c_Q\}$$

The inputs can be continuous or discrete, as this will be accommodated through the appropriate distance metric $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$.

The kNN algorithm for discrete outcomes is similar to the continuous version but adapting the *predictor* function. Schematically, the steps are:

(1) Choose k – same as before;

(2) For $\mathbf{x}^{\text{in}}$, at fixed k, find the neighbourhood $\mathcal{N}_k(\mathbf{x}^{\text{in}})$ – same as before;

(3) The final prediction is different here: $\hat{y} = \hat{f}(\mathbf{x}^{\text{in}}) \in \mathcal{C}_Q$ by the majority rule.

This is best understood visually, see Figure 2.3 for a descriptive intuition. The idea is simple: $\mathbf{x}^{\text{in}}$ will be assigned the class of the majority of its neighbours.

The *majority rule* is the simplest predictor function in this case, and can be seen as a discretisation of a probabilistic predictor, as follows.

Recall that $\mathcal{C}_q = \{c_1, \ldots, c_Q\}$ represents the set of all classes and I is the indicator function. Then we define the output predictor in terms of the probability that the input

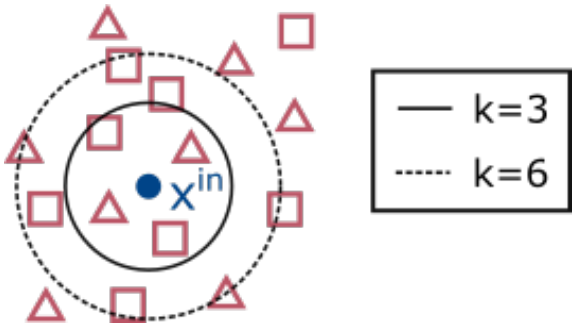


Figure 2.3. k-Nearest Neighbours for a discrete outcome variable. Here we have some data belonging to one of two classes: squares and triangles in some predictor space. For $k = 3$, we apply the majority rule to count two triangles and one square in the neighbourhood and get a prediction $\hat{y}(\mathbf{x}^{\text{in}}) = \triangle$, whereas for $k = 6$, we have $\hat{y}(\mathbf{x}^{\text{in}}) = \square$. Note that for ease of visualisation, the radii were drawn such that only those data points are counted as inside, if their symbols are fully inside the radius.

$\mathbf{x}^{\text{in}}$ belongs to each of the Q classes:

$$\hat{f}_{\text{Prob-kNN}}(\mathbf{x}^{\text{in}}) = \frac{1}{k} \begin{pmatrix} \sum_{i \in \mathcal{N}_k(\mathbf{x}^{\text{in}})} I\big(y^{(i)} \in c_1\big) \\ \vdots \\ \sum_{i \in \mathcal{N}_k(\mathbf{x}^{\text{in}})} I\big(y^{(i)} \in c_Q\big) \end{pmatrix}$$

$$= \frac{1}{k} \begin{pmatrix} \text{number of neighbour points in } c_1 \\ \vdots \\ \text{number of neighbour points in } c_Q \end{pmatrix}$$

Hence each component of $\hat{f}_{\text{Prob-kNN}}(\mathbf{x}^{\text{in}})$ denotes the probability of $\mathbf{x}^{\text{in}}$ belonging to a specific class. Then the majority rule corresponds to the *argmax function* applied across the components of this probabilistic predictor, i.e., choosing the class with the maximum probability among the Q classes:

$$\hat{f}_{\text{MajRule-kNN}}(\mathbf{x}^{\text{in}}) = \operatorname*{argmax}_q \hat{f}_{\text{Prob-kNN}}(\mathbf{x}^{\text{in}})$$

so the class assignment for a certain input data point is 1 for a certain class, and 0 for all the others.

> *Remark.* This discretisation (the argmax step), in which a probabilistic outcome is transformed into a discrete outcome by choosing the maximum probability, is typical in many classification methods, including neural networks (chapter 7) and logistic regression (chapter 3), as we will see during the rest of the course. In general, we talk about *soft* classification when the predicted assignment to classes is probabilistic, *hard* classification when we perform the argmax discretisation.

Finally, note that to evaluate the classification outcome one will need to define an *error function* (or, conversely, a *quality function*) that works for categorical data. (The MSE is obviously not appropriate, but we will see these measures later on, starting with Section 2 of Chapter 3). Once our quality function is defined, the concept of T-fold cross-validation remains the same as above.

Logistic regression

In contrast to the previous chapter, we now turn towards solving *classification problems with a global approach*. The classic model in this category is **logistic regression**, and we study it in its simplest formulation: when we have a binary (2-class) outcome variable to be predicted.

> *Preliminary comment:* You might be asking yourselves, since we are carrying out a *classification task*, is 'logistic regression' actually a *regression*? The answer is yes, and you can understand this based on our discussion at the end of Section 3.2, Chapter 2 regarding the argmax discretisation. In short, **logistic regression predicts the *probability* that our output belongs to each of the two classes** and then selects the class based on a threshold. In the simplest case, the threshold leads to selecting the outcome with the higher probability, a.k.a. argmax discretisation.

1. Logistic regression

Once again, we set up our problem in the by now standard setting:

$$\mathbf{x}^{(i)} \in \mathbb{R}^p; \quad y^{(i)} \in \{0, 1\} \qquad i = 1, \ldots, N$$

where we have continuous input variables and binary outcome variables to be predicted.

The logistic regression model is global and linear, but adapted to the discrete nature of the outcomes. Since our output variables can only have two outcomes $\{0, 1\}$, one way to understand logistic regression is by considering the so-called *log odds* (or *logit function*):

$$\log \frac{P(y = 1|\mathbf{x})}{P(y = 0|\mathbf{x})} = \log \frac{P(y = 1|\mathbf{x})}{1 - P(y = 1|\mathbf{x})} \tag{3.1}$$

where $P(y = 1|\mathbf{x})$ (respectively $P(y = 0|\mathbf{x})$) is the probability that our output variable will take the value 1 (respectively 0).

> **Key message:** The logistic regression model assumes that the logit function (3.1) is a *linear* combination of the input variables:
>
> $$\mathbf{x}^T \boldsymbol{\beta} = \log \frac{P(y = 1|\mathbf{x})}{1 - P(y = 1|\mathbf{x})} \tag{3.2}$$
>
> we have already switched to the shorthand notation where $\mathbf{x}^T$ is a $(p+1)$ length vector $(1, x_1, \ldots, x_p)$, allowing for a bias/intercept term, and $\boldsymbol{\beta} = \begin{pmatrix} \beta_0 & \beta_1 & \cdots & \beta_p \end{pmatrix}^T$.

In other words, logistic regression aims at predicting a **decision boundary, i.e., a function separating the data predicted in different classes, that is *linear*** (see Figure 3.1). Indeed, the decision boundary is given by the set of points where $P(y = 1|\mathbf{x}) = P(y = 0|\mathbf{x})$, hence they form the hyperplane, parametrised by $\boldsymbol{\beta}$, given by $\mathbf{x}^T\boldsymbol{\beta} = 0$.

With some rearranging of (3.2), we also see:

$$P(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{x}^T\boldsymbol{\beta}}} = h\left(\mathbf{x}^T\boldsymbol{\beta}\right) =: h_{\boldsymbol{\beta}}(\mathbf{x}) \tag{3.3}$$

This is called the *logistic function*, which is one of the key functions in statistical learning—it will reappear down the line when you look at neural networks. Clearly, we also have:

$$P(y = 0|\mathbf{x}) = 1 - h_{\boldsymbol{\beta}}(\mathbf{x}) \tag{3.4}$$

You already know how the logistic function looks but for a reminder, see Figure 3.1.

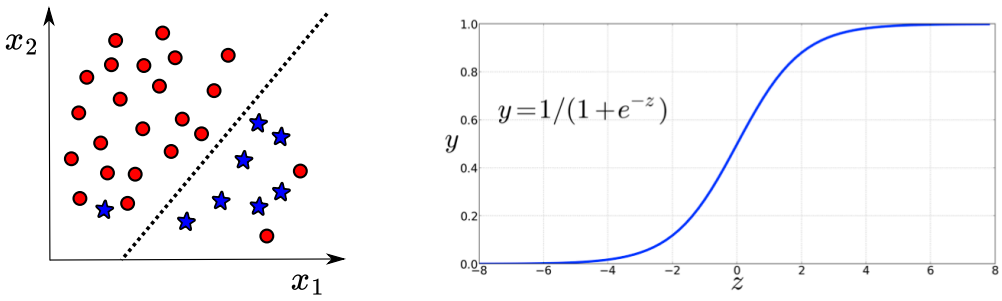


Figure 3.1. (Left) Logistic regression learns a linear decision boundary (a line in the two-dimensional case). (Right) The logistic function for a one-dimensional variable z.

The above expressions (3.3)–(3.4) can be immediately understood as the result of assuming that our output variable is a *Bernoulli* random variable with a probability of success $P(y = 1|\mathbf{x})$ (and of failure $P(y = 0|\mathbf{x})$) :

$$P(Y = y \mid \mathbf{x}, \boldsymbol{\beta}) = \big(h_{\boldsymbol{\beta}}(\mathbf{x})\big)^y \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x})\big)^{1-y}$$

In summary, the logistic regression model is a probabilistic model that makes the specific assumption that **the probability distribution of the output conditional on the data features is a Bernoulli distribution** with probability of success that depends on the parameters $\boldsymbol{\beta}$ and is given by the logistic function $h_{\boldsymbol{\beta}}(\mathbf{x})$ (3.3).

> *Quick Aside:* There are many ways to introduce and motivate logistic regression within broader families of approaches to classification: indeed, logistic regression is one member of the family of linear classification methods (i.e., methods aimed at learning linear decision boundaries), and of the one of probabilistic classifiers that rely on specific assumptions on the conditional probability density of output labels. In Chapter 4 of Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*, you can read more about logistic regression (section 4.4) in comparison to other methods from these families. You will also find more information on extensions of logistic regression to multi-class classification.

1.1. Estimating the parameters. To estimate the parameters $\boldsymbol{\beta}$ **we maximise the log-likelihood of the model over the data** (where here the log-likelihood is the one

of outputs conditional on the data features). Given that the N samples can be assumed to be independent, we can factorise probabilities to get:

$$P\big(\{y^{(i)}\} \,\big|\, \{x^{(i)}\}, \boldsymbol{\beta}\big) = \prod_{i=1}^{N} \big(h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\big)^{y^{(i)}} \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\big)^{1-y^{(i)}}$$

This leads us to the log-likelihood:

$$\mathcal{L} = \sum_{i=1}^{N} y^{(i)} \log h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\big) \tag{3.5}$$

The optimisation follows a similar pattern to what we saw in linear regression in Chapter 1. As with linear regression, we set $\nabla_{\boldsymbol{\beta}}\mathcal{L} = 0$ to find the maximum. Note that:

$$P(y = 0|\mathbf{x}) = 1 - h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)}) = \frac{e^{-(\mathbf{x}^{(i)})^T\boldsymbol{\beta}}}{1 + e^{-(\mathbf{x}^{(i)})^T\boldsymbol{\beta}}} = e^{-(\mathbf{x}^{(i)})^T\boldsymbol{\beta}}\; h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})$$

so that:

$$\log\Big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\Big) = -(\mathbf{x}^{(i)})^T\boldsymbol{\beta} + \log h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})$$

We can now use these expressions (fill in the easy steps) to simplify the log-likelihood to get:

$$\mathcal{L} = \sum_{i=1}^{N} \log h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)}) - (1 - y^{(i)})(\mathbf{x}^{(i)})^T\boldsymbol{\beta}$$

In order to obtain the gradient, first note that:

$$\begin{aligned}\nabla_{\boldsymbol{\beta}}\big(\log h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\big) &= \frac{e^{-(\mathbf{x}^{(i)})^T\boldsymbol{\beta}}}{1 + e^{-(\mathbf{x}^{(i)})^T\boldsymbol{\beta}}}\mathbf{x}^{(i)} \\ &= \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\big)\mathbf{x}^{(i)}\end{aligned}$$

Finally, using the above, we can write down the gradient of the log-likelihood:

$$\nabla_{\boldsymbol{\beta}}\mathcal{L} = \sum_{i=1}^{N} \big(1 - h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\big)\mathbf{x}^{(i)} - (1 - y^{(i)})\mathbf{x}^{(i)} = \sum_{i=1}^{N} \big(y^{(i)} - h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})\big)\mathbf{x}^{(i)}$$

Now, we look for the maximum (i.e., the equivalent of the Normal Equation in linear regression), which is where:

$$\nabla_{\boldsymbol{\beta}}\mathcal{L}|_{\boldsymbol{\beta}^\star_{\log}} = \sum_{i=1}^{N} \Big(y^{(i)} - h_{\boldsymbol{\beta}^\star_{\log}}(\mathbf{x}^{(i)})\Big)\,\mathbf{x}^{(i)} = \mathbf{0} \tag{3.6}$$

where the notation $\boldsymbol{\beta}^\star_{\log}$ indicates that this is the solution of the maximisation for logistic regression. Unlike the case of linear regression, this system of $p + 1$ equations can not be solved analytically. However, **since $\mathcal{L}$ is concave** (try to show it as an exercise), **standard optimisation algorithms (e.g., gradient ascent) are able to find the maximum**.

In practice, one can formulate the logistic regression learning task as the minimisation of a mean sample loss given by the mean negative log-likelihood:

$$E(L) = -\frac{1}{N}\sum_{i=1}^{N} y^{(i)} \log h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)}))$$

which gives the same solution as maximising the log-likelihood (3.5).

Once we have obtained the $\boldsymbol{\beta}^\star_{\log}$ that fulfils the equation (3.6) by gradient methods, we can then plug this into the equation (3.3) to get the 'probability of success'.

Key message: In summary, training a logistic regression model gives a probabilistic estimator for the input variables, hence a *real* outcome (a probability):

$$P(y=1|\mathbf{x}^{\text{in}}) = h_{\boldsymbol{\beta}^\star_{\log}}(\mathbf{x}^{\text{in}}) = h\big((\mathbf{x}^{\text{in}})^T\boldsymbol{\beta}^\star_{\log}\big) \in [0,1] \tag{3.7}$$

To get a hard classifier, we perform a *discretisation step* by defining a threshold τ for the classifier such that the sample is assigned to each of the two classes:

$$P(y=1 \mid \mathbf{x}^{\text{in}}) > \tau \implies \mathbf{x}^{\text{in}} \in \{1\} \tag{3.8}$$

The threshold is a hyperparameter of the model, which can be tuned *a posteriori* to improve the performance of the method, as we will see below. Clearly, if we take $\tau = \frac{1}{2}$, then the input variable is assigned to the class with maximum probability of the two classes (hence equivalent to argmax).

Quick aside: We can also summarise the solution to the logistic regression problem in matrix-vector form, as follows. Consider the vector version of (3.3) where $\mathbf{h}(\mathbf{X}\boldsymbol{\beta}^\star_{\log})$ is a N-dimensional vector function where each individual entry is:

$$h_i(\mathbf{X}\boldsymbol{\beta}^\star_{\log}) = \left(\frac{1}{1+e^{-(\mathbf{x}^{(i)})^T\boldsymbol{\beta}^\star_{\log}}}\right), \quad i=1,\dots,N$$

For some data set $\mathbf{X}$ (dimension $N \times (p+1)$), we have obtained the following probabilistic estimator:

$$\mathbf{h}(\mathbf{X}\boldsymbol{\beta}^\star_{\log}) = \hat{\mathbf{y}}$$

where $\hat{\mathbf{y}}$ is a vector of dimension $N \times 1$. The 'Normal Equation'-equivalent (3.6) is:

$$\mathbf{X}^T\big[\mathbf{y} - \mathbf{h}(\mathbf{X}\boldsymbol{\beta}^\star_{\log})\big] = 0$$

Compare this to the Normal Equation for linear regression, which was:

$$\mathbf{X}^T\big[\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^\star_{\text{LS}}\big] = 0.$$

2. Quality function for the classifier: Confusion matrix and Receiver Operating Characteristic (ROC)

We mentioned in Chapter 2, when discussing kNN classifiers, that a quality function (or error function) needs to be developed to quantify the performance of the model, since error functions like the MSE are not appropriate for discrete outcomes.

2.1. Confusion matrix and associated quality measures. We now discuss how to obtain quality measures for binary classification, as needed for logistic regression. To do this, we introduce the *confusion matrix*.

For the logistic regression with outcomes in $\{0,1\}$, Table 3.1 shows an overview of the different fields of a confusion matrix. With the four numbers (TP, FP, FN, TN) of this confusion matrix, one can create different 'quality measures' for the classifier. Below we show several such measures, all of which are used for different applications (i.e., depending on what is the objective of our computations and the type of data and application):

The *true positive rate* or *sensitivity* or *recall*:

$$\text{TPR} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP}+\text{FN}} = \text{recall}$$

		Truth	
		1	0
Predicted	1	true positives (TP)	false positives (FP)
	0	false negatives (FN)	true negatives (TN)

Table 3.1. Confusion matrix schematic. A good model will have most cases on the diagonal, i.e. either TP or TN. Note that FP is sometimes called the *Type I error* and FN is sometimes the *Type II error*. The sum over the first column gives the total amount of actual positives in the data P = TP+FN; the sum over the second column gives the total amount of actual negatives N = FP+TN.

The *true negative rate* or *specificity*:

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}}$$

The *accuracy*:

$$\mathrm{accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{N_{\mathrm{validation}}}$$

where $N_{\mathrm{validation}}$ is the number of data on which the model has been 'validated', and is simply the sum of all entries of the confusion matrix ($N_{\mathrm{validation}} = \mathrm{FN} + \mathrm{FP} + \mathrm{TN} + \mathrm{TP}$). Furthermore, the *precision* (or *Positive Predictive Value*):

$$\mathrm{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

And finally, the *F-score*:

$$\mathrm{F} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

F-score is the harmonic mean of precision and recall. It rewards models with similar recall and precision. Which of these functions is chosen depends on the particular problem—how much tolerance of FPs or FNs our application can accept. This will depend heavily on the field of application.

The case of imbalanced data. In imbalanced datasets, the sizes of the different data subsets are expected to be different. In the case of binary classification, one typically has a minority class (positives) which are present in much lower quantity than the majority class (negatives). This setting represents what is called a 'rare-class learning problem', which can appear in many applications, for example: the medical diagnosis of a disease (where disease cases are rarer than healthy cases); problems of anomaly detection (detection of frauds among normal financial transactions, detection of risk situations, errors in texts, etc.).

If one cares about the retrieval of the positive class, and less about the discrimination or balance of retrieval between classes, **the recommendation is to focus on measures that do not depend on the number of True Negatives (TN)**, being they many more than the positives in the cases under consideration. Such measures are recall, precision and F-score. Precision is particularly important since it gives the proportion of predicted positives that are true positives, hence it takes into account false positives as a source of misclassification. Precision tells how much we can trust the model when it predicts positives - which is what we are interested in in rare-class learning problems.

Accuracy and AUC on the other hand can assume misleadingly high performance: for instance, the accuracy can be high due to the fact that the majority class only is well

predicted. **Generally speaking, recall, precision and F-score are the measures more suitable for assessing the prediction of a minority class.**

In addition, there exist *ad hoc* adjustments of the classification measures to better handle the imbalanced data case, an example is 'balanced accuracy', which is a re-definition of accuracy that gives more weight to the minority class. It is given by:

$$\text{Balanced Accuracy} = \frac{\frac{\text{TP}}{\text{P}} + \frac{\text{TN}}{\text{N}}}{2}$$

i.e., the true positives (respectively, true negatives) are re-weighted by 1/P (respectively, 1/N), where P (respectively, N) is the total number of actual positives (respectively, negatives) in the dataset (see Table 3.1).

Importantly, imbalanced data are a vast area of research that involves also strategies at the level of algorithms and of the use of the data during the training that should be applied to ensure a good performance in this imbalanced cases (e.g., one strategy is to re-balance class distribution by weighting data, or by re-sampling them either by under-sampling the majority of the class, or over-sampling the minority class, or a mix of both.) For a discussion of these more advanced aspects see Sun et al., *Classification of imbalanced data: a review*, International Journal of Pattern Recognition and Artificial Intelligence (2009), which is available as additional reading.

> *Quick Aside.* In the case of **multi-class classification**, one can build the equivalent of the confusion matrix (here called contingency matrix), with the correct predictions along the diagonal and the incorrect predictions in the off-diagonal. There are some measures that are easily generalisable such as the accuracy (given by the number of correct predictions normalised by all predictions); for other measures, one can always use the same measures defined above for pairwise comparisons between classes and then take the average. The terminology *micro* or *macro* average is typically used depending on whether the averaging takes into account the class relative proportion or not - for a detailed discussion, see e.g. Grandini et al., *Metrics for multi-class classification: an overview* (2020) in 'additional reading'.

2.2. ROC curve. In addition to the 'zoo' of various quality functions that can be used to gauge a model's quality, a widely used representation is the *receiver operating characteristic (ROC) curve*, which allows to represent the balance of errors graphically, see Figure 3.2.

Remember that our classifier has a hyperparameter (the threshold τ) that can be tuned to fit our purpose. In Figure 3.2, we scan across a range of values of τ and plot the true positive rate (TPR) against the false positive rate (FPR) to obtain a curve that ideally bulges towards the point where TPR $= 1$ and FPR $= 0$. The area under the curve is then an overall measure of quality for the model. Trivially, one can see that we want the area under the curve to be > 0.5 since that will mean that we are performing better than a random classifier.

The ROC curve can therefore be used also to select the hyperparameter τ by establishing the balance between TPR and FNR appropriate for our problem.

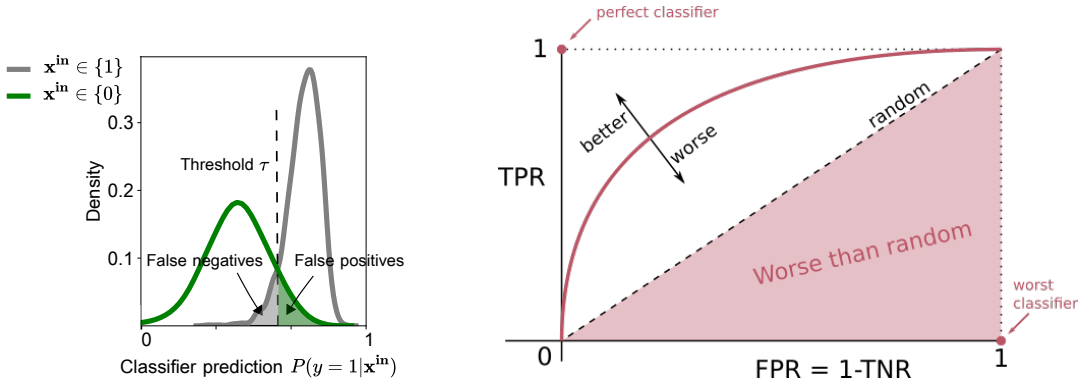


Figure 3.2. (Left) Cartoon illustration of a distribution of probabilistic predictions by a classifier. (Right) The receiver operating characteristic (ROC) curve. When scanning across a range of values for the threshold (τ) parameter of a logistic regression (or some other binary outcome model) and plotting the true positive rate (TPR) against the false positive rate (FPR), we may expect to get a curve similar to the thick upwards curving line in red. If this curve is above the diagonal (dashed) line, the model is doing better than a completely random classifier and vice versa.

Naive Bayes classifier

In this chapter, we consider a very simple multi-class classifier that is based on a different principle directly drawn from Bayes' theorem under some strongly simplifying assumptions. Even under such strong assumptions, this classifier is shown to work very well in many practical applications.

As usual, we start by stating our setup. Our input and output variables in our dataset with N samples are:

$$\begin{aligned} \mathbf{x}^{(i)} &= (x_1^{(i)}, \ldots, x_p^{(i)}) \qquad i = 1, \ldots, N \\ y^{(i)} &\in \mathcal{C}_q = \{c_1, \ldots, c_Q\} \end{aligned}$$

i.e., the outcome variable is categorical with Q classes (rather than just two), and $\mathcal{C}_q$ denotes the set of all Q classes.

In general a Bayes classifier uses for prediction the posterior probabilities, i.e. the probabilities of the output class conditional on the data features, using Bayes' theorem:

$$P(Y = c_q \mid X = \mathbf{x}) = \frac{\overbrace{P(X = \mathbf{x} \mid Y = c_q)}^{\spadesuit} \overbrace{P(Y = c_q)}^{\clubsuit}}{\underbrace{P(X = \mathbf{x})}_{\bigstar}}, \quad q = 1, \ldots, Q \tag{4.1}$$

As usual, you can think of the three ingredients of Bayes' theorem as: $\spadesuit$ = 'likelihood', $\clubsuit$ = 'prior' and $\bigstar$ = 'evidence'.

We are now interested in estimating the three expressions ($\spadesuit, \clubsuit$ and $\bigstar$) on the right hand side of equation (4.1) from our data in order to obtain the classifier's estimate. To do so, we will need to make some (strong) assumption - the so-called *'naive assumption'* - hence the name *Naive Bayes*.

Estimation of the components of the classifier (4.1)

'Prior' ($\clubsuit$): Firstly, we turn towards the *prior*, which is labelled $\clubsuit$ in the equation above.

In the absence of any further information, the prior will be given by the frequency of each class in the training data set:

$$P(Y = c_q) = \frac{\sum_{i=1}^{N} I(y^{(i)} = c_q)}{N}, \tag{4.2}$$

where, just as in the previous chapter, $I(\cdot)$ denotes the indicator function:

$$I(b) = \begin{cases} 1, & \text{if } b \text{ is true} \\ 0, & \text{if } b \text{ is false} \end{cases}$$

Of course, if there is additional information in our data set or experimental setup that can be used to inform the prior, then an appropriate prior that incorporates such information could be used instead.

'Likelihood' (♠): Secondly, we consider the second part of Bayes' theorem's numerator, labelled ♠.

Key message: In the likelihood term we make the *naive assumption*, which is to assume the following:

$$P(X = \mathbf{x} \mid Y = c_q) = P(X_1 = x_1, X_2 = x_2, \ldots, X_p = x_p \mid Y = c_q) = \prod_{j=1}^{p} P(X_j = x_j \mid Y = c_q) \tag{4.3}$$

i.e., the independence of the p descriptors conditionally on the class.

Generally, this assumption is only an approximation, which in practice is very convenient and works well in many instances. **A key advantage conferred by the naive assumption is scalability**, as it enables estimates for each of the descriptors to be computed separately, without trying to fit all possible correlations among them (indeed, the latter becomes quickly computationally expensive with large p and requires many data points to be statistically accurate). It also allows to deal with mixed-type variable vectors seamlessly (e.g. with both continuous and discrete features).

To estimate (4.3), there are two standard options:

- For discrete variables, we can simply count the cases in the data:

$$P(X_j = x_j \mid Y = c_q) \approx \frac{\sum_{i=1}^{N} I(x_j^{(i)} = x_j \wedge y^{(i)} = c_q)}{\sum_{i=1}^{N} I(y^{(i)} = c_q)}, \quad j = 1, \ldots, p \tag{4.4}$$

  where $\wedge$ denotes the Boolean operation 'and'. In other words, $I(a \wedge b)$ will only return 1 if both a and b are true.

  Many entries might end up being empty (due to finite sampling), and so to account for this, one can use *Laplace smoothing*:

$$P(X_j = x_j \mid Y = c_q) \approx \frac{\sum_{i=1}^{N} I(x_j^{(i)} = x_j \wedge y^{(i)} = c_q) + 1}{\sum_{i=1}^{N} I(y^{(i)} = c_q) + p}$$

  This is a standard technique in statistical sampling to avoid having to deal with zero probabilities.

- Alternatively, for continuous input data, we can assume a distribution for the data and then construct estimators for the parameters of the distribution.

  A typical assumption, unsurprisingly, is that the data within each class follows a Gaussian distribution. This leads to what is called *Gaussian naive Bayes*. Specifically, assuming that each feature of the data is Gaussian in each class:

$$P(X_j = x_j \mid Y = c_q) \approx \frac{1}{\sqrt{2\pi\sigma_{j,q}^2}} e^{\frac{-(x_j - \mu_{j,q})^2}{2\sigma_{j,q}^2}}, \quad j = 1, \ldots, p \tag{4.5}$$

where we have the standard sample estimators for the mean and variance for each class q, which can be computed from our data:

$$\mu_{j,q} = \frac{1}{N_q} \sum_{y^{(i)} \in c_q} x_j^{(i)}$$

$$\sigma_{j,q}^2 = \frac{1}{N_q} \sum_{y^{(i)} \in c_q} (x_j^{(i)} - \mu_{j,q})^2$$

Here N_q is the number of samples in class q.

Normalisation factor (★): Finally, we are left with the denominator of Bayes' theorem, labelled with ★.

Using the law of total probability, we can directly rewrite the normalisation ★ in terms of our previous two estimates ♣ and ♠:

$$\begin{aligned} P(X = \mathbf{x}) &= \sum_{q=1}^{Q} P(X = \mathbf{x} \mid Y = c_q) P(Y = c_q) \\ &= \sum_{q=1}^{Q} [\clubsuit \cdot \spadesuit]_q \end{aligned}$$

This last step is just to show that our probabilities are properly normalised. Indeed, putting it all together, we end up with the following expression for the Naive Bayes classifier:

$$P(Y = c_q \mid X = \mathbf{x}) = \frac{P(Y = c_q) \prod_{j=1}^{p} P(X_j = x_j \mid Y = c_q)}{\sum_{q=1}^{Q} [\clubsuit \cdot \spadesuit]_q} = \frac{[\clubsuit \cdot \spadesuit]_q}{\sum_{q=1}^{Q} [\clubsuit \cdot \spadesuit]_q} \tag{4.6}$$

Making predictions with Naive Bayes. For a new sample $\mathbf{x}^{\text{in}}$, the Naive Bayes classifier predicts the conditional probability (4.6), thus it returns vector $\boldsymbol{\pi}^{(\text{NB})}$ of the probabilities that $\mathbf{x}^{\text{in}}$ belongs to each of the Q classes:

$$\boldsymbol{\pi}^{(\text{NB})} = \begin{bmatrix} \pi_1^{(\text{NB})} \\ \vdots \\ \pi_Q^{(\text{NB})} \end{bmatrix}, \text{ where each component is:}$$

$$\pi_q^{(\text{NB})} = P(Y = c_q \mid X = \mathbf{x}^{\text{in}}) = \frac{[\clubsuit \cdot \spadesuit]_q}{\sum_{q=1}^{Q} [\clubsuit \cdot \spadesuit]_q}$$

Clearly, we have the following property by construction:

$$\sum_{q=1}^{Q} \pi_q^{(\text{NB})} = 1$$

As in previous cases of classifiers, our output is actually a *probabilistic assignment* (soft classification). In order to get a discrete class prediction (hard classification), we need to take a discretisation, usually by taking the class with the largest probability (i.e., the argmax choice):

$$\hat{y}^{(\text{NB})} = \operatorname*{argmax}_q \boldsymbol{\pi}^{(\text{NB})}$$

Chapter 5

Decision trees and random forests

In this chapter, we will be introducing a very powerful and popular method called *random forests*. In order to do so we first need to start with *decision trees*, since they are the fundamental building blocks of random forests: a random forest is 'set of trees', as we will see.

1. Decision trees

As usual, we start with the following setting:

$$\mathbf{x} = (x_1, ..., x_p) \rightarrow y$$

We need not place restrictions on the type of data we can handle with decision trees. That is to say, the input data can be either discrete or continuous. The same can be said for the output variable y. Hence, **decision trees (and random forests) can be used for both classification and regression tasks**. We first consider the problem of classification (i.e., with categorical output variable), before moving on to regression (i.e., quantitative output variable).

1.1. Classification task: Given a new (unseen) sample $\mathbf{x}^{\text{in}}$ with p descriptors, our objective is to predict its class $y \in \mathcal{C}_q = \{c_1, \ldots, c_Q\}$.

The intuition behind a decision tree is to partition the space of predictors and classify according to the splits - see Figure 5.1. Briefly, a decision tree will successively split the input space into regions. This can be visualised as a binary tree: each region is assigned a node, also known as *leaf nodes*. At any depth of the tree, the leaf nodes correspond to regions whose union gives us the full input space. The key aim is to **find a set of splits into regions separating samples of different classes into different regions**, while keeping samples of the same class in the same region (as much as possible).

Setting up the problem: Each split can be thought of as a *decision* that generates a tree. The split is strictly limited to <u>one</u> input variable at a time and can be defined by a

pair (j, s), where j is the input variable used for the decision and s is the threshold chosen:

$$\text{split } (j,s) = \left\{ \begin{array}{l} \{x_j : x_j < s\} =: R_1(j,s)\} \\ \{x_j : x_j \geq s\} =: R_2(j,s)\} \end{array} \right\}$$

which splits the space of input feature x_j into two regions $R_1(j, s)$ and $R_2(j, s)$.

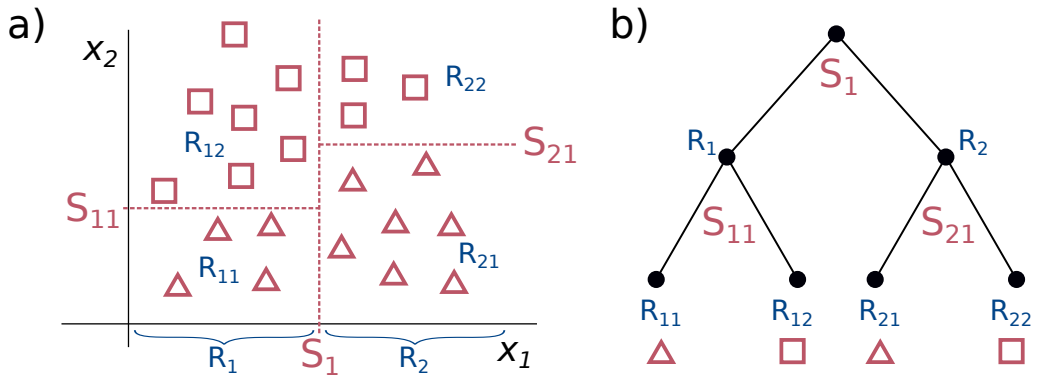


Figure 5.1. Visual intuition of a decision tree in a classification task. **a)** Using a simple case of two predictor variables as well as one categorical outcome with two classes, we can see that a decision tree is nothing other than a partition of the input variables' space. **b)** By computing different thresholds in a binary-tree-like manner, we find regions which contain only one class of data points. The corresponding prediction is denoted by the class symbols below.

Figure 5.1 shows us the intuition of how successive splits can be used to find regions for classification. In that case, we start with the S_1 split, which is only a 'decision' on the x_1 variable, denoted by $(j = 1, s = S_1)$. This is followed by S_{11} and S_{21}, which are solely based on x_2. If we choose the thresholds appropriately in each of the splits S_1, S_{11}, S_{21}, we find regions where we have a majority of triangles or squares. Indeed, it's almost always impossible to achieve a perfect partitioning , i.e., with 100% of cases in each leaf node being of a single class, and a majority rule will need to be applied.

Key message: A *decision tree* is given by the set of splits (j, s) associated with the branching points of the binary tree. Given a new sample, we use this binary tree to determine in which region the new instance falls, to give a prediction for the class based on the majority of samples in that region.

The learning task, in this case, is about making these successive decisions, i.e., choose a descriptor and find a threshold in that descriptor that will improve the tree's capability to separate the classes in the training set.

The algorithm. Since evaluating all possible combinations of splits across a number of levels of splits is a combinatorially and computationally intractable problem, **we utilise a *greedy* algorithm** that computes optimally one level at a time. In other words, the algorithm will first determine the first split optimally (i.e. S_1 in Figure 5.1), and then move on to the next level (i.e. S_{11} and S_{21}) to obtain optimal splits, but without reconsidering other possibilities for S_1. This helps keep the computational cost low and works quite well nonetheless in most cases. Note that in order to find an optimal split, the algorithm is not

limited to using any particular input variable, but rather chooses the optimal variable for the optimal split regardless of whether that variable was used before or not.

To determine the feature j and the threshold s for an optimal split at each step, we need to define a loss (or cost) function, just like we have been doing throughout the course.

The loss/cost function. The loss function is based on the classification quality achieved by a possible split (recall our discussion in Chapter 3 on the confusion matrix and the measure of its quality). Here, we will expand on those concepts with related measures imported from information theory.

We will need a few definitions. For a given region $R_{\boldsymbol{\alpha}}$ and a particular class c_q, the probability of being in that region and belonging to that class is:

$$\pi_q(R_{\boldsymbol{\alpha}}) = \frac{\sum_{i=1}^{N} I\big(\mathbf{x}^{(i)} \in R_{\boldsymbol{\alpha}} \wedge y^{(i)} \in c_q\big)}{\sum_{i=1}^{N} I(\mathbf{x}^{(i)} \in R_{\boldsymbol{\alpha}})}$$

This can now be computed for each class $q = 1, \ldots, Q$ and summarised in a $Q \times 1$ vector of probabilities:

$$\boldsymbol{\pi}(R_{\boldsymbol{\alpha}}) = \begin{bmatrix} \vdots \\ \pi_q(R_{\boldsymbol{\alpha}}) \\ \vdots \end{bmatrix}$$

where each component of the vector corresponds to the proportion of class q observations in the region (or node) with index α and is also called the *impurity* of node α.

Now, we can define the cost function for the splits in the decision tree. The two most broadly used measures of classification quality are the *Gini index* and the *cross-entropy*, which are both information theoretical measures. With these, **we want to achieve maximal *information* in our split**.

The Gini index is defined as follows, using notation from above:

$$\mathrm{GI}\big[\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})\big] = \sum_{q=1}^{Q} \pi_q(R_{\boldsymbol{\alpha}})\big(1 - \pi_q(R_{\boldsymbol{\alpha}})\big)$$

Similarly, the cross entropy is defined as:

$$\mathrm{CE}\big[\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})\big] = -\sum_{q=1}^{Q} \pi_q(R_{\boldsymbol{\alpha}}) \log \pi_q(R_{\boldsymbol{\alpha}})$$

which corresponds to the entropy of the distribution $\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})$.

To better understand the logic of using these information-theoretic measures as optimisation criteria in machine learning, let us consider a simple case with $Q = 2$ classes and let us focus on only one region α (hence we will drop the index α below). We will call π the proportion of data in class '1', so that the proportion of data in class '2' is given by $(1 - \pi)$. In this simple case we then have:

$$\begin{aligned} \mathrm{GI}\big[\pi\big] &= 2\pi\big(1 - \pi\big) \\ \mathrm{CE}\big[\pi\big] &= -\pi \log \pi - \big(1 - \pi\big) \log \big(1 - \pi\big) \end{aligned}$$

Both these functions are plotted in Figure 5.2, showing that they are maximal when $\pi = 0.5$, i.e., when the proportion of data in one region is the same for all classes (see *Quick aside 1* for a brief illustration of this fact for the GI). Conversely, both the Gini index and CE decrease when π and $(1 - \pi)$ are more dissimilar, hence where the distribution of data points upon splitting increases the information about classes.

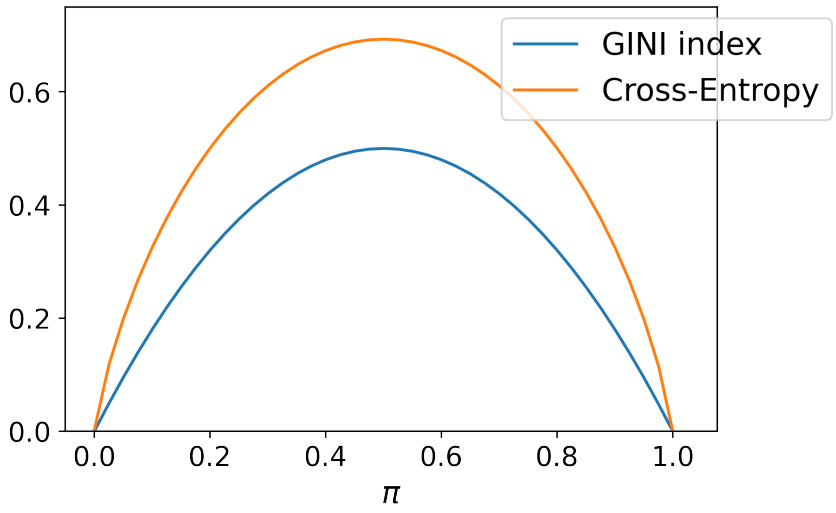


Figure 5.2. Gini index and entropy (or cross-entropy) for a two-class case.

Quick aside 1. Let us quickly illustrate how to show that the GI is maximised when all the p_i's are equal, i.e., for a uniform distribution across classes. For J classes, i.e. $i \in \{1, \dots, J\}$ and a corresponding probability vector $\mathbf{p} = (p_1, \dots, p_J)^T$, we have:

$$\mathrm{GI}(\mathbf{p}) = \sum_{i=1}^{J} p_i(1 - p_i) = 1 - \sum_{i=1}^{J} p_i^2$$

To show this, we use the method of Lagrange multipliers, a classic method to carry out optimisation of functions under constraints.

We will maximise GI under a constraint that enforces the normalisation of probabilities. To do so, we define a Lagrangian:

$$L = \mathrm{GI}(\mathbf{p}) - \lambda \left(\sum_{i=1}^{J} p_i - 1 \right)$$

and obtain the stationary point:

$$\left\{ \begin{array}{l} \nabla_{\mathbf{p}} L = -2\,\mathbf{p}^\star - \lambda^\star\,\mathbf{1} = \mathbf{0} \\ \dfrac{\partial L}{\partial \lambda} = \displaystyle\sum_{i=1}^{J} p_i^\star - 1 = \mathbf{1}^T \mathbf{p}^\star - 1 = 0 \end{array} \right\}$$

which gives finally

$$\mathbf{p}^\star = \frac{-\lambda^\star}{2}\mathbf{1} \quad \text{and} \quad \lambda^\star = \frac{-2}{J}$$
$$\Longrightarrow \; p_i^\star = \frac{1}{J}, \; \forall i$$

and one can further prove that this stationary point is a maximum.

Therefore, **the Gini index and the CE can be used to measure the deviation from a uniform probability distribution**, exactly like entropy, whose interpretation in terms of information is well established in information theory (and the CE is an entropy at the end of the day, see *Quick aside 2*). As such, these measures play the role of **functions to optimise in order to perform a classification task**, whose objective is to obtain a vector of class-assignment probability that is as concentrated on one class as possible.

(Note: it would be perfectly legitimate to use for this purpose classification metrics such as the misclassification error (given by 1-accuracy). In practice, however, the misclassification error has a few shortcomings compared to both the Gini index and cross-entropy (e.g., it is not differentiable everywhere) and hence is much less used, see e.g. Section 9.2 of Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*).

Quick aside 2. The terminology "cross-entropy" requires some clarification. Consider a certain region $R_{\boldsymbol{\alpha}}$, where the node impurity $\pi_q(R_{\boldsymbol{\alpha}})$ is given by (1.1). Let us denote by $\hat{\pi}_q(R_{\boldsymbol{\alpha}})$ the model probability that points in $R_{\boldsymbol{\alpha}}$ belong to class q. Then the cross-entropy between the *data distribution* and the *model distribution* is given by:

$$\mathrm{CE}\big[\boldsymbol{\pi}, \hat{\boldsymbol{\pi}}\big] = -\sum_{q=1}^{Q} \pi_q(R_{\boldsymbol{\alpha}}) \log \hat{\pi}_q(R_{\boldsymbol{\alpha}}) \tag{5.1}$$

Typically in machine learning the cross-entropy is used as a loss function to be minimised in order to set the optimal model distribution, because it is easy to show (try it as an exercise) that the CE is *minimal* when the model distribution is equal to the data distribution.
By substituting the model distribution $\hat{\boldsymbol{\pi}}$ with the data distribution $\boldsymbol{\pi}$ we then obtain:

$$\mathrm{CE}\big[\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})\big] = -\sum_{q=1}^{Q} \pi_q(R_{\boldsymbol{\alpha}}) \log \pi_q(R_{\boldsymbol{\alpha}}) \tag{5.2}$$

This is the reason why the CE-based splitting criterion in decision trees consists eventually of minimising the entropy of the data distribution $\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})$.

Implementation in the algorithm. *Quick Aside 1* above tells us that the Gini index (or the cross-entropy) is maximal when all probabilities in the vector are equal, i.e. no information was gained, which is precisely what we do not want. Therefore, **the greedy estimation algorithm will try to minimise the Gini index (or cross-entropy)**. This means that at given split that would produce two daughter nodes corresponding to the regions R_1 and R_2, we find (j, s) such that:

$$\min_{j,s} \Big[\mathbb{1}_{R_1} \mathrm{GI}\big(\boldsymbol{\pi}(R_1), j, s\big) + \mathbb{1}_{R_2} \mathrm{GI}\big(\boldsymbol{\pi}(R_2), j, s\big) \Big]$$

where $\mathbb{1}_{R_1}$ and $\mathbb{1}_{R_2}$ is the fraction of points from the parent node that ends up in, respectively, R_1 and R_2.

Hence, the algorithm proceeds by greedily finding splits that will minimise the Gini index (or the cross-entropy, since both have similar properties). By finding the (j, s) split that minimise such quantities, we obtain regions where the distribution of points across classes is maximally inhomogeneous and hence dominated by data belonging only to one class. You will explore this implementation in your Python notebook.

The splits will continue until a stopping criterion is met. Examples of stopping criteria:

- plateau in the optimisation, i.e., no gain in the quality of the classification;
- small number of points in region, i.e., we establish an *a priori* limit in how finely we will split our regions so that the samples are not too sparse;
- maximum depth of the tree, i.e., we establish *a priori* a limitation on the number of splits;
- and many others derived from practice.

Once the algorithm to estimate the decision tree from the data has terminated, we have our model. To make predictions from the decision tree we have the following:

(1) Given a new data point $\mathbf{x}^{\text{in}}$,
(2) Find $R_{\boldsymbol{\alpha}}$, such that $\mathbf{x}^{\text{in}} \in R_{\boldsymbol{\alpha}}$,
(3) Obtain $\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})$,
(4) $\hat{y} = \hat{f}^{\text{DT}}(\mathbf{x}^{\text{in}}) = \text{argmax}_q \, \boldsymbol{\pi}\big(R_{\boldsymbol{\alpha}}(\mathbf{x}^{\text{in}})\big)$.

In words, find the region in which the new sample lies, and assign the majority class of the samples in that region.

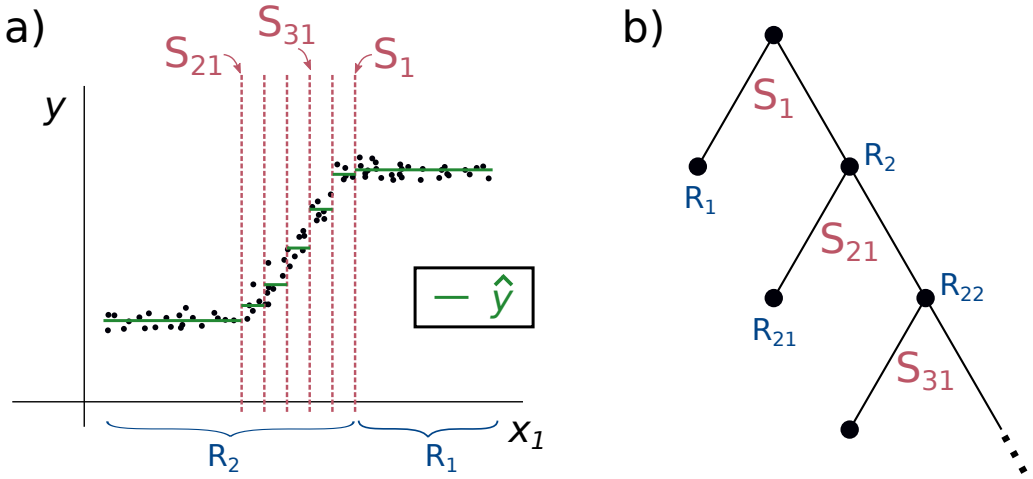


Figure 5.3. Visual intuition of a decision tree in a regression task. **a)** Given some data consisting of one input and one output variable, a decision tree will partition the space of x_1 into chunks where the data is relatively level (this vague expression will be made more precise later on). When given a new input instance, the prediction will correspond to the average of the particular region, shown as green lines. **b)** Similar to the classification case, this can also be drawn as a binary tree where each decision corresponds to one region in the input space.

1.2. Regression task: In this case, our task is to predict the continuous outcome variable $y \in \mathbb{R}$. Much of what we saw in the previous section regarding the intuition and setup of a decision tree still holds for the case of doing regression with decision trees. Just like before, the input space is divided into regions through a series of decisions, which will yield a corresponding binary tree. For some visual intuition, we use the simplest case of one continuous input and one continuous output variable. See Figure 5.3 as well as its caption for a more in-depth explanation. We chose a non-linear data set as an example to highlight the fact that decision trees are good at finding trends and making predictions in highly non-linear data.

The only difference in the procedure lies in the cost function that is optimised at each split. The greedy algorithm will choose (j, s) that minimises the following loss function, called the Residual Sum of Squares (RSS). For a split (j, s) giving rise to regions $R_1(j, s)$ and $R_2(j, s)$ reads:

$$\min_{j,s} RSS(\mathbf{y}; j, s), \qquad RSS(\mathbf{y}; j, s) = \left[\sum_{\mathbf{x}^{(i)} \in R_1(j,s)} (y^{(i)} - \overline{y}_{R_1})^2 + \sum_{\mathbf{x}^{(i)} \in R_2(j,s)} (y^{(i)} - \overline{y}_{R_2})^2 \right]$$

where $\overline{y}_{R_1}$ is the mean of the output variable in region R_1, and similarly for $\overline{y}_{R_2}$:

$$\overline{y}_{R_1} = \frac{\sum_{i=1}^{N} I(x_j^{(i)} < s) \cdot y^{(i)}}{\sum_{i=1}^{N} I(x_j^{(i)} < s)}$$

The splits aim at producing regions of the input space such that, within each, the values of the outcome are rather homogeneous. The splits continue, as indicated above, until a stopping criterion is met (usually plateau in the optimisation, or reaching a pre-established depth of the tree or a number of minimal points in a region).

Finally to obtain a prediction for the regression problem using a decision tree, we compute the average of the outcome variable in the particular region where the new sample lies. In other words, given a new instance $\mathbf{x}^{\text{in}}$, we find the region R_α in which it lies and obtain

$$\hat{y}^{\text{DT}}(\mathbf{x}^{\text{in}}) = \overline{y}_{R_\alpha} \text{ with } \mathbf{x}^{\text{in}} \in R_\alpha$$

2. Random Forests

2.1. Bagging. A major flaw of decision trees is that they tend to overfit. The smallest changes in the input data or any new training data will usually lead to vastly different trees. In order to overcome this shortcoming, we introduce a certain type of randomised algorithms called *ensemble learning/methods*. The idea is that through randomisation, we are able to reduce the variability in the estimator arising from new data. There exist a large variety of ensemble methods, amongst which are those that mix different models or a method called *boosting*.

Here we introduce another specific, possible the most popular type of ensemble learning, namely the *bagging* method, which is short for *Bootstrap aggregating*. The 'bootstrap' is a general procedure for assessing statistical accuracy of quantities estimated from a dataset, based on drawing a set of random samples from the original dataset and repeating the estimation for each of them (see Hastie, Tibshirani, Friedman, *The Elements of statistical learning*, Chap. 7, section 7.11, for more details). Note that here we apply *bagging* to decision trees specifically, but the method can be applied to almost any classification or regression model, and even to different models at the same time.

Starting with our data sample, let's call it $\mathbf{Z} = \{\mathbf{x}^{(i)}, y^{(i)}\}, i = 1, \ldots, N$, we have the following algorithm:

(1) **Bootstrapping:** Produce B random samples $\mathbf{Z}^{*b}$ for $b = 1, \ldots, B$ from $\mathbf{Z}$, all of size $N' \leq N$, by random sampling with replacement. (Sampling with replacement will lead to each $\mathbf{Z}^{*b}$ potentially containing repeated samples and/or not containing certain samples (absent samples) from the original $\mathbf{Z}$).

(2) **Ensemble of models:** In bootstrap methods, one re-obtains the quantity of interest, let's call it $S(\mathbf{Z})$, from each of the $\mathbf{Z}^{*b}$, $b = 1, \ldots, B$ - these are the bootstrap replications $S(\mathbf{Z}^{*b})$. In this case, we re-train a decision tree for each bootstrap sample, getting:

$$\left\{ \hat{f}_b^{\text{DT}} \right\}_{b=1}^{B}$$

In other words, we produce an *ensemble* of models, hence the term *ensemble learning*.

(3) **Aggregating:** We use the ensemble of models to predict our variable:

(a) *Regression:* Given a new sample $\mathbf{x}^{\text{in}}$, we simply obtain the average of all the predictions made by the ensemble of decision trees:

$$\hat{y} = \hat{f}(\mathbf{x}^{\text{in}}) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b^{\text{DT}}(\mathbf{x}^{\text{in}})$$

(b) *Classification:* In this case we will get a collection of B probability vectors, each of them being Q-dimensional:

$$\mathbf{x}^{\text{in}} \to \left[\boldsymbol{\pi}_1^{\text{DT}}, \ldots, \boldsymbol{\pi}_B^{\text{DT}}\right]$$

$$\boldsymbol{\pi}_b = \begin{pmatrix} \pi_{b,1} \\ \vdots \\ \pi_{b,Q} \end{pmatrix}$$

From this collection, we obtain the aggregated, average probability vector:

$$\boldsymbol{\pi} = \frac{1}{B}\sum_{b=1}^{B} \boldsymbol{\pi}_b^{\text{DT}}$$

And finally, our prediction is the class with the highest probability:

$$\hat{y} = \underset{q}{\operatorname{argmax}}\, \boldsymbol{\pi}$$

Whilst this does introduce a certain amount of robustness, in many cases, the bagging method will yield somewhat correlated outcomes, which is usually due to the dominance of some predictor variables. For example, if there is some predictor that is very determining, many decision trees will predominantly split along that variable. This is usually not a good property to have, since some particular predictor may be very good for describing the data at hand, but may not generalise well at all.

Furthermore, if the set of models for a given new instance are correlated, then averaging them will not reduce the variance beyond a limit, even if we increase B. Indeed, if one has B random variables (in this case, the trees), that are simply i.d. (identically distributed, but not necessarily independent), with positive pairwise correlation ρ and each of them with variance σ^2, the variance of their average as an estimator is:

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

(showing this is actually Exercise 15.1 in Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*). Thus, we need to 'uncorrelate' the trees more radically.

2.2. Random Forests (via feature bagging). *Random forests* add an additional level of randomisation, namely using the *random subspace method* (also known as *feature bagging*).

Since much of the correlated outcomes discussed above come from certain predictors being dominant, **we restrict at random the number of predictors among which the split is chosen to $m \leq p$ predictors**. This means that, compared to bagging alone (subsection 2.1), steps (1) and (3) are unchanged, while in step (2) we train every tree only on a subset of m features randomly sampled at each split, producing an ensemble of modified decision trees that are less correlated among them.

Let's summarise schematically the steps for training and evaluating a random forest:

Algorithm: Random Forests for Regression and Classification

Training.
For $b = 1$ to B:

(1) Draw a bootstrap sample $\mathbf{Z}^{*b}$ from the training data.

(2) Train a tree $\hat{f}_b^{\mathrm{DT}}$ on the bootstrapped sample as follows.
For each terminal node at the current level, repeat:
(i): Draw at random a subset of m features from the p features.
(ii): Find the optimal split (j, s) (according to the relevant cost function), choosing j only among the m features randomly sampled in *(i)*.
(iii) Perform the split into two daughter nodes.
Until the stopping criterion is reached.

Making predictions (aggregating). Given a new data point $\mathbf{x}^{\mathrm{in}}$:

(1) *Regression:*

$$\hat{y} = \hat{f}(\mathbf{x}^{\mathrm{in}}) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b^{\mathrm{DT}}(\mathbf{x}^{\mathrm{in}})$$

(2) *Classification:*

$$\hat{y} = \underset{q}{\operatorname{argmax}}\, \boldsymbol{\pi}, \text{ with } \boldsymbol{\pi} = \frac{1}{B}\sum_{b=1}^{B} \boldsymbol{\pi}_b^{\mathrm{DT}}$$

Hyperparameters. There are certain parameters that will need to be decided before running a random forest model. Some of these are the following:

- B = number of trees: There is usually no need to make this excessively big, it tends to converge fairly quickly.
- Minimum leaf size: A stopping criterion that will stop further splits as soon as the leaf reaches a minimum number of instances, with the goal of preventing overfitting. By default, this is usually set to rather small values (1-5).
- m = number of predictors sampled randomly at each split. If this is small, we obtain a large variety of decision trees, guaranteed to not have high correlations and vice versa. The guidelines by the authors of random forests is to choose:

$$\begin{cases} m \approx \frac{p}{3} & \text{for regression} \\ m \approx \sqrt{p} & \text{for classification} \end{cases}$$

Out-of-bag error. Using the bootstrapping method has many advantages, amongst which is the possibility of using *out-of-bag* (OOB) samples for error estimation. For every bootstrap sample $\mathbf{Z}^{*b}$, there are some samples that were not picked: these are OOB samples, and they play the role of unseen samples for the corresponding decision tree, hence we can compute on those an error measure (OOB error) which plays the role of a validation error. This can be useful to perform a cross-validation-type hyper-parametric search without appealing to T-fold cross-validation. To compute the OOB error, the key step is to take, as predictor for the data point $\{\mathbf{x}^{(i)}, y^{(i)}\}$, the average of predictions by only those trees corresponding to bootstrap samples in which $\{\mathbf{x}^{(i)}, y^{(i)}\}$ did not appear.

Interpretability and Variable Importance. A disadvantage of this approach of progressive partitioning of the feature space is that there is a loss of interpretability, in the sense that we lose a direct connection with the predictor variables compared e.g. to linear

regression, where the coefficient values bring information on the relevance of each predictor. But there are ways of gaining interpretability by attributing "importance" to the different predictors towards the prediction of the outcome variable, see Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*, Chapters 10.13, 15.3 for more details.

One of these ways uses precisely the OOB samples, as instances of unseen data for each tree in the forest. For each decision tree, we have a set of OOB samples, let's denote them as $\mathbf{X}^{OOB}$ and let's evaluate a metric of performance of the tree on $\mathbf{X}^{OOB}$, calling it M^0. For each feature $j = 1, ..., p$ of which we want to gauge the importance, we perform a randomisation over that feature by permuting the elements of the corresponding column $\mathbf{X}^{OOB}$, in such a way as to destroy its correlation with the other features. We re-compute the metric on this version of $\mathbf{X}^{OOB}$ with the values at column j permuted and we denote its value in this case by $M^0_{j\,\text{permuted}}$. We can take as measure of importance of feature (or variable) j for the tree's prediction the performance gap given by:

$$Importance = |M^0 - M^0_{j\,\text{permuted}}| \qquad \text{for a given tree.}$$

The absolute value ensures that the variable importance is positive (depending on the metric, the randomisation of one feature might lead to higher or lower values of the metric). Variable importance for the random forest is then found by averaging importance over the different trees of the forest; typically, importance is analysed and visualised by expressing it as a percentage of the maximal importance across features (see section 15.3.2 in see Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*).

Final Aside. A powerful alternative to standard random forests is Gradient Boosting Decision Trees (GBDTs), see Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*, Chapter 10.10 for more details. Its heuristic is the iterative addition of weak ('easy') models to a strong model aimed at reducing the error of the strong model (i.e., the model we aim to improve through this procedure).

Let's take first the example of a strong model solves a regression task minimising the squared prediction errors (like the MSE). At each iteration t, the strong model is evaluated on the training data and a weak model is trained to predict its *error*. Let's denote the prediction by the weak (respectively, strong) model at iteration t by w_t (respectively S_t). At iteration $t+1$ the strong model is updated by subtracting the weak model's prediction (so as to reduce the strong model's error):

$$S_{t+1} = S_t - \nu w_t \tag{5.3}$$

with a hyperparameter ν called shrinkage. ν is a tunable hyperparameter which plays the role of the learning rate in gradient descent, i.e., it controls the rate of updates of the strong model. The iterations are repeated until a stopping criterion is met (maximum number of iterations, signals of overfitting of the strong model).

More generally, one has the loss function of the supervised learning task under consideration: $L(y, S_t)$, which depends on the output y and the strong model's prediction at iteration t. The weak models are trained iteratively to predict the *gradient* of the loss with respect to the predictions: $w_t = \partial L(y, S_t)/\partial S_t$ and the strong predictions are updated according to (5.3). In the case of a loss given by the squared error $L(y, S_t) = (y - S_t)^2$:

$$\frac{\partial L(y, S_t)}{\partial S_t} = -2(y - S_t)$$

that is, one recovers the signed error as the learning target of the weak model. By requiring the weak model to fit the gradient of the loss that the strong model has been trained to minimise, the update of the strong model follows the direction of steepest descent of the loss.

Chapter 6

Support Vector Machines

Support Vector Machines (SVMs) are a very important tool in classification and are widely used in a variety of settings. The support vector machine (SVM) model was first introduced by Vladimir Vapnik (it is described in Vapnik, V., *The Nature of Statistical Learning Theory*, 1996, see Additional Reading). It is largely based on the idea of geometrically solving classification problems using optimisation, so, the flavour of this chapter is less statistical than the previous ones.

Vapnik's main idea was the following. Consider a binary classification problem: **a good way to classify discrete binary samples is to find a (hyper)plane that optimally separates the samples belonging to the two classes**. This objective can be posed as an optimisation problem where our outcome will be the parameters that define the hyperplane. So our target is to split our space into two half-spaces where the majority of class '1' lies in one of the half-spaces (and vice versa for class '0').

We will study the problem of *hard* classification of binary samples via the SVM. (Recall that by *hard* classification we mean i.e. a strict, non-probabilistic assignment of samples to one class, in opposition to *soft* classification, where we assign, to each sample, a probability of belonging to each class.

The SVM is a *linear* classification method, like logistic regression (Chapter 3) - this should come as no surprise as we look for a *separating hyperplane*.We will discuss later in the chapter how to extend these ideas to *soft* (i.e., probabilistic) classification, as well as to *nonlinear* classifiers.

> *Quick Aside.* SVMs are naturally binary classifiers (i.e., for two-class outcomes). With some considerable extra work (e.g. one-to-one and one-to-all strategies) that would exceed the scope of this course, they can be generalised to multi-class problems as well, but we will stick here to binary classifiers.

1. Formulation of the problem: Hard-margin SVMs

We start with our familiar setup, where the input variables are continuous and the output variable is binary:

$$\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_p^{(i)}) \in \mathbb{R}^p \quad \rightarrow \quad y^{(i)} \in \{-1, +1\}, \quad i = 1, \ldots, N$$

Without loss of generality, we have chosen -1 and +1 as our outcome 'classes' to make the mathematical derivations further down a little easier and more compact, as will become clear immediately.

To grasp the general concept visually, we consider the case where $\mathbf{x}^{(i)} \in \mathbb{R}^2$, i.e. $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)})$, which can be easily drawn as a sketch (Figure 6.1). As described above, the goal of SMV's is intuitively easy: find the hyperplane in our input space that separates one class from the other (for $\mathbf{x} \in \mathbb{R}^2$ the hyperplane is a line; for $\mathbf{x} \in \mathbb{R}^3$, the hyperplane is a plane). The hyperplane gives the "decision boundary" of this classification problem, that is, the boundary that allows us to decide to what class each data point belongs to.

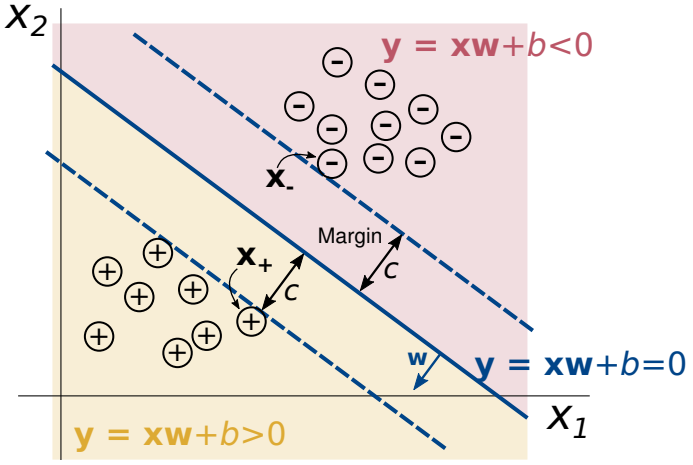


Figure 6.1. Sketch of an SVM with two predictors and a binary outcome. The separating hyperplane (here a line) is given in blue, each of its sides (coloured in either red or yellow) corresponds to a predicted class.

For $\mathbf{x} \in \mathbb{R}^2$, the hyperplane we are looking for is a line parameterised as:

$$x_2 = wx_1 + b$$

This can be written in vector notation as:

$$\mathbf{x} \cdot \mathbf{w} + b = 0$$
$$\mathbf{w} = \begin{pmatrix} w \\ -1 \end{pmatrix}$$

where $\mathbf{w}$ is the vector normal to the line (see Figure 6.1). Of course, this can be generalised to p dimensions directly. For $\mathbf{x} \in \mathbb{R}^p$, our hyperplane will be given by:

$$\mathbf{x} \cdot \mathbf{w} + b = 0$$
$$\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_p \end{pmatrix}$$

and $\mathbf{w}$ is again the vector normal to the hyperplane.

Our goal is to obtain $\mathbf{w}$ and b that parametrise our hyperplane. Once we have determined both, we can use the following rule:

Key message: the prediction by a **SVM classifier** is:

$$\text{Given } \mathbf{x}^{\text{in}}, \text{ if } \begin{cases} \mathbf{x}^{\text{in}} \cdot \mathbf{w} + b \geq 0 & \text{then } \hat{y} = +1 \\ \mathbf{x}^{\text{in}} \cdot \mathbf{w} + b < 0 & \text{then } \hat{y} = -1 \end{cases} \tag{6.1}$$

because the SVM decision boundary is the hyperplane $\mathbf{x} \cdot \mathbf{w} + b = 0$.

This corresponds to the point $\mathbf{x}^{\text{in}}$ falling in the yellow or red region, respectively, in Figure 6.1.

As can be seen in Figure 6.1, there could be an infinite number of lines that would separate the data. Intuitively, the optimally separating line is here meant as the one that allows for the 'widest street' possible through the data. SVMs approach this problem through the concept of *margin*, which denotes the smallest distance between the decision boundary (the line in the simple example of Figure 6.1) and any of the samples. Maximising the margin is what gives of the 'widest street' between data points, hence, **in SVMs, the decision boundary is chosen to be the one maximising the margin in any direction**.

Let's look at this in a few steps focussing on the example of Figure 6.1. First of all, the line that is maximally far away from both sets of data points is half-way in-between, hence we have used the same symbol c to denote the magnitude of the margin on both sides. Let $\mathbf{x}_+$ be the closest data point of class +1 to the dividing line and similarly for $\mathbf{x}_-$ for class -1 (Figure 6.1). Then we have the following:

$$\begin{cases} \mathbf{x}_+ \cdot \mathbf{w} + b = c \\ \mathbf{x}_- \cdot \mathbf{w} + b = -c \end{cases} \tag{6.2}$$

Geometrically, the width of the 'street' between points is given by:

$$\text{width} = (\mathbf{x}_+ - \mathbf{x}_-) \cdot \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

and it follows immediately from (6.2) that

$$\text{width} = \frac{2c}{\|\mathbf{w}\|}$$

Note that c is arbitrary through re-scaling of the data. Therefore, it can be fixed to be unity:

$$c := 1$$

and we finally have:

$$\text{width} = \frac{2}{\|\mathbf{w}\|} \tag{6.3}$$

This expression tells us that in order to maximise the width (and hence the margin), we have to minimise $\|\mathbf{w}\|$.

We call the **margin constraint** the condition that the margin should lie entirely between training data points, or, in other words, that training data points should be outside of (or, at most on) the margin. Mathematically:

$$(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \geq 1 \quad \text{if} \quad \mathbf{x}^{(i)} \in \{1\}, \qquad (\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \leq -1 \quad \text{if} \quad \mathbf{x}^{(i)} \in \{-1\}$$

Our choice of notation for the outcome classes as $\{-1, +1\}$ now comes in handy, as it allows us to write the margin constraint compactly as:

$$y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \geq 1$$

This, combined with (6.3), allows us to obtain:

Key message: the hard classification SVM optimisation problem is:

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to} \quad y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \geq 1, \qquad i = 1, \dots, N \tag{6.4}$$

In words, we are looking to minimise $\|\mathbf{w}\|$ such that all points $\mathbf{x}^{(i)}$ are above or below (depending on their class) the dividing line $\mathbf{x}^{(i)} \cdot \mathbf{w} + b = 0$ and at a distance of at least 1 (which is the distance to the hyperplane from both $\mathbf{x}_+$ and $\mathbf{x}_-$).

Quick Aside: The solution of the optimisation problem (6.4) is not analytical, but can be obtained via numerical techniques of convex optimisation that we will not cover systematically in this course. A few pointers to **Quadratic Programming** are however important.

Quadratic programming is used when the objective is to optimise a *non-linear* (e.g. quadratic) function with respect to *linear* inequality constraints:

$$\min_{\mathbf{z}} f(\mathbf{z}) \text{ subject to } g_i(\mathbf{z}) \leq 0 \text{ for } i = 1, \dots, m$$

This is our problem (6.4), where the optimisation variables $\mathbf{z}$ are $\mathbf{w}$ and b.

The solution to this problem is given by the Karush-Kuhn-Tucker (KKT) conditions, derived first in 1939 and, in more general form, in 1951. Briefly, we construct a Lagrangian function with Lagrange multipliers $\alpha_i \geq 0$:

$$\mathcal{L}(\mathbf{z}; \boldsymbol{\alpha}) = f(\mathbf{z}) + \sum_{i=1}^{m} \alpha_i g_i(\mathbf{z})$$

Based on the KKT conditions (whose proof is beyond the scope of this course), one has that, at the minimum $\mathbf{z}^\star$, there exists

$$\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_m)$$

such that:

$$\begin{aligned}
\nabla f(\mathbf{z}^\star) + \sum_{i=1}^{m} \alpha_i \nabla g_i(\mathbf{z}^\star) = 0 \quad & \text{(Lagrangian vanishes)} \\
g_i(\mathbf{z}^\star) \leq 0 \qquad \forall i \quad & \text{(primal costraints)} \\
\alpha_i \geq 0 \qquad \forall i \quad & \text{(dual constraints)} \\
(\spadesuit) \qquad \alpha_i \cdot g_i(\mathbf{z}^\star) = 0 \qquad \forall i \quad & \text{(complementary slackness conditions)}
\end{aligned}$$

Without going into the details of this optimisation, the α_i are found by solving the *dual* problem (hence the name "dual" constraints), which is:

$$\max_{\boldsymbol{\alpha}} \left\{ \inf_{\mathbf{z}} \mathcal{L}(\mathbf{z}; \boldsymbol{\alpha}) \right\} \quad \text{subject to} \quad \alpha_i \geq 0, \forall i$$

associated to our original minimisation task (the *primal* problem). The *duality principle* states that an optimisation be formulated in two specular ways – the primal and dual formulation. In short, if the primal problem is a minimisation (like here), the dual problem is a maximisation, and viceversa (see again the book *Convex Optimisation* by Stephen Boyd for an exhaustive discussion).

As noted in the aside, one can resort to quadratic programming since we have a non-linear objective function with linear constraints. To fit the notation from the aside, we can now rewrite the problem slightly:

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 \text{ subject to } \underbrace{1 - y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b)}_{g_i(\mathbf{w}, b)} \leq 0 \text{ for } i = 1, \dots, N$$

and we construct the Lagrangian:

$$\mathcal{L}(\mathbf{w}, b; \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{N} \alpha_i \big(1 - y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b)\big)$$

As seen in the aside, we take the gradient of the Lagrangian:

$$\left\{ \begin{array}{l} \dfrac{\partial \mathcal{L}}{\partial b} = -\displaystyle\sum_{i=1}^{N} y^{(i)} \alpha_i \\ \nabla_{\mathbf{w}} \mathcal{L} = \mathbf{w} - \displaystyle\sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)} \end{array} \right\}$$

At the minimum, the equations above equal to 0 and so we get:

$$\left\{ \begin{array}{l} \displaystyle\sum_{i=1}^{N} y^{(i)} \alpha_i = 0 \\ \mathbf{w}^\star = \displaystyle\sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)} \end{array} \right\}$$

Now we expand and rewrite $\mathcal{L}$ in terms of $\boldsymbol{\alpha}$ only:

$$\mathcal{L}(\boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + \sum_{i=1}^{N} \alpha_i - \underbrace{\sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)}}_{=\mathbf{w}} \cdot \mathbf{w} - \underbrace{\sum_{i=1}^{N} \alpha_i y^{(i)} b}_{=0}$$

And then we substitute the results from before (as can be seen from the underbraces):

$$\begin{aligned} \mathcal{L}(\boldsymbol{\alpha}) &= \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + \sum_{i=1}^{N} \alpha_i - \mathbf{w} \cdot \mathbf{w} = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\mathbf{w} \cdot \mathbf{w} \\ &= \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \end{aligned} \tag{6.5}$$

Here, the last equation was obtained by substituting $\mathbf{w}^\star = \sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)}$ into it. This expression shows that the Lagrangian can be expressed only in terms of the samples $\{\mathbf{x}^{(i)}, y^{(i)}\}$ and the Lagrange multipliers α_i, which are found (see aside) by solving the associated dual problem:

$$\max_{\boldsymbol{\alpha}} \left\{ \inf_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b; \boldsymbol{\alpha}) \right\} \quad \text{subject to} \quad \alpha_i \geq 0, \, \forall i$$

To get to the key point of Vapnik's SVM result, we look back at the aside above, and specifically at the complementary slackness conditions (♠) that appear from the KKT conditions. They imply that either the primal constraint is tight or the dual constraint is tight—one of them has to be strictly zero for the complementary slackness conditions to be verified. From that equation it then follows that:

$$\alpha_i = 0 \text{ if } \mathbf{x}^{(i)} \neq \mathbf{x}_+ \text{ or } \mathbf{x}_-$$

In words: *the Lagrange multipliers are only present to enforce the constraint at the minimal points* $\mathbf{x}_+$ *and* $\mathbf{x}_-$, that is, we recover the construction (6.2):

$$\begin{cases} \mathbf{x}_+ \cdot \mathbf{w} + b = 1 \\ \mathbf{x}_- \cdot \mathbf{w} + b = -1 \end{cases}$$

which can be used to estimate b. Putting together these different results, we obtain:

Key message: Only the two minimally distant points $\mathbf{x}_+$ and $\mathbf{x}_-$ define the vector $\mathbf{w}^\star$ and b of the margin-maximising hyperplane of a hard-margin SVM, that is:

$$\mathbf{w}^\star = \sum_{i=1}^{N} \alpha_i \, y^{(i)} \mathbf{x}^{(i)} = \alpha_+ \mathbf{x}_+ - \alpha_- \mathbf{x}_-$$

$$b = 1 - \mathbf{x}_+ \cdot \mathbf{w}^\star = 1 - \mathbf{x}_+ \cdot [\alpha_+ \mathbf{x}_+ - \alpha_- \mathbf{x}_-] = 1 - \alpha_+ \mathbf{x}_+ \cdot \mathbf{x}_+ + \alpha_- \mathbf{x}_+ \cdot \mathbf{x}_-$$

where α_+ and α_- are determined via the dual-problem formulation. The two points $\mathbf{x}_+$ and $\mathbf{x}_-$ are called the *support vectors* of the hyperplane (hence the name SVM).

Once we have obtained the parameters of the hyperplane, we can use the model to classify any new sample using the classifier (6.1).

2. Soft-margin SVMs

In this section, we will now look at ways to extend SVMs to more realistic scenarios. Indeed, the hard-margin SVM as formulated above requires the data points to be separable in p-dimensional space, but this is not possible in most applications. When the separating hyperplane does not exist, we say the hard-margin classification SVM problem is infeasible. Our formulation above contains the seeds of how to extend the method naturally to these cases when a strictly separating hyperplane does not exist, i.e. the case of *imperfect separation*. In short, **to deal with imperfect separation, we relax the optimisation by introducing a *penalty* term in the loss function (soft-margin SVM)**.

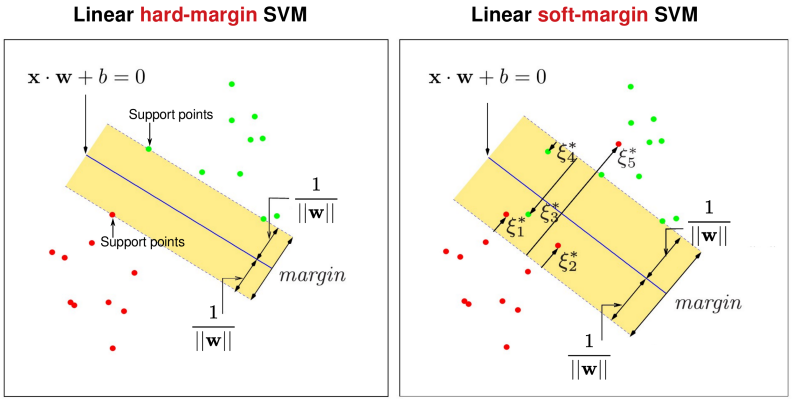


Figure 6.2. In the case of imperfect separation (right), contrary to the linearly separable case (left), one learns the hyperplane allowing some data points (here marked by ξ_j^*) to be on the wrong side of the margin (by an amount given by ξ_j^* itself). Figure adapted from Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*, Chapter 12.

As discussed above, the margin constraint for a given data point $\mathbf{x}^{(i)}$ is $y^{(i)}(\mathbf{x}^{(i)}\mathbf{w}+b) \geq 1$. Since it's impossible to find a margin-maximising hyperplane such that all the data points respect this constraint, in the soft-margin SVM optimisation one adopts a *relaxed* version of the margin constraint:

$$y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \geq 1 - \xi_i \quad \text{or, equivalently} \quad 1 - y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \leq \xi_i$$

where ξ_i represents violations of the hard margin constraint. In other words, one learns a hyperplane that still tries to maximise the margin, but also allows for points to violate

the margin constraint (i.e., to be on the wrong side of the margin, see Figure 6.2), while penalising the magnitude of this violation.

To do this, one models the violations through the *hinge loss*, defined as follows:

$$\xi_i = \max\left\{0,\ \underbrace{1 - (\mathbf{x}^{(i)} \cdot \mathbf{w} + b)y^{(i)}}_{\text{size of violation}}\right\} \tag{6.6}$$

such that $\xi_i \geq 0$ by construction (see Figure 6.3). More specifically $\xi_i \neq 0$ only for the points that violate the margin constraint, with its value reflecting the size of the violation (Figure 6.2).

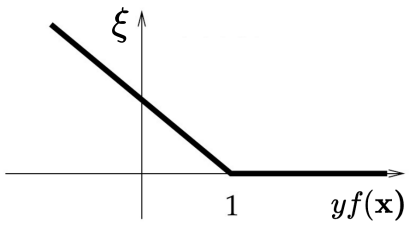


Figure 6.3. Hinge loss ξ as a function of $yf(\mathbf{x})$, where $f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{w} + b$. The soft-margin optimisation is defined in such a way as either to keep ξ at zero (for $yf(\mathbf{x}) > 1$, i.e., where the hard-margin constraint is satisfied because $\mathbf{x}$ is on the correct size of the margin) or to keep the size of the violation small (for $yf(\mathbf{x}) < 1$).

Since $\sum_{i=1}^{N} \xi_i$ measures the total magnitude of the margin constraint violation, the idea is to add this hinge term to the loss in such a way as to learn the parameters of the hyperplane while keeping the magnitude of the violation small.

Key message: The optimisation to learn the soft-margin SVM is:

$$\min_{\mathbf{w},b}\left(\frac{1}{2}\|\mathbf{w}\|^2 + \lambda\sum_{i=1}^{N}\xi_i\right) \text{ subject to } 1 - y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \leq \xi_i \text{ for } i = 1, \ldots, N$$

where ξ_i is the hinge loss (Equation 6.6, Figure 6.3) and serves to penalise margin constraint violations. The hyperparameter λ controls how 'hard' the margin constraint can be (hardness), allowing for larger or smaller violations.

This hyperparameter can be tuned by various methods such as cross-validation. In the limit of large λ, the soft-margin SVM optimisation approaches the hard-margin SVM.

Following this formulation, the same optimisation procedure relying on the dual problem described in Section 1 can be applied, simply with the additional hyperparameter λ. Its details are beyond the scope of this course, but you can read about them in Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*, Chapter 12, Section 12.2.1. In particular, from the KKT conditions in this case one finds that only for the data points $\mathbf{x}^{(i)}$ for which the relaxed margin constraint is exactly met:

$$y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) = 1 - \xi_i$$

the Lagrange multipliers are non-zero, i.e., these data points are support vectors. Thus, in the soft-margin SVM there are support vectors such that:

- $\xi_i = 0$: Support vectors on the boundary of the margin (correctly classified);
- $0 < \xi_i < 1$: Support vectors that are within the margin, and are correctly classified;
- $\xi_i > 1$: Support vectors that are within the margin, but are misclassified.

2.1. Training SVMs: gradient descent and minibatch SGD. To train an SVM, both in the hard and soft-margin version, we need to solve an optimisation problem. For instance, in the soft-margin SVM we find the optimal To train an SVM, both in the hard and soft-margin version, we need to solve an optimisation problem. For instance, in the soft-margin SVM we find the optimal $\mathbf{w}$ and b by minimising the loss function:

(6.7)
$$L(\mathbf{w}, b; \mathcal{S}) := \left(\frac{1}{2}\|\mathbf{w}\|^2 + \lambda \sum_{i=1}^{|\mathcal{S}|} \xi_i\right) \text{ subject to } 1 - y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \leq \xi_i \text{ for } \{\mathbf{x}^{(i)}, y^{(i)}\} \in \mathcal{S}$$

where we have denoted by $\mathcal{S} = \{\mathbf{x}^{(i)}, y^{(i)}\}$ the training set, of size $|\mathcal{S}| = N^{\text{training}}$. Let's denote in a compact way the model parameters as $\boldsymbol{\theta} = \{\mathbf{w}, b\}$. As we have seen in Chapter 1, the gradient $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_0; \mathcal{S})$ evaluated at $\boldsymbol{\theta}_0$ defines the direction of steepest ascent in parameter space at the point $\boldsymbol{\theta}_0$. The gradient descent algorithm takes an initial guess for the parameters $\boldsymbol{\theta}_0$ and updates, at each iteration t, the parameter values according to the rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S})$$

where $\eta_t > 0$ is a small learning rate which may depend on t. For a suitably chosen η_t, the iterations converge to a minimum for $\boldsymbol{\theta}$ - specifically, a *global* minimum for linear soft-margin SVM, as the loss is convex.

However, computing $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S})$ as above requires computing the gradients of the per-example loss for every element in the training set. For large data sets (and large models) this can be prohibitively expensive. Minibatch stochastic gradient descent (minibatch SGD) provides a cheaper estimate of the full gradient, by computing the gradient on a minibatch of data points, instead of the full data set.

> **Key message:** In minibatch SGD, the parameter update rule is:
>
> (6.8)
> $$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$$
>
> where $L(\boldsymbol{\theta}_t; \mathcal{S}_m)$ is the mean loss over $\mathcal{S}_m$, a randomly drawn sample (minibatch) of training data points, of size $|\mathcal{S}_m|$ typically much smaller than $|\mathcal{S}|$.

This update provides a stochastic approximation to the full-dataset gradient that is far more efficient to compute, and provides a huge speed up in the training process for large data sets, while still converging to the global minimum.

One can even use minibatch size $|\mathcal{S}_m| = 1$, i.e., consisting of just one data point at a time - this version is simply referred to as SGD.

You will see the implementation of minibatch SGD in the coding tasks for this week.

3. Beyond linear classification: Kernelised SVMs

The second important extension of SVMs is to consider non-linear classification, suitable for non-linearly separable problems. So far, the goal has been to find a *linear* hyper-plane to separate the data (hardly or softly). If the data is not linearly separable (even softly), then such methods will not work well (i.e., there will not be a 'good' hyperplane that can separate the classes).

To extend the method, we invoke *kernel* functions, which bring the following key advantage: **they allow one to map non-linearly separable feature space into a higher-dimensional space (the kernelised feature space) where data become linearly separable**.

3.1. Kernel functions. A kernel function is an analogue of a dot (or inner) product. In general, a kernel function maps a pair of vectors in a lower dimensional space (e.g., $\mathbb{R}^d$) to give the dot product of non-linearly transformed vectors in a higher-dimensional space ($\mathbb{R}^D$). This is a function of the following form:

$$k(\mathbf{x}, \mathbf{y}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{y}) \tag{6.9}$$

where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ (vectors of size d) and $\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{y}) \in \mathbb{R}^D$. The function k is a kernel with associated $\boldsymbol{\phi}$ if and only if k is positive semi-definite.

> *Quick Aside.* The theory of kernels is rich and very important both in mathematics and in machine learning, but in this course we only use them sparingly to study the non-linear extensions of SVMs. For further reading on kernels in relation to SVMs and machine learning generally, see Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*, Chapters 12, 5, and 14.

Some widely used kernels in applications are:

- Polynomial (of degree up to n):
$$k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + c)^n$$
- Polynomial of degree n:
$$k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^n$$
- Gaussian kernel (or radial basis function kernel):
$$k(\mathbf{x}, \mathbf{y}) = e^{-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{\sigma}}$$
- Sigmoid:
$$k(\mathbf{x}, \mathbf{y}) = \tanh\big(\beta(\mathbf{x} \cdot \mathbf{y}) + c\big)$$

Starting from these standard kernels, one can also construct new kernels by combining them according to sets of well-defined rules - see chapter 6.2 of Bishop, *Pattern Recognition and Machine Learning* - enabling additional flexibility in the kernel choice.

3.2. The kernel trick. Kernels are interesting for the following property, which we illustrate with an example. Consider the following:

$$\text{Given} \quad \mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$$
$$\text{define:} \quad \boldsymbol{\phi}(\mathbf{x}) = \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \in \mathbb{R}^3$$

Note that we have the following dot product:

$$\boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{y}) = x_1^2y_1^2 + x_2^2y_2^2 + 2x_1x_2y_1y_2$$

We can then (cleverly) define a function k such that:

$$\begin{aligned} k(\mathbf{x}, \mathbf{y}) = k(\mathbf{x} \cdot \mathbf{y}) &:= (\mathbf{x} \cdot \mathbf{y})^2 = (x_1y_1 + x_2y_2)^2 \\ &= x_1^2y_1^2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 \\ &= \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{y}) \end{aligned}$$

Hence, we do not need to compute the non-linear mapping $\boldsymbol{\phi}(\mathbf{x})$ to effectively lift our data vectors $\mathbf{x}$ to a D-dimensional space, *if* $\mathbf{x}$ *enters our framework only through pairwise dot products*, which allow us to manipulate them only implicitly. All we need to do is compute

the usually simpler dot product $\mathbf{x} \cdot \mathbf{y}$ and apply the kernel function k associated to $\boldsymbol{\phi}(\mathbf{x})$ by (6.9). Whilst this toy example was simply going from $d = 2$ to $D = 3$ dimensions (not great savings), we could have defined a kernel function in $D = 1000$ dimensions, which much greater savings.

> **Kernel trick** (or *kernel substitution*): an algorithm defined for d-dimensional vectors that can be formulated only in terms of pairwise dot products can be applied to non-linearly transformed, D-dimensional vectors (with $D >> d$ and potentially infinite) by replacing each dot product by a kernel function.

The kernel trick actually fits neatly into the SVM formulation, because all the expressions in our derivation of the optimisation contained 'dot products of vectors'. Let's point out the places where the kernel functions could be included in lieu of the dot products. We have the following model for the hard-margin SVM:

$$\mathbf{w}^\star = \alpha_+ \mathbf{x}_+ - \alpha_- \mathbf{x}_-$$

$$b = 1 - \alpha_+ \underbrace{\mathbf{x}_+ \cdot \mathbf{x}_+}_{\bigstar} + \alpha_- \underbrace{\mathbf{x}_+ \cdot \mathbf{x}_-}_{\bigstar}$$

where the coefficients α_i are found by maximising the Lagrangian (6.5):

$$\mathcal{L}(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \underbrace{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}}_{\bigstar}$$

Furthermore, the soft-margin SVM is trained by minimising the loss function:

$$\frac{1}{2} \underbrace{\mathbf{w} \cdot \mathbf{w}}_{\bigstar} + \lambda \sum_{i=1}^{N} \max \left\{ 0, 1 - y^{(i)} (\underbrace{\mathbf{x}^{(i)} \cdot \mathbf{w}}_{\bigstar} + b) \right\}$$

Finally, given a new data point $\mathbf{x}^{\text{in}}$, the classifier (6.1) is:

$$\underbrace{\mathbf{x}^{\text{in}} \cdot \mathbf{w}^\star}_{\bigstar} + b \geq 0 \implies \hat{y} = +1$$

$$\underbrace{\mathbf{x}^{\text{in}} \cdot \mathbf{w}^\star}_{\bigstar} + b \leq 0 \implies \hat{y} = -1$$

From all the starred ★ bits, we note that not only the SVM optimisation solution, but also the SVM prediction boils down to computing dot products. This is hence where we will use kernel functions to accommodate non-linear data.

3.3. Training a kernelised SVM. In the case of SVMs, we choose an appropriate kernel and train the corresponding SVM exploiting the kernel trick. To see more precisely how this optimisation is carried out, let us introduce the notation $\mathbf{w}_{\boldsymbol{\phi}}$ for the parameters of the SVM with kernel $k(\mathbf{x}, \mathbf{y})$ associated to $\boldsymbol{\phi}(\mathbf{x})$. The optimisation of the kernelised hard-margin SVM would then be written as:

$$\min_{\mathbf{w}_{\boldsymbol{\phi}}} \frac{1}{2} \|\mathbf{w}_{\boldsymbol{\phi}}\|^2 \text{ subject to } 1 - y^{(i)} (\boldsymbol{\phi}(\mathbf{x}^{(i)}) \cdot \mathbf{w}_{\boldsymbol{\phi}} + b) \leq 0 \text{ for } i = 1, \ldots, N \tag{6.10}$$

We introduce a vector $\mathbf{u}$ of size $N \times 1$ such that:

$$\mathbf{w}_{\boldsymbol{\phi}} = \sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}^{(i)}) u_i$$

If we consider the (transpose) full data matrix $\mathbf{X}^T$ (of size $p \times N$) having as columns the data points $\mathbf{x}^{(i)}$, the introduction of the vector $\mathbf{u}$ allows us to write, first:

$$\mathbf{w}_{\boldsymbol{\phi}} \cdot \boldsymbol{\phi}(\mathbf{X}^T) = \mathbf{u}^{\mathrm{T}}\mathbf{K}$$

where $\boldsymbol{\phi}(\mathbf{X}^T)$ (of size $D \times N$) gives the mapping of the data matrix onto the D transformed space. Next we have:

$$\mathbf{w}_{\boldsymbol{\phi}} \cdot \mathbf{w}_{\boldsymbol{\phi}} = \mathbf{u}^{\mathrm{T}}\mathbf{K}\mathbf{u}$$

where the $N \times N$ matrix $\mathbf{K}$ (also called Gram matrix) is defined as:

$$\mathbf{K}_{ij} = \boldsymbol{\phi}(\mathbf{x}^{(i)}) \cdot \boldsymbol{\phi}(\mathbf{x}^{(j)}) \equiv k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$$

(the Gram matrix has to be positive semi-definite). As a result of these substitutions we obtain, directly for the soft-margin kernelised SVM:

Key message: The optimisation to learn a kernelised soft-margin SVM can be formulated as:

$$\min_{\mathbf{u},b} \left(\frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathbf{K}\mathbf{u} + \lambda \sum_{i=1}^{N} \xi_i \right) \text{ subject to } 1 - y^{(i)}(\mathbf{K}^{(i)}\mathbf{u} + b) \leq \xi_i \text{ for } i = 1, \ldots, N \tag{6.11}$$

where $\mathbf{K}^{(i)}$ denotes row i of the Gram matrix $\mathbf{K}$, and the hinge loss reads:

$$\xi_i = \max\left\{0, 1 - y^{(i)}(\mathbf{K}^{(i)}\mathbf{u} + b)\right\}$$

Note here the advantage of the kernel trick: the dimension of the optimisation problem in (6.11) is N (the size of the training set), hence we don't need to specify the dimensions of $\mathbf{w}_{\boldsymbol{\phi}}$, nor to work explicitly with an optimisation of such dimension (as it would be the case instead with Equation 6.10), leaving the freedom to introduce potentially very high-dimensional transformations $\boldsymbol{\phi}(\mathbf{x})$.

Quick Aside: The kernel trick can be even more conveniently applied to the dual problem - see again Equation (6.5) - and this is often mentioned as a motivation for working with the dual formulation. In many textbooks and tutorials, you will find descriptions of kernelised SVMs based on the solution of the dual problem via quadratic optimisation since the dual form has the advantages of being, in general, faster and more computationally efficient for large datasets (due to the sparsity of the final solution). In the coding task of this course, we instead focus on the primal formulation (Equation 6.11), which can be solved by gradient-descent methods.

3.4. Why use kernelised SVMs? The idea is that non-linearity and high dimensionality can help with separability and thus, classification. For data points $\mathbf{x}, \mathbf{y}$ in $\mathbb{R}^p$, the kernel trick allows us to work with $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^D$, where $D \gg p$ potentially. Choosing the $\boldsymbol{\phi}$ with the right properties will, in principle, allow us to separate (classify) the data using a hyperplane in the lifted (higher dimensional) space.

An intuition of why the kernel trick can work is given in Figure 6.4 with an example. There we see a set of binary (red, blue) samples $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$ that we need to separate. Figure 6.4(a) shows that there is no 'good' hyperplane (line) in $\mathbb{R}^2$ that will separate the blue and red classes with low violation penalties. However, it is also clear that if we lifted our samples to $\mathbb{R}^3$ in a clever way, we would be able to separate the classes using a plane (i.e., there is a good hyperplane in $\mathbb{R}^3$). This is shown in Figure 6.4(b), where one applies the mapping $\phi(\mathbf{x}) = (x_1, x_2, x_3)$ with $x_3 = x_1^2 + x_2^2$: in the transformed space, the blue and red samples can now be separated with a plane normal to x_3. (A kernel corresponding to this transformation can be constructed via polynomial kernels.) Hence, by lifting our

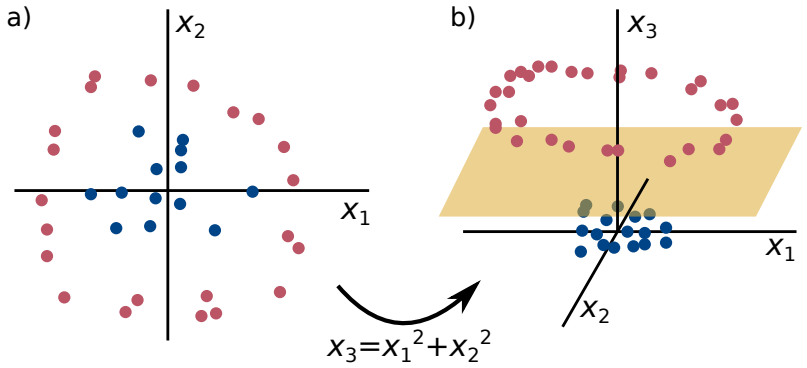


Figure 6.4. Cartoon drawing of the kernel trick. a) shows a set of binary samples that are linearly non-separable in $\mathbb{R}^2$. b) shows the kernel trick in action. Through the addition of a further dimension to the data $x_3 = x_1^2 + x_2^2$, we can make them separable with a (hyper)plane in $\mathbb{R}^3$.

data to a higher dimensional space, we were able to obtain an "easier" problem and use a separating hyperplane for classification.

This principle makes kernel SVMs very successful in applications, and thus widely used. There are typical kernels (including the ones we listed above: polynomial, radial basis functions, sigmoid kernels, etc.) which are standard choices. In many cases, these kernels can deal with nonlinearity present in the data, allowing for good hyperplane separability in the lifted, higher-dimensional space.

> *Final Aside.* Kernel versions of many other algorithms (beyond SVM) exist as a way to extend linear methods to nonlinear data. We will mention some of these methods (e.g., PCA) towards the end of the course.

Neural Networks and Deep Learning

Chapter based on notes by Dr Kevin Webster

In this chapter we introduce neural network architectures, and with them the idea of so-called *deep learning* by stacking together several layers of computing nodes.

1. The mathematical neuron

Early research into neural networks was based on models of learning in the brain using mathematical (or artificial) neurons as fundamental building blocks. These are simple information-processing units, originally meant as models of neurons in the brain, that receive a set of inputs, which are weighted and summed before being passed through an activation function to give the output y:

$$y = \sigma\left(\sum_i w_i x_i + b\right) \tag{7.1}$$

Here the inputs to the neuron are denoted by x_i, the weights by w_i, and the activation function by σ, while b stands for a bias added to the weighted sum of neurons.

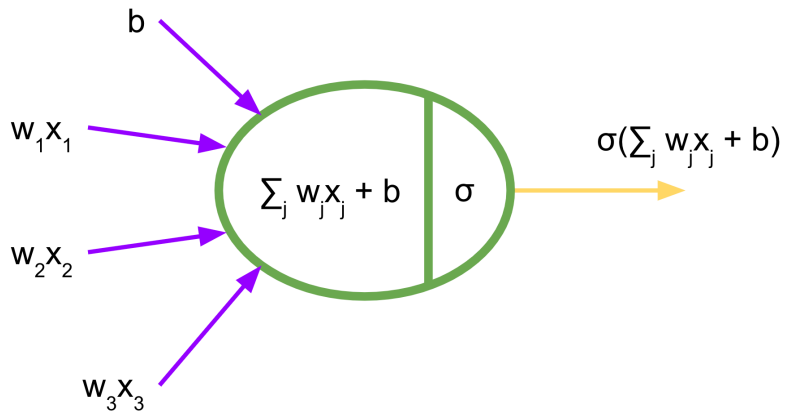


Figure 7.1. Sketch of a mathematical neuron.

The weights and bias are the parameters that need to be tuned (or *learnt*) for the given prediction task in the training of networks of such basic computing units (neural networks). This step is made, as we have seen for other methods, by optimisation of a properly defined loss function through gradient descent algorithms, which motivated research on smooth activation functions (see aside).

> *Quick Aside.* The first artificial neuron was developed by McCulloch and Pitts [9], which used a simple threshold activation function (step function) only on binary inputs, and produced a binary output. Later, Rosenblatt [14] developed the *perceptron*, which also used a step function threshold for binary classification (but with more general weights and inputs), and importantly also introduced a learning algorithm for the weights. The perceptron learning algorithm is guaranteed to converge for linearly separable data. However, the limitations of linear models was largely responsible for the decline in interest in neural networks until its revival in the 1980s. The second wave of interest in neural networks in the 80s was driven in large part by the connectionist movement (see e.g. Rumelhart et al. [15]), which focused on the concept of intelligent behaviour arising out of many simple computations composed together, with knowledge being distributed across many units. Smooth activation functions were increasingly studied, as they allowed gradient-based methods to be used in the optimisation of model parameters. For a more complete historical account on the research into neural networks, we refer to the book Goodfellow, Bengio, Courville *Deep learning* (2016), chapter 2, section 1.2.

1.1. Activation functions. A typical example of a smooth activation function is the sigmoid (or logistic function, that we have seen already in Chapter 3):

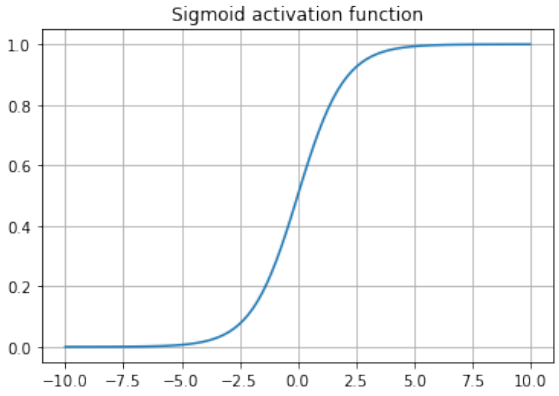


Figure 7.2. Sigmoid activation function.

Indeed, both linear regression and logistic regression can be viewed as basic artificial neuron models, with linear activation function and sigmoid activation function, respectively.

Other activation functions that later started to be commonly used in deep learning models are the ReLU (rectified linear unit), tanh, ELU (exponential linear unit, Clevert et al. 2016 [1]), SELU (scaled exponential linear unit, Klambauer et al. 2017 [6]), softplus, swish (Ramachandran et al. 2018 [13]), which can be seen in Figure 7.3.

2. Multilayer Perceptron

The simplest type of deep learning model is the **Multi-Layer Perceptron** (MLP), also simply known as a **feedforward network**. This type of neural network can be viewed as

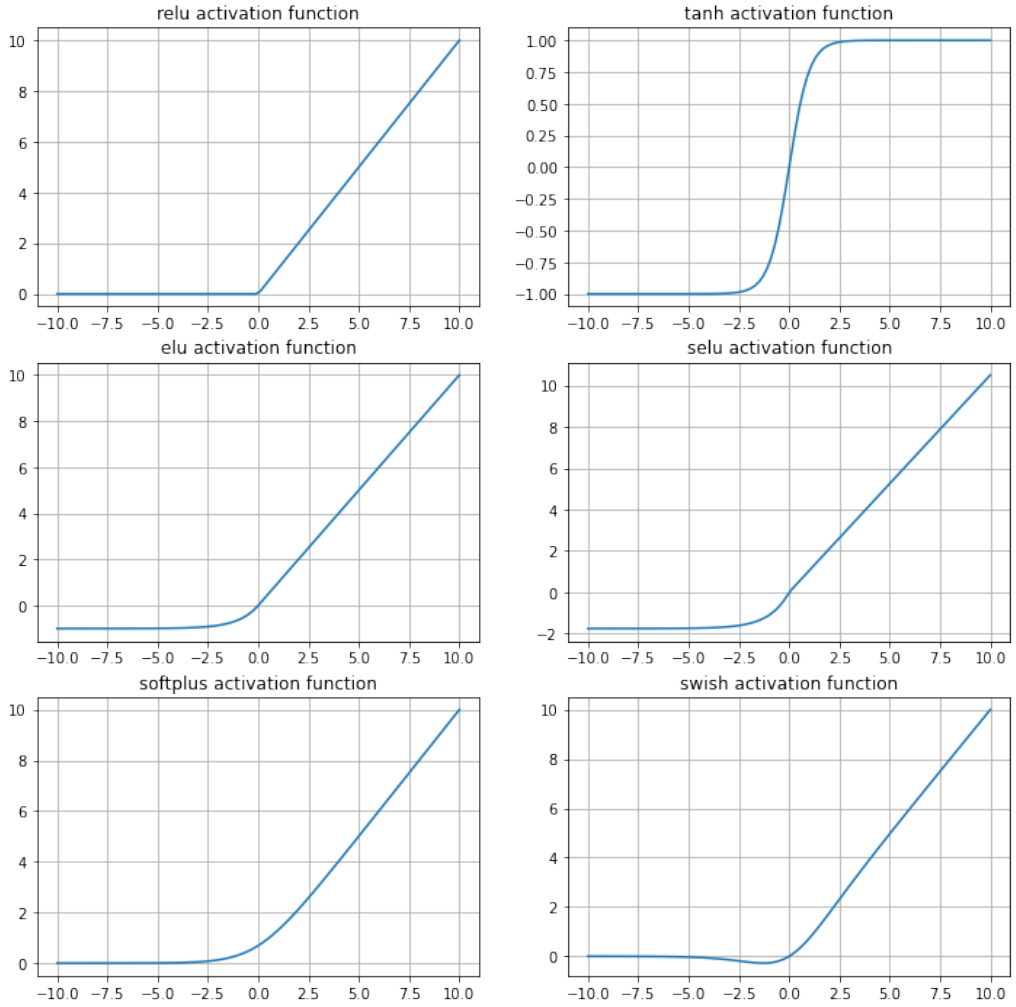


Figure 7.3. A variety of activation functions.

an architecture consisting of layers of mathematical neurons, linked together in a directed acyclic graph, i.e., by a graph where the information is processed uni-directionally from the input to the output.

2.1. MLP architecture. A key property of deep learning models is the fact that they are *compositional* instead of *additive*. By this we mean that deep learning models increase complexity by composing multiple simple functions φ_l together, to evaluate on some input data $\mathbf{x} = (x_1, \cdots, x_p)$:

$$f(\mathbf{x}) = \varphi_{N_L}(\varphi_{N_L-1}(\dots \varphi_2(\varphi_1(\mathbf{x}))\dots))$$

up to a certain N_L, which stands for the number of 'hidden' (i.e., internal) layers. The functions φ_l are defined to be affine transformations followed by an element-wise activation function.

Let's start with a simple example of such an $f(\mathbf{x})$, an MLP with a single hidden layer (i.e., $N_L = 1$):

$$h_j^{(1)} = \sigma\left(\sum_{i=1}^{p} w_{ji}^{(0)} x_i + b_j^{(0)}\right), \qquad j = 1, \ldots, n_h, \tag{7.2}$$

$$\hat{y} = \sigma_{out}\left(\sum_{j=1}^{n_h} w_j^{(1)} h_j^{(1)} + b^{(1)}\right). \tag{7.3}$$

In the above, $n_h \in \mathbb{N}$ is the number of hidden units of this (single-hidden-layer) network, $\sigma, \sigma_{out} : \mathbb{R} \mapsto \mathbb{R}$ are activation functions, $w_{ji}^{(0)} \in \mathbb{R}$ and $w_{ji}^{(1)} \in \mathbb{R}$ are weights, and $b_j^{(0)} \in \mathbb{R}$ and $b^{(1)} \in \mathbb{R}$ are biases.

This construction is summarised in Figure 7.4.

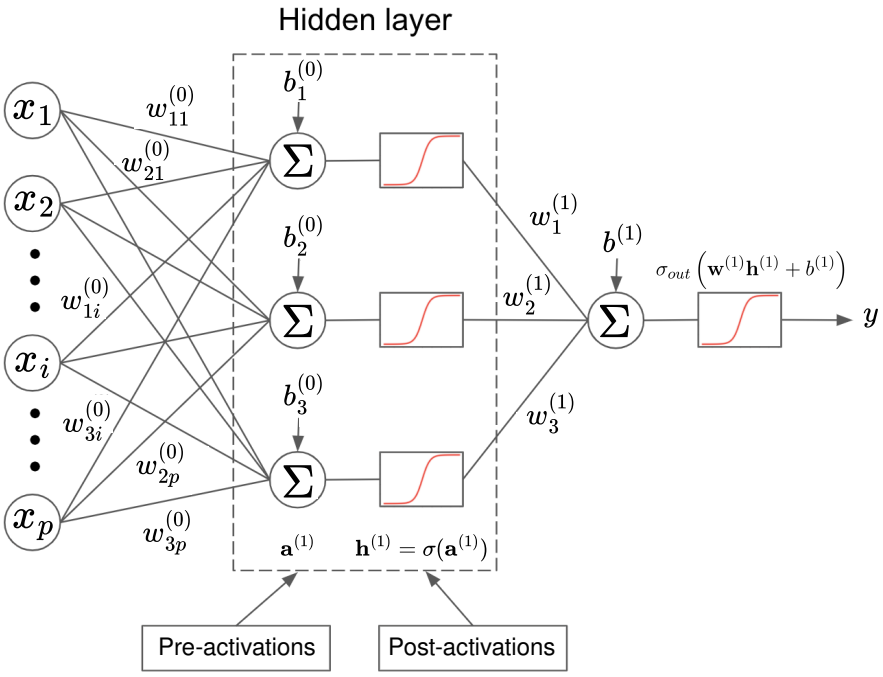


Figure 7.4. Multilayer perceptron with a single hidden layer consisting of $n_h = 3$ neurons.

We will usually write equations 7.2 and 7.3 in the more concise form:

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right), \tag{7.4}$$

$$\hat{y} = \sigma_{out}\left(\mathbf{w}^{(1)}\mathbf{h}^{(1)} + b^{(1)}\right), \tag{7.5}$$

where $\mathbf{x} \in \mathbb{R}^p$, $\mathbf{W}^{(0)} \in \mathbb{R}^{n_h \times p}$, $\mathbf{b}^{(0)} \in \mathbb{R}^{n_h}$, $\mathbf{h}^{(1)} \in \mathbb{R}^{n_h}$, $\mathbf{w}^{(1)} \in \mathbb{R}^{1 \times n_h}$, $b^{(1)} \in \mathbb{R}$ and the activation functions $\sigma, \sigma_{out} : \mathbb{R} \mapsto \mathbb{R}$ are applied element-wise in the above. This hidden layer is a type of neural network layer that is often referred to as a **dense** or **fully connected** layer, i.e., with all-to-all connections (all input features are connected to all hidden units and in turn all hidden units to the output unit by the weights).

Key message: Generalising this construction to multiple hidden layers N_L, the general MLP architecture is:

$$\mathbf{h}^{(0)} := \mathbf{x}, \tag{7.6}$$

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)}\right), \qquad l = 1, \ldots, N_L, \tag{7.7}$$

$$\hat{y} = \sigma_{out}\left(\mathbf{w}^{(N_L)}\mathbf{h}^{(N_L)} + b^{(N_L)}\right), \tag{7.8}$$

where $\mathbf{W}^{(l)} \in \mathbb{R}^{n_{l+1} \times n_l}$, $\mathbf{b}^{(l)} \in \mathbb{R}^{n_{l+1}}$, $\mathbf{h}^{(l)} \in \mathbb{R}^{n_l}$, and we have set $n_0 := p$, and n_l is the number of units in the l-th hidden layer. We define the $\mathbf{h}^{(l)}$ the **post-activations** (or frequently, just **activations**):

$$\mathbf{h}^{(l)} = \sigma(\mathbf{a}^{(l)}), \tag{7.9}$$

where:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)} \tag{7.10}$$

gives the **pre-activations**, with $l = 1, \ldots, N_L$.

Figure 7.5 shows the MLP architecture with two hidden layers.

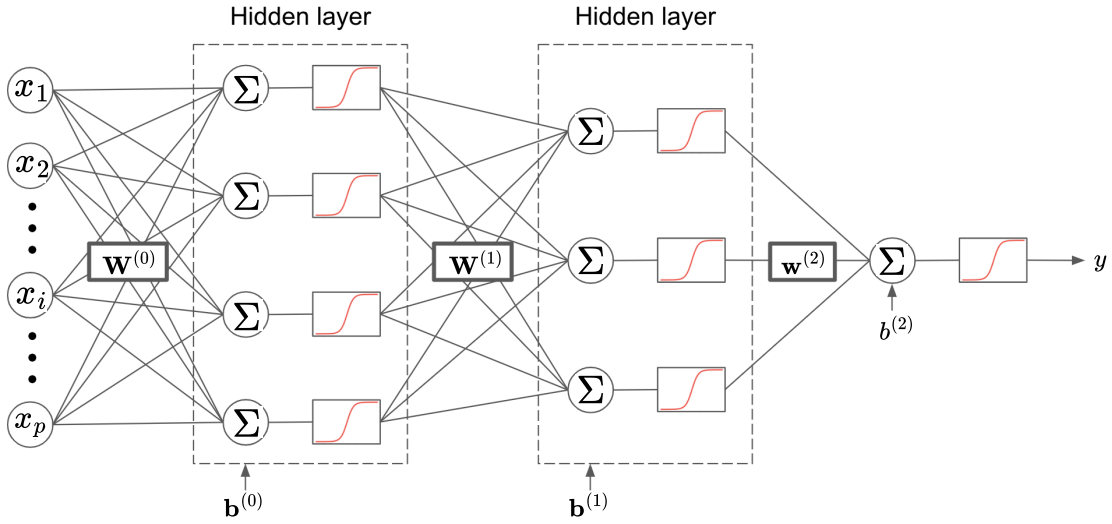


Figure 7.5. Multilayer perceptron with $N_L = 2$ hidden layers.

The hidden layers inside a deep network can be viewed as *learned feature extractors*. The weights of the network learn to encode the data in such a way as to represent progressively more complex or abstract features of the data that are useful for solving the problem task at hand. **This hierarchy of representations is a core property of the expressive power of deep learning** - see the review [7] which is also proposed in the "additional reading" folder.

2.2. Output layers. One of the strengths of deep learning models is their applicability to a wide range of data set types and problem tasks. In equation 7.8 we have considered a single unit output y, which is produced by passing the pre-activation $a^{(N_L+1)} := \mathbf{w}^{(N_L)}\mathbf{h}^{(N_L)} + b^{(N_L)}$ through the activation function σ_{out}.

Note how linear regression (chapter 1) and logistic regression (chapter 3) can both be viewed as a neural network without a hidden layer. In this case, if σ_{out} is be the identity (or linear) activation, then we are left with a simple linear regression model. Likewise, if σ_{out} is the sigmoid function, then we have the logistic regression model.

The architecture can also be easily modified to output multiple target variables $\hat{\mathbf{y}}$ by replacing Equation 7.8 with:

$$\hat{\mathbf{y}} = \sigma_{out}\left(\mathbf{W}^{(N_L)}\mathbf{h}^{(N_L)} + \mathbf{b}^{(N_L)}\right), \tag{7.11}$$

where $\hat{\mathbf{y}}$ consists of $n_{N_L+1} > 1$ elements, as can be seen in Figure 7.6 for $n_{N_L+1} = 3$.

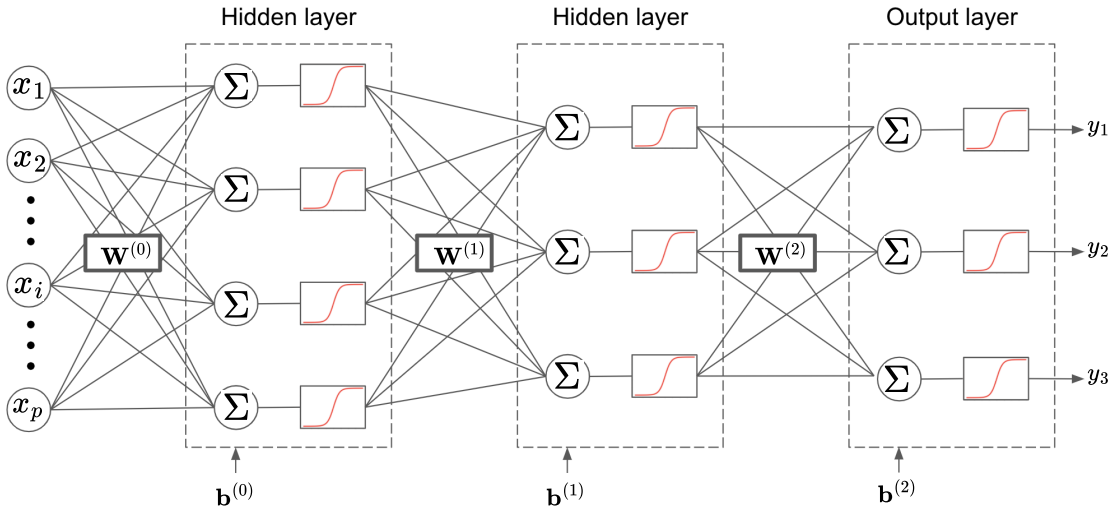


Figure 7.6. Multilayer perceptron with multiple outputs.

Moreover, the activation functions in the output layer can be chosen according to the requirements of the target variables. For example, if the network should output an estimate for a standard deviation parameter, then we will want to constrain the output to be positive. This can be achieved by passing the pre-activation through a softplus or exponential activation function, for example. It is common for a sigmoid activation to be used where the output should be interpreted as a probability (as in logistic regression). More generally, for target variables that should be constrained to an interval, then a sigmoid or tanh activation can be used followed by a suitable rescaling. Different activation functions could be applied to different units in the output layer, if appropriate.

A common output layer used for multiclass classification models is the softmax. The softmax layer outputs a normalised array, which can be interpreted as a probability vector specifying a categorical distribution. We have pre-activations:

$$\mathbf{a}^{(N_L+1)} := \mathbf{W}^{(N_L)}\mathbf{h}^{(N_L)} + \mathbf{b}^{(N_L)}$$

with $\mathbf{W}^{(N_L)} \in \mathbb{R}^{Q \times n_{N_L}}$, $\mathbf{b}^{(N_L)} \in \mathbb{R}^{Q}$ where Q is the number of classes (i.e., the number of output units). The softmax function is given by:

$$\hat{\mathbf{y}}_q := \text{softmax}(\mathbf{a}^{(N_L+1)})_q = \frac{\exp(a_q)}{\sum_{q'=1}^{Q}\exp(a_{q'})}, \qquad q = 1, \cdots, Q$$

where each element takes values between 0 and 1 and the vector is normalised over the Q classes by construction. As such, the soft-max is suitable to build a probabilistic classifier (soft classification). Note that the softmax function operates on all pre-activations in the output layer, in contrast to the usual element-wise application of most activation functions - see Figure 7.7 for a schematic.

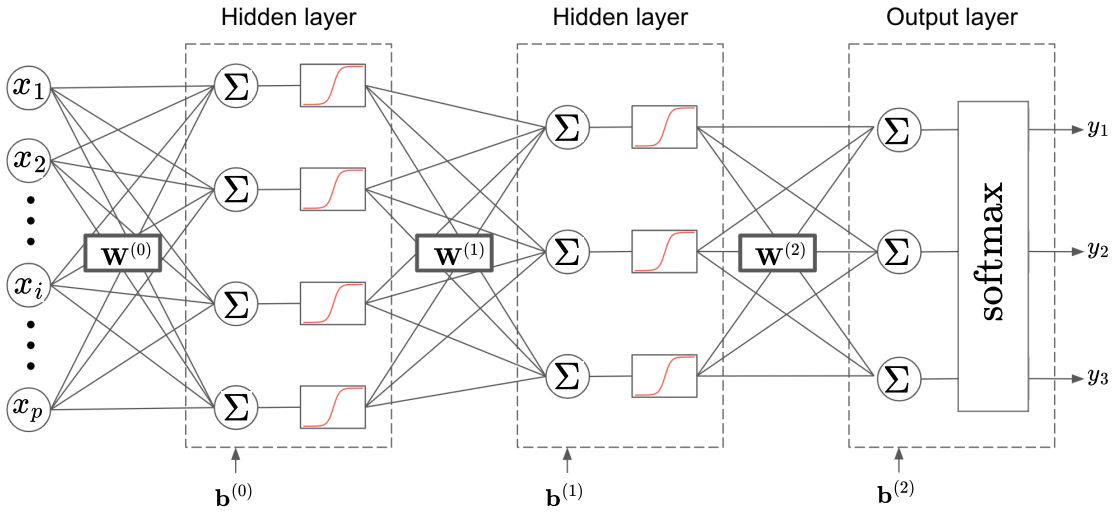


Figure 7.7. Multilayer perceptron with a softmax output layer for soft classification.

3. Neural Network training: error backpropagation

Let us consider the usual setup, where the data available for a supervised learning task is:

$$\{\mathbf{x}^{(i)}, y^{(i)}\} \qquad i = 1, \ldots, N, \qquad \mathbf{x}^{(i)} \in \mathbb{R}^p \qquad y^{(i)} \in Y$$

and where the target space Y can be either continuous (for a regression task) or categorical (for a classification task). Suppose we have constructed our neural network model, which we represent as the function $f_{\boldsymbol{\theta}} : \mathbb{R}^p \mapsto Y$, that depends on a set of parameters $\boldsymbol{\theta}$ (containing weights and biases) to be learnt. Suppose also that we have defined a suitable loss function for training:

$$L(\boldsymbol{\theta}; \mathcal{S}) := \frac{1}{|\mathcal{S}|} \sum_{\mathbf{x}^{(i)}, y^{(i)} \in \mathcal{S}} L(y^{(i)}, f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})), \tag{7.12}$$

where $\mathcal{S} = \{\mathbf{x}^{(i)}, y^{(i)}\}$ stands for the training set, with $i = 1, \ldots, N^{\text{training}}$, and where $L(y^{(i)}, f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})$ is the per-example loss. Also **in neural network architectures the parameters $\boldsymbol{\theta}$ (i.e., weights and biases) are found by solving an optimisation problem, that is, by minimising the loss function** (7.12).

To this end, we appeal to gradient-based optimisation methods, and in particular to the minibatch stochastic gradient descent (minibatch SGD) algorithm introduced in Chapter 6. Schematically, gradient-based neural network optimisation can be seen as iterating over the following two main steps:

(1) Computation of the (stochastic) gradient of the loss function with respect to the model parameters;

(2) Use of the computed gradient to update the parameters.

The parameter update rule (for step 2) according to minibatch SGD for a neural network model $f_{\boldsymbol{\theta}} : \mathbb{R}^p \mapsto Y$ is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m) \tag{7.13}$$

In the above equation, the minibatch loss at iteration t $L(\boldsymbol{\theta}_t; \mathcal{S}_m)$ is calculated on a randomly drawn sample of data points from the training set, as we have seen in Chapter

6:

$$L(\boldsymbol{\theta}_t; \mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{x}^{(i)}, y^{(i)} \in \mathcal{S}_m} L(y^{(i)}, f_{\boldsymbol{\theta}_t}(\mathbf{x}^{(i)})), \tag{7.14}$$

where $\mathcal{S}_m$ is the randomly sampled minibatch, $|\mathcal{S}_m| \ll |\mathcal{S}|$ is its size. We will denote by $L_i : Y \times Y \mapsto \mathbb{R}$, given by $L_i := L(y^{(i)}, f_{\boldsymbol{\theta}_t}(\mathbf{x}^{(i)}))$, the per-example loss. For a suitably chosen η_t, minibatch SGD converges to a minimum in the parameter space - but typically only a *local* minimum, since the loss parametrised by neural networks is highly non-convex.

The update 7.13 requires the computation of the gradient of the loss function with respect to all of the model parameters, evaluated at the current parameter settings $\boldsymbol{\theta}_t$.

The algorithm for computing these derivatives in an efficient manner is known as *backpropagation*, and is based on the chain rule of differentiation. In this section, we will derive the main steps of the backpropagation algorithm for the MLP.

> *Quick Aside.* The backpropagation algorithm was popularised for use in neural network optimisation in Rumelhart et al. 1986b [16] and Rumelhart et al. 1986c [17], although the technique dates back earlier, see e.g. Werbos [19] which includes Paul Werbos' 1974 dissertation.

First recall the layer transformations of the MLP (Equations 7.6 - 7.8), potentially with a multi-dimensional output as in Equation 7.11, as well as the definitions of pre-activations and post-activations (Equations 7.10-7.9).

We will consider the gradient of the loss computed on a single data example $\nabla_{\boldsymbol{\theta}} L_i$, given the sum 7.14. We first compute the **forward pass** 7.6 - 7.8 and store the preactivations $\mathbf{a}^{(l)}$ and post-activations $\mathbf{h}^{(l)}$.

Consider the derivative of L_i with respect to $w_{kq}^{(l)}$ and $b_k^{(l)}$. We have:

$$\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} h_q^{(l)}, \tag{7.15}$$

where the second step follows from 7.10. Similarly,

$$\frac{\partial L_i}{\partial b_k^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial b_k^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}}. \tag{7.16}$$

We introduce the notation $\delta_k^{(l)} := \frac{\partial L_i}{\partial a_k^{(l)}}$, called the **error**. We then write:

$$\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \delta_k^{(l+1)} h_q^{(l)}, \tag{7.17}$$

$$\frac{\partial L_i}{\partial b_k^{(l)}} = \delta_k^{(l+1)}. \tag{7.18}$$

We therefore need to compute the quantity $\delta_k^{(l+1)}$ for each hidden and output unit in the network. Again using the chain rule, we have:

$$\delta_k^{(l)} \equiv \frac{\partial L_i}{\partial a_k^{(l)}} = \sum_{j=1}^{n_{l+1}} \frac{\partial L_i}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} \tag{7.19}$$

Combining 7.10 and 7.9 we see that:

$$a_j^{(l+1)} = \sum_{n=1}^{n_l} w_{jn}^{(l)} \sigma(a_n^{(l)}) + b_j^{(l)} \tag{7.20}$$

$$\frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = w_{jk}^{(l)} \sigma'(a_k^{(l)}) \tag{7.21}$$

where σ' is the derivative of the activation. From the above equations and 7.19 one has:

$$\delta_k^{(l)} \equiv \frac{\partial L_i}{\partial a_k^{(l)}} = \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = \sigma'(a_k^{(l)}) \sum_{j=1}^{n_{l+1}} w_{jk}^{(l)} \delta_j^{(l+1)} \tag{7.22}$$

Equation 7.22, giving a rule determining the errors in the layer l as a function of the ones in the layer $l+1$, describes the *backpropagation* of errors through the network.

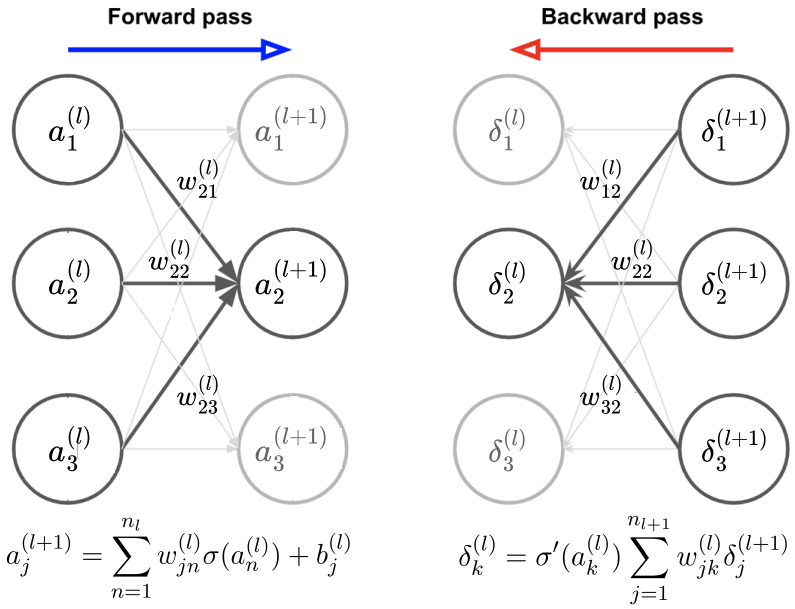


Figure 7.8. Forward and backward passes.

Key message: The backpropagation algorithm works as follows:

(1) Propagate the signal forwards by passing an input vector $\mathbf{x}^{(i)}$ through the network and computing all pre-activations and post-activations using:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l-1)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)} \qquad l = 1, \cdots, N_L.$$

(2) Evaluate $\delta^{(N_L+1)} = \frac{\partial L_i}{\partial \mathbf{a}^{(N_L+1)}}$ for the output neurons (i.e., at layer N_L+1).

(3) Backpropagate the errors to compute $\delta^{(l)}$ for each hidden unit using Equation (7.22), whose compact form reads:

$$\delta^{(l)} = \sigma'(\mathbf{a}^{(l)}) (\mathbf{W}^{(l)})^T \delta^{(l+1)},$$

where $\sigma'(\mathbf{a}^{(l)}) = \text{diag}([\sigma'(a_k^{(l)})]_{k=1}^{n_l})$ and the activation functions are applied element-wise.

(4) Obtain the derivatives of L_i with respect to the weights and biases using: $\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \delta_k^{(l+1)} h_q^{(l)}$, $\quad \frac{\partial L_i}{\partial b_k^{(l)}} = \delta_k^{(l+1)}$.

The derivatives with respect to the weights and biases are then used in the gradient-descent algorithm to train the network.

4. Neural Network training: Optimisers

Recall the two main steps to training neural networks:

(1) Computation of the (stochastic) gradient of the loss function with respect to the model parameters;

(2) Use of the computed gradient to update the parameters.

Now that we have seen how gradients of the loss with respect to the parameters can be efficiently computed using the backpropagation algorithm (step 1), we will take a look at several popular gradient-based optimisation algorithms used in deep learning (step 2).

We have already seen that minibatch SGD can be applied to optimise neural network parameters, and its advantages (SGD reduces redundancy in the gradient computation, and is faster than standard gradient descent). However some challenges remain:

- Convergence can still be very slow with minibatch SGD;
- Setting the learning rate correctly can be difficult, involving trial and error;
- Different weights might operate on different scales, thus requiring different rates of learning.

Several optimisation algorithms have been proposed to help cure these problems.

4.1. Momentum. One common tweak to accelerate the slow convergence of minibatch SGD is to add momentum (Qian 1999 [12]):

$$\begin{aligned} \mathbf{g}_t &:= \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m), \\ \mathbf{v}_{t+1} &= \beta \mathbf{v}_t + \eta \mathbf{g}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_{t+1}, \end{aligned}$$

where $\beta \geq 0$ is the momentum term, and as before, $\eta > 0$ is the learning rate. When $\beta = 0$ then we recover plain minibatch SGD, but with $\beta > 0$ (a typical value is around 0.9), this gives the gradient a short term memory which often accelerates convergence.

4.2. Nesterov momentum. A common variant of momentum is to use Nesterov momentum (Nesterov 1983 [10]), which computes the gradient correction after the accumulated gradient, instead of before:

$$\begin{aligned} \mathbf{g}_t &= \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t - \beta \mathbf{v}_t; \mathcal{S}_m), \\ \mathbf{v}_{t+1} &= \beta \mathbf{v}_t + \eta \mathbf{g}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_{t+1}. \end{aligned}$$

The accumulated gradient approximates the next value of the parameters, and so evaluating the gradient $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t - \beta \mathbf{v}_t; \mathcal{S}_m)$ gives the optimiser a capability of 'look-ahead'.

4.3. Adagrad. The Adagrad optimiser (Duchi et al. 2011 [3]) adapts the learning rate for each parameter, to account for different weights learning on different scales. Parameters that receive a gradient less frequently have larger updates, making Adagrad well suited to sparse data, where most of the features are zero in the data. It is used, for example, in Pennington et al. 2014 [11] to train GloVe word embedding vectors.

The update rule is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m),$$

where G_t is a diagonal matrix where the diagonal element $(G_t)_{ii}$ is the sum of squares of gradients with respect to θ_i up to the iteration t. In the above, the division and square root operations are performed element-wise, and $\odot$ is the Hadamard product.

Note that the resulting learning rates per parameter are monotonically decreasing, and eventually the algorithm effectively stops learning.

4.4. RMSprop. RMSprop is an optimisation method that aims to resolve the vanishing learning rates of Adagrad (it appeared in Geoff Hinton's Coursera course[1] in lecture 6e). It uses a decaying average of past squared gradients. Denoting the gradient by $\mathbf{g}$, the update rule is:

$$\begin{aligned}
\mathbb{E}[\mathbf{g}^2]_t &= \rho \mathbb{E}[\mathbf{g}^2]_{t-1} + (1-\rho)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m))^2, \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\mathbb{E}[\mathbf{g}^2]_t + \varepsilon}} \odot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m).
\end{aligned}$$

As before, the division and square root are performed element-wise, and $\odot$ is the Hadamard product. The ρ term is typically set similar to momentum, around 0.9.

4.5. Adam. The Adam optimiser (Kingma 2015 [5]) is a very popular optimisation algorithm, that also computes adaptive learning rates per parameter. It estimates first and second moments of the gradients, and the name stands for Adaptive moment estimation.

Again, denoting the gradient by $\mathbf{g}$, the update rule is:

$$\begin{aligned}
\mathbb{E}[\mathbf{g}]_t &= \beta_1 \mathbb{E}[\mathbf{g}]_{t-1} + (1-\beta_1) \nabla_\theta L(\theta_t; \mathcal{D}_m), \\
\mathbb{E}[\mathbf{g}^2]_t &= \beta_2 \mathbb{E}[\mathbf{g}^2]_{t-1} + (1-\beta_2)(\nabla_\theta L(\theta_t; \mathcal{D}_m))^2, \\
\mathbf{m}_t &= \mathbb{E}[\mathbf{g}]_t / (1-\beta_1), \\
\mathbf{v}_t &= \mathbb{E}[\mathbf{g}^2]_t / (1-\beta_2), \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\mathbf{v}_t + \varepsilon}} \odot \mathbf{m}_t.
\end{aligned}$$

The $\mathbf{m}_t$ and $\mathbf{v}_t$ terms correct for an initial bias towards zero. Typical values are $\beta_1 \approx 0.9$, $\beta_2 \approx 0.999$ and $\varepsilon \approx 10^{-7}$.

5. Weight regularisation, dropout and early stopping

Deep learning models are typically very over-parameterised, often with millions of parameters over many layers in the model. The fitting power enabled by all these parameters makes them **universal approximators** (see e.g. Cybenko [2] for the large width case, or Lu et al. [5] for the large depth case), but on the other hand **the risk of overfitting is a major problem**. When training neural networks, it is important to regularise them to prevent overfitting. There are several forms of regularisation to prevent overfitting, but in this section we will look at three in particular: weight regularisation, dropout and early stopping. Of course, all of these regularisation techniques mentioned can be used together in deep learning models (and they often are).

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

5.1. L_2 and L_1 regularisation. Recall that for linear regression (Chapter 1), a typical regularisation to add a penalty term containing the squared norm of the parameters to the loss, to discourage the parameters from growing too large. We can write schematically a regularised loss of this type, when the parameters are neural networks' weights $\mathbf{w}$, as:

$$L(\mathbf{w}, \alpha) = L_0(\mathbf{w}) + \alpha_2 \sum_i w_i^2,$$

where L_0 is the unconstrained loss function, and α_2 is a regularisation hyperparameter. This is the L_2 regularisation, in neural networks often referred to as weight decay.

> *Quick Aside.* Note that L_2 regularisation and weight decay, in general, are technically not the same. Weight decay (Hanson & Pratt [4]) is defined as a modification to the update rule, rather than to the loss function itself:
>
> $$\boldsymbol{\theta}_{t+1} \leftarrow (1-\lambda)\boldsymbol{\theta}_t - \eta \eth_t,$$
>
> where λ and η are hyperparameters, and $\eth_t$ is the t-th batch update. In the case of stochastic gradient descent, the update $\eth_t = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$ and the two formulations are equivalent. However, this is not the case for all gradient-based optimisers commonly used in deep learning.

An alternative weight regularisation is the L_1 regularisation, in which the sum of absolute values of the weights are added to the loss term:

$$L(\mathbf{w}, \alpha) = L_0(\mathbf{w}) + \alpha_1 \sum_i |w_i|.$$

This form of regularisation, akin to LASSO, encourages sparsity in the weights. Both L_1 and L_2 regularisation discourage the weights from growing too large, restricting fitting capacity of the network while keeping it away from overfitting, like one has seen for Ridge and Lasso regression (Chapter 1). It is also possible to add a weighted combination of both L_2 and L_1 regularisation to the loss function, similarly to an elastic net.

5.2. Dropout. Dropout was introduced in 2014 by Srivastava et al. [18] (see folder "additional reading") as a regularisation technique for neural networks, that also has the effect of modifying the behaviour of neurons within a network.

The method of dropout is to randomly 'zero out' neurons (or equivalently, weight connections) in the network during training according to a Bernoulli mask whose values are independently sampled at every iteration.

Take $\mathbf{W}^{(l)} \in \mathbb{R}^{n_{l+1} \times n_l}$, the a weight matrix mapping neurons in layer l to layer $l+1$:

$$\mathbf{h}^{(l+1)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l)} + \mathbf{b}^{(l)}\right).$$

We can view dropout as randomly replacing each column of $\mathbf{W}^{(l)}$ with zeros with a certain probability, known as the *dropout rate*. We can write this as applying a Bernoulli mask:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} \cdot \mathrm{diag}(\mathbf{z}^l)$$

where, for layer l, $\mathbf{z}^l$ is a vector of independent n_l Bernoulli random variables. Each of them has probability p_l of being 1:

$$z_j^l \sim \mathrm{Bernoulli}(p_l), \qquad j = 1, ..., n_l,$$

with $\mathbf{W}^{(l)} \in \mathbb{R}^{n_{l+1} \times n_l}$ and $l = 1, \ldots, N_L$. hence, p_l is the probability of keeping a neuron, and the dropout rate is $1 - p_l$. Figures 7.9 and 7.10 illustrate the effect of dropout on a neural network.

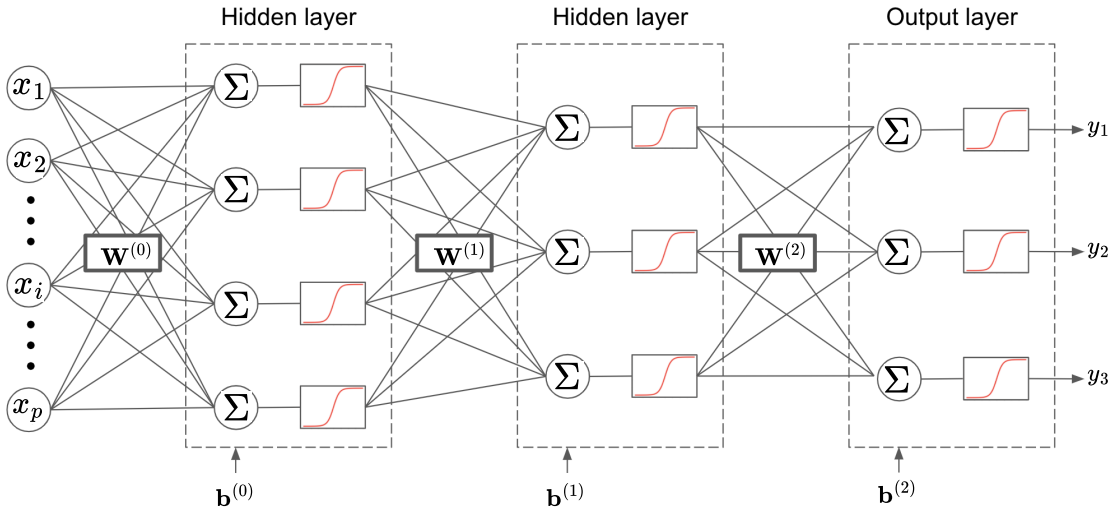


Figure 7.9. Neural network without dropout.

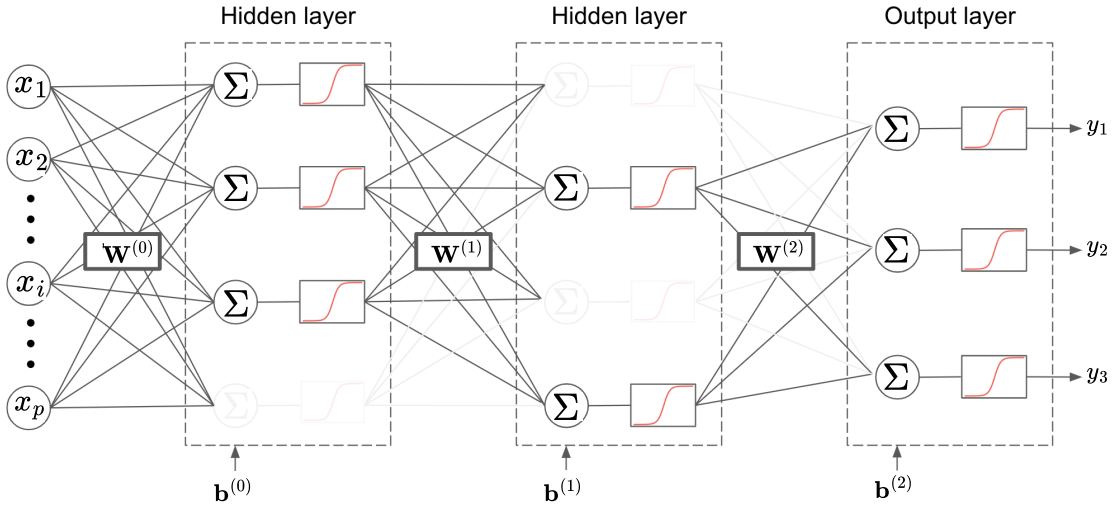
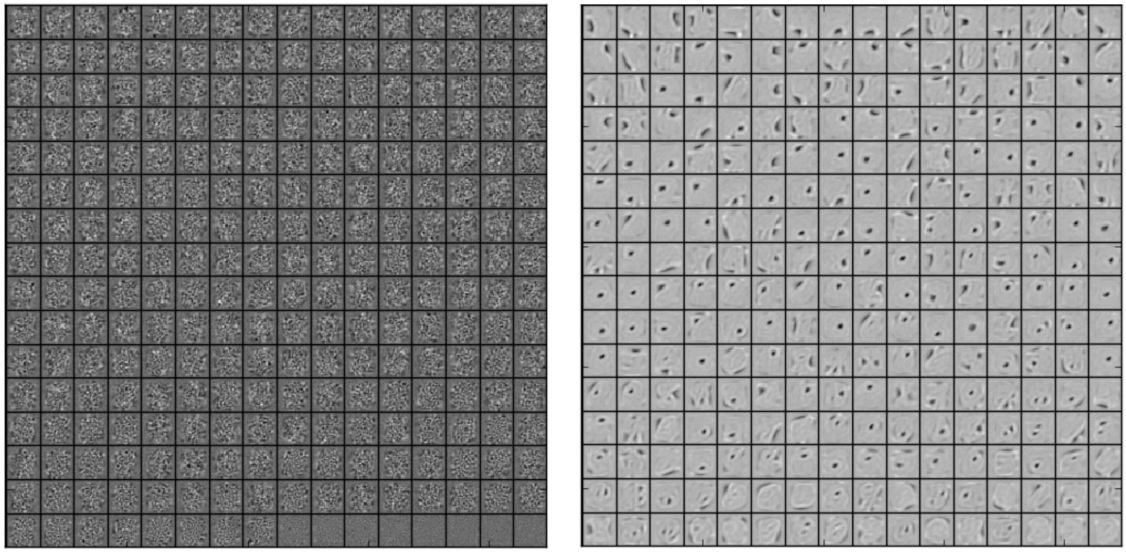


Figure 7.10. Neural network with dropout.

By randomly dropping out neurons in the network, one obvious effect is that the capacity of the model is reduced, and so there is a regularisation effect. Each randomly sampled Bernoulli mask defines a new 'sub-network' that is smaller than the original.

In addition, **a key motivation of dropout is that it prevents neurons from co-adapting too much**. Any neuron in the network is no longer able to depend on any other specific neurons being present, and so each neuron learns features that are more robust, and generalise better.

In the Figure 7.11 (taken from the original paper [18]) we see features that are learned on the MNIST data set for a model trained without dropout (left) and one trained with dropout (right). We see that the dropout model learns features that are much less noisy and more meaningful (it is detecting edges, textures, spots etc.) and help the model to generalise better. The non-dropout model's features suggest a large degree of co-adaptation, where the neurons depend on the specific combination of features in order to make good predictions on the training data.



(a) Without dropout (b) Dropout with $p = 0.5$.

Figure 7.11. Learned features in a neural network trained without dropout (left) and with dropout (right). From Srivastava et al. 2014 [18].

Typically, dropout is applied only in the training phase. When making predictions, all weight connections $\mathbf{W}^{(l)}$ are restored, but re-scaled by a factor of p_l, to take account for the fact that fewer connections were present at training.

5.3. Early stopping. In neural network training, one epoch is a set of iterations that allows for one complete pass through the all the mini-batches of the training dataset, with one iteration per mini-batch. In practice, it is difficult to set a good number of epochs to train for ahead of time. If the training is quick, one can experiment with different values, but in many cases training can take hours or days (or even longer!), so this is not an option.

Recall that deep learning models are usually vastly overparameterised, and have the capacity to drastically overfit. A simple but effective method is to simply stop the training before the model starts to overfit. The resulting picture is similar to the balance between model capacity and generalisation, see Figure 7.12.

With early stopping, the aim is to stop the training when the validation error is at a minimum. This means that the model needs to be regularly evaluated on a held-out validation set (that is not used for training), and the optimisation routine is terminated when the validation error starts to rise. Validation is normally performed once per epoch in the training run.

In practice, the validation error measurements will be noisy, and so it is not a reliable measure to simply detect when the validation error increases and immediately stop the training. What is usually done is to **periodically save model checkpoints (say once per epoch), and set a *patience* threshold**, to specify a maximum number of validation runs that are allowed where the validation error does not improve upon the best score so far. If this patience threshold is reached, the training is terminated.

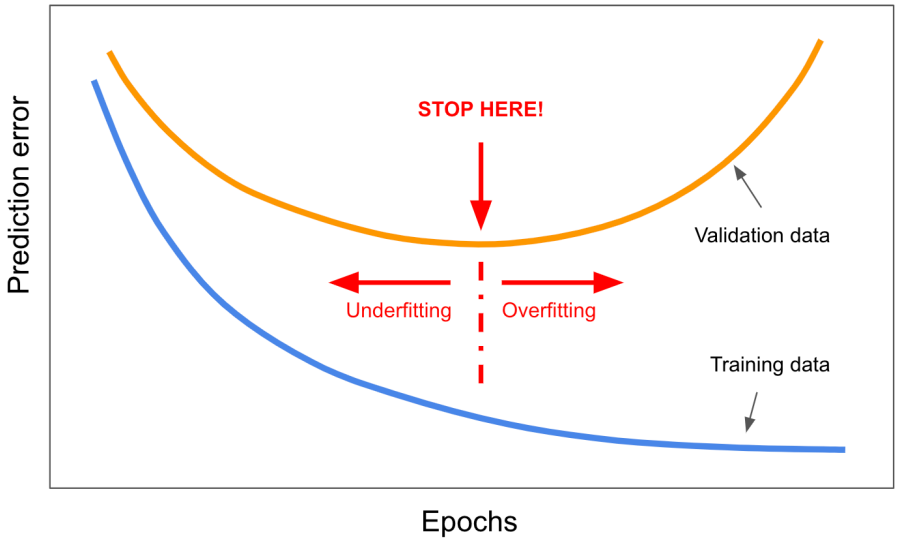


Figure 7.12. Prediction error vs number of training epochs.

References

[1] Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2016), "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", in "4th International Conference on Learning Representations, ICLR 2016", San Juan, Puerto Rico, May 2-4, 2016.

[2] Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, **2** (4), 303–314.

[3] Duchi, J., Hazan, E., & Singer, Y. (2011), "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, **12**, 2121–2159.

[4] Hanson, S. J. & Pratt, L. Y. (1988) "Comparing biases for minimal network construction with back-propagation", in *Proceedings of the 1st International Conference on Neural Information Processing Systems*, 177–185.

-

[5] Kingma, D. P. & Ba, J. L. (2015), "Adam: a Method for Stochastic Optimization", International Conference on Learning Representations, 1–13.

[6] Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017), "Self-Normalizing Neural Networks", *Neural Information Processing Systems (NIPS)*, 971-980.

[7] LeCun, Y., Bengio, Y., Hinton, G. (2015) "Deep Learning", *Nature*, 521(7553), 436-444.

[8] Lu, Z., Pu, H., Wang, F. Hu, Z., & Wang, L. (2017) "The Expressive Power of Neural Networks: A View from the Width", Advances in Neural Information Processing Systems 30. Curran Associates, Inc., 6231–6239.

[9] McCulloch, W. & Pitts, W. (1943), "A Logical Calculus of Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biophysics, **5**, 127-147.

[10] Nesterov, Y. (1983), "A method for unconstrained convex minimization problem with the rate of convergence o(1/k2)", Doklady ANSSSR (translated as Soviet. Math. Docl.), **269**, 543–547.

[11] Pennington, J., Socher, R. & Manning, C. D. (2014), "Glove: Global vectors for word representation", in *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*.

[12] Qian, N. (1999), "On the momentum term in gradient descent learning algorithms", Neural Networks: The Official Journal of the International Neural Network Society, **12** (1), 145–151.

[13] Ramachandran, P., Zoph, B. & Le, Q. V. (2018) "Searching for Activation Functions", arXiv preprint, abs/1710.05941.

[14] Rosenblatt, F. (1958), "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain", Psychological Review, 65-386.

[15] Rumelhart, D. E., McClelland, J. L. and the PDP Research Group (1986a), "Parallel Distributed Processing: Explorations in the Microstructure of Cognition", MIT Press, Cambridge.

[16] Rumelhart, D. E., Hinton, G., & Williams, R. (1986b), "Learning representations by back-propagating errors", Nature, **323**, 533-536.

[17] Rumelhart, D. E., Hinton, G., and Williams, R. (1986c), "Learning Internal Representations by Error Propagation", in Rumelhart, D. E.; McClelland, J. L. (eds.), Parallel Distributed Processing : Explorations in the Microstructure of Cognition. Volume 1: Foundations, Cambridge, MIT Press.

[18] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014), "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Journal of Machine Learning Research, **15**, 1929-1958.

[19] Werbos, P. J. (1994), "The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting", New York:, John Wiley & Sons.

Convolutional neural networks

Chapter based on notes by Dr Kevin Webster

A convolutional neural network (CNN) is a type of neural network with a special structure. It can be seen as a special case of the multilayer perceptron architecture that builds certain assumptions into the design of the model, in particular using **local connectivity** and **equivariance**.

An important motivating application for CNNs is computer vision, as the architectural design of these networks mimics the visual system, where neurons respond to stimulus in a restricted region of the visual field (Hubel 1959 [3]). This concept led initially to the development of the neocognitron (Fukushima 1980 [2]), and later to the modern convolutional neural network trained by backpropagation (LeCun et al 1989 [4]).

In this exposition, **we will focus on CNNs for image processing**, using 2-D convolutions. However, convolutional neural networks have also been very successful when applied to time series data, using 1-D convolutions. They can also be applied to 3-D image processing tasks, or video analysis, with 3-D convolutions.

1. The convolution operation

The convolution operation for two (Lebesgue integrable) functions h and k is defined as:

$$(h * k)(t) = \int_{-\infty}^{\infty} h(\tau)k(t-\tau)d\tau.$$

It can be described as the weighted average of the function h according to the weighting function (or **kernel**) k at each point in time t. As t changes, the weighting function emphasises different parts of the input function h.

In practice (and in the context of CNNs), one has discrete data, thus we need to work instead with discrete convolutions:

$$(h * k)(t) = \sum_{\tau=-\infty}^{\infty} h(\tau)k(t-\tau).$$

In this case, we assume that both h and k (and the convolution $(h * k)$) take integer arguments. In addition, when the kernel function k has finite support, we can write the above as a finite summation.

CNNs with image inputs use 2-D discrete convolutions, which are defined as:

$$(\mathbf{h} * \mathbf{k})(i,j) = \sum_{m,n} h(m,n)k(i-m, j-n).$$

In practice, in CNNs one implements the **cross-correlation** operation, which is the same as above but changing the orientation of the arguments:

$$(\mathbf{h} * \mathbf{k})(i,j) = \sum_{m,n} h(i+m, j+n)k(m,n). \tag{8.1}$$

and we refer to this operation in CNNs as a *convolution*. In the above, $h(i,j)$ and $k(i,j)$ denote the (i,j)-th elements of the matrices $\mathbf{h} \in \mathbb{R}^{n_h \times n_w}$ and $\mathbf{k} \in \mathbb{R}^{k_h \times k_w}$ respectively, where n_h and n_w are the image height and width in pixels, and k_h and k_w are the kernel height and width in pixels (typically much smaller than the ones of the image). In the convolution operation (8.1), the kernel $\mathbf{k}$ is progressively swept through the input $\mathbf{h}$, and different weights of the kernel $\mathbf{k}$ will pick out different features in the input $\mathbf{h}^{(k-1)}$.

As an example, consider a grayscale image $\mathbf{x} =: \mathbf{h}^{(0)} \in \mathbb{R}^{7\times 7}$ with height and width $n_h = n_w = 7$, as illustrated in the following Figure 8.1.

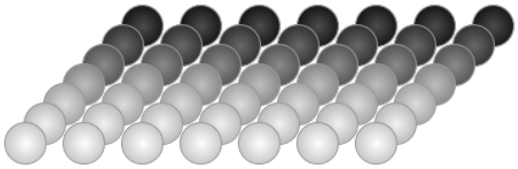


Figure 8.1. Pixels in a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times 7}$.

Suppose we define a kernel matrix $\mathbf{k} \in \mathbb{R}^{3\times 4}$. The operation 8.1 can be visualised as in Figure 8.2 as sweeping the convolutional kernel $\mathbf{k}$ across the input image. At each step, the weights of the kernel matrix $\mathbf{k} \in \mathbb{R}^{3\times 4}$ are multiplied pointwise by the values of the pixels outlined in red and summed together to produce the pre-activation of the neuron in the next hidden layer.

Key message: In CNNs, a **convolutional layer** consists of the convolution operation 8.1 plus a bias term, followed by a pointwise activation function:

$$\mathbf{h}^{(l)} = \sigma\left((\mathbf{h}^{(l-1)} * \mathbf{k}^{(l-1)}) + b^{(l-1)}\right). \tag{8.2}$$

Here the superscript (l) indexes the layer, as in Chapter 7, and the bias $b^{(l-1)} \in \mathbb{R}$ is added pointwise to the output of the convolution operation $(\mathbf{h}^{(l-1)} * \mathbf{k}^{(l-1)}) \in \mathbb{R}^{(n_h-k_h+1)\times(n_w-k_w+1)}$. The output of 8.2 is sometimes referred to as a **feature map**, and the kernel $\mathbf{k}$ is also referred to as a **filter**. Each filter $\mathbf{k}$ in the network consists of a matrix of learnable weights, which are trained via the minimisation of an appropriately defined loss function by error backpropagation (see Chapter 7).

Note that **the convolution operation introduces a translational equivariance property** in convolutional layers. That is, if the input image is translated, then the activations in the next hidden layer are also translated accordingly. Another way to view

Figure 8.2. 2-D convolution with kernel $\mathbf{k} \in \mathbb{R}^{3\times4}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times7}$

this is that the convolutional kernel searches for the same features across the input image, thanks to *weight sharing*, i.e., the use of the same small filters across the input image.

Weight sharing is the key of the success of CNNs in pattern recognition tasks for computer vision: it enables detection of localised features and the equivariance of predictions, which ensures that feature detection is robust to its exact position.

2. Multi-channel inputs and outputs

The operations 8.1 and 8.2 can easily be extended to inputs with multiple channels. Consider, for example, an input RGB image, which has three channel values per pixel (Figure 8.3).

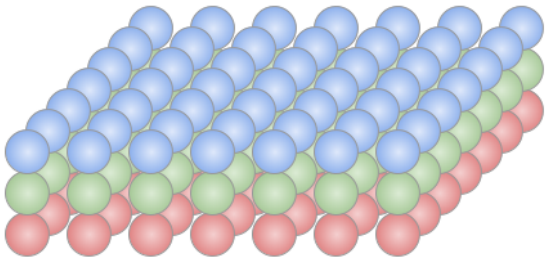


Figure 8.3. Pixels in an RGB input image $\mathbf{x} \in \mathbb{R}^{7\times7\times3}$.

The input is now a rank-3 tensor $\mathbf{x} = \mathbf{h}^{(0)} \in \mathbb{R}^{7\times7\times3}$, and correspondingly we require a rank-3 kernel tensor $\mathbf{k} \in \mathbb{R}^{k_h\times k_w\times3}$. For illustration we will again choose $k_h = 3$, $k_w = 4$. The operation 8.1 now becomes:

$$(\mathbf{h} * \mathbf{k})(i,j) = \sum_{m,n,p} h(i+m, j+n, p)k(m,n,p).$$

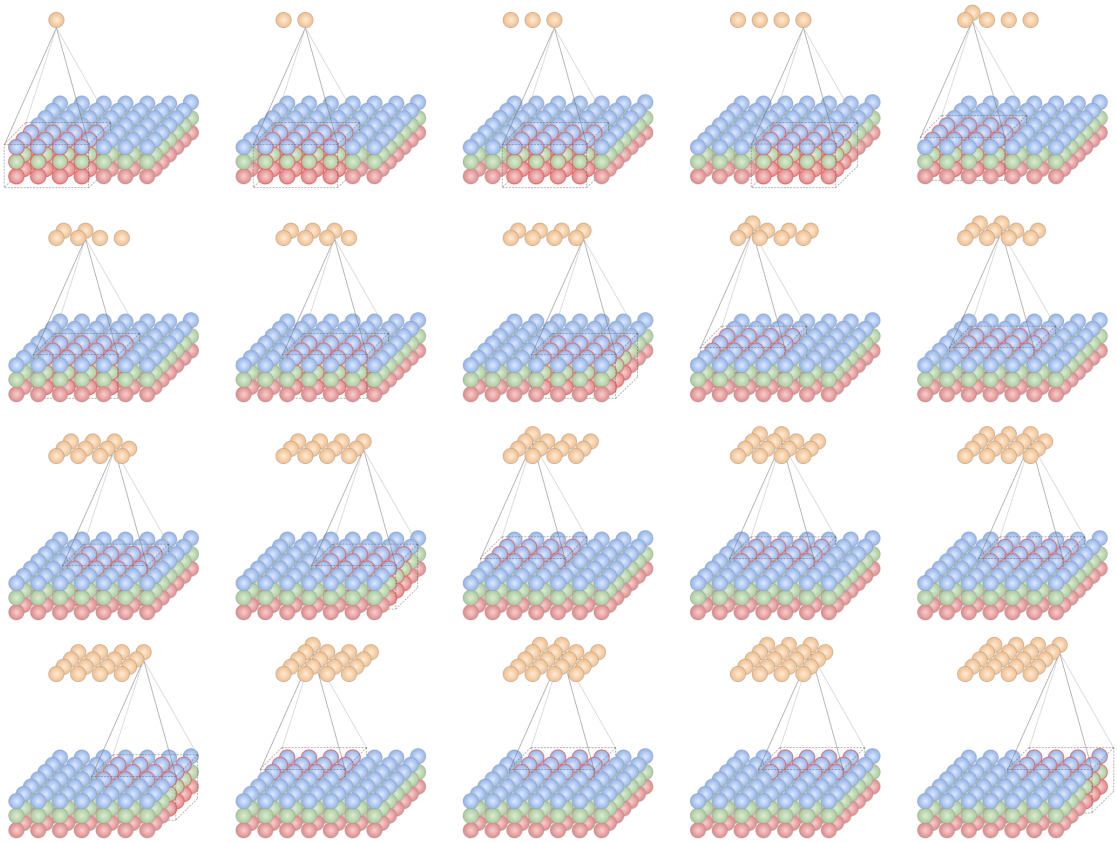


Figure 8.4. 2-D convolution with kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times3}$ operating on an RGB input image $\mathbf{x} \in \mathbb{R}^{7\times7\times3}$.

This is visualised in Figure 8.4, where the kernel $\mathbf{k}$ again sweeps over the input image, this time multiplying a $3 \times 4 \times 3$ block of input pixels (outlined in red) elementwise by the values of the kernel tensor $\mathbf{k}$, and summing the results to produce the output neuron pre-activation.

In convolutional layers, **many filters are stacked on top of each other, so as to produce a multichannel output**. In practice, we implement this with a rank-4 kernel tensor $\mathbf{k} \in \mathbb{R}^{k_h\times k_w\times c_{in}\times c_{out}}$, where c_{in} are the number of channels in the input, and c_{out} are the number of channels in the output:

$$(\mathbf{h} * \mathbf{k})(i, j, q) = \sum_{m,n,p} h(i + m, j + n, p)k(m, n, p, q).$$

This is visualised in Figure 8.5, for an input $\mathbf{x} \in \mathbb{R}^{7\times7\times3}$ and kernel tensor $\mathbf{k} \in \mathbb{R}^{3\times4\times3\times2}$.

The convolutional layer operation 8.2 now becomes the following

$$\mathbf{h}^{(l)} = \sigma\left((\mathbf{h}^{(l-1)} * \mathbf{k}^{(l-1)}) + \mathbf{b}^{(l-1)}\right),$$

where now the bias $\mathbf{b}^{(l-1)} \in \mathbb{R}^{c_{out}}$ is added pixel-wise to the output of the convolution operation:

$$(\mathbf{h}^{(l-1)} * \mathbf{k}^{(l-1)}) \in \mathbb{R}^{(n_h-k_h+1)\times(n_w-k_w+1)\times c_{out}}.$$

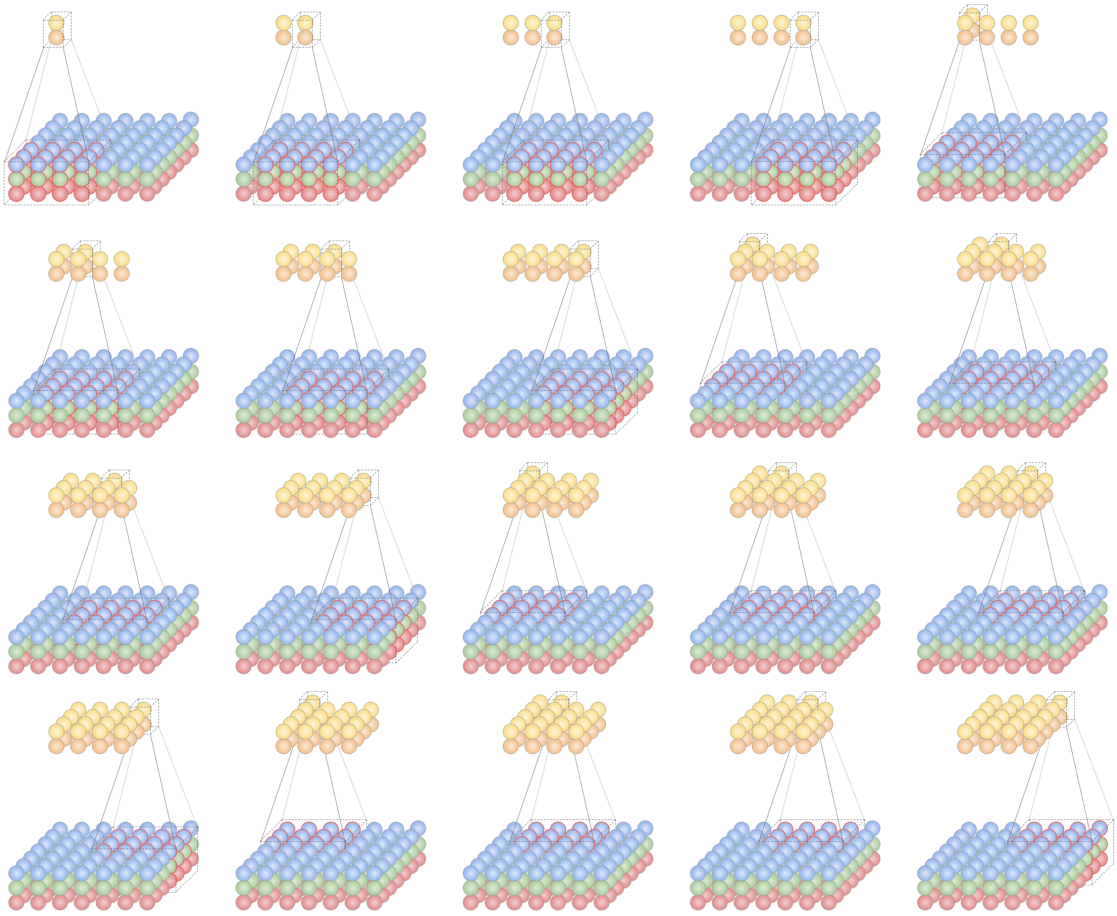


Figure 8.5. 2-D convolution with kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times3\times2}$ operating on an RGB input image $\mathbf{x} \in \mathbb{R}^{7\times7\times3}$ with 2 filters.

3. Pooling layers

In many CNN models, **convolutional layers are alternated with pooling layers that downsample the spatial dimensions of a layer**, by computing a summary statistic of (often non-overlapping) regions of the input layer's post-activations.

A typical pooling layer type is the **max pooling** layer (Zhou & Chellappa 1988 [6]) which takes the maximum activation in a region as the single neuron activation for that region. For example, we could divide the input layer into 2×2 squares and take the maximum value for each square. This results in halving the spatial dimensions of the following layer.

Pooling operations are usually performed separately for each input channel, so that the spatial dimensions are downsampled, but the channel dimensions stay the same. Other common pooling operations include average pooling, or computing the L_2 norm of the pixel values within each input region.

4. Padding and strides

Padding and strides are additional properties of convolutional (and pooling) layers that can give some flexibility over the spatial dimensions of the output. In this section, we will define these properties and the effects they have on the input and output dimensions of a

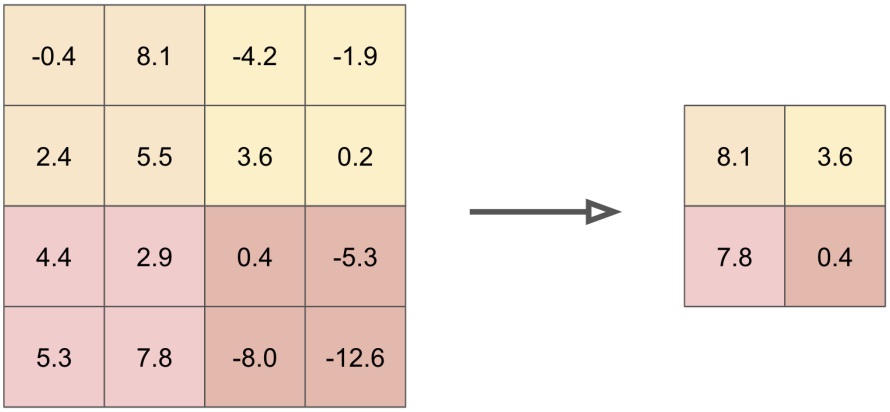


Figure 8.6. Max pooling using 2×2 pooling windows.

convolutional layer. For a more complete guide to convolutional arithmetic, see Dumoulin & Visin 2016 [1].

4.1. Padding. Recall that in our earlier example, an input grayscale image $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$ convolved with a kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times1\times1}$ produced an output of size $\mathbf{h} \in \mathbb{R}^{5\times4\times1}$, visualised in Figure 8.2. In general, for a spatial dimension of size i and a kernel of width k, the output size o is given by:

$$o = i - k + 1 \tag{8.3}$$

In many model architectures, it is desirable to keep the spatial dimensions the same in the output of a convolutional layer. This can be achieved by padding the input layer with zeros.

In the case of our kernel tensor $\mathbf{k} \in \mathbb{R}^{3\times4\times1\times1}$, if we add 2 zeros in the first dimension and 3 zeros in the second dimension (distributed on either side of the image), this will result in an output $\mathbf{h} \in \mathbb{R}^{7\times7\times1}$ that has the same spatial dimensions as the input $\mathbf{x}$. This type of padding is known as "SAME" padding - see Figure 8.7 for a complete visualisation of this.

If p zeros are added to our input size i with kernel width k, then the output size o is given by:

$$o = i + p - k + 1 \tag{8.4}$$

If $p = k - 1$ ("SAME" padding) then $o = i$. If $p = 0$ ("VALID" padding) then we recover 8.3 and $o = i - k + 1$. There is also "FULL" padding where $p = 2(k - 1)$, so that $o = i + k - 1$, although this is less common.

4.2. Strides. Convolutions may also use a stride s, which is the distance between consecutive positions of the kernel. So far, all of our examples have used $s = 1$, however it is easy to see that using $s > 1$ leads to a downsampling of the input. Figure 8.8 shows our input $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$, this time with a kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$, "SAME" padding, and a stride $s = 2$ in both spatial dimensions.

In the above example, the stride and kernel size exactly divides the input and padding size, however this does not need to be the case, such as in Figure 8.9 where $\mathbf{x} \in \mathbb{R}^{7\times6\times1}$.

In this example the input $\mathbf{x} \in \mathbb{R}^{7\times6\times1}$ has been downsampled to an output $\mathbf{h} \in \mathbb{R}^{4\times3\times1}$. Note that if the input image size was instead $7 \times 5 \times 1$ then the output size would again be $4 \times 3 \times 1$ as can be seen in Figure 8.10.

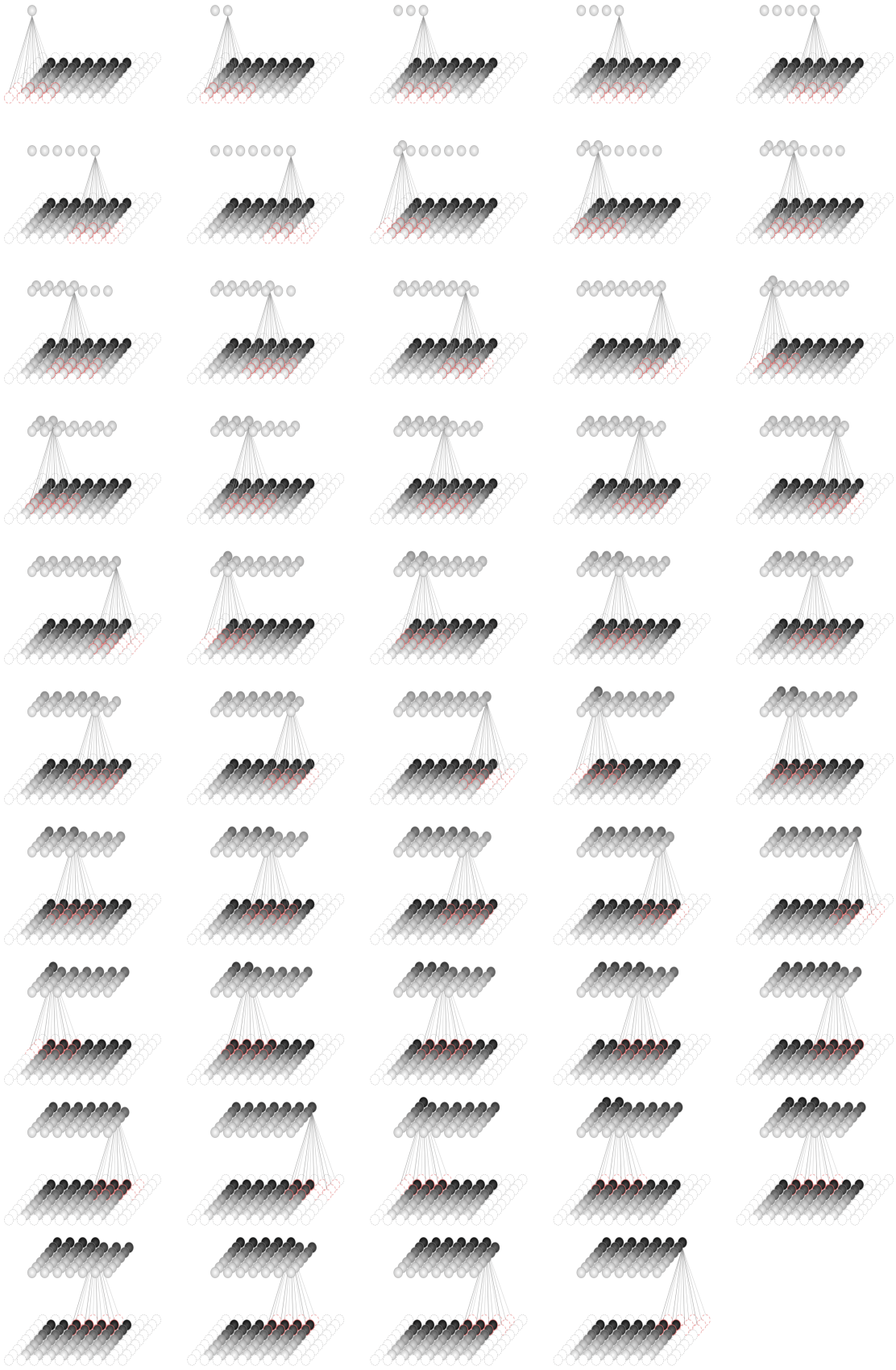


Figure 8.7. A kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$ with "SAME" padding.

Figure 8.8. A kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$ with "SAME" padding and stride of $(2, 2)$.

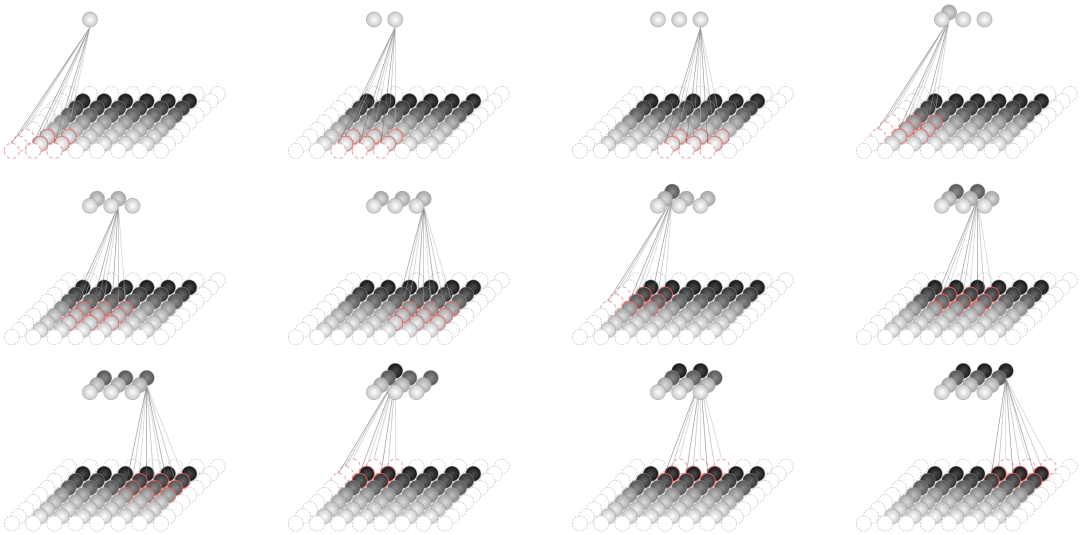


Figure 8.9. A kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times6\times1}$ with "SAME" padding and stride of $(2, 2)$

In all cases, for a spatial dimension of size i with padding p and a kernel of width k with stride s, the output size o is given by:

$$o = \left\lfloor \frac{i + p - k}{s} \right\rfloor + 1$$

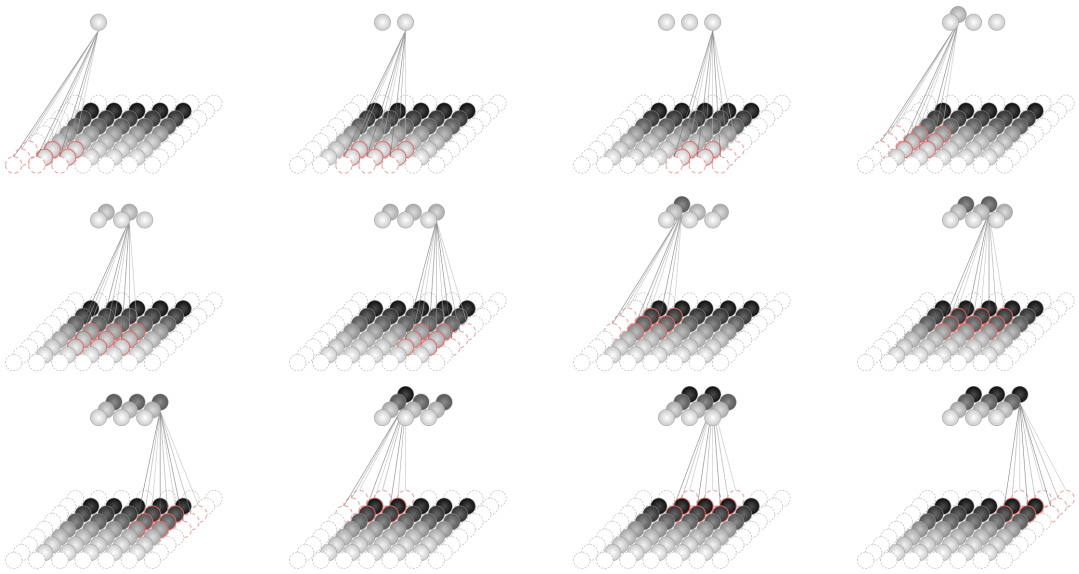


Figure 8.10. A kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times5\times1}$ with "SAME" padding and stride of $(2, 2)$.

References

[1] Dumoulin, V. & Visin, F. (2016), "A guide to convolution arithmetic for deep learning", arXiv preprint, abs/1603.07285.

[2] Fukushima, K. (1980), "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", *Biological Cybernetics*, **3** 6 (4), 193–202.

[3] Hubel, D. H. & Wiesel, T. N. (1959), "Receptive fields of single neurones in the cat's striate cortex", *Journal of Physiology* **148** (3), 574–91.

[4] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989) "Backpropagation Applied to Handwritten Zip Code Recognition", AT&T Bell Laboratories.

[5] Lu, Z., Pu, H., Wang, F. Hu, Z., & Wang, L. (2017) "The Expressive Power of Neural Networks: A View from the Width", Advances in Neural Information Processing Systems 30. Curran Associates, Inc., 6231–6239.

[6] Zhou, Y. & Chellappa, R. (1988), "Computation of optical flow using a neural network", in *IEEE International Conference on Neural Networks*, IEEE, 71-78.

Unsupervised learning

Clustering: K-means, hierarchical clustering, and Gaussian mixtures

1. Unsupervised learning *vs.* Supervised learning

Supervised learning. Up until now, we have concerned ourselves with methods in *supervised learning*.

In supervised tasks, we have N samples which we view as input-output (predictor-observable) pairs, i.e., we define a vector of input variables $\mathbf{x}^{(i)}$ (discrete or continuous) and an output variable $y^{(i)}$:

$$\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N}, \quad \text{where} \quad \mathbf{x}^{(i)} \in \mathbb{R}^p \quad \text{or} \quad \mathbf{x}^{(i)} \in A_1 \times A_2 \cdots \times A_p \tag{9.1}$$
$$\text{and} \begin{cases} y^{(i)} \in \mathbb{R} & \text{(regression)} \\ \quad \text{or} & \\ y^{(i)} \in \{c_1, \ldots, c_Q\} & \text{(classification)} \end{cases}$$

where $\{A_m\}_{m=1}^{p}$ are p sets of discrete values that p categorical predictors can take. In all the supervised methods, we take the set $\mathcal{S} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N^{\text{training}}}$ to be used for training and cross-validation, i.e., to learn the parameters (and hyper-parameters) $\boldsymbol{\theta}$ of the model $f_{\boldsymbol{\theta}}(\mathbf{x})$.

Once the training and cross-validation is finished, the resulting model can be used to make predictions on unseen samples, i.e., given a new sample $\mathbf{x}^{\text{in}}$ we use the model to predict the output $\widehat{y}$ (either its class assignment or its value):

$$\text{Given } \mathbf{x}^{\text{in}}, \quad \text{predict } \widehat{y} = \begin{cases} f_{\boldsymbol{\theta}}(\mathbf{x}^{\text{in}}) \in \mathbb{R} & \text{(regression)} \\ \quad \text{or} & \\ f_{\boldsymbol{\theta}}(\mathbf{x}^{\text{in}}) \in \{c_1, \ldots, c_Q\} & \text{(classification)} \end{cases}$$

In other words, in supervised learning we have samples that provide a *ground truth*, i.e., we take the observations of the output variable $y^{(i)}$ to be true, and our target is to learn from those *true* examples so that we can generalise the task to *unseen* samples.

Unsupervised learning. In the following chapters, we turn our attention towards *unsupervised learning*. In this type of learning tasks, there is a dataset of interest, but

we do **not** view it as a collection of input-output pairs where the target is to predict the output from the input. We now we consider all the variables equally, i.e., we do not define a separation between descriptors (or predictors) and outputs to predict. Our dataset is now viewed only as a collection of (continuous or discrete) samples:

$$\{\mathbf{x}^{(i)}\}_{i=1}^{N}, \qquad \mathbf{x}^{(i)} \in \mathbb{R}^p \quad \text{or} \quad \mathbf{x}^{(i)} \in\in A_1 \times A_2 \cdots \times A_p. \tag{9.2}$$

As a consequence, we do not treat our data as a training set that provides a ground truth for the prediction of some output. Hence, there is no training against known outputs, as we have seen in supervised learning. Likewise, we will not have measures of success computed as the quality of our predictions, since there is no ground truth against which we can assess the goodness of the models.

The main idea of unsupervised learning is to learn the intrinsic structure of the data, as it emerges from the data itself. The target is to discover informative and concise representations of the data that can help our understanding of the observations, and which can drive data-informed discovery from our observations. This implies that the objectives and quality measures arising in unsupervised learning are more open-ended. Yet, they can still be mathematically and computationally formulated, usually as optimisation tasks.

In the next chapters, we will concentrate on two of the most important types of tasks in unsupervised learning:

- **Clustering:** find groups of data points such that the samples within each group are more similar to each other as compared to samples outside of the group. A good clustering will allow us to describe our dataset in terms of the obtained groups (or 'clusters') in a concise manner. The clusters found can also drive interpretation and discovery through the investigation of the meaning of such groups of samples present in the dataset.
- **Dimensionality reduction:** find a 'good' approximate representation of the original dataset (9.2) in terms of lower dimensional variables $\mathbf{a}^{(i)} \in \mathbb{R}^m$ where $m \ll p$. This is important when we have high-dimensional data (i.e., p is very large so that each sample is described by many variables) and we want to find a more concise description in terms of fewer variables obtained (in many cases non-trivially) from the original variables. Dimensionality reduction can also improve interpretability of data and models by eliminating variables that are not very descriptive, instead focussing on those variables that capture the most relevant properties of the dataset.

In this chapter, we first introduce and define the pre-requisites common to all clustering algorithms, before showcasing these approaches with three of the most established clustering methods.

2. Similarity measures for clustering

Usually, the first step towards clustering data is to decide on the notion and definition of *similarity*, $S(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, or *dissimilarity*, $D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, between data points. This notion of dissimilarity is oftentimes closely related to a *distance*, but many (dis)similarities are not true metrics (i.e., they do not fulfil all the properties of a metric).

Continuous data: Given a continuous dataset $\mathbf{x} \in \mathbb{R}^p$, typical examples of *dissimilarity* include:

- the squared Euclidean distance (i.e., the L_2-norm):
$$D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2$$
- or the Manhattan distance (i.e., the L_1-norm):
$$D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = |\mathbf{x}^{(i)} - \mathbf{x}^{(j)}|$$

whereas classic examples of *similarity* include:

- the *cosine similarity*:
$$S(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \frac{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}}{\|\mathbf{x}^{(i)}\|\|\mathbf{x}^{(j)}\|}$$
- or measures of a statistical nature (e.g., correlations and covariances), where two variables being correlated implies similarity. For example:
$$S(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \frac{\text{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})}{\sqrt{\text{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(i)}) \cdot \text{cov}(\mathbf{x}^{(j)}, \mathbf{x}^{(j)})}}$$

Categorical or discrete data: If the variables in the dataset are categorical, $\mathbf{x}^{(i)} \in A_1 \times A_2 \cdots \times A_p$, then we can utilise a distance matrix based on pre-defined 'costs' of a mismatch between categorical values. A particular example of this is the *Hamming distance*, that we had introduced in Chapter 2:

$$D_H(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sum_{m=1}^{p} I(x_m^{(i)} \neq x_m^{(j)})$$

with the indicator function:

$$I(b) = \begin{cases} 1, & \text{if } b \text{ is true} \\ 0, & \text{if } b \text{ is false} \end{cases}$$

The Hamming distance $D_H(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ assigns a cost = 1 to each mismatch and simply sums them. In other words, it counts the number of positions at which two categorical vectors of the same length differ.

3. Definition of the clustering problem

With a chosen definition of what it means to have 'similar' data, we consider the general mathematical statement of the clustering problem:

> **Key Message:** Given N samples $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$ and a 'distance' $D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, quantifying dissimilarity, the objective is to find a *partition* of the N samples into K clusters such that the dissimilarity is smaller within clusters than across clusters.

Note again the lack of predictor/outcome pairs, i.e., there are no pre-defined class labels to be learnt, i.e., the partition of the data emerges naturally from the dataset itself.

We can visualise this as the task of finding a mapping (or 'assignment') between the N samples and K clusters as pictured in Figure 9.1. In other words, we are aiming to find groups in the data (if they exist) such that we minimise the total distances of data points within clusters and maximise the distances between data points in different clusters, as pictured for $\mathbb{R}^3$ in Figure 9.1(b).

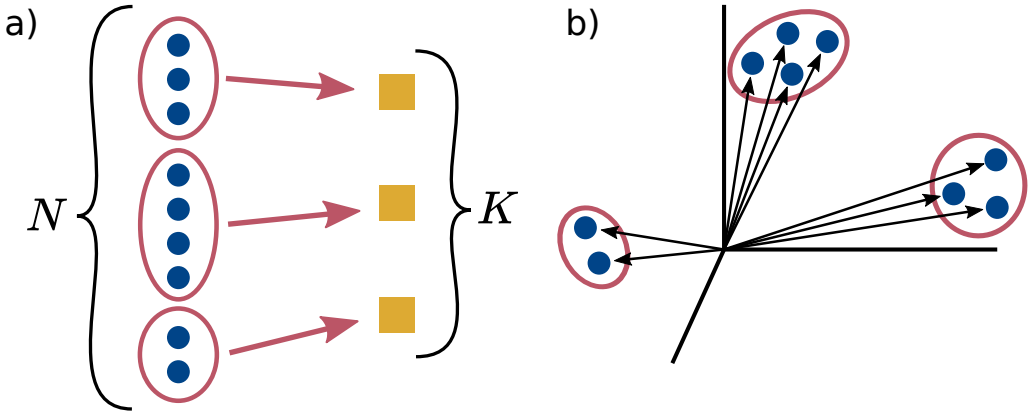


Figure 9.1. a) The clustering problem: Grouping N samples (blue dots) into K clusters (yellow boxes). b) If $\mathbf{x}^{(i)} \in \mathbb{R}^3$ (i.e. 3 variables), we can visualise the data as points in 3D space, which will be closer to each other within a cluster (red ellipses) as opposed to across clusters.

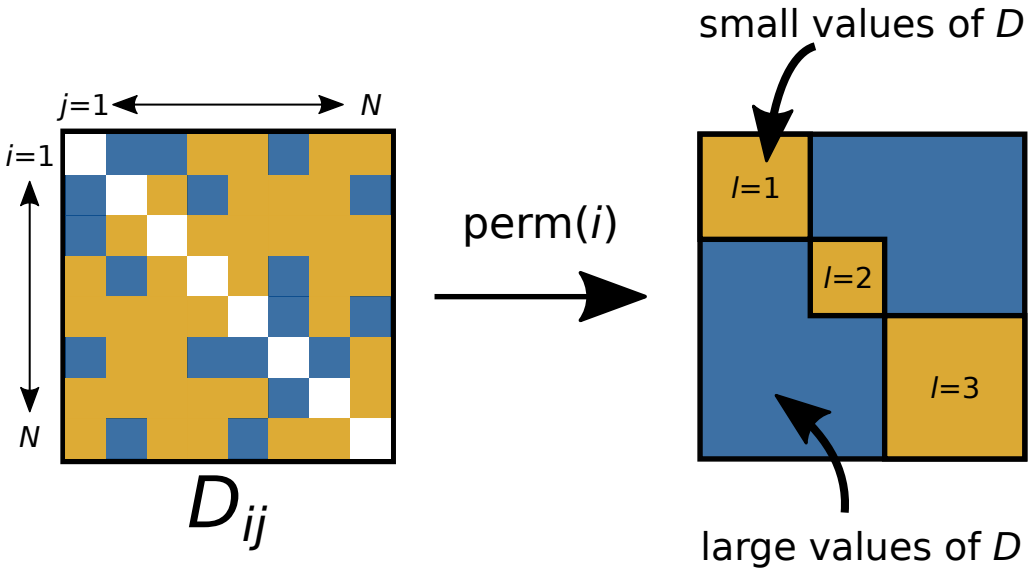


Figure 9.2. The clustering problem re-formulated as a matrix permutation problem. Starting from a distance matrix D, we find a permutation of the rows and columns, such that the resulting permuted matrix contains blocks of small values of D (in yellow) on the diagonal, whilst the off-diagonal sections contain mostly large values of D, shown in blue. Since we have $K = 3$ in this particular example, $l = 1, 2, 3$ represent the indices of each individual cluster. Note that the figure merely represents a cartoon-like depiction of the concept and should not be taken to be exact.

Another viewpoint on this problem is the following. Consider the $N \times N$ distance matrix D that contains all distances $D_{ij} := D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. The clustering problem aims to find a reordering of the rows and columns of D according to some permutation $\text{perm}(i)$, such that the reordered matrix contains blocks of small values along the diagonal and large values off the block diagonals. **In essence, we want to produce a rearrangement of the matrix D that is maximally block-diagonal** - see Figure 9.2.

Using mathematical notation, given a specific clustering $\mathcal{C} = \{c_1, \ldots, c_K\}$ that matches each sample $\mathbf{x}^{(i)}$ to one of K clusters, we aim to **minimise the within-cluster distance**

W:

$$W(\mathcal{C}) = \frac{1}{2}\sum_{k=1}^{K}\sum_{i,j\in c_k} D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{9.3}$$

or, equivalently, maximising the between-cluster distance $B(\mathcal{C})$:

$$B(\mathcal{C}) = \frac{1}{2}\sum_{k=1}^{K}\sum_{i\in c_k}\sum_{j\notin c_k} D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{9.4}$$

This viewpoint shows that clustering can be seen as a *combinatorial optimisation* problem, i.e., the problem relies on finding some 'optimal' permutation of our samples (in the sense just defined) into groups among all possible permutations of this type. As we have already discussed (see Chapter 1), such problems can be 'hard', i.e., in order to find the true global optimum of the problem, one would have to check every single possible permutation, thus making the task of finding the best clustering an NP-hard problem, thus infeasible in practice even for datasets of moderate size (see *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*, Chap. 14, section 14.3.5 for a discussion). This implies that the most established clustering methods rely on some type of heuristics.

4. The K-means clustering method

In the basic clustering method named K-means, the goal is to find a *hard assignment* between the N samples and K clusters, i.e., an assigment whereby each sample belongs to one and only one cluster (unique membership).

After having decided on a specific value of K, the goal of K-means is to find the $N \times K$ *assignment matrix* taking the following form:

$$H_{N\times K} = \begin{bmatrix} 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & & 0 \\ 1 & 0 & 0 & \cdots & & 0 \\ & & \vdots & \ddots & & \\ 0 & \cdots & & & 0 & 1 \end{bmatrix}$$

Note that each row represents one sample data point and each column corresponds to one cluster. Each row contains exactly one entry equal to 1, assigning that particular data point to a particular cluster. Elsewhere, the row is simply filled with zeros.

Quick Aside. The assignment matrix H just defined summarises a hard assignment to a clustering $\mathcal{C} = \{c_l\}_{k=1}^{K}$ that is *exhaustive*. In other words, every single data point will be assigned to just one class, and orphan data points are *not* allowed. Mathematically, this can be described through the following properties (denoting each individual cluster with c_k for some $k = 1, \ldots, K$):

$$c_k \cap c_{k'} = \emptyset \qquad k \neq k'$$

$$\bigcup_{k=1}^{K} c_k = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$$

In matrix-vector form, we have: $H_{ij} \in 0, 1$ with

$$H_{ij} \in 0, 1 \quad \text{with} \quad H\,\mathbf{1}_K = \mathbf{1}_N \quad \text{and} \quad \mathbf{1}_N^T H = (|c_1|, \ldots, |c_K|),$$

where the $\mathbf{1}$ are vectors of ones of the corresponding dimensions, e.g., $\mathbf{1}_N := \mathbf{1}_{N\times 1}$.

K-means algorithm. As explained above, we start with the data points $\mathbf{x}^{(i)} \in \mathbb{R}^p$ and a distance measure, in this case the squared Euclidean distance $D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2$. For a given clustering $\mathcal{C} = \{c_k\}_{k=1}^K$, we have an associated assignment matrix H. We then have the within distance:

$$W(\mathcal{C}) = \frac{1}{2} \sum_{k=1}^{K} \sum_{i,j \in c_k} \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \tag{9.5}$$

However, it is important to pay attention to unbalanced datasets, i.e., where the sizes of the subsets are expected to be quite different. Then one 'normalises' the above equation as follows (where $|c_k|$ is the number of data points in c_k):

$$W(\mathcal{C}) = \frac{1}{2} \sum_{k=1}^{K} \frac{1}{|c_k|} \sum_{i,j \in c_k} \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \tag{9.6}$$

We can re-formulate this in terms of the assignment matrix framework introduced above, so that the $W(\mathcal{C})$ in (9.6) can be rewritten in terms of H:

$$W(\mathcal{C}) = \frac{1}{2} \mathrm{Tr}\left[(H^T H)^{-1} \left[H^T_{K \times N} D_{N \times N} H_{N \times K} \right] \right] \tag{9.7}$$

(see derivation in the aside below).

Quick Aside: To understand better how (9.7) is obtained, we take each element of it and explicitly write out the corresponding matrices. The key insight is that the operation $H^T D H$ can be understood as effecting a sum over the blocks implicitly defined by H. Take for example $K = 3$, then we have:

$$H^T_{K \times N} D_{N \times N} H_{N \times K} = \begin{bmatrix} \clubsuit_1 & \square & \square \\ \square & \clubsuit_2 & \square \\ \square & \square & \clubsuit_3 \end{bmatrix},$$

where each $\clubsuit_k$ corresponds to the total 'within-cluster' distances for cluster c_k (which we are aiming to minimise):

$$\clubsuit_k = \sum_{i,j \in c_k} \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2$$

Furthermore, we have:

$$H^T H = \begin{bmatrix} \spadesuit_1 & 0 & 0 \\ 0 & \spadesuit_2 & 0 \\ 0 & 0 & \spadesuit_3 \end{bmatrix},$$

where each $\spadesuit_k$ corresponds exactly to the cardinality of each cluster c_k:

$$\spadesuit_k = |c_k|$$

Now we can see that (9.7) is indeed equivalent to (9.6).

The objective is to minimise $W(\mathcal{C})$, i.e., the diagonal elements of $(H^T H)^{-1}[H^T D H]$. Conversely, the off-diagonal elements can be written in similar matrix-vector notation. Consider the total sum of distances:

$$T = \frac{1}{2} \mathbf{1}_N^T D \mathbf{1}_N \tag{9.8}$$

$$= \frac{1}{2} \mathbf{1}_k^T [H^T D H] \mathbf{1}_K \tag{9.9}$$

where we have used $H_{N\times K}\mathbf{1}_{K\times 1} = \mathbf{1}_{N\times 1}$. Since T is fixed for the dataset, minimising $W(\mathcal{C})$ necessarily implies maximising $B = T - W(\mathcal{C})$, which in fact brings us back to the equation of the between-cluster distance (9.4).

K-means algorithm: The optimisation algorithm of K-means is as follows:

- **Step 0:** Given a number of clusters K, assign every sample to one of the K clusters at random.
- **Step 1:** Compute the *centroid* of each of the K clusters:
$$\mathbf{m}_k = \frac{1}{|c_k|}\sum_{i\in c_k}\mathbf{x}^{(i)}, \quad k = 1,\ldots,K$$
- **Step 2:** Reassign each $\mathbf{x}^{(i)}$ to the closest centroid. Mathematically speaking, we consider the cluster assigned to $\mathbf{x}^{(i)}$ at iteration t, which is denoted by $k_i^{(t)}$, and evaluate the following:
$$k_i^{(t+1)} = \underset{k}{\operatorname{argmin}}\|\mathbf{x}^{(i)} - \mathbf{m}_k\|^2$$
- **Iterate** steps 1 and 2 until convergence, i.e. $W(\mathcal{C})$ does not improve much (see below) or the assignments do not change.

In the above, one has introduced cluster centroids and formulated the algorithm in terms of them. Indeed, it can be shown that minimising the objective function (9.6) is equivalent to assigning data points to clusters by minimising their distance to the centroids of the cluster. Given a clustering $\mathcal{C} = \{c_k\}_{k=1}^K$, the objective function (9.6) can be rewritten as:

$$\begin{aligned}
W(\mathcal{C}) &= \frac{1}{2}\sum_{k=1}^{K}\frac{1}{|c_k|}\sum_{i,j\in c_k}\|(\mathbf{x}^{(i)} - \mathbf{m}_k) - (\mathbf{x}^{(j)} - \mathbf{m}_k)\|^2 \\
&= \sum_{k=1}^{K}\frac{1}{2}\frac{1}{|c_k|}\sum_{i,j\in c_k}\left[\|\mathbf{x}^{(i)} - \mathbf{m}_k\|^2 + \|\mathbf{x}^{(j)} - \mathbf{m}_k\|^2 - 2(\mathbf{x}^{(i)} - \mathbf{m}_k)(\mathbf{x}^{(j)} - \mathbf{m}_k)\right] \\
&= \sum_{k=1}^{K}\sum_{i\in c_k}\Bigg[\|\mathbf{x}^{(i)} - \mathbf{m}_k\|^2 - \underbrace{(\mathbf{x}^{(i)} - \mathbf{m}_k)\frac{1}{|c_k|}\sum_{j\in c_k}(\mathbf{x}^{(i)} - \mathbf{m}_k)}_{=0}\Bigg]
\end{aligned}$$

Leading to:

$$W(\mathcal{C}) = \frac{1}{2}\sum_{k=1}^{K}\frac{1}{|c_k|}\sum_{i,j\in c_k}\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 = \sum_{k=1}\sum_{i\in c_k}\|\mathbf{x}^{(i)} - \mathbf{m}_k\|^2$$

Note that this optimisation can be likened to a gradient method, since at every step $W(\mathcal{C})$ is decreased. However, since the problem is combinatorial, there is no guarantee that this procedure will converge to the global minimum (which would have to be found through complete enumeration). However, **the algorithm will converge to a local minimum, depending on the random initialisation** (the random assignment of points to clusters in Step 0 above).

Recognising the inherent lack of guarantees for global optimality in the algorithm, the standard procedure for K-means involves running K-means many times with different random initialisations to obtain an ensemble of optimised clusterings. One can then examine the ensemble of solutions found and evaluate the robustness of the obtained clusterings as indication that a partition into K groups is indeed capturing an intrinsic property of the

dataset. In many cases, one will then **choose the optimal clustering (according to the cost function W) from this ensemble of optimised clusterings** obtained from the runs of K-means.

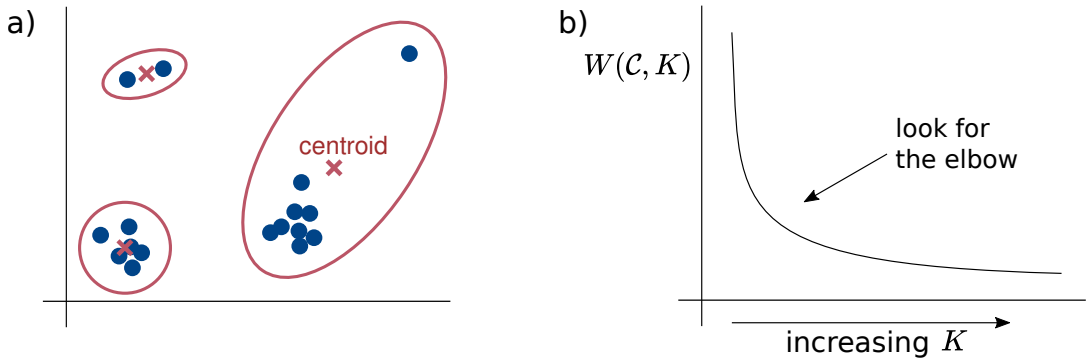


Figure 9.3. a) Cartoon depiction of the possible effect of outliers on K-means clustering: due to an outlier being included in the right cluster, the centroid of this cluster is not representative of the data. b) Optimisation of K: as K is increased, $W(\mathcal{C}, K)$ will decrease. The objective is to look for the 'elbow' of this curve, where increasing K brings no significant further improvement.

A couple of additional comments on K-means:

- **Outliers can pose problems for the K-means algorithm** as can be seen from Figure 9.3a. In some cases, it might happen that outliers will be counted towards a certain cluster, despite the large distance. In order to counteract this sensitivity to outliers, one can utilise an extension to the algorithm called K-medoids. In this method, instead of computing the centroid (which is not a sample point), a specific, representative sample point of those assigned to each cluster is used as the 'centroid' of the cluster, see for more details *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*, Chap. 14, section 14.3.10.
- Crucially, the question of choosing K remains: this is a hard and at the same time crucial task. This falls under the category of tuning the 'hyperparameters' of the model. In short, the answer to this question is to **try a range of increasing K's, until there is no longer any significant improvement in the clustering** (in terms of $W(\mathcal{C}, K)$). This is sometimes termed **'looking for the elbow'**. The basic idea is to use the principle of parsimony: if we view the clusters (e.g., the K centroids) as providing a concise description of the dataset, we want to find the smallest k that produces a small W. An illustration of this can be seen in Figure 9.3b, illustrated with a plateauing of $W(\mathcal{C}, K)$ as K is increased to larger values. Note that other measures of 'goodness' can also be used to decide on how to choose K, including measures of change in the assignment matrix H as K is increased. For a more detailed discussion, see also *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*, Chap. 14, section 14.3.11.

5. Hierarchical clustering

Hierarchical clustering is used broadly for data exploration, particularly in fields such as bioinformatics, as it produces an effective visualisation of the internal substructures of the data across different levels of resolution. In this sense, it gives a global description of the intrinsic organisation of the dataset without having to choose *a priori* a number of clusters.

We start with the usual description of our dataset: $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$, and we also have some distance/similarity measure as defined above which produces our $N \times N$ matrix of distances between all points : $D_{ij} = D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, $i, j = 1, \dots N$.

Key Message: Hierarchical clustering represents whole dataset $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$ as a binary tree - a *dendrogram* - tracking how samples (leaf nodes) get agglomerated into larger groupings as one goes up to the root node (the whole dataset).

Each bifurcation (binary split) of the tree: *i)* corresponds to a merge of two groups of samples; *ii)* happens at a specific depth, enforcing a*strict hierarchy* (i.e., no cross-over of data points into other clusters as depth progresses).

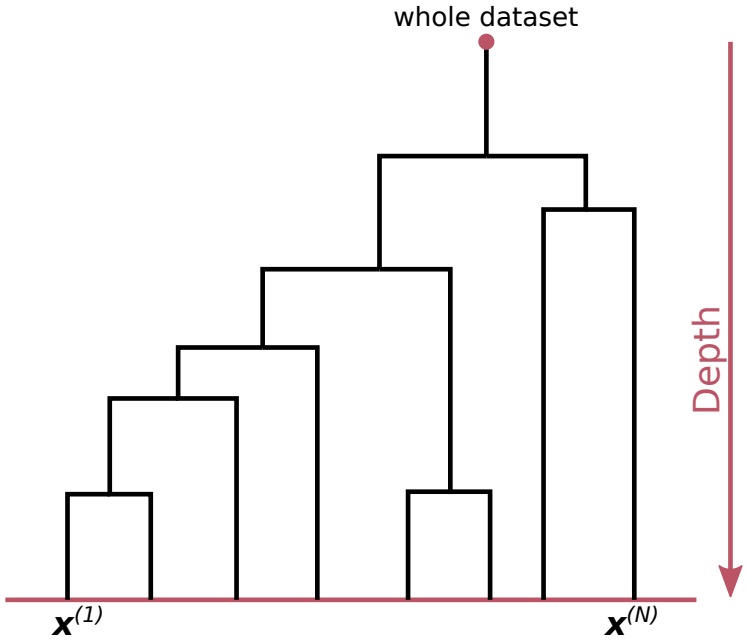


Figure 9.4. A typical dendogram, i.e., the binary tree summarising the groupings of the data samples obtained in hierarchical clustering as a function of the depth. The N leaves at the bottom are samples and the root at the top is the whole dataset. Starting at the top and moving down the tree, splits are made at various stages until finally, at the maximum depth, all individual data points are represented by a leaf of the tree each. Conversely, starting from the N leaves at the bottom (i.e., the N data points), we merge them into groups of increasing coarseness as go up towards the root.

A basic sketch of this idea can be seen in Figure 9.4. In order to produce the dendogram, there are in essence two approaches: bottom-up or top-down, i.e., agglomerative or divisive, respectively.

5.1. Agglomerative schemes for hierarchical clustering. Starting from each sample in its own individual cluster, the algorithm proceeds by merging samples into clusters of successive coarseness, to finally arrive at the whole dataset. So the aim is to reduce the number of clusters one-by-one from N down to 1. There are different versions of this agglomerative scheme depending on the criterion that decides the merge. Some of simplest and most popular of these criteria are:

Simple linkage (SL):. Given two clusters G, H, we consider:

$$d_{\mathrm{SL}}(G, H) = \min_{i \in G,\, j \in H} D_{ij}$$

This scheme can also be described as looking for the nearest neighbours across clusters. This minimum distance is then computed for every pair of clusters at a given depth level of the dendrogram and the minimal of such distances determines the next merge that will reduce the number of clusters by one.

Complete linkage (CL):. Given two clusters G, H, we consider:

$$d_{\mathrm{CL}}(G,H) = \max_{i\in G,\, j\in H} D_{ij}$$

This is essentially looking for furthest points across the two clusters G, H as a measure of distance between G and H, and then one looks for the minimal distance across all pairs of clusters present at a level of the dendrogram.

Average Linkage (GA):. The optimisation function in this case is:

$$d_{\mathrm{GA}}(G,H) = \frac{1}{N_G N_H} \sum_{i\in G, j\in H} D_{ij}$$

where N_G and N_H are the numbers of points respectively in cluster G and H. This computes the distance between clusters as an average over all pairs of points, and uses the distance between all pairs of clusters to decide on the next merge.

Ward's criterion: In its simplest form, Ward's criterion minimises the total within-cluster *variance*, i.e, at each level, we merge the pair of clusters such that the increase in total within-cluster variance is minimal.

A couple of comments about agglomerative schemes:

- The strictly hierarchical nature of the clusterings produced makes agglomerative methods **prone to variability due to sampling**, i.e., small amounts of noise in the data can lead to very different dendrograms.
- If just one partition is desired, it is difficult in most cases to decide what is the right level of resolution to pick in the dendrogram. The y-axis of the dendrogram is a function of the cluster distance used to decide on the merges (or splits), i.e., the amount by which the distance function has increased in the merge. Usually, stable levels of the dendrogram are picked, signalled by large jumps in the depth of the dendrogram between levels.
- As depth increases (i.e., as we move up the hierarchy, merging clusters), the total dissimilarity between grouped samples increases, the dissimilarity here being meant as the quantity minimised at each merge. This results in a *monotonic* increase in this dissimilarity moving from the dendrogram's leaves to the root.

5.2. Divisive schemes for hierarchical clustering. In opposition to the aforementioned agglomerative schemes, divisive schemes represent the 'top-down' approaches. This is usually done by recursive K-means with $K = 2$, which yields a binary tree, *but*: (1) monotonicity is not preserved; and (2) it depends on the initialisation at every split, leading to even more variable results.

6. Probabilistic clustering: Gaussian Mixtures

6.1. Probabilistic clustering. An alternative approach to clustering is that similarities in the data arise from some hidden common "cause". In this view, a cluster is then associated with a hidden variable, to form a so-called *latent variable model*. In brief, one

considers a model:

$$P(\boldsymbol{x}, z), \tag{9.10}$$

where $\boldsymbol{x}$ is an observed datapoint (with continuous features) and z is a categorical latent variable assuming K values corresponding to K clusters. According to this model, we can estimate the probability of a data point $\mathbf{x}^{(i)}$ belonging to the k-th cluster via

$$r_{ik} = P(z = k|\boldsymbol{x} = \mathbf{x}^{(i)}). \tag{9.11}$$

The above expression is called the **cluster probability or responsibility matrix**. For this reason, such methods are referred to as *probabilistic clustering* or *soft clustering*, in contrast to the hard clustering approaches we have seen in the previous sections. Any soft clustering can be converted to a hard one using the argmax discretisation we have seen for classification:

$$\underset{k}{\operatorname{argmax}}\, P(z = k|\boldsymbol{x} = \mathbf{x}^{(i)}). \tag{9.12}$$

An advantage of probabilistic clusterings is that they perform also density estimation and, as such, they yield a *generative model*: **a generative model is a model that can generate more data for each cluster by sampling** from the probability (9.11).

6.2. Gaussian mixture models. Mixture models are probabilistic models commonly used for clustering. The model consists of mixture components $P(\boldsymbol{x} = \mathbf{x}^{(i)}|z = k) = p_k(\mathbf{x}^{(i)}|\boldsymbol{\theta})$ and mixture weights $P(z = k) = \pi_k$ such that the marginal distribution of the data is a convex combination:

$$P(\boldsymbol{x} = \mathbf{x}^{(i)}) = \sum_{k=1}^{K} P(z = k)P(\boldsymbol{x} = \mathbf{x}^{(i)}|z = k) = \sum_{k=1}^{K} \pi_k p_k(\mathbf{x}^{(i)}|\boldsymbol{\theta}). \tag{9.13}$$

The mixture weights $\{\pi_k\}_{k=1}^K$ should satisfy:

$$0 \leq \pi_k \leq 1$$

$$\sum_{k=1}^{K} \pi_k = 1$$

in order to be valid probabilities. The model involves parameters $\boldsymbol{\theta}$ and mixture weights $\boldsymbol{\pi}$ that we need to choose given the data.

We have N observations $\{\mathbf{x}^{(i)}\}_{i=1}^N$, the cluster probabilities can then be computed as:

$$\begin{aligned} r_{ik}(\boldsymbol{\theta}) &= P(z = k|\boldsymbol{x} = \mathbf{x}^{(i)}, \boldsymbol{\theta}) \\ &= \frac{P(z = k, \boldsymbol{x} = \mathbf{x}^{(i)}, \boldsymbol{\theta})}{P(\boldsymbol{x} = \mathbf{x}^{(i)}, \boldsymbol{\theta})} = \frac{\pi_k p_k(\mathbf{x}^{(i)}|\boldsymbol{\theta})}{\sum_{k'=1}^{K} \pi_{k'} p_{k'}(\mathbf{x}^{(i)}|\boldsymbol{\theta})}. \end{aligned} \tag{9.14}$$

The most popular mixture models are Gaussian Mixture Models for which:

$$p_k(\boldsymbol{x}|\boldsymbol{\theta}) = (2\pi)^{-p/2} \det(\boldsymbol{\Sigma}_k)^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_k)^T {\boldsymbol{\Sigma}_k}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_k)\right) \tag{9.15}$$

In unsupervised learning we do not have access to labels and therefore all parameters $\boldsymbol{\theta} = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1,..,K}$ should be chosen such that the model approximates the data as close as possible. A common choice is to **maximise the log-likelihood function of the probabilistic model**, in such a way that the probability modelled by the generative model closely matches the distribution of the data, as we discuss in the next section.

6.3. Expectation-Maximization algorithm (EM). In principle, the maximum likelihood estimate of the mixture is obtained, for numerical convenience, by maximising the log-likelihood:

(9.16)

$$\boldsymbol{\theta}_{\text{MLE}} = \underset{\boldsymbol{\theta}}{\text{argmax}} \log \mathcal{L}(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\text{argmax}} \sum_{i=1}^{N} \log P(\boldsymbol{x} = \mathbf{x}^{(i)}|\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\text{argmax}} \sum_{i=1}^{N} \log \sum_{k=1}^{K} \pi_k p_k(\mathbf{x}^{(i)}|\boldsymbol{\theta}),$$

where $\mathcal{L}(\boldsymbol{\theta})$ is the data likelihood of the GMM, obtained from the data marginal distribution. In principle, the maximum likelihood estimate can be obtained using gradient ascent. The theoretical difficulty is that the likelihood involves a sum over the states of the latent variables z (standing from the K clusters). **EM is an iterative algorithm to find such a maximum likelihood solution by alternating 'expectation' and 'maximisation' steps**. It exploits the fact that if both $\boldsymbol{x}$ and z were fully observed, then the maximum likelihood estimate would be easy to compute.

Assume we had access to the full data $(\mathbf{x}^{(i)}, z^{(i)})$, $i = 1, ..., N$, then best parameters of the mixture model would be given by:

(9.17)

$$\pi_k(\boldsymbol{z}) = \frac{1}{N} \sum_{i=1}^{N} \delta_{z^{(i)},k}, \quad \boldsymbol{\mu}_k(\boldsymbol{z}) = \frac{1}{N_k} \sum_{i=1}^{N} \delta_{z^{(i)},k} \mathbf{x}^{(i)}, \quad \boldsymbol{\Sigma}_k(\boldsymbol{z}) = \frac{1}{N_k} \sum_{i=1}^{N} \delta_{z^{(i)},k} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_k)(\mathbf{x}^{(i)} - \boldsymbol{\mu}_k)^T$$

where $\delta_{z^{(i)},k}$ is the indicator function $\mathbf{1}(z^{(i)} = k)$, N_k denotes the number of data points belonging to cluster k. However, in practice, the clustering $\boldsymbol{z}$ (i.e., the set assignments of a value k to the categorical variables $z^{(i)}$) is uncertain and we require another step to fill in this information. The general idea of the EM procedure is to replace the empirical averages with weighted averages whose weights correspond to the cluster probabilities.

The Expectation-Maximisation (EM) algorithm:

Initialise at random the parameters $\boldsymbol{\theta} = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1,..,K}$

At each iteration t, perform the E and M-steps, as follows:

Expectation step (E-step): uses current parameter values $\boldsymbol{\theta}^t = \{\pi_k^t, \boldsymbol{\mu}_k^t, \boldsymbol{\Sigma}_k^t\}_{k=1,..,K}$ to compute cluster probabilities $r_{ik}(\boldsymbol{\theta}^t)$ and a set of weights:

(9.18)
$$w_{ik}(\boldsymbol{\theta}^t) = \frac{r_{ik}(\boldsymbol{\theta}^t)}{\sum_{i'=1}^{N} r_{i'k}(\boldsymbol{\theta}^t)}$$

Maximisation step (M-step): updates the model's parameters for the next iteration $t+1$ maximising of the log-likelihood. One updates the mixture weight for the k-th component via the corresponding data-averaged cluster probability:

(9.19)
$$\pi_k^{t+1} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{z^{(i)}|\mathbf{x}^{(i)},\boldsymbol{\theta}^t}[\delta_{z^{(i)},k}] = \frac{1}{N} \sum_{i=1}^{N} r_{ik}(\boldsymbol{\theta}^t) = \frac{N_k}{N} \quad \text{Mixture weights}$$

One updates the parameters of the Gaussian mixture components using $w_{ik}(\boldsymbol{\theta}^t)$:

(9.20)
$$\boldsymbol{\mu}_k^{t+1} = \sum_{i=1}^{N} w_{ik}(\boldsymbol{\theta}^t) \mathbf{x}^{(i)} \quad \text{Gaussian means}$$

(9.21)
$$\boldsymbol{\Sigma}_k^{t+1} = \sum_{i=1}^{N} w_{ik}(\boldsymbol{\theta}^t)(\mathbf{x}^{(i)} - \boldsymbol{\mu}_k^{t+1})(\mathbf{x}^{(i)} - \boldsymbol{\mu}_k^{t+1})^T \quad \text{Gaussian covariances}$$

which are simply a weighted version of (9.17).

Iterating between the E and M steps, we obtain a sequence of parameters $\{\boldsymbol{\theta}^t\}_{t=1,2,\ldots}$. The theoretical justification of this approach is that this sequence yields an increasing sequence marginal likelihoods $\mathcal{L}(\boldsymbol{\theta}^0) \leq \mathcal{L}(\boldsymbol{\theta}^1) \leq \ldots \leq \mathcal{L}(\boldsymbol{\theta}^t)$, guaranteeing **convergence to a local maximum** (see chapter 9 of Bishop, *Pattern Recognition And Machine Learning* for a proof). Since this optimum is only local, a common mitigation strategy is to start the algorithm multiple times from different initial guesses for the parameters $\boldsymbol{\theta}$, allowing one to get closer to the global optimum.

6.4. Relation to K-means. You may have wondered about the relation between the EM-algorithm for Gaussian mixture models and K-means, which are in fact related.

Assume to fix in a Gaussian mixture model $\boldsymbol{\Sigma} = \mathrm{diag}(1)$ and $\pi_k = 1/K$, so only the parameter to estimate is the cluster mean $\boldsymbol{\mu}_k$. Assume that we enforce a hard clustering taking the argmax of (9.14) via:

$$r_{ik}(\boldsymbol{\theta}) = \begin{cases} 1 & \text{if } k = \mathrm{argmax}_j\, r_{ij}(\boldsymbol{\theta}) = \mathrm{argmin}_j\, ||\mathbf{x}^{(i)} - \boldsymbol{\mu}_j||^2 \\ 0 & \text{otherwise} \end{cases} \tag{9.22}$$

This reduces the EM iteration of the mean (9.20) to the K-means algorithm, showing that Gaussian mixture models reduce to K-means in the limit of $\boldsymbol{\Sigma} = \mathrm{diag}(1)$, $\pi_k = 1/K$ and hard clustering assignments via the argmax discretisation. Conversely, **the EM-algorithm for Gaussian mixture models can be seen a generalisation of K-means to soft clustering**.

6.5. Choosing K. In K-means, as it is a *distance-based* method, we have seen that an optimal K can be set through the elbow rule, i.e., finding the elbow of the within-cluster distance as a function of the number of clusters K. In a GMM, as a distribution-based method, one appeals to criteria from information theory that include the likelihood of the distribution $\mathcal{L}(\boldsymbol{\theta})$, maximised during the training, as a measure of the goodness of fit. These criteria are the:

- Akaike Information Criterion (AIC):
  $AIC = 2N_{\boldsymbol{\theta}} - 2\ln(\mathcal{L}(\boldsymbol{\theta}))$
  where $N_{\boldsymbol{\theta}}$ denotes the number of parameters in the model.
- Bayesian Information Criterion (BIC):
  $BIC = N_{\boldsymbol{\theta}} lnN - 2\ln(\mathcal{L}(\boldsymbol{\theta}))$

where we have stressed that the log of the likelihood here is meant as the natural log. They are designed to balance the goodness of fit, measured by the log-likelihood maximised by the learnt parameters, and the model complexity (measured by the overall number of parameters in the model), in such a way as to discourage overfitting. We see that BIC replaces the factor 2 in AIC by $\ln N$: for typical sample sizes, let's say $N > e^2 \sim 7.4$, BIC will tends to penalise complex models more heavily than AIC; BIC adapts this model complexity penalty to the sample size, making it particularly severe for small datasets. You can read more about the use of AIC and BIC in model selection in Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*, Chap. 7.

The optimal K can be set by looking for the *minimum* of the AIC or BIC.

7. Comparing clusterings

A brief note on the comparison of clusterings is in order. In many cases, one needs to compare how similar two clusterings are. This topic is related to the confusion and

contingency matrices we have seen for classification (Chapter 3), but here, again, we do not have a ground truth against which we are comparing, and clusterings can have different numbers of groups.

Typically, the way to compare clusterings is through the creation of a **count matrix** that matches the number of coincident elements in each cluster of the two clusterings. Then a summary measure is compiled as a quality of the overlap of elements across clusters. Main examples are the following.

Adjusted Rand Index[1]**.** Given a set of N elements, and two groupings or partitions (e.g. clusterings) of these elements with respectively r and s elements, namely $X = \{X_1, X_2, \ldots, X_K\}$ and $Y = \{Y_1, Y_2, \ldots, Y_{K'}\}$, the overlap between X and Y can be summarized in a contingency table $[n_{ij}]$ where each entry n_{ij} denotes the number of objects in common between X_i and Y_j : $n_{ij} = |X_i \cap Y_j|$.

	Y_1	Y_2	$\cdots$	$Y_{K'}$	sums
X_1	n_{11}	n_{12}	$\cdots$	$n_{1K'}$	a_1
X_2	n_{21}	n_{22}	$\cdots$	$n_{2K'}$	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
X_K	n_{K1}	n_{K2}	$\cdots$	$n_{KK'}$	a_K
sums	b_1	b_2	$\cdots$	$b_{K'}$	

The Adjusted Rand Index is then defined by:

$$ARI = \frac{\sum_{ij} \binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \Big/ \binom{N}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \Big/ \binom{N}{2}}$$

Mutual Information and Adjusted Mutual Information[2]**.** Given two clusterings X and Y of size K and K' respectively, the Mutual Information (MI) score is defined as:

$$MI(X, Y) = \sum_{i=1}^{K} \sum_{j=1}^{K'} P_{XY}(i, j) \log \frac{P_{XY}(i, j)}{P_X(i) P_Y(j)}$$

where $P_{XY}(i, j)$ denotes the probability that a point belongs to both the cluster X_i in X and cluster Y_j in Y:

$$P_{XY}(i, j) = \frac{|X_i \cap Y_i|}{N}$$

N being the total number of data points.

The Adjusted Mutual Information (AMI) is an adjustment of MI to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared.[3] It is defined as:

$$AMI(X, Y) = \frac{MI(X, Y) - E\{MI(X, Y)\}}{\max\{H(X), H(Y)\} - E\{MI(X, Y)\}}$$

[1] https://en.wikipedia.org/wiki/Rand_index#Adjusted_Rand_index

[2] https://en.wikipedia.org/wiki/Adjusted_mutual_information

[3] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_mutual_info_score.html

Here, we have the following definitions:

$$H(X) = -\sum_{i=1}^{K} P_X(i) \log P_X(i) \qquad \text{The clustering entropy;}$$

$$E\{MI(X,Y)\} = \sum_{i=1}^{K}\sum_{j=1}^{K'} \sum_{n_{ij}=(a_i+b_j-N)^+}^{\min(a_i,b_j)} \frac{n_{ij}}{N} \log\left(\frac{N \cdot n_{ij}}{a_i b_j}\right) \times \frac{a_i! b_j! (N-a_i)! (N-b_j)!}{N! n_{ij}! (a_i - n_{ij})! (b_j - n_{ij})! (N - a_i - b_j + n_{ij})!}$$

Normalised Variation of Information[4]**.** From the Mutual Information $MI(X,Y)$ and the Entropy measures $H(X)$ and $H(Y)$ we can directly compute the Normalised Variation of Information (NVI) given by:

$$NVI(X,Y) = \frac{H(X) + H(Y) - 2MI(X,Y)}{H(X) + H(Y) - MI(X,Y)} = \frac{VI(X,Y)}{H(X,Y)}$$

where $VI(X,Y) = H(X) + H(Y) - 2MI(X,Y)$ is the variation of information and $H(X,Y) = H(X) + H(Y) - MI(X,Y)$ is the joint entropy of the partitions.

The NVI is a metric on the space of partitions that is easier to interpret, as it is bounded between 0 and 1: $NVI = 0$ implies that the two clusterings are identical, $NVI = 1$ indicates that the clusterings are completely independent.

[4] http://arxiv.org/abs/q-bio/0311039

Chapter 10

Dimensionality reduction: PCA & NMF

Continuing with the theme of unsupervised learning, this chapter introduces key approaches to *dimensionality reduction*.

1. Principal Component Analysis (PCA)

As usual, we have the standard setup for unsupervised learning, with a dataset:

$$\{\mathbf{x}^{(i)}\}_{i=1}^{N} \qquad \mathbf{x}^{(i)} \in \mathbb{R}^p$$

where, as discussed already for the general setting of unsupervised learning, we do **not** distinguish between input or output variables. Our aim is to extract information and structural understanding of the data in an unsupervised manner, guided by the dataset itself.

In this particular case, the dimensionality of the samples is very large, $p \gg 1$, and one might have the intuition (or prior knowledge) that many of those variables are unnecessary or redundant (e.g., correlated variables, noise, etc.), and the dataset might be better described by fewer, more informative variables. This is the case where it would be convenient to represent the data in a lower dimension.

> **Key Message:** The goal of dimensionality reduction is to find a description of the dataset that captures as much information as possible from the original data in reduced dimensions, i.e., going from the original number of dimensions p to a reduced dimension $m < p$.

Many times, dimensionality reduction is also a good step to facilitate other downstream tasks (e.g. classification or clustering), which might struggle to utilise very high-dimensional datasets. By summarising the data through a reduction of dimensions, many models become easier to train and computationally feasible, whilst losing next to nothing in terms of accuracy (or other quality measures).

The classic tool for dimensionality reduction is *principal component analysis* (PCA), which, in its basic form, is a *spectral linear method*. First, *spectral* methods are called in this way because they rely on properties of matrix decompositions closely aligned with matrix diagonalisation, a.k.a. 'spectral decomposition'. As you know, **the spectrum of a matrix is the set of its eigenvalues, with its associated eigenvectors**. These methods exploit spectral properties of matrices associated with data. Second, it is a *linear* method for dimensionality reduction because it transforms data to lower-dimensional representations using *linear projections*, as we now make clear.

1.1. Mathematical derivation of PCA. In PCA we aim for a description of every data point in terms of a *linear* combination basis functions $\mathbf{v}_j$:

$$\mathbf{x}^{(i)} = \sum_{j=1}^{p} a_j^{(i)} \mathbf{v}_j, \text{ such that } \mathbf{v}_j \in \mathbb{R}^p, \tag{10.1}$$

and we search for an orthonormal basis such that:

$$\mathbf{v}_j \cdot \mathbf{v}_k = \delta_{jk} := \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

From these definitions, it immediately follows that the coefficients in (10.1) are obtained as dot products (projections) on the basis:

$$a_j^{(i)} = \mathbf{v}_j \cdot \mathbf{x}^{(i)} = \mathbf{x}^{(i)} \cdot \mathbf{v}_j$$

One way to view dimensionality reduction is that we want to approximate $\mathbf{x}^{(i)}$ by:

$$\widehat{\mathbf{x}}_m^{(i)} = \sum_{j=1}^{m} a_j^{(i)} \mathbf{v}_j + \sum_{j=m+1}^{p} b_j \mathbf{v}_j \tag{10.2}$$

where the b_j are not dependent on i (i.e., we want to find the $b_j, \mathbf{v}_j$ such that they apply to all samples). In essence, we will describe each sample $\mathbf{x}^{(i)}$ in lower dimension through the m coefficients $a_j^{(i)}$, whereas the sum over the b_j terms in (10.2) reflects the error (common to all samples) that is associated with the elements of the basis $\{\mathbf{v}_j\}_{j=m+1}^{p}$.

We can write down the error made from this assumption:

$$\Delta\mathbf{x}^{(i)} = \mathbf{x}^{(i)} - \widehat{\mathbf{x}}_m^{(i)} = \sum_{j=m+1}^{p} [a_j^{(i)} - b_j]\mathbf{v}_j$$

We perform an optimisation for this approximation to find the best description of the data. Specifically, we minimise the associated mean squared error:

$$\begin{aligned} \text{MSE} &= \frac{1}{N}\sum_{i=1}^{N} \|\Delta\mathbf{x}^{(i)}\|^2 \\ &= \frac{1}{N}\sum_{i=1}^{N} \left[\sum_{j,k=m+1}^{p} (a_j^{(i)} - b_j)(a_k^{(i)} - b_k)\mathbf{v}_j \cdot \mathbf{v}_k \right] \\ &= \frac{1}{N}\sum_{i=1}^{N} \sum_{j=m+1}^{p} (a_j^{(i)} - b_j)^2 \end{aligned} \tag{10.3}$$

As we are looking to optimise both b_j and $\mathbf{v}_j$, we first start with the former and compute the following:

$$\frac{\partial \text{MSE}}{\partial b_j} = -\frac{1}{N}\sum_{i=1}^{N} 2(a_j^{(i)} - b_j)$$

At the optimum, we have:

$$\frac{\partial \text{MSE}}{\partial b_j}\Big|_{b_j} = 0 \quad \implies \quad b_j = \frac{1}{N}\sum_{i=1}^{N} a_j^{(i)}$$

Plugging the above expression into (10.3), we get:

$$\begin{aligned}
\text{MSE} &= \sum_{j=m+1}^{p} \frac{1}{N}\sum_{i=1}^{N} \Big(\underbrace{\mathbf{x}^{(i)} \cdot \mathbf{v}_j}_{a_j^{(i)}} - \underbrace{\frac{1}{N}\sum_{i=1}^{N} \mathbf{x}^{(i)} \cdot \mathbf{v}_j}_{b_j} \Big)^2 \\
&= \sum_{j=m+1}^{p} \frac{1}{N}\sum_{i=1}^{N} \left[\left(\mathbf{x}^{(i)} - \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}^{(i)} \right) \cdot \mathbf{v}_j \right]^2 \\
&= \sum_{j=m+1}^{p} \mathbf{v}_j^T \Bigg(\underbrace{\frac{1}{N}\sum_{i=1}^{N} \left[\left(\mathbf{x}^{(i)} - \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}^{(i)} \right) \left(\mathbf{x}^{(i)} - \frac{1}{N}\sum - -^{N} \mathbf{x}^{(i)} \right)^T \right]}_{\clubsuit} \Bigg) \mathbf{v}_j
\end{aligned}$$

Note that the 'clubsuit' block can be rewritten as follows:

$$\clubsuit = \Big\langle \left(\mathbf{x} - \langle \mathbf{x} \rangle \right) \left(\mathbf{x} - \langle \mathbf{x} \rangle \right)^T \Big\rangle \quad \text{with} \quad \langle \mathbf{x} \rangle = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}^{(i)},$$

which is the covariance matrix $(C_{\mathbf{x}})_{p\times p}$ of the data features. Hence the MSE can be written compactly as:

$$\text{MSE} = \sum_{j=m+1}^{p} \mathbf{v}_j^T C_{\mathbf{x}}\, \mathbf{v}_j \tag{10.4}$$

To find the *orthonormal* $\mathbf{v}_j$, we need to carry out a constrained optimisation. As you know by now, we do this using Lagrange multipliers to enforce equality constraints. Let us write down the Lagrangian:

$$\mathcal{L} = \sum_{j=m+1}^{p} \mathbf{v}_j^T C_{\mathbf{x}} \mathbf{v}_j + \sum_{j=m+1}^{p} \lambda_j (1 - \mathbf{v}_j^T \mathbf{v}_j),$$

where the λ_j are Lagrange multipliers. In order to optimise this, we follow the same approach as before:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}_j} = 2\mathbf{v}_j^T C_{\mathbf{x}} - 2\lambda_j \mathbf{v}_j^T$$

where the the derivative $\frac{\partial \mathcal{L}}{\partial \mathbf{v}_j}$ is meant as the row vector $\frac{\partial \mathcal{L}}{\partial \mathbf{v}_j} = \left(\frac{\partial \mathcal{L}}{\partial v_j^1}, ..., \frac{\partial \mathcal{L}}{\partial v_j^p} \right)$. The solution is then given as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}_j} = 0 \quad \implies \quad C_{\mathbf{x}} \mathbf{v}_j = \lambda_j \mathbf{v}_j \tag{10.5}$$

This solution shows that the Lagrange multipliers λ_j are simply the eigenvalues of $C_{\mathbf{x}}$ with corresponding eigenvectors $\mathbf{v}_j$: **the optimal basis to expand is the basis of the**

eigenvectors of $C_{\mathbf{x}}$. In other words, if we expand the data in this basis, we will be optimally reducing the MSE defined previously. Its optimal value is given by:

$$\text{MSE} = \sum_{j=m+1}^{p} \mathbf{v}_j{}^T C_{\mathbf{x}} \mathbf{v}_j = \sum_{j=m+1}^{p} \mathbf{v}_j{}^T \lambda_j \mathbf{v}_j = \sum_{j=m+1}^{p} \lambda_j$$

that is, the MSE of the lower-dimensional approximation is the sum of the eigenvalues that are discarded when reducing dimensions from p to m.

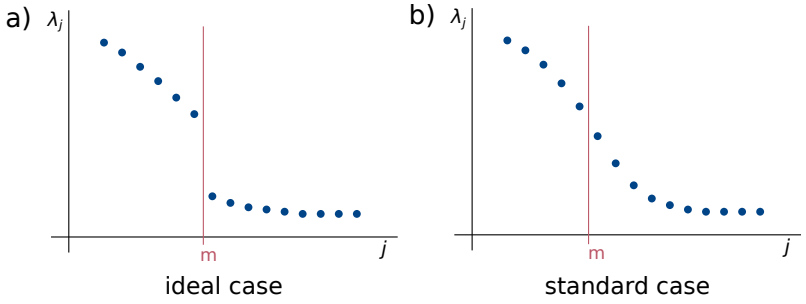


Figure 10.1. Finding an optimal m for dimensionality reduction is not always easy. In a), we can see an ideal case scenario, where some eigenvalues are large, but at some point there is a drastic drop, leading to the remaining eigenvalues to be quite small. In this case, deciding on a good m is natural. However, the usual case is seen in b), where the eigenvalues are on a much more steady decline and one must fix m according to some tolerance ε.

Finding an optimal m is then a decision made based on the eigenvalues. This can be visualised by plotting the eigenvalues, ordered according to decreasing value, as can be seen in Figure 10.1. In general, one must decide on a tolerance ε and find an m, such that $\sum_{j=m+1}^{p} \lambda_j < \varepsilon$, which will keep the error within that tolerance.

From (10.5), the total variance of the data V can be calculated as $V = \sum_{j=1}^{p} \lambda_j$, hence the fraction of variance explained by a certain component j (also called 'explained variance ratio') is given by λ_j / V. Ideally, one would choose the number of components to include by adding the **explained variance ratio** of each component until one reaches a total of **around 80%**.

Key Message: In PCA, having chosen the number of dimensions m, the approximated data in the lower, m-dimensional representation are given by:

$$\widehat{\mathbf{x}}^{(i)} = \sum_{j=1}^{m} \left(\mathbf{x}^{(i)} \cdot \mathbf{v}_j \right) \mathbf{v}_j \quad \text{where} \quad C_{\mathbf{x}} \mathbf{v}_j = \lambda_j \mathbf{v}_j$$

i.e., they are given by an expansion in the eigenvectors $\mathbf{v}_j$ of the data covariance matrix $C_{\mathbf{x}}$. These are called the *principal components*, thus the name of the method.

In Figure 10.2, we illustrate visually the difference between linear regression and PCA. Whilst in linear regression, the aim is to find a hyperplane that minimises the mean squared error of prediction (see Chapter 1), in PCA the view shifts towards errors normal to the hyperplane, i.e., the goal is to find the hyperplane that minimises the sum of distances normal to the plane. Hence, the hyperplanes obtained through each method are *not the same*. The intuition of why this happens is clear: standard linear regression is a supervised learning method where we divide our variables into inputs and outputs (i.e., independent and dependent variables). The error is then assigned to the output

observations (dependent variables), with no error in the inputs (hence the definition of the MSE in that case). In the case of PCA, there is no separation between input-output variables (as it is an unsupervised method). Hence the error is assigned to *all* variables, and the MSE depends on the errors normal to the hyperplane spanned by the principal components because PCA performs an orthogonal projection onto them.

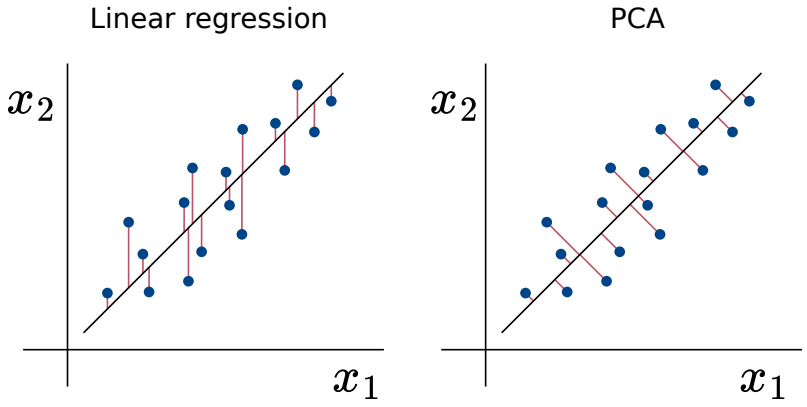


Figure 10.2. Geometric picture of PCA as opposed to linear regression.

Quick Aside: connection of PCA to Singular Value Decomposition. Singular Value Decomposition (SVD) is a standard matrix factorisation method, analogue to the diagonalisation of square matrices, but for rectangular matrices. In short, if we have a rectangular matrix $\mathbf{A}$ of dimensions $N \times p$ (let us take here $N > p$) and of rank R, the SVD decomposition of $\mathbf{A}$ has the form:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where $\mathbf{U} = \big(\mathbf{u}_1, \ldots, \mathbf{u}_N\big)$ is a $N \times N$ matrix of the left eigenvectors $\mathbf{u}_j$, $\mathbf{V} = \big(\mathbf{v}_1, \ldots, \mathbf{v}_p\big)$ is a $p \times p$ matrix of the right eigenvectors $\mathbf{v}_j$, the matrix $\mathbf{\Sigma}$ is a $N \times p$ matrix containing non-zero values $\sigma_1, \ldots, \sigma_R$ only along the diagonal - the *singular values*. They are the square root of the eigenvalues of $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$:

$$\left\{ \begin{array}{l} (\mathbf{A}^T\mathbf{A})\mathbf{v}_k = \sigma_k^2 \mathbf{v}_k \\ (\mathbf{A}\mathbf{A}^T)\mathbf{u}_k = \sigma_k^2 \mathbf{u}_k \end{array} \right\}$$

meaning that left and right eigenvectors (or singular vectors) come in pairs with the *same* σ_k^2. The fact that the eigenvalues of $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ are the square of the singular values also tells us that these matrices must be positive semi-definite.

A rank m approximation of $\mathbf{A}$ is then given by:

$$\mathbf{A} \approx \widehat{\mathbf{A}}_m = \sum_{j=1}^{m} \sigma_j \, \mathbf{u}_j \mathbf{v}_j^T =: \mathbf{U}_m \mathbf{\Sigma}_m \mathbf{V}_m^T$$

where $\mathbf{U}_m$ and $\mathbf{V}_m$ are truncated matrices with only the first m vectors $\mathbf{u}_j$ and $\mathbf{v}_j$, respectively. For more details, we refer to Deisenroth, Faisal, Ong *Mathematics for Machine Learning*, section 4.5.

If the matrix $\mathbf{A}$ is the dataset matrix $\mathbf{X}$:

$$\mathbf{X}_{N\times p} = \begin{pmatrix} x_1^{(1)} & \ldots & x_p^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \ldots & x_p^{(N)} \end{pmatrix} = \begin{pmatrix} {\mathbf{x}^{(1)}}^T \\ \vdots \\ {\mathbf{x}^{(N)}}^T \end{pmatrix}$$

then, we can apply SVD to effectively reduce the dimensions of our data. Let us look at the matrix of *centred data*, obtained as:

$$\mathcal{X}_{N\times p} := \mathbf{X} - \frac{1}{N}\mathbf{1}\mathbf{1}^T\mathbf{X} = \underbrace{\left(I - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right)}_{centring\ matrix}\mathbf{X},$$

In the above, we have introduced column vector of ones $\mathbf{1}_{N\times 1}$ such that:

$$\frac{1}{N}\mathbf{1}\mathbf{1}^T\mathbf{X} = \begin{pmatrix} \langle x_1\rangle & \dots & \langle x_p\rangle \\ \vdots & \ddots & \vdots \\ \langle x_1\rangle & \dots & \langle x_p\rangle \end{pmatrix}_{N\times p}$$

having denoted $\langle x_j\rangle = \frac{1}{N}\sum_{i=1}^{N} x_j^{(i)}$. In this way:

$$(\mathcal{X}^T\mathcal{X})_{p\times p} = N\,C_{\mathbf{x}} \qquad (C_{\mathbf{x}})_{p\times p} : \text{Data covariance matrix} \tag{10.6}$$

Building upon these results, we see that PCA follows from the SVD of $\mathcal{X}$:

$$\mathcal{X}_{N\times p} = \mathbf{U}_{N\times p}\mathbf{\Sigma}_{p\times p}\mathbf{V}^T_{p\times p}$$

and a rank m (with $m < p$) approximation of $\mathcal{X}$, $\widehat{\mathcal{X}}_{N\times p}$, can be found by:

$$\widehat{\mathcal{X}}_{N\times p} = \mathbf{U}_m\mathbf{\Sigma}_m\mathbf{V}^T_m =: \mathbf{X}_{\text{PCA}}\mathbf{V}^T_m$$

where the $N \times m$ matrix:

$$\mathbf{X}_{\text{PCA}} = \mathbf{U}_m\mathbf{\Sigma}_m \tag{10.7}$$

expresses the approximation $\widehat{\mathcal{X}}$ in terms of the m vectors contained in $\mathbf{V}^T_m$, which are the eigenvectors of the covariance matrix (see (1.1) and (10.6)). (Note that $\mathbf{\Sigma}$ contains the *ordered* singular values.) The matrix $\mathbf{X}_{\text{PCA}}$ thus contains the coordinates of each of the N data points in the m-dimensional space of the principal components (what we called $a_j^{(i)}$ in (10.2), with $j = 1, \dots, m$ and $i = 1, \dots, N$).

2. Extensions of PCA

The basic ideas of dimensionality reduction that have been introduced through PCA can be extended in many directions. We now briefly introduce two of them:

- how to deal with nonlinear data;
- how to produce sparse representations.

2.1. Nonlinear extensions: Kernel PCA. PCA is a linear method, i.e., it finds linear projections onto a space of reduced dimensionality. This begs the question of how to deal with nonlinear data that elude such methods, e.g., see Figure 10.3a. A first extension of PCA introduces the concept of kernels to dimensionality reduction. This should come as no surprise, since we have already introduced kernel functions as nonlinear extensions of SVMs in Chapter 6. The idea here is similar: to obtain a non-linear version of PCA by applying the kernel trick, substituting dot products in the mathematical formulation of the PCA algorithm by kernel functions.

The basic intuition behind the applicability of kernelisation is that PCA relies on the eigenvalues of $(\mathcal{X}^T\mathcal{X})_{p\times p}$, and/or $(\mathcal{X}\mathcal{X}^T)_{N\times N}$ (see aside above). The latter is precisely what can be thought of as 'kernel' matrices in this context. Indeed, the elements of the matrix $\mathcal{X}\mathcal{X}^T$ correspond to dot products of the vectors describing the *centred* p-dimensional samples:

$$\left(\mathcal{X}\mathcal{X}^T\right)_{ij} = \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \quad \text{Assuming: } \mathbf{x}^{(i)} \leftarrow (\mathbf{x}^{(i)} - \langle\mathbf{x}\rangle) \tag{10.8}$$

i.e., that the original variables $\mathbf{x}^{(i)}$ are already centred. Instead of these dot products, we can now consider a kernel function $k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, as properly defined in Chapter 6.

More precisely, like in kernel SMVs, the main idea in kernel PCA is to introduce a non-linear mapping $\boldsymbol{\phi}(\mathbf{x})$ that lifts the data to D dimensions, with $D > p$ (potentially $D >> p$). Let us assume that the transformed data vector $\boldsymbol{\phi}(\mathbf{x})$ is already centred, i.e., $\sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}^{(i)}) = 0$. We define the $N \times N$ matrix $\mathbf{K}$ (also called the Gram matrix) as:

$$K_{ij} = \boldsymbol{\phi}(\mathbf{x}^{(i)}) \cdot \boldsymbol{\phi}(\mathbf{x}^{(j)}) \equiv k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{10.9}$$

While in PCA we are interested in the spectral decomposition of $(C_{\mathbf{x}})_{p\times p}$, in kernel PCA we look at the one of:

$$(C_{\boldsymbol{\phi}})_{D\times D} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}^{(i)})\boldsymbol{\phi}(\mathbf{x}^{(i)})^T \tag{10.10}$$

which has eigenvalues λ_j and eigenvectors $\mathbf{v}_j$:

$$C_{\boldsymbol{\phi}}\mathbf{v}_j = \lambda_j \mathbf{v}_j \qquad j = 1, .., D \tag{10.11}$$

The eigenvectors $\mathbf{v}_j$ play the role of *non-linear* principal components. They can be expressed as a function of the set of $\boldsymbol{\phi}(\mathbf{x}^{(i)})$ through a set of coefficients $a_j^{(i)}$:

$$\mathbf{v}_j = \sum_{i=1}^{N} a_j^{(i)} \boldsymbol{\phi}(\mathbf{x}^{(i)}) \qquad j = 1, .., D \tag{10.12}$$

As we saw for SVMs (chapter 6), the introduction of kernels is convenient because we can work directly with them, without having to work out explicitly $\boldsymbol{\phi}(\mathbf{x})$. This is the key point of the kernel trick, and is valid also in this setting.

Kernel PCA construction: By plugging (10.12) into (10.11) and using the kernel definition (10.9), one obtains:

$$\mathbf{K}\mathbf{a}_j = N\lambda_j \mathbf{a}_j \quad \text{where: } \mathbf{a}_j = \{a_j^{(i)}\}_{i=1}^{N} \tag{10.13}$$

i.e., the coefficients $a_j^{(i)}$ in (10.12) are found as the eigenvectors of the Gram matrix $\mathbf{K}$. As a result, for a given data point $\mathbf{x}$ non-linearly mapped into $\boldsymbol{\phi}(\mathbf{x})$, its projection onto the j^{th} non-linear principal component is given by:

$$\boldsymbol{\phi}(\mathbf{x}) \cdot \mathbf{v}_j = \sum_{i=1}^{N} a_j^{(i)} \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{x}^{(i)}) = \sum_{i=1}^{N} a_j^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) \tag{10.14}$$

i.e., it can also be cast in terms of the kernel function only.

The result of kernel PCA can be seen as performing standard PCA in the non-linearly transformed space, resulting in non-linear dimensionality reduction in the original data space.

Kernel PCA in practice: centring the kernel. In equation (10.10), we have assumed $\boldsymbol{\phi}(\mathbf{x})$ to be already centred. This is not guaranteed to be the case even if $\mathbf{x}$ is centred, because $\boldsymbol{\phi}(\mathbf{x})$ lives in a different, D-dimensional space. In practice, $\boldsymbol{\phi}(\mathbf{x})$ won't be typically centred, hence one needs to work with:

$$\tilde{\boldsymbol{\phi}}(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{x}) - \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}^{(i)})$$

and the corresponding kernel $\tilde{K}_{ij} = \tilde{\boldsymbol{\phi}}(\mathbf{x}^{(i)}) \cdot \tilde{\boldsymbol{\phi}}(\mathbf{x}^{(j)})$. With some algebra, it can be shown that this centred version of the kernel can be found as:

$$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_N\mathbf{K} - \mathbf{K}\mathbf{1}_N + \mathbf{1}_N\mathbf{K}\mathbf{1}_N \tag{10.15}$$

where $\mathbf{1}_N$ is a $N \times N$ matrix with all elements equal to $1/N$. As such, this centred version of the Gram matrix performs a rescaling of its elements by N along with the mean subtraction. (See section 12.3 in Bishop, *Pattern Recognition and Machine Learning* for more details on the derivation). Once $\tilde{\mathbf{K}}$ has been computed from the data, the construction of kernel PCA (expressions 10.13 and 10.14) is the same, with $\mathbf{K} \to \tilde{\mathbf{K}}$.

Normalising the eigenvectors. The normalisation condition for the eigenvectors $\mathbf{v}_j$ translates into a normalisation condition for the eigenvectors $\mathbf{a}_j$ in (10.13):

$$1 = \mathbf{v}_j \cdot \mathbf{v}_j = \mathbf{a}_j^T \mathbf{K} \mathbf{a}_j = N\lambda_j \mathbf{a}_j \cdot \mathbf{a}_j$$

that we have to enforce before using their elements for the projections (10.14). With the centred Gram matrix (10.15), which already includes a rescaling by N, the normalisation condition becomes:

$$1 = \mathbf{v}_j \cdot \mathbf{v}_j = \mathbf{a}_j^T \tilde{\mathbf{K}} \mathbf{a}_j = \lambda_j \mathbf{a}_j \cdot \mathbf{a}_j$$

λ_j being now the eigenvalues of $\tilde{\mathbf{K}}$.

Example. An example of a kernel is the radial basis (or Gaussian) kernel function:

$$K_{ij} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = e^{-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{c}} \tag{10.16}$$

where c is a hyperparameter controlling the width of the kernel. Kernel PCA would work well with the data shown in Figure 10.3a, and Figure 10.3b shows the outcome of applying precisely this kernel PCA with a radial kernel. By effectively projecting the data onto a non-linear manifold spanned by the non-linear principal components, kernel PCA produces a representation that separates more straightforwardly data points with different properties (as indicated by the colour).

Finally, note that the standard PCA algorithm is recovered as a special case if we use the linear kernel (10.8).

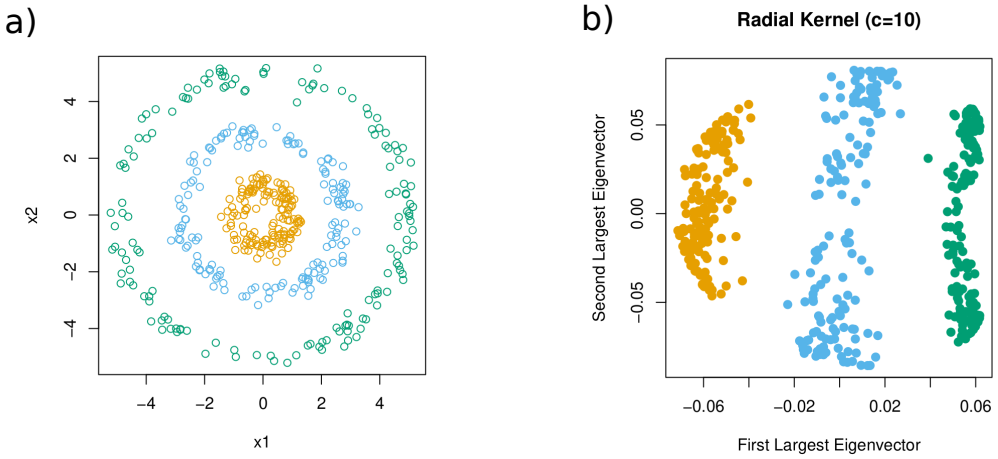


Figure 10.3. a) Example of non-linearly distributed data, that standard PCA is not able to handle very well. Instead, we need to use kernel PCA methods to obtain a more informative representation of such data. Figure adapted from Hastie, Tibshirani, Friedman, *The Elements of statistical learning*, Chap. 14.

2.2. Sparse PCA. In this second extension to PCA, *sparse PCA*, the aim is to alleviate the fact that PCA yields non-sparse descriptions of the data. In many cases this is not desirable, and so the goal of this method is to increase interpretability through sparsity. This concept is not new either—we have seen similar ideas previously when considering sparse versions of linear regression in Chapter 1.

First, recall our concise representation extracted from PCA in (10.7):

$$(\mathbf{X}_{\text{PCA}})_{N\times m} = \mathbf{U}_m\mathbf{\Sigma}_m$$

whose rows contain the coordinates of each data point in terms of the m principal vectors $\mathbf{V}_m = \big(\mathbf{v}_1 \dots \mathbf{v}_m\big)_{p\times m}$. By construction, $\mathbf{V}$ is a *non-sparse* basis in terms of the p original variables of the dataset, which in turn leads to limited interpretability.

To increase the sparsity of the representation, sparse PCA considers penalty terms that shrink the magnitude of the basis' vectors. To introduce its formulation, we need first to highlight a nice property of PCA: it can be shown that the principal components $\big(\mathbf{v}_1 \dots \mathbf{v}_p\big)$ are such that the linear combination $\mathcal{X}\mathbf{v}_1$ has the highest variance among all linear combinations of the p features; $\mathcal{X}\mathbf{v}_2$ has the highest variance among all linear combinations under the condition that $\mathbf{v}_2$ is orthogonal to $\mathbf{v}_1$, etc. In other words, principal components identify, in order, the directions of maximal variance in the data. An alternative way of deriving the original PCA is therefore by maximisation of the data variance:

$$\max_{\mathbf{v}} \mathbf{v}^T(\mathcal{X}^T\mathcal{X})\mathbf{v}, \text{ where } \mathbf{v} \in \mathbb{R}^p \text{ such that } \mathbf{v}^T\mathbf{v} = 1$$

We can induce sparsity on the vectors $\mathbf{v}_j$ by adding extra constraints, similarly to what we did in later sections of chapter 1 with LASSO:

$$\sum_{j=1}^{p} |\mathbf{v}_j| \leq t$$

This will essentially lead to PCA attempting to find a basis that tries to maximise the data variance (or, equivalently, to minimise the quadratic error) under sparsity constraints, which we can write down as:

$$\{\mathbf{v}_j^{\text{LASSO}}\}_{j=1}^{p}$$

and the $p\times m$ matrix $\mathbf{V}_m^{\text{LASSO}}$ will be sparser than the ordinary $\mathbf{V}_m$, with $\mathbf{v}_j^{\text{LASSO}}$ containing many zeros:

$$\mathbf{v}_j^{\text{LASSO}} = \begin{pmatrix} \blacksquare \\ 0 \\ \vdots \\ \blacksquare \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

The ■ stand for the non-zero elements indicating the fact that the sparse principal components can be expressed in terms of few of the original coordinates of the data. The fewer non-zero elements give a sparser representation of the main direction of variation in the data, helping identify the few features that contribute most to it. This is how sparsity in this context enhances interpretability: it performs feature selection, facilitating the generation of hypothesis based on a focussed set of relevant features. Please see Hastie, Tibshirani, Friedman, *The Elements of statistical learning*, Section 14.5.5 for the discussion of sparse PCA in real data analysis tasks.

3. Non-negative Matrix Factorisation (NMF)

Another possible limitation of PCA, i.e., the standard SVD matrix factorisation is not restricted to positive values in the matrices. However, in many applications, the data variables are restricted to be non-negative. An example of such datasets is images (where the description of the image is a grayscale value) or datasets of counts of events. Hence one would want to have a description that respects this *positivity*, and expresses the reduced representation as a linear combination of non-negative components.

The method that achieves this is called *Non-negative Matrix Factorisation* (NMF). This method was known from the 1970's and 80's but it was revitalised by Seung and Lee in a 1999 seminal paper `https://www.nature.com/articles/44565`.

To see how this is done, we once again remind ourselves of standard PCA. The approximation of our centred data, $\widehat{\mathcal{X}}$, by a projection onto the first m principal components can be written in terms of a matrix factorisation via the connection of PCA to SVD (see previous aside):

$$\widehat{\mathcal{X}} = \mathbf{U}_m \mathbf{\Sigma}_m \mathbf{V}_m^T =: \mathbf{X}_{\text{PCA}} \mathbf{V}_m^T$$

Similarly to PCA, in NMF the basic idea is to perform dimensionality reduction via matrix factorisation. We start from our usual $N \times p$ data matrix $\mathbf{X}$:

$$\mathbf{X}_{N\times p} = \begin{pmatrix} x_1^{(1)} & \dots & x_p^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \dots & x_p^{(N)} \end{pmatrix} = \begin{pmatrix} {\mathbf{x}^{(1)}}^T \\ \vdots \\ {\mathbf{x}^{(N)}}^T \end{pmatrix} =: \begin{pmatrix} X_{11} & \dots & X_{1p} \\ \vdots & \ddots & \\ X_{N1} & & X_{Np} \end{pmatrix},$$

where we have introduced the notation X_{ij} for convenience.

> **Key Message:** In NMF, the data matrix $\mathbf{X}$ is approximated by a lower-dimensional representation given by:
>
> $$\boldsymbol{X} \approx \boldsymbol{W}\boldsymbol{H}, \quad \text{such that} \quad W_{ij} \geq 0, H_{ij} \geq 0 \quad \text{for all} \quad i, j$$
>
> i.e., where we require a factorisation into matrices with non-negative elements.

In the above, $\boldsymbol{W}$ is $N \times r$ and $\boldsymbol{H}$ is $r \times p$ matrix, where $r \ll p$ is the number of basis vectors, as follows:

$$\begin{pmatrix} X_{11} & \dots & X_{1p} \\ \vdots & \ddots & \vdots \\ X_{N1} & \dots & X_{Np} \end{pmatrix} \approx \begin{pmatrix} W_{11} & \dots & W_{1r} \\ \vdots & \ddots & \vdots \\ W_{N1} & \dots & W_{Nr} \end{pmatrix} \begin{pmatrix} H_{11} & \dots & H_{1p} \\ \vdots & \ddots & \vdots \\ H_{r1} & \dots & H_{rp} \end{pmatrix} =: \begin{pmatrix} {\mathbf{w}^{(1)}}^T \\ \vdots \\ {\mathbf{w}^{(N)}}^T \end{pmatrix} \boldsymbol{H}$$

Hence each sample is represented in terms of the r factors:

$$\mathbf{x}_{p\times 1}^{(i)} = \boldsymbol{H}^T \mathbf{w}_{r\times 1}^{(i)}, \quad i = 1, \dots, N.$$

r being the reduced dimensions of this representation. $\boldsymbol{H}$ contains the non-negative equivalent of the basis for expansion (the principal components) of PCA, and the matrix $\boldsymbol{W}$ contains the non-negative coefficients expressing the data $\mathbf{x}^{(i)}$ in the basis of vectors in $\boldsymbol{H}$, thus specifying the coordinates of the lower-dimensional NMF data representation.

3.1. NMF learning algorithms. To find $\boldsymbol{H}$ and $\boldsymbol{W}$, we need to solve an optimisation problem, where we have an objective (or cost) function to minimise that quantifies the quality of the approximation, similarly to PCA.

In the literature on NMF, there exist variants of such cost function. The two most standard cost functions are the ones analysed by Lee and Seung in `https://papers.nips.cc/paper/2000/file/f9d1152547c0bde01830b7e8bd60024c-Paper.pdf`. The first is the squared Euclidean distance between the data $\boldsymbol{X}$ and the product $\boldsymbol{WH}$:

$$||\boldsymbol{X} - \boldsymbol{WH}||^2 = \sum_{ij}(X_{ij} - (\boldsymbol{WH})_{ij})^2$$

Note that this function is not convex in both matrices together, therefore one uses techniques from numerical optimisation to find local minima. To this end, Lee and Seung introduced **multiplicative update rules** to iteratively learn a non-negative matrix factorisation. Since the joint optimisation is a very hard problem, the basic idea of the multiplicative update rules is to first fix the factor $\boldsymbol{W}$ and to optimise with regard to $\boldsymbol{H}$, and then to fix the learned factor $\boldsymbol{H}$ and to optimise with regard to $\boldsymbol{W}$. The multiplicative rules that guarantee a non-increasing Euclidean distance are:

$$W_{ij} \leftarrow W_{ij}\frac{(\mathbf{X}\mathbf{H}^T)_{ij}}{(\mathbf{W}\mathbf{H}\mathbf{H}^T)_{ij}} \tag{10.17}$$

$$H_{jk} \leftarrow H_{jk}\frac{(\mathbf{W}^T\mathbf{X})_{jk}}{(\mathbf{W}^T\mathbf{W}\mathbf{H})_{jk}} \tag{10.18}$$

The multiplicative update rules are guaranteed to converge to limiting matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ forming a local minimum of the cost function. One major benefit of using multiplicative update rules is that the non-negativity constraints are enforced automatically. Furthermore, the multiplicative update rules are easy to implement and the convergence is relatively fast (see coding task).

The second type of cost function is a divergence defined as:

$$D(\boldsymbol{X}||\boldsymbol{WH}) = \sum_{ij}\left[X_{ij}\log\left(\frac{X_{ij}}{(\boldsymbol{WH})_{ij}}\right) - X_{ij} + (\boldsymbol{WH})_{ij}\right]$$

Using the multiplicative update rules above, one obtains that the divergence reaches a plateau, however the update rules that make it strictly non-increasing are slightly different:

$$W_{ij} \leftarrow W_{ij}\frac{\sum_k H_{jk}X_{ik}/(\boldsymbol{WH})_{ik}}{\sum_k H_{jk}} \tag{10.19}$$

$$H_{jk} \leftarrow H_{jk}\frac{\sum_i W_{ij}X_{ik}/(\boldsymbol{WH})_{ik}}{\sum_i W_{ij}} \tag{10.20}$$

They provide an alternative algorithm to learn the NMF of a dataset of interest, and they are as well explored in the coding task.

> *Final Aside.* For more information about NMF, you can read the original paper by Seung and Lee: `https://www.nature.com/articles/44565` that we make available as additional reading. It illustrates nicely how learning non-negative factorisations can lead to more interpretable reduced representations in the case of images. The non-negative vectors correspond to parts of images that are more recognisable and interpretable to the human eye. You will gain some more intuition of the differences between NMF and PCA through your coding tasks.

Graph-based learning

In this final chapter, we will demonstrate how ideas and tools from graph theory can be leveraged to build powerful models, and how graph-based learning is on the rise in a wide range of current research applications. We will see that methods inspired by graph theory are both efficient and effective for data analysis. Furthermore, many aspects will be closely related to concepts described in the chapters above, and have served as inspirations for extensions of algorithms that were originally formulated from different perspectives.

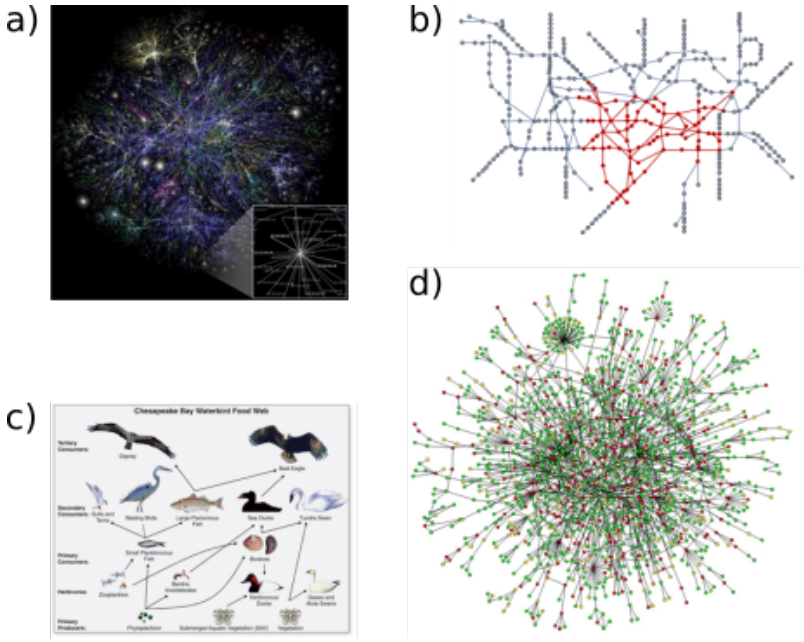


Figure 11.1. A few real-world examples of graphs. a) An illustration of the internet, where each node is a server and the edges are coloured according to traffic between them. b) A schematic of an Underground Map (unfortunately, the real map for the London underground is under heavy copyright rules and cannot be used here). The tube network is a graph of stations connected by the lines between them. c) Food webs are also a type graph, with each interaction being a predator-prey relation between species of animals. d) The interactions between proteins in a cell or a bacterium (in this case *Saccharomyces cerevisiae* or more commonly known as yeast) form a graph too.

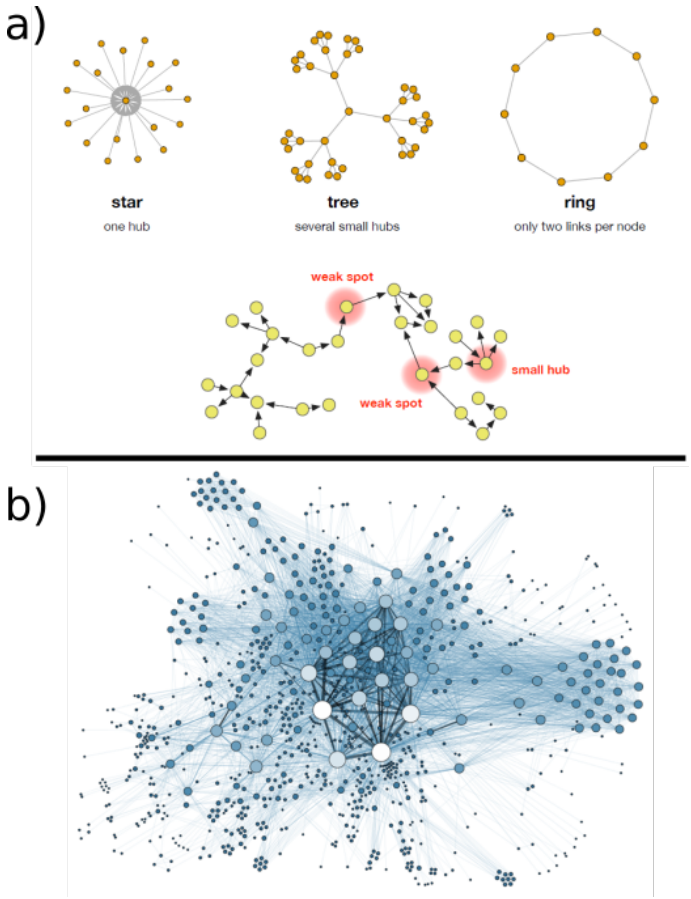


Figure 11.2. a) Small networks are intuitive, b) but large ones are not (at all). This motivates the use of graph-based learning methods to understand highly complex data.

1. Graph theory preliminaries

1.1. Graphs and data. To link the use of graphs for data analysis with our previous chapters, we start once more with a data set, with N samples, written in the usual format as a vector in p-dimensional space:

$$\{\mathbf{x}^{(i)}\}_{i=1}^{N} \qquad \mathbf{x}^{(i)} \in \mathbb{R}^p \qquad \mathbf{x}^{(i)} = \begin{pmatrix} x_1^{(i)} \\ \vdots \\ x_p^{(i)} \end{pmatrix}$$

As before, this can be summarised in matrix form:

$$X_{N\times p} = \begin{pmatrix} {\mathbf{x}^{(1)}}^T \\ \vdots \\ {\mathbf{x}^{(N)}}^T \end{pmatrix}$$

In the previous chapters, the goal was to find separable subsets, which was coupled with some notion of distance. Likewise, with the dimensionality reduction methods, we were aiming to obtain projections in lower dimensions while capturing most of the variation in the data. In both cases, the geometry of the dataset played the most important role.

The key ingredient to many of these methods was the notion of similarity or dissimilarity (in some cases a distance) between samples. This yielded an $N \times N$ matrix summarising the

pairwise relationships. Examples of both similarities (S) and dissimilarities (D) included:

$$\begin{aligned}
(\textit{cosine similarity}) \qquad & S_{ij} = \frac{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}}{\|\mathbf{x}^{(i)}\| \, \|\mathbf{x}^{(j)}\|} \\
(\textit{statistical similarity}) \qquad & S_{ij} = \frac{\mathrm{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})}{\sqrt{\mathrm{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(i)})}\sqrt{\mathrm{cov}(\mathbf{x}^{(j)}, \mathbf{x}^{(j)})}} \\
(\textit{Euclidean distance}) \qquad & D_{ij} = \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\| \\
(\textit{radial kernel matrix }) \qquad & D_{ij} = \exp\left[-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{c}\right]
\end{aligned}$$

In most of the unsupervised learning methods, we had to define pairwise (dis)similarities between samples, which were the important determinants of the structure of the dataset. In other words, what mattered most about our data sets where the pairwise interactions between them. This is the insight that leads into graph representations of data.

Even further, in many applications, the problem is in fact directly and only defined by the pairwise (dis)similarities, that is we do not know anything about the $\mathbf{x}^{(i)}$, but rather simply have a $N \times N$ matrix S or D, compiling the binary relationships between samples.

This is the basis for the definition of graphs in data analysis: **we will consider graphs with N nodes, where each sample is a *node* and the relationships between pairs of nodes (as given by the matrix S or D) will define E *(weighted) edges* between the nodes.** We now see a few definitions on graphs. (Many of you will have seen many of these concepts in the course in Network Science.)

> *Quick Aside:* Alternatively, in many applications, a $p \times p$ matrix of pairwise measures of statistical dependency of the features of the dataset can be estimated instead. This yields a network of interdependencies between the features of the dataset. These measures include statistical relationships such as *Pearson correlations*, *Mutual Information*, *Granger Causality*, *Cosine Similarity* etc. The networks obtained from these measures are useful in datasets where features represent specific parts of a system that the data represent. For instance, time-series data recorded from different parts of the brain can be transformed into a network of interactions among the brain regions to study the interaction structure of the brain. For a review of network construction and analysis used in neuroscience, see *Bassett, D., Sporns, O. Network neuroscience., Nat Neurosci 20, 353–364 (2017).*

1.2. Graphs: some definitions. A *graph* is defined to be a combinatorial object on two sets: a set of nodes $V = \{i\}_{i=1}^N$ (aka vertices) and a set of edges $E = \{(i,j)\}_{i \sim j}$ (aka links). In mathematical terms, a graph is then defined as $G(V, E)$.

In the context of graph-based learning, the parallels are now obvious: the set of nodes is the set of samples:

$$\{\mathbf{x}^{(i)}\}_{i=1}^N \mapsto \{i\}_{i=1}^N$$

and the edges are defined from the entries of S or D.

In Figure 11.3, we show a very simple and basic example of a graph where $N = 4$. In this specific example, we have the following node and edge sets:

$$\begin{aligned}
V &= \{1, 2, 3, 4\} \\
E &= \{(1,2), (2,3), (2,4)\}
\end{aligned}$$

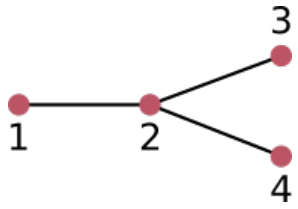


Figure 11.3. A simple example of a graph with $N = 4$.

This can be easily and equivalently summarised using the *adjacency matrix* $A_{N\times N}$. In the example above, this is:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The definition of the adjacency matrix is therefore:

$$A_{ij} = \begin{cases} 1, & \text{if nodes } i \text{ and } j \text{ are connected, i.e. } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

From the adjacency matrix, we can also obtain the *degree vector*:

$$\mathbf{d} = A\mathbf{1} = \begin{bmatrix} 1 \\ 3 \\ 1 \\ 1 \end{bmatrix}$$

As you can see, the *degree* of a particular node is simply the number of edges it makes with other nodes. In our example above, node 1 has degree 1, node 2 has degree 3, and so on.

Note that the graph above is *undirected*, i.e., there is no direction associated to the edges, which implies that $A = A^T$.

1.3. Directed graphs. For the case of a *directed graph*, the latter equality does not necessarily hold and thus in most cases in fact we have $A \neq A^T$. An example of a directed graph is given in Figure 11.4.

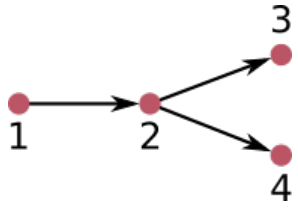


Figure 11.4. A simple example of a *directed* graph.

The corresponding adjacency matrix is given by:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Note that for directed graphs, there are two types of degrees: the *in-degree* (the number of incoming edges of a particular node) as well as the *out-degree*, which can be written from the adjacency matrix as:

$$\mathbf{d}_{\text{in}} = A\mathbf{1} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathbf{d}_{\text{out}} = A^T\mathbf{1} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

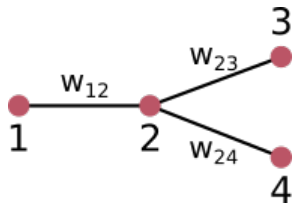


Figure 11.5. An example of a weighted graph.

1.4. Weighted graphs. These are graphs where each edge has an associated *weight*, in most cases a number in $\mathbb{R}$. We then denote the weight between the nodes i and j by w_{ij}. In the example of Figure 11.5, the adjacency matrix would then look like the following:

$$A = \begin{bmatrix} 0 & w_{12} & 0 & 0 \\ w_{12} & 0 & w_{23} & w_{24} \\ 0 & w_{23} & 0 & 0 \\ 0 & w_{24} & 0 & 0 \end{bmatrix}$$

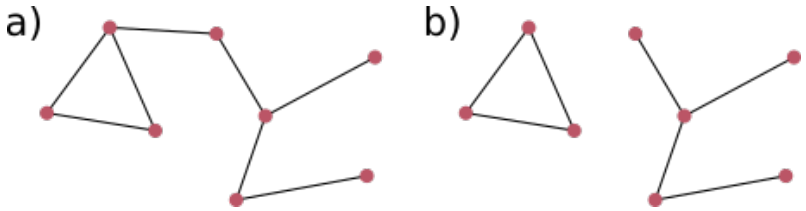


Figure 11.6. a) An example of a connected graph. **b)** An example of a disconnected graph with two components.

1.5. Connected graphs. Finally, we also define the concept of *connected graph*, as a graph where every possible pair of nodes is connected by a path. Figure 11.6 gives examples of both a connected as well as a disconnected graphs.

A disconnected graph has multiple components, whereas a connected graph has only one component. It can be shown trivially that if the graph is connected, the adjacency A has full rank and cannot be rearranged in block-diagonal form (i.e., the matrix is irreducible). You can immediately see that the clustering problem on graphs is directly related to making the adjacency matrix as block-diagonal as possible by relabelling the nodes (as seen in Chapter 9). Remembering that the nodes are samples, a good clustering implies

that the graph is 'close' to being disconnected, i.e., it could be made disconnected by cutting a few edges (the *graph partitioning* problem in graph theory).

Note that for directed graphs there exist two types of connectivity, strong and weak. In the weaker case, the definition is the same as for undirected graphs: If all directed edges were to be replaced by regular edges, one would get a connected graph. However, being strongly connected means that there is a *directed* path for every pair of nodes.

1.6. Distances on graphs. Another important concept in graph-based learning is that of *graph distance*, to be understood as *distance on the graph*. As with the concept of (dis)similarities, there are many different definitions that can be used. Two of the most popular measures of graph-based distances use the concept of paths on the graph:

- The **geodesic distance** is perhaps the most popular and is simply defined to be the *minimal* number of edges between two nodes, i.e., if we denote the set of all paths between i and j as $\{i, j\}$, we can write the geodesic distance as:

$$\min_{\{i,j\}}(\#\text{ edges in }\{i,j\})$$

- The **average distance** between two nodes i and j is also simply the average length of all paths connecting them:

$$\langle \#\text{ edges}\rangle_{\{i,j\}}$$

Note that if the graph is weighted, then the length of a path is the sum of the weights of the edges on the path. This allows us to take into account some of the geometric information of the data set.

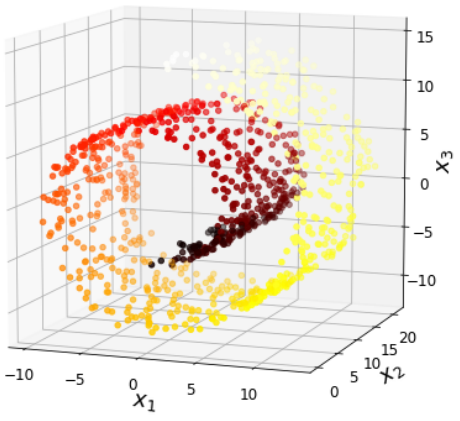


Figure 11.7. The 'Swiss roll' data set.

As we will see later, **distances on graphs can sometimes provide a much better representation of the intrinsic geometry of the data than Euclidean distances in** $\mathbb{R}^p$. One such dataset is the so-called *swiss roll* dataset, which is plotted in Figure 11.7. You can immediately see that a distance compiled on a well chosen graph representing the data set will be much more informative of the structure of this data set than the set of Euclidean distances in $\mathbb{R}^3$.

There are other measures of distance and of connectivity that emerge from considering dynamical processes on graphs related to diffusion. To introduce those, we now define the Laplacian matrix of a graph.

1.7. The Laplacian matrix of a graph. This matrix is essentially an extension of the Laplacian operator to graphs. It stems from the heat equation (aka the diffusion equation):

$$u = u(x,t), \quad \frac{\partial u}{\partial t} = \nabla^2 u$$

Here, ∇^2 is the Laplacian operator, defined as:

$$\nabla^2 u = \frac{d}{dx}\left(\frac{d}{dx}u\right)$$

If we apply a discretisation to space, i.e., $\mathbf{u} = \begin{pmatrix} u_1 & \dots & u_N \end{pmatrix}^T$, then we can write:

$$\begin{aligned} \nabla^2 u &= \frac{d}{dx}\left(\frac{d}{dx}u\right) \\ &\approx \Delta(\Delta u) = \Delta(u_{k+1} - u_k) \\ &= (u_{k+2} - u_{k+1}) - (u_{k+1} - u_k) \\ &= u_{k+2} - 2u_{k+1} + u_k \end{aligned}$$

In terms of graphs, the discretisation of the space can essentially be expressed as a graph of nodes along a line as shown in Figure 11.8.

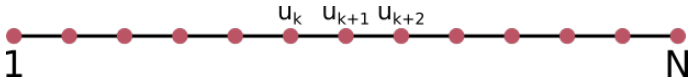


Figure 11.8. A discretisation of 1D space resulting in a graph consisting of a string of nodes connected to the next node by one edge each.

The heat equation on the discretised space can now be rewritten in terms of a tri-diagonal $N \times N$ matrix:

$$\frac{d\mathbf{u}}{dt} = \underbrace{\begin{pmatrix} -1 & 1 & & & & & \\ 1 & -2 & 1 & & & 0 & \\ & \ddots & \ddots & \ddots & & & \\ & & 1 & -2 & 1 & & \\ & & & \ddots & \ddots & \ddots & \\ & 0 & & & 1 & -2 & 1 \\ & & & & & 1 & -1 \end{pmatrix}}_{\text{Laplacian matrix: } -L} \begin{pmatrix} u_1 \\ \vdots \\ u_k \\ u_{k+1} \\ u_{k+2} \\ \vdots \\ u_N \end{pmatrix} \tag{11.1}$$

Note that the adjacency matrix of the graph in Figure 11.8 is given by:

$$A = \begin{pmatrix} 0 & 1 & & \\ 1 & 0 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 0 \end{pmatrix}$$

and the degree matrix is $\mathcal{D} = \mathrm{diag}(\mathbf{d}) = \mathrm{diag}(A\mathbf{1})$.

It is then easy to see that the matrix in (11.1) is simply:

$$L = \mathcal{D} - A$$

L is called the *Laplacian matrix*. More specifically, L is commonly termed the *Combinatorial Graph Laplacian*. The spectral decomposition of the Laplacian matrix plays an important role in clustering and dimensionality reduction in graph-based learning, which we will explore in the following sections.

2. Clustering on graphs – graph partitioning

As mentioned above, there are two main problems that frequently appear in the field of unsupervised learning: dimensionality reduction and clustering. Graph-based learning methods are particularly suited for the latter but can also be used to do the former. We will first introduce approaches of how to tackle the problem of clustering data through graph-based methods. Then in Section 3 we will see how graphs allow us to perform dimensionality reduction – in particular, *nonlinear* dimensionality reduction.

As discussed in Chapter 9, clustered data should be similar within clusters and dissimilar between clusters. In graphs, this is analogue to the idea of *graph partitioning*, i.e. finding splits of the graph into subgraphs so that the nodes are partitioned into mutually exclusive subsets. Equivalently, one is trying to find cuts across a set of edges to produce partitions. If cutting a graph into separate subgraphs is 'easy', this is indicative of the presence of potentially good clusters.

2.1. Graph construction. In the introduction to this chapter, we mentioned that in some cases data is given directly in the form of a graph (e.g., a social network, or a citation graph), whereas in other cases our data might be a collection of samples in the form of high-dimensional vectors $\mathbf{x}^{(i)} \in \mathbb{R}^p, \quad i = 1, \dots, N$. In this second case, it might still be advantageous to describe the data set as a graph so that, e.g., one can approach the clustering problem as a graph partitioning problem.

Naturally, the first step to doing clustering using graphs is the graph construction from data. If we have traditional tabular data, graph construction usually starts from matrices of (dis)similarities $D_{N\times N}$ or $S_{N\times N}$, as described in Section 1.1. As an alternative viewpoint to multivariate statistics/calculus, these matrices can then be viewed as *adjacency matrices* of weighted complete graphs, i.e. a graph where every node is connected to every single other node via a weighted edge such that the adjacency matrix is 'full' (the graph in these cases is also called 'fully connected'). However, from a graph-theoretical perspective such 'weighted complete graphs' are usually not very helpful—they are too dense, with little structure and are not suited for the analysis using graph properties. Hence they tend not to be used in this form, and a *sparsification* method is usually applied to reduce the number of edges, i.e., to create zeros in the adjacency matrix. There are a few methods that are applied widely (but this is a very active area of current mathematical research):

- **Thresholding** is simply removing all edges below (or above) a certain threshold. This is usually not the best strategy, but it is done sometimes in very simple applications. Things that can go wrong are graphs that are sensitive to tiny changes in the threshold, where parts can become disconnected as a result.
- **Geometric graphs:**
  - **ε-ball:** This method only connects nodes that are within some distance ε of each other, where ε is a hyperparameter to be tuned.
  - **k-NN:** This is similar to the above, except the k nearest neighbours of each node will be connected to it. Here k is a hyperparameter to be tuned. There

is a wide set of variations on k-NN constructions: standard k-NN, mutual k-NN, and continuous k-NN. [1]

Quick Aside: sparsifications based on the minimum spanning tree. The minimum spanning tree (MST) of the graph connects all N nodes of the graph with just $N-1$ edges such that the sum of the weights of the chosen edges is minimal. Once the MST is found (there are good algorithms for this), then edges are added according to different criteria (spectral or distance based) up to a given sparsity regulated by a given hyperparameter. The spectral sparsification algorithm by Spielman and Srivastava[a] is a beautiful example of this approach.

[a] *Graph Sparsification by Effective Resistances*, Daniel A. Spielman and Nikhil Srivastava, SIAM Journal on Computing 2011 40:6, 1913-1926

It should be evident from our descriptions above that the level of sparsity can be adjusted with one (or more) parameters (e.g., k in k-NN or ε in ε-ball). These can be thought in many cases as hyperparameters to be chosen with a particular task in mind so that the properties of the data set are best captured by the graph.

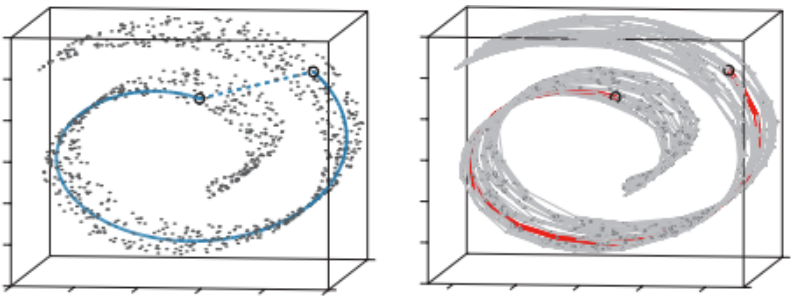


Figure 11.9. The Swiss roll data set with a geometric graph. This example gives a visual intuition of why a geometric graph (say ε-ball or k-NN) with a well chosen sparsity could be a better representation of this data set (as compared to a full distance matrix $D_{N\times N}$ in Euclidean space). Why random walks or diffusions on such a graph could be used to capture the geometry of the data set is also immediately evident from the picture.

2.2. Spectral partitioning of graphs. Once we have a graph, we can apply some of the results we have obtained during this course and in this chapter to compute a partitioning of the graph into separate subgraphs. Similarly to methods described previously in this course, this process is usually linked to a cost function. In the case of graph partitioning, the cost function is oftentimes (but not always!) related to the 'size of the cut' (i.e., the number of edges or the sum of the weights of the edges) that are necessary to be cut so as to split the graph into k separate subgraphs. The separated subgraphs will correspond to the clusters of data points. Mathematically, we can formalise this notion as follows. We will consider in detail the problem of bipartitions, i.e., splitting a given graph into $k = 2$ clusters S_1 and S_2. (The ideas behind this approach can be extended to $k > 2$ but the extensions are not trivial and will be mentioned at the end of the section.)

Given S_1 and S_2, the cost function that counts the number of edges needed to split the graph into two subgraphs (i.e., the *cut*) can be written in terms of the graph adjacency matrix A as follows:

$$C = \frac{1}{2} \sum_{i\in S_1, j\in S_2} A_{ij}$$

[1] *Geometric graphs from data to aid classification tasks with graph convolutional networks*, Y. Qian, P. Expert, P. Panzarasa, M. Barahona, Patterns 2, 4 (2021): 100237.

Note how we are selecting the edges that link a node i in S_1 with a node j in S_2. In order to simplify our derivation, it is helpful to define an indicator variable s_i:

$$s_i = \begin{cases} +1 & \text{if } i \in S_1 \\ -1 & \text{if } i \in S_2 \end{cases}$$

from which we have:

$$t_{ij} = \frac{1}{2}(1 - s_i s_j) = \begin{cases} 1 & \text{if } i,j \text{ are in different clusters} \\ 0 & \text{if } i,j \text{ are in the same cluster} \end{cases}$$

With these definitions, we rewrite the cost function as:

$$C = \frac{1}{4}\sum_{i,j} t_{ij} A_{ij} = \frac{1}{4}\left(\sum_{i,j} A_{ij} - \sum_{i,j} A_{ij} s_i s_j\right)$$

Note that we can rewrite:

$$\sum_{i,j} A_{ij} = \sum_i \left(\sum_j A_{ij}\right) = \sum_i d_i = \sum_i d_i s_i^2 = \sum_{i,j} d_i s_i s_j \delta_{ij}$$

where δ_{ij} is the Kronecker delta. Then we finally have:

$$C = \frac{1}{4}\sum_{i,j} \big(\underbrace{d_i \delta_{ij} - A_{ij}}_{L_{ij}}\big) s_i s_j = \frac{1}{4}\mathbf{s}^T L \mathbf{s} \tag{11.2}$$

where we have rewritten the cost function in terms of L, the (combinatorial) graph Laplacian.

Our goal is thus to find a partition with a small cut. That is, we want to minimise the cost function:

$$\min_{\mathbf{s}\in\{-1,+1\}^N} \mathbf{s}^T L \mathbf{s} \qquad \text{under the constraints } \begin{cases} \mathbf{s}^T \cdot \mathbf{s} & = n_1 + n_2 = N \\ \mathbf{s}^T \cdot \mathbf{1} & = n_1 - n_2 \end{cases} \tag{11.3}$$

Note that $\mathbf{s} \in \{-1,+1\}^N$ by definition. Unfortunately, the optimisation 11.3 cannot be solved analytically as it is a combinatorial optimisation (the aim of the optimisation is to assign two discrete labels to the nodes of the graph). Hence this is a 'hard' optimisation problem and, as we know from earlier chapters, complete enumeration of the space of graph partitions is out of computational bounds for even small graphs.

To solve this problem in practice, we rely on optimisation techniques. As discussed from the beginning of the course, one such method is called *relaxation*, which aims to solve an easier problem that is closely related to the original one. In this case, the relaxation is to solve the optimisation (11.3) for $\mathbf{s} \in \mathbb{R}^N$ rather than in the original discrete space $\{-1,+1\}^N$ (a hypercube). Now that we have a continuous optimisation, we can use the method of Lagrange multipliers for constrained optimisation. We thus define the Lagrangian with two Lagrange multipliers λ, μ:

$$\mathcal{C}(\mathbf{s},\lambda,\mu) = \mathbf{s}^T L \mathbf{s} + \lambda(N - \mathbf{s}^T \cdot \mathbf{s}) + 2\mu\big((n_1 - n_2) - \mathbf{s}^T \cdot \mathbf{1}\big), \qquad \mathbf{s} \in \mathbb{R}^N, \lambda \in \mathbb{R}, \mu \in \mathbb{R}.$$

As we have done many times before, we now seek $\mathbf{s}^\star$ such that:

$$\nabla_{\mathbf{s}} \mathcal{C}\big|_{\mathbf{s}^\star} = 0.$$

At this $\mathbf{s}^\star$, we then have:

$$L\mathbf{s}^\star = \lambda \mathbf{s}^\star + \mu \mathbf{1} \tag{11.4}$$

Using $L\mathbf{1} = 0 = \mathbf{1}^T L$, we know that:

$$\mathbf{1}^T L\mathbf{s}^\star = \lambda \underbrace{\mathbf{1}^T \cdot \mathbf{s}^\star}_{=(n_1-n_2)} + \mu \underbrace{\mathbf{1}^T \cdot \mathbf{1}}_{=N} = 0$$

And hence:

$$\frac{\mu}{\lambda} = -\frac{n_1 - n_2}{N} \tag{11.5}$$

We then solve equation (11.4) for $\mathbf{s}^\star$:

$$L\mathbf{s}^\star = \lambda\left(\mathbf{s}^\star + \frac{\mu}{\lambda}\mathbf{1}\right)$$

Again, because $L\mathbf{1} = 0$, we can add a term to the right hand side so that:

$$L\mathbf{s}^\star = L\left(\mathbf{s}^\star + \frac{\mu}{\lambda}\mathbf{1}\right) = \lambda\left(\mathbf{s}^\star + \frac{\mu}{\lambda}\mathbf{1}\right)$$
$$\implies L\mathbf{v} = \lambda\mathbf{v}$$

where we have defined:

$$\mathbf{v} := \mathbf{s}^\star + \frac{\mu}{\lambda}\mathbf{1}. \tag{11.6}$$

This is the main result: the solution to this (relaxed) optimisation problem is given by **an eigenvector of the Laplacian matrix v and its associated eigenvalue** λ.

The remaining question is now: *which eigenvector is it?* If we plug this result into the cost function (11.2), we get:

$$C^\star = \frac{1}{4}\mathbf{s}^{\star T} L\mathbf{s}^\star = \frac{1}{4}\mathbf{v}^T L\mathbf{v} = \frac{1}{4}\lambda\,\mathbf{v}^T\mathbf{v}$$

where we have:

$$\begin{aligned}\mathbf{v}^T\mathbf{v} &= \mathbf{s}^{\star T}\mathbf{s}^\star + 2\frac{\mu}{\lambda}\mathbf{1}^T\mathbf{s}^\star + \frac{\mu^2}{\lambda^2}\mathbf{1}^T\mathbf{1} \\ &= N - 2\frac{(n_1-n_2)^2}{N} + \frac{(n_1-n_2)^2}{N} = \frac{(n_1+n_2)^2 - (n_1-n_2)^2}{N} = 4\frac{n_1 n_2}{N}.\end{aligned}$$

Thus, the cost function is

$$C^\star = \lambda\frac{n_1 n_2}{N}.$$

In order for $C^\star$ to be minimal, we will want λ to be the smallest possible eigenvalue (and $\mathbf{v}$ to be its associated eigenvector). However, note that $\lambda_1 = 0$ with $\mathbf{v}_1 = \mathbf{1}$ *cannot* be the solution we are looking for. To see this, use the definition (11.6) and multiply on the left by the vector $\mathbf{1}^T$:

$$\mathbf{1}^T\mathbf{v} = \mathbf{1}^T\mathbf{s}^\star + \frac{\mu}{\lambda}\mathbf{1}^T\mathbf{1} = (n_1 - n_2) - \frac{n_1 - n_2}{N}N = 0.$$

This shows that the solution must be orthogonal to $\mathbf{v}_1$.

Key message: The lowest value of the graph partitioning cost function $C^\star$ is achieved by choosing the second (i.e., the first non-trivial) eigenvector $\mathbf{v}_2$ with eigenvalue λ_2, which gives:

$$C^\star_{\min} = \lambda_2\frac{n_1 n_2}{N}$$

This is the classic result by Miroslav Fiedler, who obtained it in 1973[a].

[a] Algebraic connectivity of graphs, Miroslav Fiedler, Czechoslovak Mathematical Journal 23, 298-305 (1973)

The second eigenvector of the Laplacian, $\mathbf{v}_2$, is thus called the *Fiedler* eigenvector. Note that the quality of the bipartition is directly related to the eigenvalue λ_2, which is also called the *algebraic connectivity* of the graph. If $\lambda_2 \ll 1$ (is very small) then the cost of splitting the graph into two subgraphs is low, i.e., there exists a good bipartition of the graph given by $\mathbf{v}_2$. If, on the contrary, λ_2 is large (i.e., it is well separated from the zero eigenvalue $\lambda_1 = 0$) then the bipartition will not be 'good' since the cut cost function $C^\star$ cannot be made small.

Now you can see why λ_2 is called the algebraic connectivity of the graph. If the algebraic connectivity λ_2 is large, there is no good/easy bipartition of the graph because the cost of the bipartition is large; hence the graph is well connected. A small value of the algebraic connectivity means that a good bipartition with low cost exists; hence the graph is not well connected and it is easy to split into two subgraphs.

The Fiedler eigenvector is then used to produce the optimised bipartition according to its components. To see this, we need to 'undo' the relaxation from $\mathbb{R}^N$ back to a solution in $\{-1,+1\}^N$. Remember that the vector solution to the relaxation (in the reals) is:

$$\mathbf{s}^\star = \mathbf{v}_2 + \frac{n_1 - n_2}{N}\mathbf{1}, \quad \mathbf{s}^\star \in \mathbb{R}^N.$$

We would like to find a vector $\mathbf{s} \in \{-1,+1\}^N$ as close as possible to $\mathbf{s}^\star$ by maximising the projection of $\mathbf{s}$ on $\mathbf{s}^\star$:

$$\max_{\mathbf{s}\in\{-1,+1\}^N} \mathbf{s}^T \cdot \left(\mathbf{v}_2 + \frac{n_1 - n_2}{N}\mathbf{1}\right) = \max_{\mathbf{s}\in\{-1,+1\}^N} \mathbf{s}^T\mathbf{v}_2 + \frac{(n_1 - n_2)^2}{N}, \tag{11.7}$$

where we have used the fact that $\mathbf{1}^T\mathbf{s} = n_1 - n_2$.

There are a couple of procedures to maximise the alignment with the solution of the relaxation given by $\mathbf{v}_2$:

- If n_1 and n_2 are given, compute $\mathbf{v}_2 = (v_2^{(1)}, \ldots, v_2^{(N)})$ and order the elements $v_2^{(i)}$ from most positive to most negative. Assign the top n_1 elements to cluster 1 and the rest $n_2 = N - n_1$ to cluster 2.
- If we do not have a prescribed size for the split subgraphs, compute $\mathbf{v}_2 = (v_2^{(1)}, \ldots, v_2^{(N)})$ and assign all elements with $v_2^{(i)} < 0$ to cluster 1 and all elements $v_2^{(i)} > 0$ to cluster 2.

Both of this procedures aim to maximise the alignment (11.7) of the discrete solution.

2.2.1. ***Beyond bipartitions:*** The bipartition procedure described above can be generalised to arbitrary $k > 2$ in a number of ways. Here we briefly mention a couple with a very different flavour.

Sequential bipartitions: One approach is divisive ('top-down') by applying Fiedler bipartitions recursively. One would start from the full graph and iterate this process to generate a series of bipartitions, where each subgraph obtained as the result of a bipartition is again split until a particular end condition is met.

Using more Laplacian eigenvectors: Another approach is a generalisation that uses at once the full eigendecomposition of the Laplacian L. The standard algorithm is as follows. Given a diagonalisation of the Laplacian, i.e. $L = V\Lambda V^T$, we have the following steps:

(1) Take the $N \times m$ matrix $V_m = \begin{bmatrix}\mathbf{v}_1 & \ldots & \mathbf{v}_m\end{bmatrix}$ containing the ordered (by ascending eigenvalues) m eigenvectors.

(2) Each row (denoted $\mathbf{w}_i$) of this matrix, will give a description of each of the N nodes of the graph in terms of this m dimensional eigenspace. We use these as descriptors for each node.

(3) Carry out K-means on the vectors $\{\mathbf{w}_i\}_{i=1}^N$ to find K groups of nodes.

Quick Aside. Although our derivation has been based on the *combinatorial* Laplacian L, similar algorithms exist based on the *normalised* Laplacians $L_{\text{rw}}, L_{\text{sym}}$. These are defined below.

$$L_{\text{rw}} := I - A\mathcal{D}^{-1} = L\,\mathcal{D}^{-1}$$
$$L_{\text{sym}} := I - \mathcal{D}^{-\frac{1}{2}} A \mathcal{D}^{-\frac{1}{2}} = \mathcal{D}^{-\frac{1}{2}} L \mathcal{D}^{-\frac{1}{2}}$$

In such cases, the vectors $\mathbf{w}_i$ can either be obtained from L_{rw} — as in the **normalised cuts** algorithm of Shi and Malik; see *Normalized cuts and image segmentation,* IEEE Trans. Pattern Anal. Mach. Intell. 22(8), 888–905 (2000) — or from L_{sym} with an additional row-normalisation of the vectors $\mathbf{w}_i$ — as in the famous **spectral clustering** algorithm of Ng, Jordan and Weiss; see *On Spectral Clustering: Analysis and an algorithm*, Advances in Neural Information Processing Systems 14, 849–856 (2001). The normalised cuts algorithm using L_{rw} yields more balanced partitions and is therefore preferable. The spectral clustering algorithm using L_{sym}, on the other hand, does not extensively mention graphs, yet you can now see that some of the foundational ideas for data clustering are in fact intimately linked to ideas from graph partitioning. Between the two, we note that using L_{rw} is generally preferable over L_{sym}, since the eigenvectors of L_{rw} directly encode the cluster indicators whereas the eigenvectors of L_{sym} are additionally scaled by the square-root of node degrees which can result in undesirable behaviours — especially when the entries of the eigenvectors are small; for further details, see von Luxburg, *A tutorial on spectral clustering,* Statistics and Computing 17, 395–416 (2007) in 'additional reading'.

2.3. Modularity. As opposed to clustering methods based on spectral decomposition seen above, the *Modularity* method is somewhat based on combinatorial notions from graph theory. The idea is to induce a labelling of the nodes of a graph that lead to a maximally block-diagonal adjacency matrix. Interestingly, some of these notions can be related to dynamical aspects such as diffusions, conductance, and random walks on graphs.

In essence, Modularity is a cost function that has a similar starting point to those above, with some subtle differences. Of course, the overall aim is to find groups in the graph that are similar within and different to the rest. In other words, we want to find a partition of the graph that will have a maximally block-diagonal structure in the adjacency matrix, and has more 'blocks' than expected at random. The idea behind Modularity is to count the edges within blocks as well as between blocks. So this is very similar to our setup for clustering introduced in Chapter 9. To aid our explanations, let us consider the very

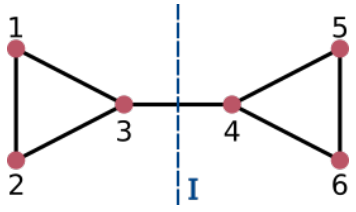


Figure 11.10. A simple example to illustrate Modularity. The blue dashed line is a good clustering split.

simple graph given in Figure 11.10 with adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 1 & & & 0 \\ 1 & 0 & 1 & & & \\ 1 & 1 & 0 & 1 & & \\ & & 1 & 0 & 1 & 1 \\ & 0 & & 1 & 0 & 1 \\ & & & 1 & 1 & 0 \end{bmatrix}$$

Modularity is then a cost function for a given graph partition. In the simple graph in Figure 11.10, splitting the nodes into two clusters ($k = 2$) along the dashed blue line (denoted by the split number I) seems to be a good idea. Such a clustering can be succinctly written in matrix form through the *membership matrix* (see Chapter 9):

$$H_I = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}_{N \times k}$$

Here each row corresponds to a node in the graph (in this case nodes 1-6 as given in Figure 11.10) and each column represents one of the clusters (left, right).

Let us consider the following expression:

$$\frac{1}{2}(H_I^T A H_I)_{k \times k} = \begin{bmatrix} 3 & \frac{1}{2} \\ \frac{1}{2} & 3 \end{bmatrix}$$

This matrix captures the number of edges within each cluster (on the diagonal, 3 in each) as well as how many edges are between the clusters (on the off-diagonal, $\frac{1}{2}$ each). This information is in line with what we stated at the beginning: Modularity counts the number of edges within as well as between clusters, and attempts to find clusterings that will have most of the edges within the clusters and few edges between the clusters.

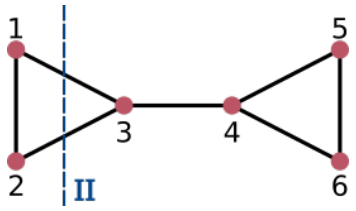


Figure 11.11. A simple example to illustrate Modularity. The blue dashed line is not a good clustering split.

In contrast, if we now consider a 'worse' split (denoted with the number II), shown by the blue dashed line in Figure 11.11, then we have a different membership matrix:

$$H_{II} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}_{N \times k}$$

and we compute again:

$$\frac{1}{2}(H_{II}^T A H_{II})_{k \times k} = \begin{bmatrix} 1 & 1 \\ 1 & 4 \end{bmatrix}$$

Again, this tells us that the first (left-hand side) cluster has only 1 edge, and the other cluster has 4. Since the split is through two edges this time, the inter-cluster counts amount to $2 \times \frac{1}{2} = 1$ for each cluster.

It is becoming apparent that a good partition will be achieved by maximising the total number of edges inside the defined clusters, i.e.,

$$\max_H \operatorname{Tr}\left[H^T A H\right].$$

Since the total number of edges E is fixed, maximising the number of edges inside clusters (diagonal elements of $H^T AH$) will reduce the number of edges between clusters (off diagonal elements of $H^T AH$).

The second part that makes Modularity so popular and successful is that it introduces a 'null model', against which the given partition is measured. This is usually called the *configuration model*, and is an essential ingredient to define an intrinsic measure of goodness of a given partition. The intuition behind the null model is to discount edges that are expected to be present at random given the properties of the graph. This can be formalised in the following way. We reduce the graph down to mere knowledge of the degrees of all the nodes, i.e. we only have the degree vector $\mathbf{d} = A\mathbf{1} = \begin{pmatrix} d_1 & \dots & d_N \end{pmatrix}^T$. Then we ask the question: If the nodes in the graph (with degree vector $\mathbf{d}$) were rewired at random, what would be the likelihood of the nodes being connected? The probability of two nodes i, j being connected is then given by:

$$R_{ij} = \frac{d_i d_j}{2E},$$

where E is the number of edges, i.e. $E = \frac{1}{2}\sum_{i=1}^{N} d_i$. In vector form, we can write this as:

$$R = \frac{1}{2E}\mathbf{d}\mathbf{d}^T$$

Now, the final step is to simply put the two concepts together to create the cost function which we call *modularity* :

$$Q = \frac{1}{2E}\operatorname{Tr}\left[H^T \underbrace{\left[A - \frac{1}{2E}\mathbf{d}\mathbf{d}^T\right]}_{Z} H\right]$$

The matrix Z is typically known as the Modularity matrix.

Key message: The aim of the Modularity method is simply to maximise the modularity Q over all membership matrices H:

$$\max_H Q. \tag{11.8}$$

In essence, we are looking for graph partitions with more in-cluster edges than you would expect at random. The graph clusters so found are usually called *graph communities* and modularity maximisation is usually referred as a *community detection* algorithm.

An important, distinctive feature is that the modularity Q is a compound cost function by definition since it has two balancing terms: one measuring the block-clustering, and another one that subtracts the expected clustering score of a null random model. Hence maximising modularity allows one to find an optimised number of clusters directly, without prescribing the number of clusters *a priori*, as we do in K-means.

As can be expected at this point, we recognise (11.8) as a combinatorial optimisation, since we need to optimise over the space of partitions. This is a 'hard' problem (in fact, it has been shown to be NP-hard) which needs to be solved through heuristic optimisations. There are many approaches to optimise modularity, from simulated annealing to specialised relaxations. However we mention two different approaches to the optimisation of Q in the space of partitions:

- Similar to spectral clustering, it can be shown that we can use the leading eigenvector of Z to maximise Q. Then we can effect bipartitions in a recurrent manner until Q does not increase. The stopping criterion is therefore $\Delta Q \leq 0$ in the iteration.
- In practice, modularity is optimised almost always through a greedy agglomerative algorithm that merges nodes of the graph into 'super-nodes' and proceeds sequentially until no improvement of Q is achieved. This method, which performs better than any other method currently known, is called the *Louvain algorithm*[2] (it was developed at UC Louvain in Belgium) and has become the industry standard and used very broadly in many areas of data science.

3. Dimensionality reduction using graphs

In this section we show how graphs enable us to perform dimensionality reduction in nonlinear settings. Specifically, we will explore how graph-based approximations can be used to learn nonlinear transformations of data that somehow capture the implicit geometry of data in the high-dimensional space. Here, we assume that data lie on some low-dimensional *manifold* of dimensionality m, embedded in a high-dimensional space of dimensionality $p > m$ – this is often called the *manifold assumption*. The notion of a manifold is central to the field of geometry in mathematics and is defined in terms of a collection of points such that locally the neighbourhood of each point resembles Euclidean space of dimensionality m, albeit the dimensionality of the "ambient space" in which the manifold lives can be $p > m$.

> *Quick Aside.* The dimensionality m of a m-manifold can be seen as the "number of linearly independent axes" if one were to zoom into the local neighbourhood of a point. Consequently, a 1-manifold is like a curve (say, a circle), a 2-manifold is like a surface (say, a disc), a 3-manifold is like a volume (say, a ball), and so on. To illustrate with some parametric examples: (a) the function $x^2 + y^2 = 1$ in $\mathbb{R}^2$ is a curve (the unit circle) in the XY-plane — a 1-manifold embedded in a 2-dimensional ambient Euclidean space — whereas in $\mathbb{R}^3$ it is a surface (a cylinder) in the XYZ-volume — a 2-manifold embedded in a 3-dimensional ambient Euclidean space, and (b) the parametrised curve $x(t) = \cos(t)$, $y(t) = \sin(t)$, and $z(t) = t$ is a curve (right-handed helix) in the XYZ-volume — a 1-manifold embedded in a 3-dimensional ambient Euclidean space.

The key idea is the following: **since graphs are also characterised by local neighbourhoods — a node connected to its neighbours via edges — constructing an "appropriate" graph can approximate the structure of the manifold on which data lie**. Consequently, we can use the techniques of graph construction from Section 2 to obtain a discrete approximation of the underlying manifold of the data.

[2] Blondel, V. D., Guillaume, J.-L., Lambiotte, R. & Lefebvre, E. Fast unfolding of communities in large networks. J. Stat. Mech. Theory Exp. 2008, P10008 (2008).

Figure 11.12 shows one such construction of a geometric graph using k-nearest neighbours with $k = 7$ on a Swiss roll dataset with $N = 1000$ data points. Once a graph has been (appropriately) constructed, we can now focus on the actual problem at hand — of mapping the data to its corresponding manifold which will be of a lower dimension. Similar to how PCA is motivated by the optimisation problem of minimising reconstruction error, **we can define optimisation problems that preserve desirable graph properties in the mapped space**, which is done using broadly two kinds of approaches.

3.1. Embeddings that preserve local structure. Since manifolds (and graphs) encode the local structure around each point, it is natural to desire that a node's neighbours are mapped to locations that lie close to the mapped location of the node itself. As before, let $\{\hat{\mathbf{x}}^{(i)}\}_{i=1}^N$ refer to the mapped location of all data points, and A_{ij} encode the (possibly weighted) adjacency of points $\mathbf{x}^{(i)}, \mathbf{x}^{(j)}$, i.e. $A_{ij} > 0$ if they are neighbours in the graph constructed from $\{\mathbf{x}^{(i)}\}_{i=1}^N$. Then we would like to find the mapped locations that minimise the cost function:

$$C := \frac{1}{2} \sum_{i,j=1}^{N} \|\hat{\mathbf{x}}^{(i)} - \hat{\mathbf{x}}^{(j)}\|_2^2 A_{ij},$$

subject to some constraint that keeps the locations scale-invariant. This cost function is very similar to the one we previously obtained for graph bipartitioning in Section 2. Indeed, it is a multivariate generalisation where the cost function can be written in terms of the combinatorial Laplacian L:

$$C = \mathrm{tr}(\widehat{X}^T L \widehat{X}),$$

where $\widehat{X}$ is the matrix obtained by stacking each data point's mapped location row-wise. It is no surprise then that the solution $\widehat{X}$ that minimises the cost function is given by the first m eigenvectors of a graph Laplacian, where the eigenvectors are arranged by non-decreasing eigenvalues — after ignoring the trivial eigenvector(s) corresponding to zero eigenvalue(s). If the constraint that renders the locations scale-invariant is given by (a) $\widehat{X}^T \widehat{X} = \mathbf{I}$ (where $\mathbf{I}$ is the identity matrix) then it corresponds to using eigenvectors of the combinatorial Laplacian L, whereas if it is given by (b) $\widehat{X}^T \mathcal{D} \widehat{X} = \mathbf{I}$ (where $\mathcal{D}$ is the degree matrix) then it corresponds to using eigenvectors of the normalised random-walk Laplacian L_{rw}. In other words, **the eigenvectors based on the Laplacian that are used to perform graph-based clustering can very well be used as a graph-based low-dimensional representation of the data**, and such a representation is referred to **spectral embedding** or **Laplacian eigenmaps**[3].

Extra: link between geodesic and geometry.
The notion of geodesic — or the shortest path — between points is intrinsically linked to the geometry of the underlying space (or manifold); it generalises the standard notion of a straight line in Euclidean space to non-Euclidean spaces. For instance, the "straight line" or geodesic between any two locations on the surface of the Earth — which is a sphere and thus exhibits spherical geometry — is given by the great circle that joins the two locations — a fact that is used to determine optimal flight paths. The behaviour of geodesics characterise the "shape" of the underlying space, and therefore the geodesic distances carry a lot of information about its intrinsic structure. In Euclidean space, for any straight line (geodesic) l and a point p not on l there is exactly one straight line l' through p that does not intersect l

[3]Belkin, M., & Niyogi, P. (2003). Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation, 15(6)*, 1373-1396 (see 'additional reading').

— this reflects Euclid's fifth postulate also termed as the parallel postulate. Hence, Euclidean space is said to be *flat* or with *zero curvature*. In spherical (or more generally elliptical) space, on the other hand, all "straight lines" intersect l, i.e. geodesics tend to converge and such a space is said to have a *positive* curvature. Finally, in hyperbolic space there are infinitely many such "straight lines" l' that do not intersect l, i.e. geodesics tend to diverge and such a space is said to have a *negative* curvature. For discrete objects like graphs, the "straight line" or geodesic between two nodes corresponds to a shortest path connecting them. The behaviour of geodesics on graphs can characterise their "curvature" and shape as well — for instance, a lattice-type graph is flat whereas a tree-like graph has negative curvature.

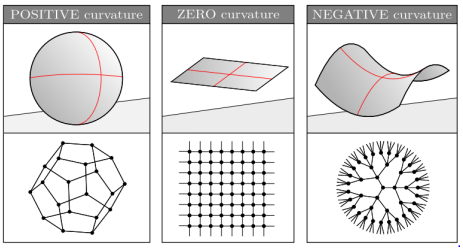


Figure from Devriendt, K., & Lambiotte, R. (2022). Discrete curvature on graphs from the effective resistance. *Journal of Physics: Complexity, 3(2)*, 025008.

3.2. Embeddings that preserve global structure. A potential problem with a method that preserve only local structure is that while it ensures that nodes that are adjacent (directly connected) on the graph are mapped close to one another in the low-dimensional space, it does not ensure that nodes that are far apart on the graph are mapped far away from one another. Consequently, if an embedding can preserve the (geodesic) distances or shortest path lengths between *all* pairs of nodes in the graph in the mapped space, then it preserves the global structure of the manifold.

In Chapter 10, although PCA was motivated as a dimensionality reduction method that minimises the reconstruction error, we saw that it ends up capturing the (maximum possible) variance in the data — with the sum of eigenvalues of the chosen top-m eigenvectors of the covariance matrix encoding the total variance explained (see the discussion after Figure 10.1). In other words, it tries to approximately preserve the Euclidean distance between every pair of points — thus preserving some global structure. Generalising beyond Euclidean distance, multidimensional scaling (MDS) is a class of methods that attempts to preserve pairwise distances, i.e., to minimise the cost function:

$$C := \sum_{i,j} \left[\|\hat{\mathbf{x}}^{(i)} - \hat{\mathbf{x}}^{(j)}\|_2^2 - d(i,j)^2 \right]^2,$$

where $d(i,j)$ refers to the distance between points i, j. In other words, when $d(i,j)$ is the Euclidean distance then we recover PCA. However, as we wish to preserve geodesic distances on the corresponding graph, we define $d(i,j)$ to be the geodesic distance or shortest path lengths in the graph. This algorithm is referred to as **Isomap**[4]. One can summarise the full algorithm as follows:

(1) Construct a (weighted) graph G (using techniques in Section 2 like k-nearest neighbours).

(2) Compute all pairwise geodesic distances in G (using Dijkstra's algorithm for graphs with positive-valued edge weights).

[4]Tenenbaum, J. B., Silva, V. D., & Langford, J. C. (2000). A Global Geometric Gramework for Nonlinear Dimensionality Reduction. *Science, 290(5500)*, 2319-2323 (see 'additional reading').

(3) Apply multidimensional scaling (MDS) to the matrix $D_{N\times N}$ of geodesic distances to obtain the mapping to a lower-dimensional space (equivalent to performing kernel PCA where the kernel matrix K is given by the (centered) negative element-wise squared distance matrix: $K_{ij} := -D_{ij}^2$)

Figure 11.12 demonstrates the Isomap algorithm applied to the Swiss roll dataset. Evidently, if the graph construction method trivially assumes a full-connected network ($\varepsilon \to \infty$ or $k \to \infty$) and weights the edges with the Euclidean distance in the original feature space, then Isomap is equivalent to PCA. On the other hand, assuming a fully-connected network and using a kernel matrix — such as one obtained from the radial basis function kernel — to provide edge weights is equivalent to kernel PCA. This shows that Isomap is a generalisation of kernel PCA, which in turn is a generalisation of PCA.

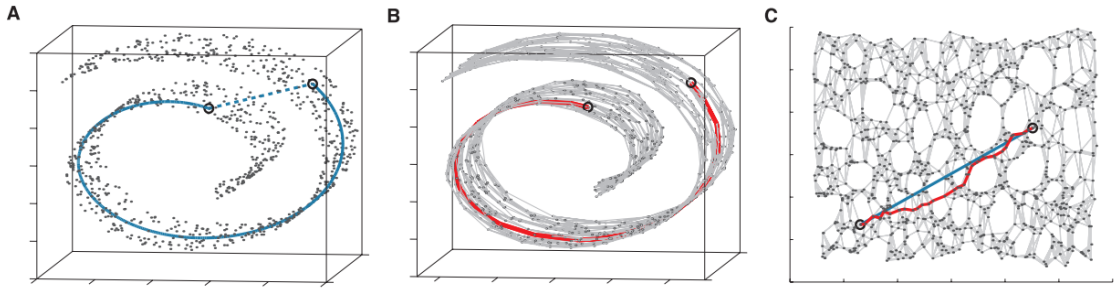


Figure 11.12. Using the Swiss roll dataset ($N = 1000$) to demonstrate the use of Isomap in performing nonlinear dimensionality reduction. Figure from Tenenbaum, Silva, & Langford (2000). (A) The Swiss roll dataset is not Euclidean as Euclidean distances do not capture the intrinsic non-linearity of the data. (B) Constructing a k-NN graph (with $k = 7$ neighbours) and using geodesic distances captures the non-linear structure well. (C) Performing MDS preserves pairwise distances on the graph (underlying 2D manifold) and "unrolls" the Swiss roll into two dimensions.

Quick Aside : As discussed before, the datasets can be transformed into a network of features as nodes and the edges represent a measure of linear or non-linear interdependency among the features, estimated across the N samples. Such a transformation provides an adjacency matrix of the dimension $p \times p$. Deploying the community detection algorithms discussed above on this graph will cluster groups of features that are strongly interdependent into say $c < p$ clusters. Now, if a coarse graining that combines the cluster of features into one (or more) components is applied on the original dataset, it will result in a dataset $\tilde{X}_{N\times c}$ with reduced dimensionality.

This transformation (also known as graph renormalisation) is used to reduce dimensionality in datasets with large number of features and each feature represent a different part of the system studied. Various coarse graining and renormalization algorithms have been proposed recently, see *Villegas, P., Gili, T., Caldarelli, G. et al. Laplacian renormalization group for heterogeneous networks. Nat. Phys. 19, 445–450 (2023)* and *Zhang, Z., Ghavasieh, A., Zhang, J. et al. Coarse-graining network flow through statistical physics and machine learning. Nat. Commun. 16, 1605 (2025).* This method of dimensionality reduction is a popular choice for time series datasets obtained from different regions of the brain, price of different stocks in the financial markets, population of different species in an ecosystem, spike trains obtained from a group of neurons etc. In all of these datasets, the data corresponding to each part of the system is the same in most cases (either electrical activity, price, population, spiking).

4. Graph centralities

In this very final section, we will introduce another concept from graph theory that can be very useful in understanding data from an alternative viewpoint.

The concept of *graph centrality* has to do with how to find the most important nodes or edges in a graph effectively establishing an importance ranking for nodes/edges. This notion is of course not uniquely defined, but rather a concept that can be viewed from many perspectives, and many notions of centrality are commonly used.

Recall that, according to our setup, the nodes/edges correspond to samples/relationships between samples in our data set. Hence when looking for the most important nodes, one is in essence making an assessment of the importance of samples, with the potential to rank samples or to find representative samples in the data set, and in turn, also outliers can be found thus. But of course, many different aims can be achieved through centrality measures.

The area of graph centralities is extremely well-developed in the literature and still the focus of open research, so in the following we give but a snippet, non-comprehensive list of some of the most commonly used centrality measures:

- **Degree centrality.** This is perhaps the simplest centrality. The importance of a node is simply defined to be its degree:

  $$\mathbf{c}_d = \frac{\mathbf{d}}{2E} = \frac{A\mathbf{1}}{2E}$$

  When applying this to a graph of citations for example, the importance of a single document is measured by the number of citations it received. In a social network, it would measure the number of contacts.

- **Betweenness centrality.** In this case we are not interested in searching for nodes that have a lot of connections, but rather nodes that have many paths going through them. Intuitively, the higher the betweenness, the more likely it is that removing the node would disconnect the graph or at least reduce the connectivity in the graph. In order to compute this measure, the set of shortest paths (i.e. the shortest path between every pair of nodes) in the whole graph needs to be found first. Then, the (normalised) number of minimal paths traversing each node is the betweenness centrality.

- **Closeness centrality.** This measure is based on distances in the graph: the closer a node is to all other nodes, the more central it is. For a given node, all shortest paths to all other nodes are computed and the closeness centrality is then defined to be the reciprocal of the average length of those paths:

  $$\mathbf{c}_c = \frac{1}{\frac{1}{N}\sum_j d_{ij}}$$

  Here we have defined the shortest path to be the measure of distance from one node to another.

- **Eigenvector centrality.** As opposed to the previous centralities, which are inspired by combinatorial graph-theoretical properties, we now look towards spectral graph properties once more. The first of such measures stems from the eigendecomposition of the adjacency matrix. Intuitively, this measure assigns higher centrality to those nodes that are themselves connected to other highly central nodes. What sounds like a circular argument, is simply based on the following statement: "The centrality of node i is a function of the centralities of

its neighbours", which can be written as

$$\mathbf{c}_e(i) = \alpha \sum_j A_{ij} \mathbf{c}_e(j),$$

Where α is a proportionality constant. This can be immediately identified as an eigenvalue/eigenvector problem:

$$A\mathbf{c}_e = \lambda \mathbf{c}_e$$

Hence the eigenvector centrality of each node is given by the component of the leading eigenvector of the adjacency matrix.

- **PageRank.** Arguably the most famous of them all, PageRank was introduced in the 90's at Stanford by Larry Page and Sergei Brin[5], who subsequently went on to found the company behind a search engine that is nowadays responsible for saving the lives of thousands of students every day. PageRank was of course originally conceived for ranking webpages on the internet, which is nothing other than one massive *directed* graph with websites for nodes and *directed* links for edges. Intuitively, PageRank will assign higher centrality to nodes where many searches will reside with higher probability, where the centrality of the nodes of those incoming edges also plays a role, i.e., being pointed at from an important node increases your score.

  PageRank is an iterative algorithm that is defined by a simple, yet powerful process (that is worth billions):

$$\mathbf{c}_{\mathrm{PR}\ t+1} = \alpha \left(A\mathcal{D}^{-1}\right) \mathbf{c}_{\mathrm{PR}\ t} + (1-\alpha)\frac{1}{N}\mathbf{1} \tag{11.9}$$

  For a given graph with adjacency matrix A, the PageRank centrality of each node is then given by the elements of the stationary eigenvector of (11.9), which is obtained by solving for $\mathbf{c}_{\mathrm{PR}}$ in:

$$\mathbf{c}_{\mathrm{PR}} = \alpha \left(A\mathcal{D}^{-1}\right) \mathbf{c}_{\mathrm{PR}} + (1-\alpha)\frac{1}{N}\mathbf{1}, \quad \alpha \in [0,1]$$

  The process (11.9) corresponds to a weighted *random walk* on the graph where the random walker follows the transition matrix of the graph $M = A\mathcal{D}^{-1}$ with probability α and transitions to any node in the graph with probability $(1-\alpha)$. Note that in this case the graph is directed and $A \neq A^T$. The parameter α, called the teleportation parameter, regulates how much the random walk follows the graph or jumps unrestricted by it. This parameter is needed to guarantee ergodicity of the random walk and is customarily set to $\alpha = 0.85$. This model of the random walk was inspired by web surfing.

  What that big company is doing now, only they know, but it all started with this formula and this simple idea based on random walks on graphs.

[5] Page, L., Brin, S., Motwani, R. and Winograd, T. (1999) "The PageRank Citation Ranking: Bringing Order to the Web". Technical Report, Stanford InfoLab (see 'additional reading').

Glossary

- **Accuracy**: In a classification problem, the accuracy gives the number of well predicted cases divided by the overall number of cases. For this and other indicators of classification performance, see chapter 3, section 2.
- **Bias**: The bias of a statistical estimator is a measure of the average error of the estimated parameters compared to the true parameters, see chapter 1.
- **Classification**: The task of predicting the class (or label) of a categorical variable (see chapter 0).
- **Clustering**: The task of sorting data points into different based on intra-group similarities, see chapter 9.
- **Confusion matrix**: Matrix summarizing false positives, false negatives, true positives and true negatives in the predictions of a classification algorithm. It serves to estimate the indicators of classification performance (see chapter 3, section 2).
- **Cross-validation**: Statistical procedure used to search for optimal hyperparameters by assessing the method's performance on a validation dataset, set aside from the dataset used for training the method (training dataset). A typical cross-validation procedure is the T-fold cross-validation (see chapter 2, section 2).
- **Dimensionality Reduction**: The task of finding a representation of data points characterized by many features that is specified by a small number of coordinates. The most basic and yet representative algorithm for dimensionality reduction is Principal Component Analysis, see chapter 10.
- **Dropout**: regularization method for deep neural networks consisting of masking out a certain fraction of the neurons and their outgoing/incoming connections at each training iteration. It helps prevent overfitting and improve interpretability by avoiding co-adaptation between neurons (see chapter 7, section 5).
- **False Negatives/Positives**: Positives/Negatives (e.g. data that belong/don't belong to a certain class) that are predicted by a classification method as Negatives/Positives (see chapter 3, section 2).
- **Generalisation power (or generalisability)**: Model's ability to perform well on unseen data (see chapters 0 and 1).
- **Graph**: Representation of the relationships in a dataset given by edges connecting nodes (the data points), see chapter 11.

- **Hidden (latent) variable**: Variables of the model that do not represent real features of the data but are used internally by the model to reach high expressive power and prediction performance. An example are the neurons of the intermediate layers of a neural network (see chapter 7).
- **Hyperparameter**: Parameters of the model that specify its structure, for example regularization terms or the depth or width of the layers in a neural network. They are not learnt during training but they are set via cross-validation (see e.g. chapters 1 and 7).
- **Likelihood**: Probability of the data under the model, see e.g. chapter 1, section 3. The criterion of maximum likelihood relies on maximizing the likelihood, averaged over the training set, to train a model.
- **Loss function**: Function that is minimized to train a model (see beginning of chapter 1).
- **Mean Square Error (MSE)**: Difference between the true value and the model's predicted value squared and averaged over data points. It's an example of loss function, hence one typically minimizes the MSE over the training set to train a regression model, see chapter 1.
- **Overfitting**: Training outcome whereby the model reproduces very well the features of the training set (high accuracy on the training set) but lacks generalisation power on an unseen test set (low accuracy on the test set). It occurs when the training has led the model to fit not only the features of the training set informative about the structure of the data, but also the noise coming e.g. from limited sampling. It is usually a consequence of overparametrisation (that is, having a model with too many or unregularised parameters), see chapter 1.
- **Penalty term**: Term added in the loss function or in the likelihood to 'penalize' certain values of the parameters to learn (for example, penalize very large values). The penalty terms implement examples of regularization, see chapter 1.
- **Receiver Operating Characteristics (ROC)**: Curve describing the false positive rate (x-axis) *vs* the true positive rate (y-axis) of a classification algorithm, varying the threshold for classification into positives/negatives. The Area Under the ROC (AUROC) is a measure of classification performance: AUROC=1 for perfect classification, AUROC=0.5 for the random case (chapter 3, section 2).
- **Regression**: Task consisting in predicting the value of a quantitative variable, called outcome variable, given an input, called predictor variable (see chapter 0).
- **Regularisation**: It generally refers to terms that help fix the issue of overfitting by imposing a penalty on the parameters in the loss function, keeping in this way under control the parameter values learnt. Examples of regularisation are the Ridge and Lasso regularization, which rely on penalty terms that, respectively, penalise large parameter values or pushes small values towards zero, see chapter 1.
- **Supervised learning**: Type of statistical learning aimed at modelling the mapping between a certain input and an output. Examples are the assignment of data points to their class, or the prediction of an outcome for a given value of the input (see chapter 0).
- **Test set**: Part of the dataset that has not been seen neither during training nor cross-validation. The model's performance on test set measures the model's generalisation power (see chapter 2, section 2).

- **Training set**: Part of the dataset that is used to *train* (i.e. learn) the model's parameters (see chapter 2, section 2).
- **True Positives/Negatives**: Positives/Negatives (e.g. data that belong/don't belong to a certain class) that are correctly predicted by a classification method (see chapter 3, section 2).
- **Unsupervised learning**: Type of statistical learning aimed at extracting and inspecting the intrinsic structure and properties of the data. Typical unsupervised learning tasks are dimensionality reduction and clustering (see chapter 0).
- **Validation set**: Part of the dataset that is used in cross-validation to evaluate the model's performance to select the hyperparameters (see chapter 2, section 2).
- **Variance**: The variance of a statistical estimator is a measure of the variation of the estimated parameters around the true parameters, see chapter 1.