Imperial College London

MATH60026/MATH70026

Methods for Data Science

Lecture notes

Spring Term 2023

Written by Prof Mauricio Barahona
with integrations by Dr Barbara Bravi, Dr Philipp Thomas, Dr Sahil Loomba and Florian Song

Chapter 8 by Dr Kevin Webster

Contents

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.

[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.

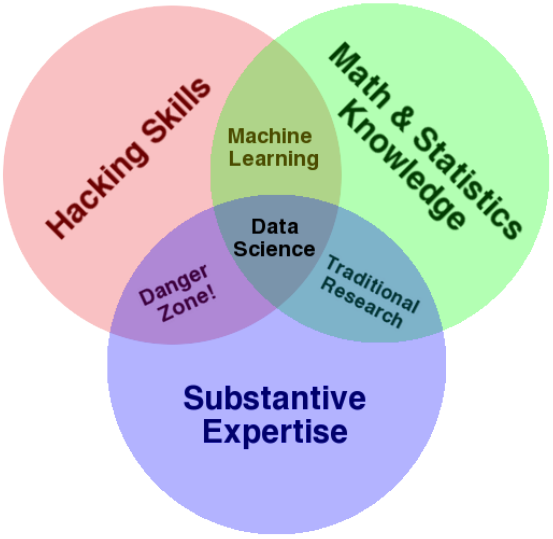


Figure 0.2. Venn diagram: Where does data science fit?

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

1. 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:

- Spam filters

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

- 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.

2. 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.

3. 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 mathematical concepts 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

Chapter 1

Preliminaries. 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?)
- Check your assumptions
- 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.)
- Select your methods and tools
- Find relationships between predictors

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

1. Some preliminary definitions

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

1.1. 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:

1.1.1. *Categorical data.* A *categorical* (or *discrete*) variable takes on one of a limited number of values (usually fixed), such as classes, or memberships. That means 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.

1.1.2. *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, blood count. 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*.

1.1.3. *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 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*.

1.2. 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*. The game of this course is to establish relationships between these two spaces. You need to be aware that this is a decision that you make. You may come across data with 100 columns, 99 of which are irrelevant and thus may not be used as predictors. Sometimes the variable that was thought to be the outcome is actually not.

1.2.1. *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.

1.2.2. *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*.

1.3. 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: county of residence, gender, company name... Outputs $y^{(i)}$ can be continuous or categorical: some number or feature or group, some knowledge about the data point. Fundamentally, machine learning is about finding ways to link the predictors $x^{(i)}$ to the outcomes $y^{(i)}$ so that we can make new predictions. There are two big, main classes of problems/algorithms:

1.3.1. *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.

Note that 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:

1.3.2. *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. To this end, we start with giving a brief summary of the typical workflow starting with some data.

The first step is to be cautious and do some exploratory data analysis, to get an idea and overview of what the data contains, but also to conduct a clean-up (e.g. eliminate empty fields, other unusable pieces of data). Then we need to decide on the variables to use in the analysis as well as the task.

This initial examination of the problem is crucial and encapsulates much of the modelling aspects of the work (i.e., trying to establish associations between variables and observables and create a model that allows us to predict the observable given new data points). 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 $y^{(i)} \in Y$. 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. The remaining data will constitute the *training set*.

Now, for a given task with training set $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, we want to find the function $f : X \to Y$, whilst defining the *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. Both f and L are intimately coupled, and therefore it is not possible to do the task with just one or the other defined.

More formally, the goal is to create a function $y = f(x)$ relating *predictor* variables x to *outcome* variable y. The data, called a *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.

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*:

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

For example: in regression, the mean sample loss is the mean squared error:

$$L_{MSE}\big(y, f(x)\big) = \frac{1}{N} \sum_{i=1}^{N} \big(y^{(i)} - f(x^{(i)})\big)^2 \tag{2.1}$$

In general, the solution for f will be the result of some form of *optimisation*, minimising the in-sample loss (error) L. That is to say, 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 desirable property is also known as *generalisability*. In Machine Learning, this is a very important property. The current models are complex, with many parameters that are optimised to the training data in relatively obscure ways. Hence there is a need to protect ourselves from overfitting (more so than in simple 'fitting' algorithms). In addition, there are usually several *hyperparameters* in the models. 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. Think, for instance, when training a deep neural network we optimise the weights of the network (parameters) but we maintain fixed, e.g., the number of layers and the number of neurons (hyperparameters), see e.g. chapter 8.

To try and avoid overfitting and to enhance the robustness and generalisability of our models to predict on unseen data, 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. And 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.

We then expect that $f(x_{\text{unknown}})$ will be a good predictor for *totally unseen* y_{unknown} for which we have no ground truth.

2. Linear regression and the Normal Equation

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; it could follow from our exploratory data analysis; or it could be based on a known model with a basis in physics, biology, economics, etc.

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:

$$\left\{x_1^{(i)}, x_2^{(i)}, \ldots, x_p^{(i)}, \quad y^{(i)}\right\}_{i=1}^{N} \tag{2.2}$$

We then write the following linear model for our observations:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p =: f_{\mathrm{LR}}(\mathbf{x}, \boldsymbol{\beta}), \tag{2.3}$$

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 found from the training data. (Note the alternative notation sometimes used in the literature: $\boldsymbol{\beta} \equiv \boldsymbol{\theta}$.)

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

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

Ideally, given the data (2.2), 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:

$$\left\{\begin{array}{l} \hat{y}^{(1)} = \beta_0 + \beta_1 x_1^{(1)} + \cdots + \beta_p x_p^{(1)} \stackrel{?}{=} y^{(1)} \\ \hat{y}^{(2)} = \beta_0 + \beta_1 x_1^{(2)} + \cdots + \beta_p x_p^{(2)} \stackrel{?}{=} y^{(2)} \\ \qquad \vdots \\ \hat{y}^{(N)} = \beta_0 + \beta_1 x_1^{(N)} + \cdots + \beta_p x_p^{(N)} \stackrel{?}{=} y^{(N)} \end{array}\right\} \tag{2.4}$$

But, of course, the system of linear equations (2.4) in the paramters is usually overdetermined ($N \gg (p+1)$, more equations than unknowns β_i), which means that unless we have a very special case where all the data points are colinear, there will *not* be one solution for $\beta_0, \ldots, \beta_p$ that satisfies each equation in the system with the $\stackrel{?}{=}$ in (2.4).

What to do? Linear regression solves an associated problem, the *least squares problem*, which has many nice and interesting properties. Essentially it finds 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 (2.4) 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:

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

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 to find the parameters 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 2.1. The vector $\mathbf{e}_{N\times 1}$ contains all the deviations between the predicted values and the observed outcomes.

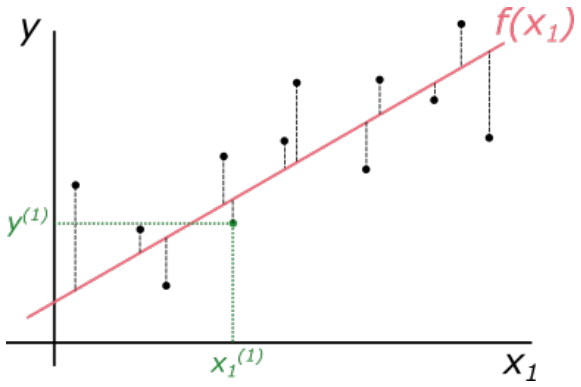


Figure 2.1. Simple linear regression task with one predictor and one outcome variable.

Least squares finds a solution of an associated system that minimises the *mean squared error* (MSE) (2.1). In our matrix-vector notation, the MSE can be written compactly as:

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

The linear regression model that minimises the MSE is obtained by solving this least-squares problem:

> 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:
>
> (2.5) $$L(\boldsymbol{\beta}) = \frac{1}{N}\|\mathbf{e}\|^2 = \frac{1}{N}\big[(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\big].$$

To understand what this least squares solution means we need to perform this optimisation. Optimisations can be (and in most interesting cases are) very complicated. In fact, you will spend this course optimising difficult loss functions. But the least squares 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.

Explicit minimisation: We have to minimise the loss function L (2.5) in the parameter space of the β_i's, i.e., the loss function is a multivariable real function of $\boldsymbol{\beta}$. To minimise,

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 least-squares 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:

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

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:

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

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 spacce, regardless of the optimisation of $\boldsymbol{\beta}$. Locally, the Hessian gives information of the curvature of the function.

Finally, 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:

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

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

Important aside: 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 $\mathbf{X}$.

There exist many computational tools that can compute this pseudoinverse. 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 $\mathbf{A}$:

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

In our case, remember 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 least-squares solution applies a pseudo-inversion:

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

Note that 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 least-squares 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

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

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:

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

This noise will be a particular instance $\varepsilon^{(i)}$ drawn i.i.d. (*independent and identically distributed*) from a Gaussian/Normal 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 Statistics, for this we can write down a *likelihood function* which is given by:

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

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. To this end, we make our life a bit easier by maximising

the logarithm of the likelihood (which is equivalent to the original problem). Therefore consider the *log-likelihood*:

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

where C is constant.

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

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

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 pseudoinverse 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.) The idea is simple: the gradient of a multivariate function gives the direction of maximum variation of the function. Hence following the gradient will maximise the function at every point. (Minimising is easily done by changing signs.) It can be shown easily that following the gradient along an

optimisation trajectory leads to a maximum of the function (you saw this in your Calculus modules).

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. As you might have guessed, the least squares problem is one such case since we have a quadratic function with positive definite Hessian everywhere, as shown above and exploited below.

> 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.

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

$$L(\boldsymbol{\beta}) = L(\boldsymbol{\beta}^\star) + \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)^T \underbrace{H}_{\frac{2}{N}(\mathbf{X}^T\mathbf{X})} (\boldsymbol{\beta} - \boldsymbol{\beta}^\star) \tag{2.6}$$

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

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

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:

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

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 2.2, where we show a sketch of this loss function, with lines indicating the level sets

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

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 $\boldsymbol{\beta}^\star$.

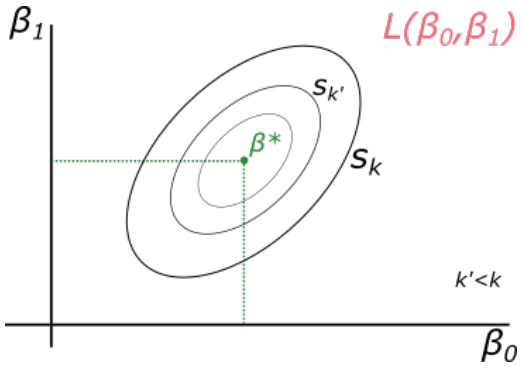


Figure 2.2. Finding $\boldsymbol{\beta}^\star$ is a convex optimisation problem in the case of linear regression. This sketch shows the optimal value of $\boldsymbol{\beta}^\star$ in green, as well as the level sets where the loss function is constant.

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.

Algorithm: Iteratively follow the direction of $-\nabla_{\boldsymbol{\beta}} L$. The iteration $k+1$ will be of the form:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \eta_k \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}_k)$$

where η_k is the step size, which can be adjusted at each step of the algorithm. Note that 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_k) to speed up convergence to the minimum in a shorter number of steps.

Another method for optimisation that uses the Hessian is using Newton's method.

$$L \simeq L(\boldsymbol{\beta}_k) + \nabla L|_{\boldsymbol{\beta}_k}(\boldsymbol{\beta} - \boldsymbol{\beta}_k) + \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\beta}_k)^T H(\boldsymbol{\beta}_k)(\boldsymbol{\beta} - \boldsymbol{\beta}_k)$$

Then, the gradient of L can be approximated accordingly:

$$\nabla L \simeq \nabla L|_{\boldsymbol{\beta}_k} + H(\boldsymbol{\beta}_k)(\boldsymbol{\beta} - \boldsymbol{\beta}_k)$$

Then the iterative process will be as such:

$$\nabla L(\boldsymbol{\beta}_k) + H(\boldsymbol{\beta}_k)(\boldsymbol{\beta}_{k+1} - \boldsymbol{\beta}_k) = 0$$

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - H(\boldsymbol{\beta}_k)^{-1} \nabla L(\boldsymbol{\beta}_k)$$

Whilst this method is guaranteed to converge faster, we have to compute H as well as its inverse, which is much more expensive than computing the gradient. Therefore, this algorithm is almost never used in practice.

As an illustration of the importance of the non-convexity of the loss function, consider Figure 2.3. 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 (or even 'good') solutions exist for this problem.

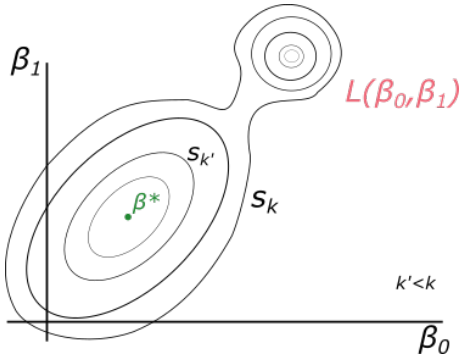


Figure 2.3. Cartoon of a non-convex loss function, with a second (deeper) well that contains the globally optimal solution that is hard to find using gradient descent methods.

5. Bias vs. variance

Here, we will describe what is sometimes known in Machine Learning as the *no-free-lunch* theorem. It describes 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.

> 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 least-squares estimate. First, we define two measures, the bias and the MSE of the estimator $\boldsymbol{\beta}^\star$:

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

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:

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

We see that 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.

However, this is not the whole story. On average, we will get the correct estimate, but what about the MSE of our estimator? This can also obtained easily from our expressions above as follows:

$$\mathbb{E}\big[(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)(\boldsymbol{\beta} - \boldsymbol{\beta}^\star)^T\big] = \mathbb{E}\big[(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T \underbrace{\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^T}_{\sigma^2\mathbf{I}}\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\big] = (\mathbf{X}^T\mathbf{X})^{-1}\sigma^2 \tag{2.7}$$

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:

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

where we use the symbol $\widehat{\cdot}$ to denote quantities estimated from data. Hence, the MSE of our estimated parameters depends on the observed 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. Recall from your Numerical Analysis and Linear Algebra courses that badly conditioning of a matrix is related to its being nearly non-invertible; being close to losing full rank; and having small singular values. The next section will deal with how we ameliorate the problems that can emerge from the bad conditioning of $(\mathbf{X}^T\mathbf{X})$.

The above arguments tell us that although LS is unbiased (good!) it can have high MSE in the estimated parameters (depending on properties of the data matrix $\mathbf{X}$). 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).

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 $\boldsymbol{\beta}^\star_{\mathrm{LS}}$ define the LS estimator such that $\hat{f}_{\mathrm{LS}}(\mathbf{x}_0) = \mathbf{x}_0^T\boldsymbol{\beta}^\star_{\mathrm{LS}}$, 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

$$\mathrm{MSE} = \mathbb{E}[(\hat{f}-y)^2] = \mathrm{var}(\hat{f}) + \big(y - \mathbb{E}[\hat{f}]\big)^2 = \mathrm{Variance} + \mathrm{Bias}^2.$$

This simple expression already highlights the main issue, i.e., that by minimising the MSE we could face a compromise between variance and bias.

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:

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

Since we have

$$\mathrm{MSE} = \mathbb{E}[(\hat{f}-y)^2] = \mathrm{var}(\hat{f}) + \big(y - \mathbb{E}[\hat{f}]\big)^2$$

then it follows directly that LS has the lowest MSE of all linear unbiased estimators.

The consequence is clear: 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 (2.7), least-squares can have large variance.

If we want to remain within linear methods, either we do something about $X^T X$ (see (2.7)), or we go for linear methods with a bias. That is the topic of the next section.

It is probably good to end with a visual representation of this discussion. Effectively, how good an estimator is depends on a combination of as low a bias and as low a variance as possible. You can see an illustration of this idea in Figure 2.4, 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*).

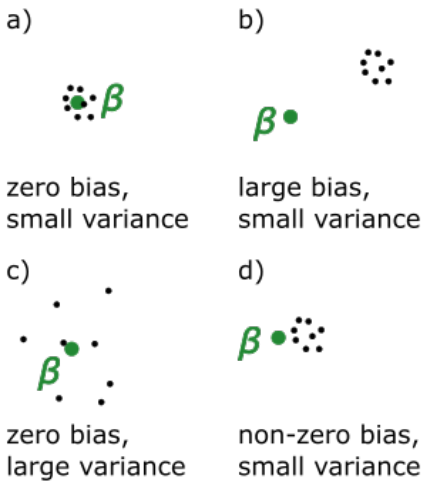


Figure 2.4. Overview of the trade-off between bias and variance. Each black dot represents an estimate $\boldsymbol{\beta}^\star$ of the true $\boldsymbol{\beta}$ (in green). In case a), we have an almost ideal case, with zero bias and small variance, which is rarely achievable in reality. Case b) shows 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).

6. Methods to reduce variance

As we just saw, least squares (i.e., plain vanilla linear regression) is unbiased but can be afflicted by large variance. We will see here a few of the approaches that are used to obtain models that ameliorate this issue either by trying to reduce the variance of LS directly, or going for methods that are no longer unbiased but might have lower variance (see Figure 2.4).

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

6.1. Reducing variance by choosing a 'good' subset of predictors. From (2.7), 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 (2.7) 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 X (i.e., the predictor variables), are nearly colinear 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. Another possibility is to use the singular value decomposition ('principal component analysis') to transform the problem onto a projected space where those redundant directions are projected out—we will see in the latter part of the course.

There is no perfect way to select a good subset of predictors. There are two main ways:

Brute-force complete enumeration: The only method with any guarantees is the 'brute force' approach, i.e., find the subset of k best parameters out of the p parameters by complete enumeration of all possibilities. Then, all k subsets of the p parameters have to be checked. An efficient algorithm to do this is called 'Leaps and bounds'. Of course, as p becomes larger than around 50, this becomes unfeasible due to combinatorial explosion. (It is left to the reader to get a sense of the number of parameter sets to be checked through this exhaustive method even for moderate p.) Also the selection of k is not trivial.

Greedy sequential selection: Another array of methods (with no guarantees) follow the sequential approach of adding/reducing descriptors until a criterion is met. There are two sub-categories of this method: Forward and Backward. In the forward approach, the algorithm will add $\mathbf{x}_i$ one at a time by choosing the predictor that reduces the error maximally. This is a typical 'greedy algorithm'. Conversely, the backward algorithm starts from the full LS model with p parameters and then, discards (one by one) the predictor that increases the error minimally. These algorithms are both implemented through the QR decomposition (also known as Gram-Schmidt):

$$\mathbf{X} = \mathbf{QR}$$
$$\mathbf{Q}^T\mathbf{Q} = \mathbf{I},$$

with $\mathbf{R}$ being upper triangular and $\mathbf{Q}$ orthogonal: the columns of $\mathbf{Q}$ provide an orthonormal basis of the space of features, i.e., it projects $\mathbf{X}$ onto orthogonal directions, thus guiding choose the largest/smallest increase at each step. The stopping criteria are usually based on heuristics, and the greedy process is not guaranteed to obtain a global minimum of the error (i.e., in this case, following at every step the smallest increase does not guarantee that you will end up at the global minimum of the error). Just to note that this is quite typical behaviour in many problems in combinatorial optimisation, and this makes them 'hard' computationally.

In summary, 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.

6.2. Shrinkage methods. There are some methods that 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).

First, we remind ourselves once more of the least squares solution of linear regression. Given $\mathbf{X}, \mathbf{y}$ (our data), the objective is to find $\boldsymbol{\beta}^\star$ by minimising the MSE loss function:

$$\min_{\boldsymbol{\beta}} L_{\text{LS}}(\boldsymbol{\beta}) = \min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2,$$

from which we then derived the following analytical solution:

$$\boldsymbol{\beta}^\star = \operatorname{argmin} L_{\text{LS}}(\boldsymbol{\beta}) = \mathbf{X}^+\mathbf{y} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

We now introduce *shrinkage methods*, in particular two alternatives: ridge regression and LASSO. Both methods entail a change in the loss function to be optimised. The proposed functions are based on an optimisation 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') these alternative methods change the loss function to try and make the *size* of coefficients 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.

6.2.1. *Ridge regression.* As mentioned above, the idea is to mimic the elimination of descriptors in a continuous fashion. In practice, we weight them low, by introducing the following altered loss function that penalises large values of $\|\boldsymbol{\beta}\|$:

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

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 regularisation 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:

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

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 X^TX 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 least squares method 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 2.5.

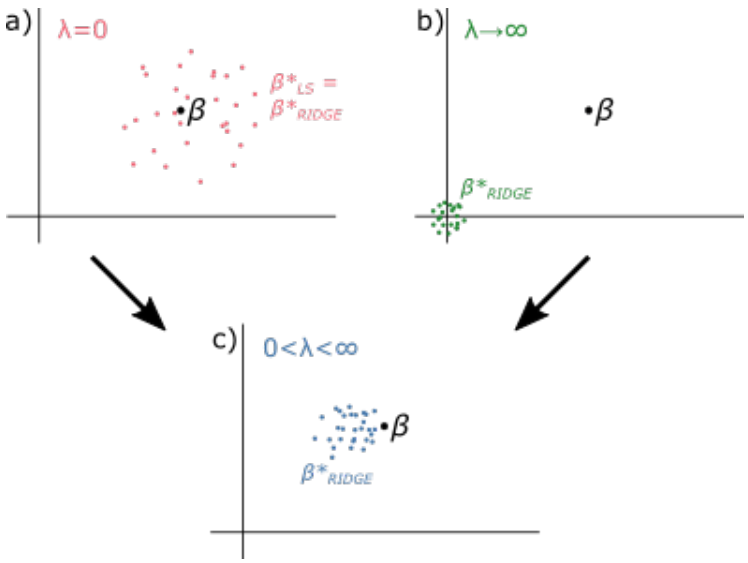


Figure 2.5. Bias and variance visualised for ridge regression. **a)** At $\lambda = 0$, we simply recover the least-squares estimate. **b)** As $\lambda \to \infty$, the bias tends towards $-\boldsymbol{\beta}$, resulting in the estimates being around 0. The variance decreases towards 0. **c)** As usual, the 'sweet spot' will be somewhere in-between.

6.2.2. *LASSO (Tibshirani).* Here, we introduce the LASSO (least absolute shrinkage and selection operator) method. As with the Ridge regression, we redefine the loss function to include a penalty (or regularisation) 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 regularisation 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 (quadratic programming in this case) to find the global optimum computationally. (This is out of remit for this course.)

In Figure 2.6, 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 makes many parameters have *small values*, so it induces a stronger sparsity than ridge. You can see this in Figure 2.6 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.

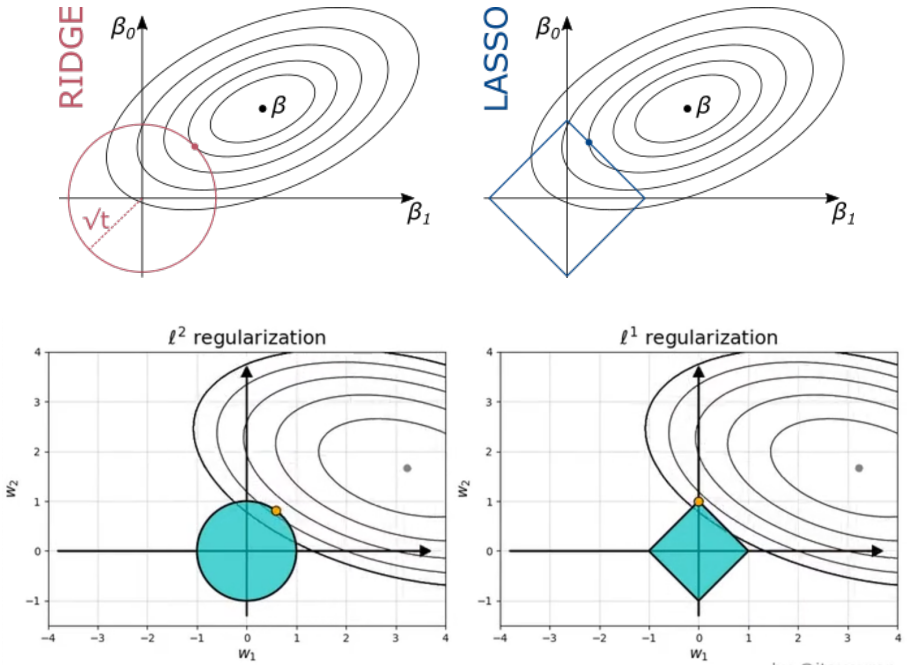


Figure 2.6. Figure showing the level curves of the least squares, i.e. such that $\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 = k$, in black. In red, we depict RIDGE regression and in blue, LASSO.

6.3. Regularisation. The above optimisation approach indicates a general procedure to introducing 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.

Here we briefly mention some other variations of regularisation, where the penalty contains 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, of course, with respectively $q = 2$ and $q = 1$.

Figure 2.7 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 we covered in Section 6.1. 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.

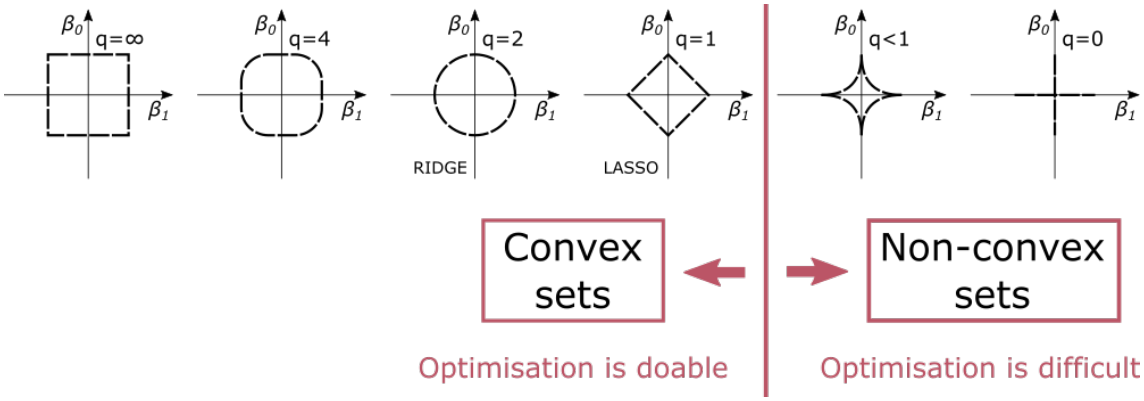


Figure 2.7. An overview of regularisation variants, i.e. $\lambda\|\boldsymbol{\beta}\|_q$ for a range of q.

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 convex combination of ridge and LASSO penalty terms:

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

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

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 regularization 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 less 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 2, 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.

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, 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, $\mathbb{R}^p$):

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

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 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 3.1a) shows an outline of these ideas.

The kNN algorithm is very simple to implement. Given an input $\mathbf{x}^{\text{in}}$:

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

$$\|\mathbf{x}^{\text{in}} - \mathbf{x}^{(i)}\|$$

(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 output samples in the neighbourhood:

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

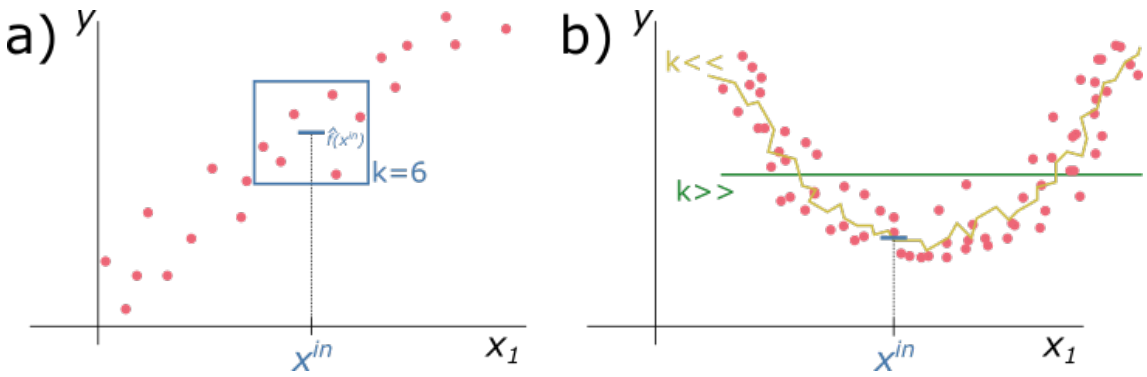


Figure 3.1. k-Nearest Neighbours. **a)** Given some arbitrary data (not necessarily linear), a new data point $\mathbf{x}^{\text{in}}$, and some k (in this case $k = 6$), the algorithm searches for the 6 nearest neighbours to $\mathbf{x}^{\text{in}}$. The prediction for $\mathbf{x}^{\text{in}}$ is then the average of the 6 points' outcome variables, marked by a blue bar. **b)** Here we can see that this also 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 evaluating different values.

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, normalising all predictor variables is key, unless the hypothesis states that some descriptors indeed are more important that others. 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 3.1, 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, but much more variable, and thus not very generalisable (hence prone to overfitting), and vice versa. In other words, k acts here equivalently to the parameter that modulates the regularisation term, 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 pervasive in Machine Learning and is applied across almost any statistical learning method. This procedure is termed **T-fold cross validation**, and it aims to control for the complexity of the model, so as to help us decide on the balance between accuracy and generalisability. Hence, 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 full training data or to parts of the training data.

More specifically, the procedure is as follows. First, we split the data $\mathcal{S} = \{\mathbf{x}^{(i)}, y^{(i)}\}$ for $i = 1, \ldots, N^{\text{training}}$ into T equal subsets called $\mathcal{S}_t$, such that:

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

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.

NB: 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 then test the model after having set the hyperparameters, we need to evaluate it on a portion of the data, of size $N^{\text{test}} = N - N^{\text{training}}$, that is not used in the neither in the training nor in the cross-validation procedure. We call it the *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 3.2, from which one can obtain an optimal $k^\star$ (or a range of 'good' k values in more realistic scenarios).

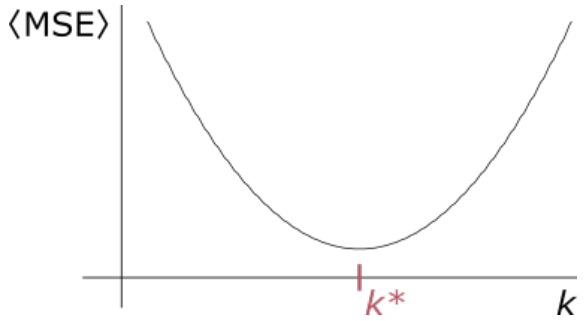


Figure 3.2. A schematic of the outcome of scanning k for a kNN model through T-fold cross-validation. This can help us choose an optimal $k^\star$. For small k, the average MSE will be large, since the model will not be generalisable (refer to figure 3.1) and thus will be performing badly when predicting 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 amount of 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 around 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. Then, the rest of the kNN algorithm will be exactly the same. This shows that defining a distance measure is key to incorporating any variety of predictor spaces.

3.2. k-Nearest Neighbours with discrete outcomes. The previous section 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 algorithm looks like the following:

(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 3.3 for a descriptive intuition. The idea is simple: $\mathbf{x}^{\text{in}}$ will be assigned the class of the majority of its neighbours.

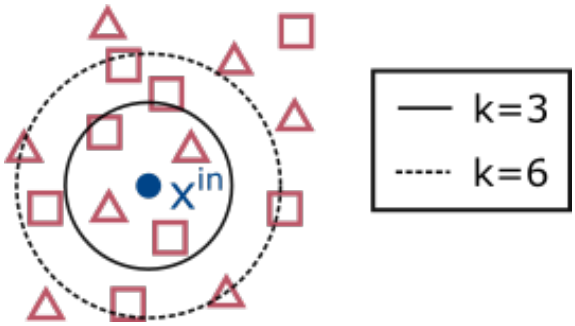


Figure 3.3. k-Nearest Neighbours for a discrete outcome variable. Here we have some data belonging to one of two classes: squares and triangles in some predictor space. For $k = 3$, we apply the majority rule to count two triangles and one square in the neighbourhood and get a prediction $\hat{y}(\mathbf{x}^{\text{in}}) = \triangle$, 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.

The *majority rule* is the simplest predictor model, which is sometimes presented as a discretisation of the following probabilistic predictor.

Remember 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 $\mathbf{x}^{\text{in}}$ 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:

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

> 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 8) and logistic regression (chapter 4), as we will see during the rest of the course.

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 some of these measures later on, starting with in Section 2 of Chapter 4, when we will introduce quality functions for classifiers). Once our quality function is defined, the concept of T-fold cross-validation remains the same as above.

Chapter 4

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**, where we have a binary (2-class) outcome variable to be predicted. (There are extensions to multi-class classification but are much less used for those problems.)

> *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 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. Hence we are addressing a classification problem.

The logistic regression model is global and linear, but adapted to the discrete nature of the outcomes. There are many ways to introduce and motivate logistic regression. You might want to read Section 4.4 of Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning* (ESL) to get some additional motivation. The logistic regression is a Generalised Linear Model and is based on some of the properties of the exponential family of distributions. Here we just give a simple description of the model and its assumptions. Chapter 4 of ESL contains an additional discussion of linear models for classification.

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*):

$$\log \frac{P(y=1)}{P(y=0)} = \log \frac{P(y=1)}{1 - P(y=1)} \tag{4.1}$$

where $P(y = 1)$ is the probability that our variable will take the value 1 (i.e., the probability of success).

The logistic regression model assumes that the logit function (4.1) is a linear combination of the input variables:

$$\mathbf{x}^T\boldsymbol{\beta} = \log \frac{P(y=1)}{1-P(y=1)} \tag{4.2}$$

Reminder: we are using here the shorthand notation we introduced in Chapter 2 (Linear Regression), where $\mathbf{x}$ is a $(p+1)$ length vector $(1, x_1, \ldots, x_p)$, allowing for a bias/intercept term, and $\boldsymbol{\beta} = \begin{pmatrix}\beta_0 & \beta_1 & \cdots & \beta_p\end{pmatrix}^T$.

With some rearranging of (4.2), we get:

$$P(y=1) = \frac{1}{1+e^{-\mathbf{x}^T\boldsymbol{\beta}}} = h\left(\mathbf{x}^T\boldsymbol{\beta}\right) =: h_{\boldsymbol{\beta}}(\mathbf{x}). \tag{4.3}$$

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

$$P(y=0) = 1 - h_{\boldsymbol{\beta}}(\mathbf{x}). \tag{4.4}$$

You already know how the logistic function looks but for a reminder, see Figure 4.1.

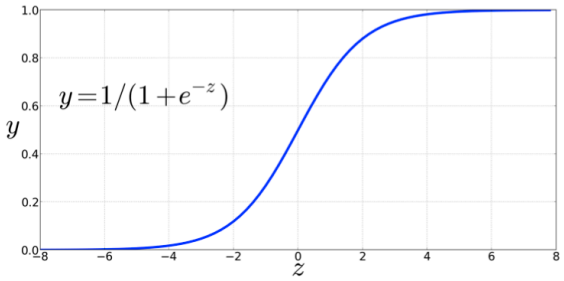


Figure 4.1. The logistic function for a one-dimensional variable z.

The above expressions (4.3)–(4.4) can be immediately understood as the result of assuming that our output variable is a *Bernoulli* random variable with a probability of success $P(y=1)$ (and of failure $P(y=0)$) :

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

In summary, the logistic regression model assumes that the output follows a Bernoulli random variable whose probability of success (or mean) is given by the logistic function $h_{\boldsymbol{\beta}}(\mathbf{x})$ given in (4.3).

> *Aside:* Logistic regression is a particular example of a Generalised Linear Model (GLM). IN GLMs, the output variable follows a particular distribution of exponential type (in this case, Bernoulli) and a particular 'link function' (in this case, the logit function) is assumed to be a linear combination of the inputs.
> By changing the distributions of the output variable and the link function, one can get various GLMs.

1.1. Estimating the parameters. To estimate the parameters $\boldsymbol{\beta}$ we maximise the log-likelihood of the model over the whole data set. If we assume independence in our samples, 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:

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

The optimisation follows a similar pattern to what we saw in linear regression in Chapter 2. As with linear regression, we set $\nabla_{\boldsymbol{\beta}} \mathcal{L} = 0$ to find the maximum. Note that:

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

so that:

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

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, which will be at $\nabla_{\boldsymbol{\beta}} \mathcal{L}|_{\boldsymbol{\beta}^\star} = 0$ (i.e., the equivalent of the Normal Equation in linear regression). This is where:

$$\nabla_{\boldsymbol{\beta}} \mathcal{L}|_{\boldsymbol{\beta}^\star_{\log}} = \sum_{i=1}^{N} \Big(y^{(i)} - h_{\boldsymbol{\beta}^\star_{\log}}(\mathbf{x}^{(i)})\Big)\, \mathbf{x}^{(i)} = \mathbf{0} \tag{4.5}$$

where the notation $\boldsymbol{\beta}^\star_{\log}$ indicates that this is the solution for the 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 (this is not shown here), standard optimisation algorithms (i.e., gradient ascent) are able to find the maximum.

Once we have obtained the $\boldsymbol{\beta}^\star_{\log}$ that fulfils the equation (4.5) by gradient methods, we can then plug this into the equation (4.3) to get the 'probability of success'. This is our probabilistic estimator for the input variables, which gives a real outcome (a probability):

$$P(y = 1|\mathbf{x}^{\text{in}}) = h_{\boldsymbol{\beta}^\star_{\log}}(\mathbf{x}^{\text{in}}) = h\big((\mathbf{x}^{\text{in}})^T \boldsymbol{\beta}^\star_{\log}\big) \in [0, 1]. \tag{4.6}$$

To get a classifier, we need the additional *discretisation step*. For that, we need to define a threshold for the classifier so that the sample is assigned to each of the two classes:

$$P(y = 1 \,|\, \mathbf{x}^{\text{in}}) > \tau \implies \mathbf{x}^{\text{in}} \in \{1\}, \tag{4.7}$$

where τ is the threshold for the classifier. 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).

Final aside: We can also summarise the solution to the logistic regression problem in matrix-vector form, as follows. Consider the vector version of (4.3) where $\mathbf{h}(\mathbf{X}\boldsymbol{\beta}^\star_{\log})$ is a N-dimensional vector function where each individual entry is:

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

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 (4.5) is:

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

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

$$\mathbf{X}^T\left[\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^\star_{\mathrm{LS}}\right] = 0.$$

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

We mentioned in Chapter 3, 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*. (If there are more than 2 classes, one will construct a *contingency table* and some of the general ideas below can be extended to that case, albeit non trivially. Here we stick for the moment with the analysis of binary outcomes.)

For the logistic regression with outcomes in $\{0,1\}$, Table 4.1 shows an overview of the different fields of a confusion matrix. With the four numbers (TP, FP, FN, TN) of this

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

Table 4.1. 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*.

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*:

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

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 sum of all entries of the confusion matrix ($N_{\mathrm{validation}}$ = FN + FP + TN + 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 diagnoses 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, 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. 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).

Multi-class classification. 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 frequency or not. For a detailed discussion see e.g. Grandini et al., *Metrics for multi-class classification: an overview* (2020).

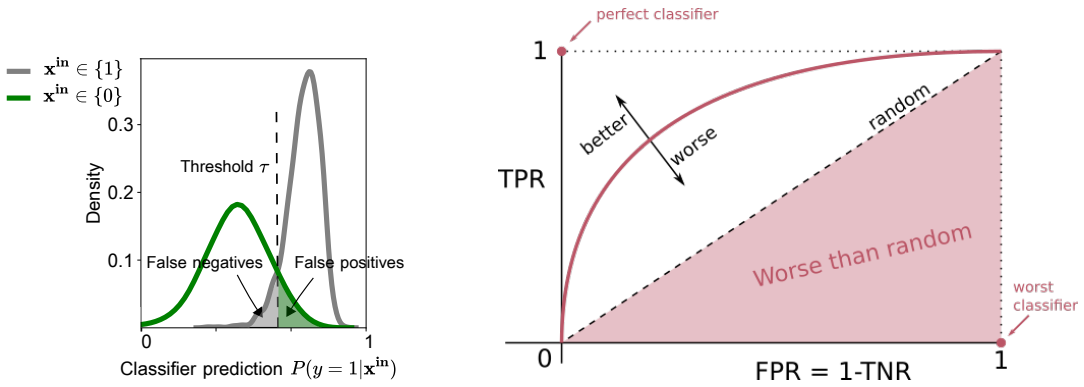


Figure 4.2. (Left) Cartoon illustration of a distribution of predictions by a classifier. (Right) The receiver operating characteristic (ROC) curve. When scanning across a range of values for the threshold (τ) parameter of a logistic regression (or some other binary outcome model) and plotting the true 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.

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 in a graphical manner, see Figure 4.2.

Remember that our classifier has a hyperparameter (the threshold τ) that can be tuned to fit our purpose. In Figure 4.2, 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 $> \frac{1}{2}$ since that will mean that we are performing better than random.

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.

Chapter 5

Naive Bayes classifier

In Chapter 4, we covered logistic regression, one of the classic algorithms for binary classification (i.e., the outcome variable is categorical, with only two outputs). This is the main use of logistic regression but, as we mentioned briefly, there are also well-known extensions of logistic regression that can be applied to output variables with more than two classes, i.e., multinomial logistic regression also known as multiclass logistic regression. Multinomial logistic regression is also a generalised linear model, as is logistic regression.

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 with such strong assumptions, this classifier is shown to work very well in many practical applications.

As usual, we start by stating our set-up. Our input and output variables in our training set with N samples are:

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

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

The Naive Bayes classifier computes the posterior probabilities using Bayes' theorem:

$$P(Y = y_q \mid X = \mathbf{x}) = \frac{\overbrace{P(X = \mathbf{x} \mid Y = y_q)}^{\spadesuit}\overbrace{P(Y = y_q)}^{\clubsuit}}{\underbrace{P(X = \mathbf{x})}_{\bigstar}}, \quad q = 1, \ldots, Q \tag{5.1}$$

where y_q denotes the value of the output variable in class c_q. 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 (5.1) from our data in order to obtain the classifier's estimate. To do so, we will need to make some (strong) assumptions along the way—hence the name *Naive Bayes*. The perhaps surprising outcome is that even with such strong assumptions, the resulting Naive Bayes classifier works surprisingly well in many applications.

1. Estimation of the components of the classifier (5.1)

'Prior' (♣): Firstly, we turn towards the *prior*, which is labelled ♣ 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:

$$P(Y = y_q) = \frac{\sum_{i=1}^{N} I(y^{(i)} = y_q)}{N}, \tag{5.2}$$

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

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

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 ♠.

Here, we now need to make the so-called *'naive assumption'* (this is where the method gets the first half of its name from), which is to assume the following:

$$\begin{aligned} P(X = \mathbf{x} \,|\, Y = y_q) &= P(X_1 = x_1, X_2 = x_2, \ldots, X_p = x_p \,|\, Y = y_q) \\ &= \prod_{j=1}^{p} P(X_j = x_j \,|\, Y = y_q) \end{aligned} \tag{5.3}$$

Generally, this assumption (independence of the p descriptors) will be wrong to make, of course. But in practice, it actually works quite well.

To estimate (5.3), there are two standard options:

- We can simply count the cases in the data:

$$P(X_j = x_j \,|\, Y = y_q) \approx \frac{\sum_{i=1}^{N} I(x_j^{(i)} = x_j \wedge y^{(i)} = y_q)}{\sum_{i=1}^{N} I(y^{(i)} = y_q)}, \quad j = 1, \ldots, p \tag{5.4}$$

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

  For discrete variables, this may lead to many entries ending up being empty, and so to account for this, one can use *Laplace smoothing*:

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

  This is a standard technique in statistical sampling to avoid having to deal with zero probabilities (which occur due to finite sampling).

- 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 sometimes called *Gaussian naive Bayes*. Specifically, assuming that each feature of the data is Gaussian in each

class:

$$P(X_j = x_j \mid Y = y_q) \approx \frac{1}{\sqrt{2\pi\sigma_{j,q}^2}} e^{\frac{-(x_j-\mu_{j,q})^2}{2\sigma_{j,q}^2}}, \quad j = 1, \ldots, p \tag{5.5}$$

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

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

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

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 ♠:

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

We don't need to compute this, of course. This is just to show that our probabilities are properly normalised as we quickly point out below.

Expression for the classifier: Putting it all together, we end up with the following expression for the Naive Bayes classifier:

$$P(Y = y_q \mid X = \mathbf{x}) = \frac{[\clubsuit \cdot \spadesuit]_q}{\sum_{q=1}^{Q} [\clubsuit \cdot \spadesuit]_q}, \tag{5.6}$$

where we have the expressions from the flat prior (5.2) and, for example, the Gaussian Naive Bayes likelihood (5.5) above:

$$\clubsuit_q = P(Y = y_q) = \frac{\sum_{i=1}^{N} I(y^{(i)} = y_q)}{N}$$

$$\spadesuit_q = \prod_{j=1}^{p} P(X_j = x_j \mid Y = y_q) = \prod_{j=1}^{p} \left[\frac{1}{\sqrt{2\pi\sigma_{j,q}^2}} e^{\frac{-(x_j-\mu_{j,q})^2}{2\sigma_{j,q}}} \right]$$

In summary, for a new sample $\mathbf{x}^{\text{in}}$, the Naive Bayes classifier will return a probability vector $\boldsymbol{\pi}^{(\text{NB})}$ of the data point belonging to each of the Q classes:

$$\boldsymbol{\pi}^{(\text{NB})} = P(Y = y_q \mid X = \mathbf{x}^{\text{in}}) = \begin{bmatrix} \pi_1^{(\text{NB})} \\ \vdots \\ \pi_Q^{(\text{NB})} \end{bmatrix}$$

$$\text{where each component is } \pi_q^{(\text{NB})} = \frac{[\clubsuit \cdot \spadesuit]_q}{\sum_{q=1}^{Q} [\clubsuit \cdot \spadesuit]_q}$$

Clearly, we have the following property by construction:

$$\mathbf{1}^T \cdot \boldsymbol{\pi}^{(\text{NB})} = 1.$$

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

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

A key advantage of the Bayes classifier is its scalability—due to our assumption that the features factorise in (5.3), the estimates for each of the descriptors is computed separately.

Chapter 6

Decision trees and random forests

In this chapter, we will be introducing a very powerful and popular method called *random forests*.

However, in order to do so we first need to start with *decision trees*, since they are the fundamental building blocks of random forests. 'A forest is set of trees', as we will see.

1. Decision trees

As usual, we start with the following setting:

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

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 regresssion tasks.

Although decision trees (and consequently, random forests) have been generalised to multivariate output variables, for simplicity, we restrict ourselves to the problem with a single output variable here.

In the following, 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 6.1 for a visual aid as well as explanation (in the caption) for the ideas that we now cover.

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 insight is to find the splits for the regions that will separate 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 one 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 variable x_j into two regions $R_1(j, s)$ and $R_2(j, s)$.

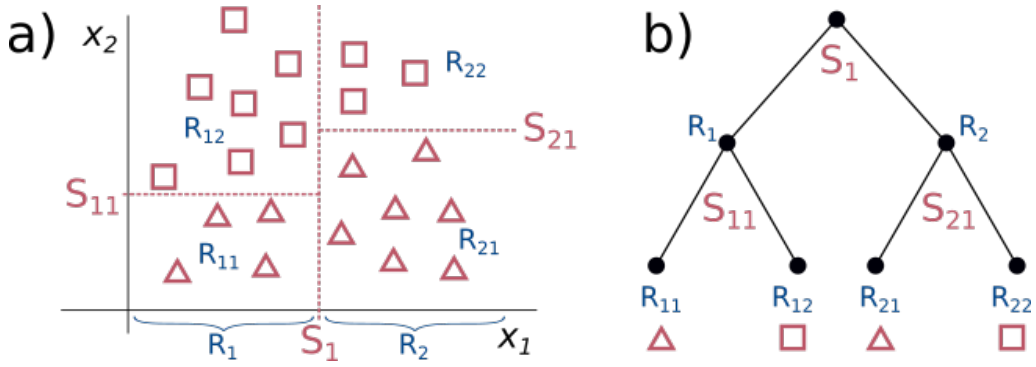


Figure 6.1. Visual intuition of a decision tree in a classification task. **a)** Using a simple case of two predictor variables as well as one categorical outcome with two classes, we can see that a decision tree is nothing other than a partition of the input variables' space. **b)** By computing different 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 6.1 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.

Our model (the *decision tree*) is then given by the set of splits (j, s) associated with the branching points of the binary tree. Given a new sample, we can either use the partition of space or the binary tree (they are equivalent) to determine in which region the new instance falls and thus, give a prediction for the class based on the majority of samples in that region.

This is the intuitive idea behind a decision tree. Our task is then to make the successive 'decisions', i.e., choose a descriptor and find a threshold in that descriptor that will improve our capability to separate the classes in the training set.

The algorithm: Clearly, achieving a perfect partitioning can not always possible be done perfectly, i.e. with 100% of cases being of a single class in each leaf node. More often than not, a majority rule will need to be applied.

The algorithm to determine/estimate a particular decision tree is greedy, since evaluating all possibilities of binary splits would be combinatorially and computationally intractable. In order to determine 'good' thresholds s, we need to define a loss/cost function, just like we have been doing throughout the course. The cost function will differ, depending on whether the problem at hand is a classification or regression task. Here, we will discuss the estimation of a decision tree for classification and leave the regression variant for the next section.

Since evaluating all possible combinations across a number of levels of splits is an intractable problem, we utilise a greedy algorithm, that will move one level at a time. In other words, the algorithm will first determine the first split optimally (i.e. S_1 in figure 6.1), and then move on to the next level (i.e. S_{11} and S_{21}) to obtain optimal splits, but without considering 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 split regardless of whether that variable was used before or not. This can lead to splits being made at the same level based on different input variables x_i.

The loss/cost function: The **loss function** is based on the classification quality achieved by a possible split (recall our discussion in Chapter 4 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, i.e. $q = 1, \ldots, Q$, 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. We could use a contingency table approach, similar to the one used in Chapter 4 on logistic regression and minimise the error rate. However, this approach has been empirically observed to not be very sensitive and therefore, won't work well in this case and is not used in practice.

Instead, other measures of classification quality are used. The two most broadly used are the *Gini index* and the *cross-entropy*. Both are instances of the same family of functions, namely 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:

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

1. Gini index. We can now write down some properties of the Gini index. For J classes, i.e. $i \in \{1, \ldots, J\}$ and a corresponding probability vector $\mathbf{p} = \begin{bmatrix} p_1 \\ \vdots \\ p_J \end{bmatrix}$ we have:

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

It is important to show that the GI is maximised when all the p_i's are equal. In order to do so, we use the method of **Lagrange multipliers**. This is a classic method to carry out optimisation of functions under constraints, which will reappear throughout the course.

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

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

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 $\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})$.

2. Cross-entropy. Consider a certain region $R_{\boldsymbol{\alpha}}$, where the node impurity is given by:

$$\pi_q(R_{\boldsymbol{\alpha}}) = \frac{\sum_{i=1}^{N} I\big(\mathbf{x}^{(i)} \in R_{\boldsymbol{\alpha}} \wedge y^{(i)} \in c_q\big)}{\sum_{i=1}^{N} I(\mathbf{x}^{(i)} \in R_{\boldsymbol{\alpha}})} \tag{6.1}$$

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:

$$\mathrm{CE}\big[\boldsymbol{\pi}, \hat{\boldsymbol{\pi}}\big] = -\sum_{q=1}^{Q} \pi_q(R_{\boldsymbol{\alpha}}) \log \hat{\pi}_q(R_{\boldsymbol{\alpha}}) \tag{6.2}$$

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:

$$\mathrm{CE}\big[\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})\big] = -\sum_{q=1}^{Q} \pi_q(R_{\boldsymbol{\alpha}}) \log \pi_q(R_{\boldsymbol{\alpha}}), \tag{6.3}$$

and the CE splitting criterion consists of minimising 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 6.2, 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. 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.

Therefore, the Gini index 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. Both of these measures play the role of functions for a classification task: for example, it can be shown that minimising the Gini index has the interpretation of minimising an error of classification at each splitting. (Note: it would be perfectly legitimate to use measures of accuracy, such as the mis-classification error for this purpose. In practice, however, the mis-classification error has a few shortcomings compared to both the Gini index and cross-entropy and hence is less used, see e.g. Section 9.2 of Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning*).

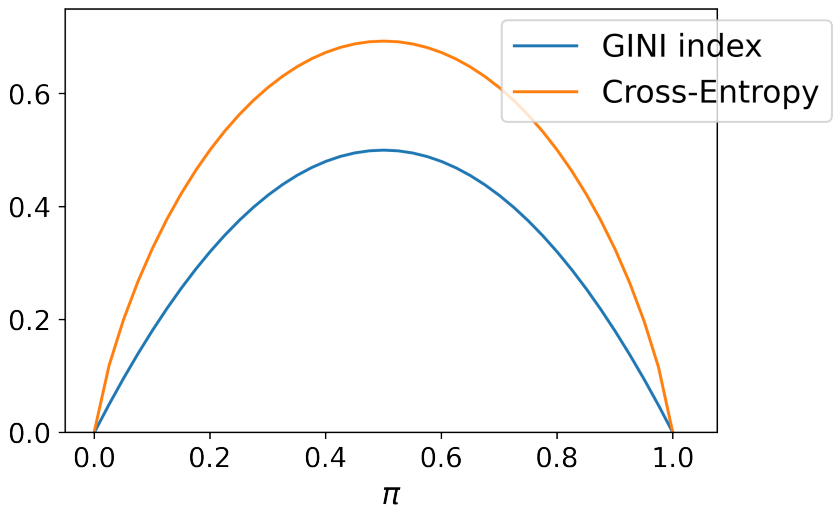


Figure 6.2. Gini index and entropy (or cross-entropy) for a two-class case.

Implementation in the algorithm: 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. In practical terms this means that, at every split, we want to find (j, s) such that:

$$\min_{j,s} \mathrm{GI}\big(\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})\big)$$

More precisely, by this expression, we mean that we need to minimise the sum of the $\mathrm{GI}\big(\boldsymbol{\pi}(R_{\boldsymbol{\alpha}})\big)$ for all regions α generated by the split, appropriately weighting each term in the sum by the fraction of points from the parent node that end up in each region. By finding the (j, s) split that minimises such quantity, 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 concept in the implementation in your Python notebook.

Hence the algorithm proceeds by greedily finding splits that will minimise the Gini index (or the cross-entropy, since both have similar properties).

When do we stop? 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.

1.2. Regression task: In this case, our task is to predict the continuous outcome variable $y \in \mathbb{R}$.

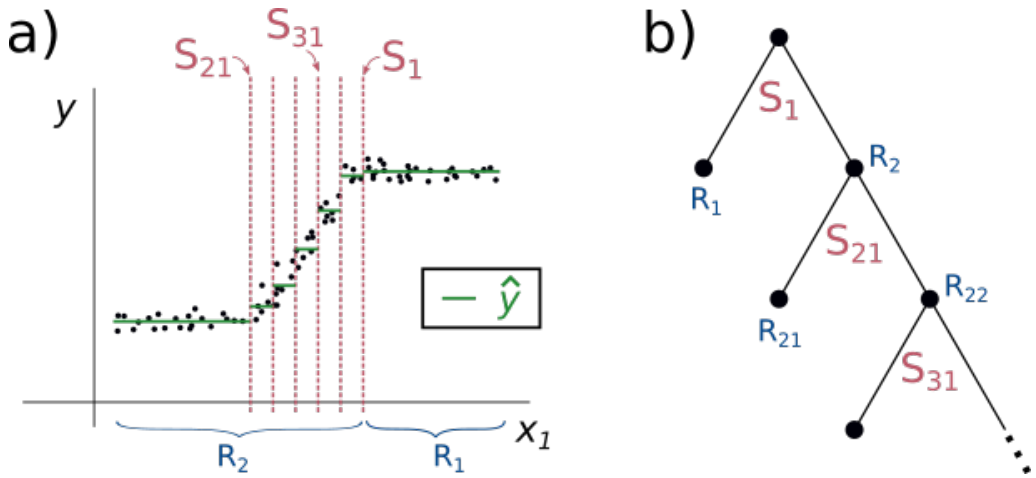


Figure 6.3. Visual intuition of a decision tree in a regression task. **a)** Given some data consisting of one input and one output variable, a decision tree will partition the space of x_1 into chunks where the data is relatively level (this vague expression will be made more precise later on). When 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.

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 6.3 as well as its

caption for a more in-depth explanation. We chose an nonlinear 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. See figure 6.4 for an example of this.

The only difference in the procedure lies in the cost function that is optimised at each split. In terms of estimating the decision tree, the greedy algorithm will choose (j, s) that minimises the following loss function:

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

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 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

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

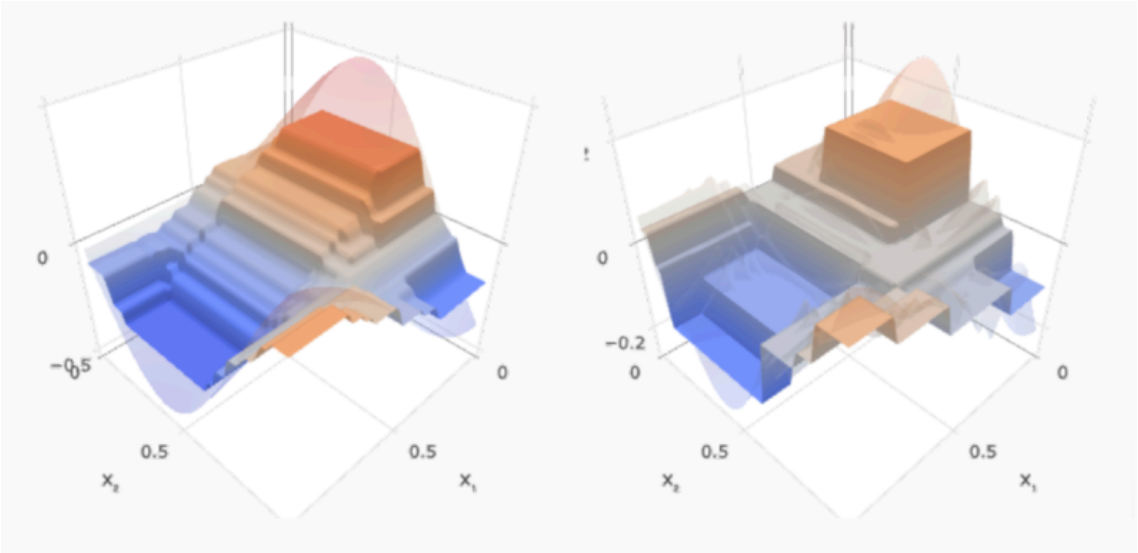


Figure 6.4. With a fine enough partition of the input space, decision trees can approximate very complex (as in complicated) non-linear functions.

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 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.

Once again, starting with the samples of training data $\mathcal{S} = \{\mathbf{x}^{(i)}, y^{(i)}\}, i = 1, \ldots, N$, we have the following algorithm:

(1) **Bootstrapping:** Produce B random samples from $\mathcal{S}$, all of size $N' \leq N$ by random sampling with replacement:

$$\mathcal{S}_b \text{ for } b = 1, \ldots, B$$

Sampling with replacement will lead to each $\mathcal{S}_b$ potentially containing repeated samples and/or not containing certain samples (absent samples) from the original $\mathcal{S}$.

(2) **Ensemble of models:** From each of the B samples, obtain a decision tree:

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

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 $\mathbf{x}^{\mathrm{in}}$, we simply obtain the average of all the predictions made by the ensemble of decision trees:

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

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

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

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

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

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

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 all correlated, then averaging them will not reduce the variance beyond a limit. Hence, we need to 'uncorrelate' them 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 used to fit each individual decision tree (in addition to the bootstrap algorithm). The steps of the random forest algorithm are then as follows:

(1) Generate random samples $\mathcal{S}_b$ for $b = 1, \ldots, B$, same as before (i.e., with replacement).

(2) Here we will introduce the change. Remember that each split in the decision trees can be written as the pair (j, s). Instead of allowing for j to be chosen from all predictor variables $\mathcal{P} = \{1, \ldots, p\}$ like before, we choose a random subset of predictors $\tilde{\mathcal{P}} \subset \mathcal{P}$, such that $|\tilde{\mathcal{P}}| = \tilde{p} < p$. Note that this is done individually at each split. Then, some $x_j \in \{x_i\}_{i \in \tilde{P}}$ is chosen as the predictor to make a decision along the threshold s as before. Since this is done at every decision point, we obtain an ensemble of modified decision trees that can be characterised by a series of subsets of $\mathcal{P}$:
$$\left\{ \hat{f}^{\mathrm{DT}}_{b,[\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2, \tilde{\mathcal{P}}_3, \ldots]} \right\}_{b=1}^{B}$$

(3) The aggregation step is then again identical to the bootstrap algorithm. For regression we use the average and for classification, we use the average probability vector.

2.3. Some comments on random forests.

2.3.1. *Loss of interpretability.* A disadvantage of following this approach is that there is a loss of interpretability. With aggregation, we lose the direct connection with the predictor variables. In other words, it becomes harder to gauge which predictors are important. But there are methods developed recently that can help with interpretability. This is an area of active research.

2.3.2. *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. Studies show that these tend to coverge fairly quickly.
- Minimum leaf size: A stopping criterion that will stop further splits as soon as the leaf reaches a minimum number of instances in the corresponding region, with the goal of preventing overfitting. By default, this is usually set to 1 for classification and to 5 for regression.
- One of the most important parameters is the number of predictors taken into consideration randomly at each split $|\tilde{\mathcal{P}}| = \tilde{p} < p$. If this is small, we will obtain a large variety of decision trees, guaranteed to not have high correlations and vice

versa. There are some guidelines for these choices:

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

Support vector machines (SVMs)

The support vector machine (SVM) model was first introduced by Vladimir Vapnik[1] and is largely based on the idea of geometrically solving classification problems using optimisation. So the flavour of this chapter is less statistical that the previous ones.

We will start by introducing Vapnik's main idea, namely: Consider a binary classification problem. A good way to classify discrete binary samples is to find a (hyper)plane that will optimally separate the samples that belong 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 '2', of course).

> 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.

We will now see how this problem can be posed and solved, and how this basic idea can be extended. SVMs are a very important tool in classification and are widely used in a variety of settings.

1. Formulation of the problem: Hard-margin SVMs

To introduce the method by Vapnik we start with the problem of *hard* classification of binary samples. By *hard* classification we mean 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. Recall our discussion on the *discretisation* step applied at the end of logistic regression (see (4.7) in Chapter 4. As we will see immediately, the method is actually *linear*. This should come as no surprise as we will be looking for a *separating hyperplane*.

> *Note:* We will discuss later in the chapter how to extend these ideas to *soft* (i.e., probabilistic) classification, as well as to *nonlinear* classifiers.

[1] It is described in Vapnik, V., *The Nature of Statistical Learning Theory*, 1996.

As per usual, we start with our familiar setup, where the input variables are continuous and the output variable is binary:

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

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 (see Figure 7.1). As described above, the goal of SMV's is intuitively easy: find the hyperplane in our input space (for $\mathbf{x} \in \mathbb{R}^2$ the hyperplane is a line; for $\mathbf{x} \in \mathbb{R}^3$, the hyperplane is a plane) that separates one class from the other. 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.

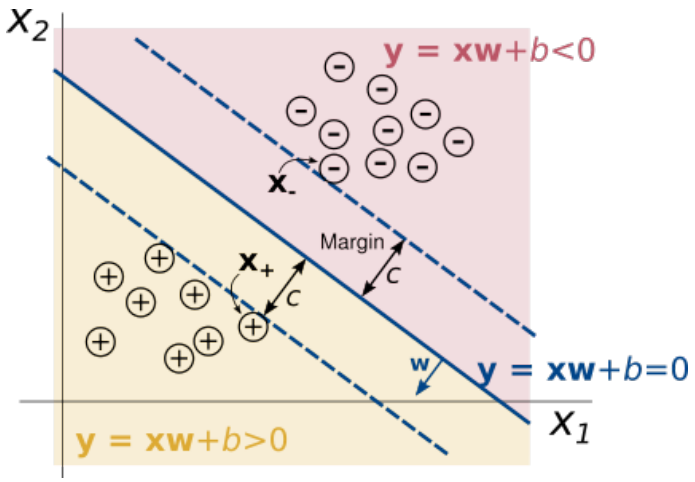


Figure 7.1. Sketch of a support vector machine with two predictors and binary outcome. The line of interest is given in blue, with each side being coloured in either red or yellow.

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 7.1). 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.

Since we are looking for hyperplanes, the method is linear. Our goal is to obtain $\mathbf{w}$ and b, which parameterise our hyperplane. Once we have determined both, we can use the following rule to make predictions with the model:

(7.1) **SVM classifier:** Given $\mathbf{x}^{\text{in}}$, if $\begin{cases} \mathbf{x}^{\text{in}} \cdot \mathbf{w} + b \geq 0 & \text{then } \hat{y} = +1 \\ \mathbf{x}^{\text{in}} \cdot \mathbf{w} + b < 0 & \text{then } \hat{y} = -1 \end{cases}$

This corresponds to the point $\mathbf{x}^{\text{in}}$ falling in the orange region or the pink region, respectively, in Figure 7.1.

As can be seen in Figure 7.1, there could be an infinite number of lines that would separate the data. Intuitively, the goal is to find the 'widest street' possible through the data. SVMs approach this problem through the concept of the *margin*, which denotes the smallest distance between the decision boundary (the line in the simple example of Figure 7.1) and any of the samples. In short, 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 7.1. First of all, the line that is maximally far away from both sets of data points (in the sense of least squares), is half-way in-between, hence we have used the same symbol c to denote the magnitude of the margin on both sides. Maximising the margin is hence equivalent to maximising the overall width of the 'street' between data points. Let $\mathbf{x}_+$ be the closest data point of class +1 to the dividing line and similarly for $\mathbf{x}_-$ for class -1 (Figure 7.1). Then we have the following:

$$\begin{cases} \mathbf{x}_+ \cdot \mathbf{w} + b = c \\ \mathbf{x}_- \cdot \mathbf{w} + b = -c. \end{cases} \tag{7.2}$$

Geometrically, the width of the street is given by:

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

and it follows immediately from (7.2) 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:

$$\text{width} = \frac{2}{\|\mathbf{w}\|} \tag{7.3}$$

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

Our choice of notation for the outcome classes as $\{-1, +1\}$ now comes in handy. We can write the decision conditions compactly as:

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

which combined with (7.3) allows us to write the (hard classification) SVM optimisation problem:

$$\text{(7.4)} \qquad \min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to} \quad y^{(i)}(\mathbf{x}^{(i)} \cdot \mathbf{w} + b) \geq 1, \qquad i = 1, \ldots, N$$

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 $\mathbf{x}^{(i)} \cdot \mathbf{w} + b = 0$ at a distance of at least 1 (which is the distance to the hyperplane from both $\mathbf{x}_+$ and $\mathbf{x}_-$).

Aside: The solution of the optimisation problem (7.4) is not analytical, but follows on ideas that we have already mentioned in passing in our discussion of Ridge regression and LASSO in Chapter 2.

We will not cover this in this course but just a few pointers follow. Recall some of the optimisation strategies that are used when we have constraints:

(1) **Lagrange multipliers:** These are used to optimise a function (linear or nonlinear) subject to *equality* constraints. That is, given the following optimisation problem:

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{subject to} \quad g_i(\mathbf{x}) = 0 \text{ for } i = 1, \ldots, m$$

From this we can construct the Lagrangian:

$$\mathcal{L} = f(\mathbf{x}) + \sum_{i=1}^{m} \alpha_i g_i(\mathbf{x})$$

This can be solved through taking the following gradients:

$$\left\{ \begin{array}{l} \nabla_{\mathbf{x}} \mathcal{L} = 0 \\ \nabla_{\boldsymbol{\alpha}} \mathcal{L} = 0 \end{array} \right\} \text{ at } (\mathbf{x}^\star, \boldsymbol{\alpha}^\star)$$

(2) **Linear programming:** Here, we want to optimise a *linear* objective function under *linear* equality and/or inequality constraints:

$$\min_{\mathbf{x}} [\mathbf{w} \cdot \mathbf{x}] \text{ subject to } \begin{cases} g_i(\mathbf{x}) = 0 \text{ for } i = 1, \ldots, m \\ h_j(\mathbf{x}) \geq 0 \text{ for } j = 1, \ldots, k \end{cases}$$

Note that both the equality constraints, g_i, and the inequality constraints, h_j, are *linear* and the constraints define a convex set. This sort of problem can be solved by the *simplex method*.

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

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

This is our problem (7.4), where the optimisation variable is $\mathbf{w}$, of course.

The solution to this problem is given by the Karush-Kuhn-Tucker (KKT) conditions, who derived them in 1939 and, in more general form, in 1951. The KKT conditions are related to Lagrange multipliers but require further proofs beyond the scope of this course.

Briefly, as above, we construct the Lagrangian with Lagrange multipliers $\alpha_i \geq 0$:

$$\mathcal{L} = f(\mathbf{x}) + \sum_{i=1}^{m} \alpha_i g_i(\mathbf{x})$$

This function has some important properties (KKT, not proved). At the minimum $\mathbf{z}^\star$, there exists

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

such that:

$$\nabla f(\mathbf{z}^\star) + \sum_{i=1}^{m} \alpha_i \nabla g_i(\mathbf{z}^\star) = 0 \quad \text{(Lagrangian vanishes)}$$

$$g_i(\mathbf{z}^\star) \leq 0 \qquad \forall i \quad \text{(primal costraints)}$$

$$\alpha_i \geq 0 \qquad \forall i \quad \text{(dual constraints)}$$

(♠) $$\alpha_i \cdot g_i(\mathbf{z}^\star) = 0 \qquad \forall i \quad \text{(complementary slackness conditions)}$$

As noted in the aside, in order to solve the optimisation problem of SVMs, we cannot directly use Lagrangian multipliers, because we are working with inequality constraints. Since we have a non-linear objective function, we cannot use linear programming, but quadratic programming will do the trick. To fit the notation from the aside, we can now rewrite the problem slightly :

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

To solve the problem, we construct the Lagrangian:

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

and, as seen in the aside, we take the gradient of the system:

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

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:

$$\mathcal{L}(\boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + \sum_{i=1}^{N} \alpha_i - \underbrace{\sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)}}_{=\mathbf{w}} \cdot \mathbf{w} - \underbrace{\sum_{i=1}^{N} \alpha_i y^{(i)} b}_{=0} \tag{7.5}$$

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

$$\mathcal{L}(\boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + \sum_{i=1}^{N} \alpha_i - \mathbf{w} \cdot \mathbf{w} = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\mathbf{w} \cdot \mathbf{w} \tag{7.6}$$

$$= \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \tag{7.7}$$

Here, the last equation was obtained by substituting $\mathbf{w}^\star = \sum_{i=1}^N \alpha_i y^{(i)} \mathbf{x}^{(i)}$ into the equation. This expression shows that the Lagrangian can be expressed only in terms of the samples and the Lagrange multipliers.

In this course, we do not go into the details of convex optimisation but this is another problem that can be solved by means of the dual optimisation, i.e. by considering:

$$\frac{\partial \mathcal{L}}{\partial \alpha_i}, \quad i = 1, \ldots, N$$

and the associated dual problem:

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

Now we can use some of our derivation above to understand the key insight of Vapnis' SVM result. To understand the key point, we look back at the aside above, and specifically at the complementary slackness conditions (♠) that appear from the KKT conditions. These equations say that either the primal constraint is tight or the dual constraint is tight—one of them has to be strictly zero. 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* $\mathbf{x}_-$*.*

Since the goal is to obtain $\mathbf{w}^\star$, it follows that only the two minimal points $\mathbf{x}_+$ and $\mathbf{x}_-$ define the vector of the hyperplane:

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

These two points $\mathbf{x}_+$ and $\mathbf{x}_-$ are called the *support vectors* of the hyperplane (hence the name of the method), which are found by the process of convex optimisation sketched briefly above.

Once we have obtained the parameters of the hyperplane, we can use the model to classify any new sample using the classifier (7.1).

2. Soft-margin SVMs

In Section 1, we introduced SVMs in its original formulation for hard classification. In this section, we will now look at ways to extend SVMs to more realistic scenarios.

The hard-margin SVM as formulated above requires the data points to be separable in d-dimensional space. This is usually not possible in most applications. When the separating hyperplane does not exist, we say the hard classification SVM problem is infeasible.

We now turn to the case when a strictly separating hyper-plane does not exist for the data, i.e. the case of *imperfect separation*. Our formulation above contains the seeds of how to extend the method naturally for such a case.

In order to deal with imperfect separation, we relax the conditions by introducing a *penalty* term in the loss function. In other words, we score the quality of the hyperplane by penalising the violations for our separating hyperplane. To do this, one can use the *hinge loss* function (Fig. 7.2), defined as follows:

$$\zeta^{(i)} = \max \Big\{ 0, \underbrace{1 - (\mathbf{x}^{(i)} \cdot \mathbf{w} + b) y^{(i)}}_{\text{size of violation}} \Big\},$$

where, as discussed above, $y^{(i)}(\mathbf{x}^{(i)}\mathbf{w} + b) \geq 1$ for all points if there is no violation of the constraints.

Given this loss function, the soft-margin SVM optimisation will then be:

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

where all the penalties $\zeta^{(i)} \geq 0$ are positive by definition (see Fig. 7.2).

Following this formulation, the same optimisation procedure applies, i.e. this optimisation problem can be solved through similar methods as above with an additional hyperparameter, λ. This parameter allows us to tune how 'hard' the boundaries can be, allowing for larger or smaller violations of the constraints. This hyperparameter can be tuned by various methods such as cross-validation. Note that, in the limit of large λ, this optimisation approaches the hard-margin SVM.

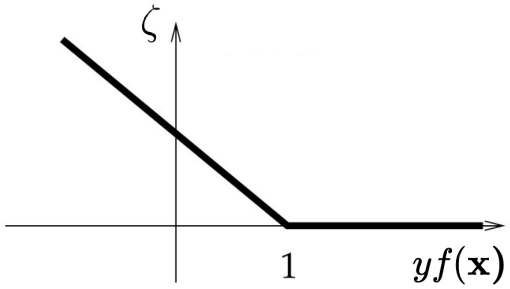


Figure 7.2. Hinge loss ζ as a function of $yf(\mathbf{x})$, where $f(\mathbf{x}) = (\mathbf{x} \cdot \mathbf{w} + b)$. The soft-margin optimization is defined in such a way as either to keep ζ at zero (for $yf(\mathbf{x}) > 1$, i.e., where the hard-margin constraint is satisfied) or to keep the size of the violation small (for $yf(\mathbf{x}) < 1$).

3. Beyond linear: Kernelised SVMs

The second important extension of SVMs is to consider non-linear classification. Again, the seed for this extension is contained in our formulation above.

Going beyond linearity, we now discuss the generalisation of SVMs to be able to consider 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 need to invoke the '*kernel trick*', which fits neatly into our previous formulation by realising that all the expressions in our derivation of the optimisation contained 'dot products of vectors', which can be seamlessly changed to (nonlinear) *kernel functions*. As we will see, a kernel function is a generalisation of inner products.

To explain this further, we recap the story so far, and point out the places where the kernel functions could be included in lieu of the dot products.

Assume there are only two support vectors (i.e., no degeneracies for simplicity). From our derivation above, we have the following model:

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

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

where the coefficients α_i are determined via the dual-problem formulation, i.e. by maximising the Lagrangian (7.5):

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

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 (7.1) 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 notice that not only the solution, but also the prediction boils down to computing dot products. This is where we will use kernel functions to accommodate non-linear data.

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 ($\mathbb{R}^d$) to give the dot product of transformed (non-linear) vectors in a higher-dimensional space ($\mathbb{R}^D$). This is a function of the following form:

$$k(\mathbf{x}, \mathbf{y}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{y}) \tag{7.8}$$

where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ 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. 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. 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.

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

$$\text{Given} \quad \mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$$

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

Note that we have the following dot product:

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

We can then (cleverly) define a function k and make the following observation:

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

Hence we do not need to compute the dot product in the higher dimensional space D; we can just compute the dot product in the lower dimensional space d and then apply the transformation k. Whilst this toy example was simply going from $d = 2$ to $D = 3$ dimensions (not great savings), suppose we had defined a kernel function in $D = 1000$ dimensions.

The kernel trick tells us that we do not always need to compute $\boldsymbol{\phi}$ to get the 100-dimensional vector, but rather (if chosen cleverly), all we need to do is compute the usually much simpler dot product $\mathbf{x} \cdot \mathbf{y}$ and work with that instead. Hence we can lift our data to higher dimensions, *if the only thing we care about is the dot products* (as is the case here in SVMs).

Some widely used kernels in applications are:

- Polynomial (up to order n):

$$k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^n$$

- Gaussian radial basis function:

$$k(\mathbf{x}, \mathbf{y}) = e^{-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{\sigma}}$$

- Sigmoid:

$$k(\mathbf{x}, \mathbf{y}) = \tanh\big(\beta(\mathbf{x} \cdot \mathbf{y}) + c\big)$$

In the case of SVMs, we can choose an appropriate kernel and train the corresponding SVM. To show how this can be done through the kernel trick, let us first introduce the notation $\mathbf{w}_{\boldsymbol{\phi}}$ for the parameters of the SVM with kernel $k(\mathbf{x}, \mathbf{y})$, which is associated to the non-linear mapping $\boldsymbol{\phi}(\mathbf{x})$ of the data matrix $\mathbf{x}$ through (7.8). Then, the optimisation of the kernelised SVM can be written as:

$$\min_{\mathbf{w}_{\boldsymbol{\phi}}} \frac{1}{2}\|\mathbf{w}_{\boldsymbol{\phi}}\|^2 \text{ subject to } 1 - y^{(i)}(\boldsymbol{\phi}(\mathbf{x}^{(i)}) \cdot \mathbf{w}_{\boldsymbol{\phi}} + b) \leq 0 \text{ for } i = 1, \ldots, N \tag{7.9}$$

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

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

which allows us to write:

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

and

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

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

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

As a result, learning a kernelised SVM can be done by minimising the loss function:

$$\frac{1}{2}\mathbf{u}^{\mathrm{T}} \mathbf{K} \mathbf{u} + \lambda \sum_{i=1}^{N} \zeta^{(i)} \text{ subject to } 1 - y^{(i)}(\mathbf{K}^{(i)} \mathbf{u} + b) \leq \zeta^{(i)} \text{ for } i = 1, \ldots, N \tag{7.10}$$

with respect to $\mathbf{u}$ and b ($\mathbf{K}^{(i)}$ represents the row i of the matrix $\mathbf{K}$). Note here the advantage of the kernel trick: the dimension of the optimisation problem in (7.10) 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 eq. 7.9), leaving the freedom to introduce potentially very high-dimensional transformations $\boldsymbol{\phi}(\mathbf{x})$. The soft-margin kernelised SVM with hinge loss then corresponds to the loss function:

$$\frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathbf{K}\mathbf{u} + \lambda \sum_{i=1}^{N} \max\left\{0, 1 - y^{(i)}(\mathbf{K}^{(i)}\mathbf{u} + b)\right\}$$

where λ is the hyper-parameter controlling boundary violation penalty (hardness). The kernel trick can be even more conveniently applied to the dual problem - this is often mentioned as a motivation for working with the dual formulation. Looking back at the SVM expressions, we plug in the kernel and the Lagrangian becomes:

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

which allows us to write:

$$\boldsymbol{\phi}(\mathbf{x}) \cdot \mathbf{w}_{\boldsymbol{\phi}} + b = \sum_{i} \alpha_i y_i k(\mathbf{x}, \mathbf{x}^{(i)}) + b$$

through the α_i which maximise $\mathcal{L}(\boldsymbol{\alpha})$. Once we determine the support vectors, using this expression allows us to determine the value of the intercept b, see section 12.3.1 of *The Elements of Statistical Learning Data Mining, Inference, and Prediction.* 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. In the coding task for this course, we instead focus on the primal-form formulation (eq. 7.10), which can be solved by first-order methods.

3.3. Why use kernel SVMs? The idea is that non-linearity and high dimensionality can help with separability and thus, classification. For $\mathbf{x}, \mathbf{y}$ in $\mathbb{R}^d$, we will then consider $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{z} \in \mathbb{R}^D$, where $D \gg d$ potentially. In some cases, if we are able to choose the right $\boldsymbol{\phi}$ with the right properties, that will allow us to separate the data nicely using a hyperplane in the lifted (higher dimensional) space.

An intuition of why the kernel trick can work is given in Figure 7.3 with a contrived 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 7.3(a) shows that there is no 'good' line in $\mathbb{R}^2$ (i.e., no good linear 'hyperplane' 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 7.3(b), where we have added a further dimension to the data $x_3 = x_1^2 + x_2^2$ so that the blue and red samples can now be separated with a plane normal to x_3. Hence by lifting our data to a higher dimensional space, we were able to use a separating hyperplane with low penalties. This principle of lifting to higher dimensions has many applications in areas of data science.

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

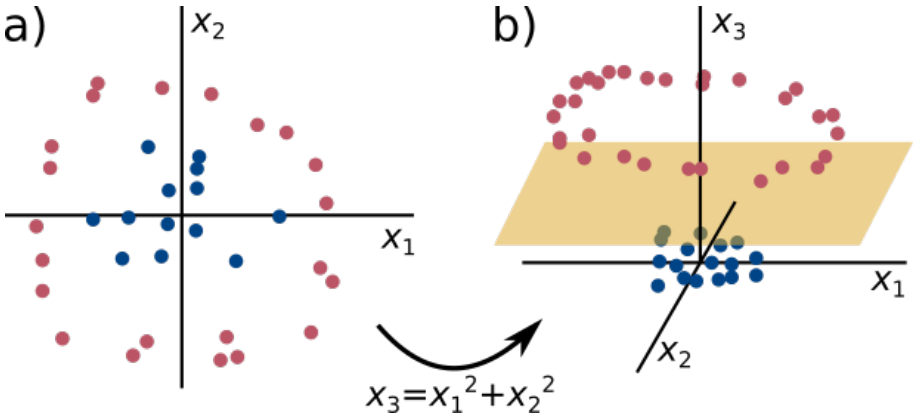


Figure 7.3. Cartoon drawing of the kernel trick. **a)** shows a set of binary samples that are linearly non-separable in $\mathbb{R}^2$. **b)** shows the kernel trick in action. Through the addition of a third variable $x_3 = x_1^2 + x_2^2$, we can make the data separable with a (hyper)plane in $\mathbb{R}^3$.

lifted, higher dimensional space. 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

Chapter based on notes by Dr Kevin Webster

1. The mathematical neuron

Early research into neural networks focused on models of learning in the brain and used mathematical (or artificial) neurons as fundamental building blocks. These are simple models of neurons in the brain that receive a set of inputs, which are weighted and summed before being passed through an activation function (or transfer function):

$$y_k = \sigma\left(\sum_j w_{kj}x_j + b_k\right) \tag{8.1}$$

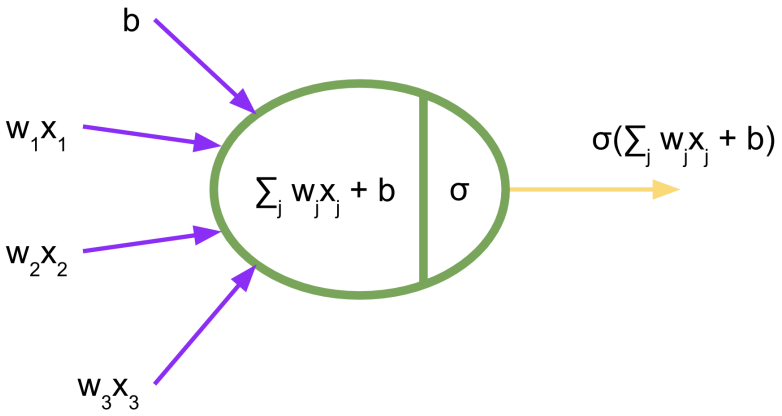


Figure 8.1. Sketch of a mathematical neuron.

In the above, the inputs to the neuron are denoted by x_j, the weights w_{kj}, bias b_k and activation function σ. The weights and bias are parameters that need to be tuned for the given task. The first artificial neuron was developed by McCulloch and Pitts [17], which used a simple threshold activation function (step function) only on binary inputs, and produce a binary output. Later, Rosenblatt [23] 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.

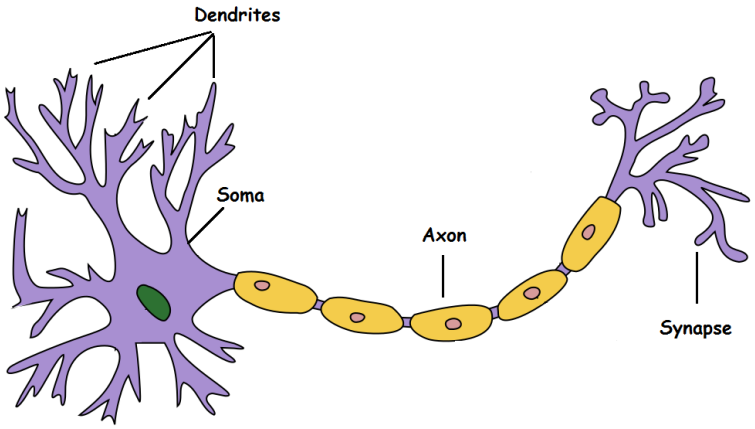


Figure 8.2. Sketch of a biological neuron. Source: Wikipedia

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.

2. Stochastic gradient descent

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 (1986a) [25]), 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 such as stochastic gradient descent (SGD, Robbins and Monro 1951 [22]) to be used in the optimisation of model parameters. For a more complete historical account on the research into neural networks, we refer to [8] (chap. 2, section 1.2).

2.1. Activation functions. A typical example of a smooth activation function is the logistic sigmoid:

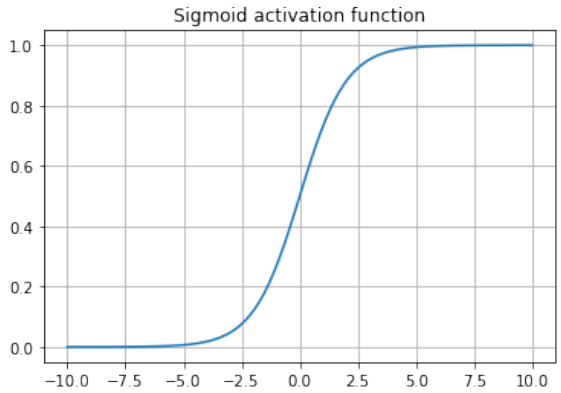


Figure 8.3. Sigmoid activation function.

Note that linear regression and logistic regression can both be viewed as artificial neuron models, with linear (or no) activation function and sigmoid activation function respectively.

Other activation functions that are commonly used in deep learning models are the ReLU (rectified linear unit), tanh, ELU (exponential linear unit, Clevert et al 2016 [2]), SELU (scaled exponential linear unit, Klambauer et al 2017 [14]), softplus, swish (Ramachandran et al 2018 [21]), which can be seen in figure 8.4.

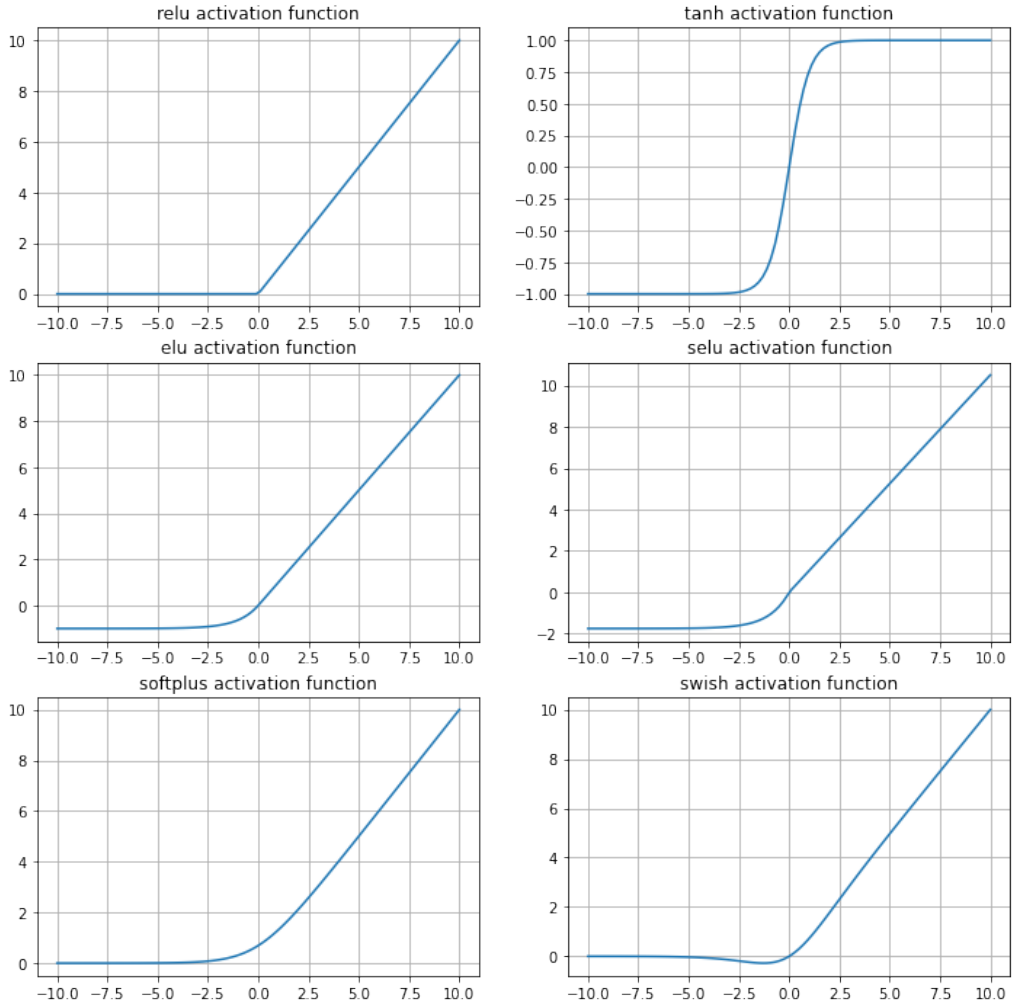


Figure 8.4. A variety of activation functions.

You can see a complete list of available activation functions in `tf.keras.activations` at `https://www.tensorflow.org/api_docs/python/tf/keras/activations`.

2.2. Gradient descent. Suppose we have constructed our neural network model, which we represent as the function $f_\theta : \mathbb{R}^D \mapsto Y$, where Y is the target space (e.g. $\mathbb{R}$ or $[0, 1]$). Suppose also that we have defined a suitable loss function

$$L(\theta; \mathcal{D}_{train}) := \frac{1}{|\mathcal{D}_{train}|} \sum_{x_i, y_i \in \mathcal{D}_{train}} l(y_i, f_\theta(x_i)),$$

where $l(y_i, f_\theta(x_i))$ is the per-example loss. Then the gradient $\nabla_\theta L(\theta_0; \mathcal{D}_{train})$ evaluated at θ_0 defines the direction of steepest ascent in parameter space at the point θ_0.

The gradient descent algorithm takes an initial guess for the parameters θ_0 and updates, at each iteration t, the parameter values according to the rule

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; \mathcal{D}_{train}), \qquad t \in \mathbb{N}_0$$

where $\eta_t > 0$ is a small learning rate which may depend on t. For a suitably chosen η_t, the iterates $L(\theta_t; \mathcal{D}_{train})$, $t \in \mathbb{N}_0$ converge to a local minimum.

2.3. Stochastic gradient descent. Note that computing $\nabla_\theta L(\theta; \mathcal{D}_{train})$ 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.

Stochastic gradient descent provides a cheaper estimate of the full gradient, by computing the gradient on a minibatch of data points, instead of the full data set. In particular, we evaluate the gradient

$$L(\theta; \mathcal{D}_m) = \frac{1}{M} \sum_{x_i, y_i \in \mathcal{D}_m} l(y_i, f_\theta(x_i)),$$

where $\mathcal{D}_m$ is a randomly sampled minibatch of training data points, $M = |\mathcal{D}_m|$ is the size of the minibatch (typically much smaller than $|\mathcal{D}_{train}|$). We then use the gradient $\nabla_\theta L(\theta_t; \mathcal{D}_m)$ to update the parameters

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; \mathcal{D}_m), \qquad t \in \mathbb{N}_0$$

This update provides a stochastic approximation to the true gradient which is far more efficient to compute, and provides a huge speed up in the training process for large data sets.

3. Multilayer perceptrons

The simplest type of deep learning model is the **multilayer perceptron**, also known as a **feedforward network**. This type of neural network can be viewed as an architecture consisting of layers of mathematical neurons, linked together in a directed acyclic graph.

3.1. MLP with single hidden layer. A key property of deep learning models is the fact that they are *compositional* instead of *additive*. Whereas linear regression models (or logistic regression, kernel regression) increase complexity by adding extra basis functions ϕ_i in the expansion

$$f(\mathbf{x}) = \sum_i w_i \phi_i(\mathbf{x}),$$

deep learning models increase complexity by composing multiple simple functions φ_k together:

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

up to a certain L, which stands for the number of 'hidden' (i.e., internal) layers. The functions φ_k are defined to be affine transformations followed by an element-wise activation function. An example is the MLP with a single hidden layer:

$$h_j^{(1)} = \sigma\left(\sum_{i=1}^{D} w_{ji}^{(0)} x_i + b_j^{(0)}\right), \qquad j = 1, \dots, n_h, \tag{8.2}$$

$$\hat{y} = \sigma_{out}\left(\sum_{i=1}^{n_h} w_i^{(1)} h_i^{(1)} + b^{(1)}\right). \tag{8.3}$$

In the above, $\mathbf{x} \in \mathbb{R}^D$ is an example input, $n_h \in \mathbb{N}$ is the number of hidden units in the 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.

Following 8.2 and 8.3 we also define the **pre-activations** $a_j^{(1)} := \sum_{i=1}^{D} w_{ji}^{(0)} x_i + b_j^{(0)}$. Correspondingly, the $h_j^{(1)}$ are referred to as the **post-activations** (or frequently, just **activations**). This construction is summarised in figure 8.5.

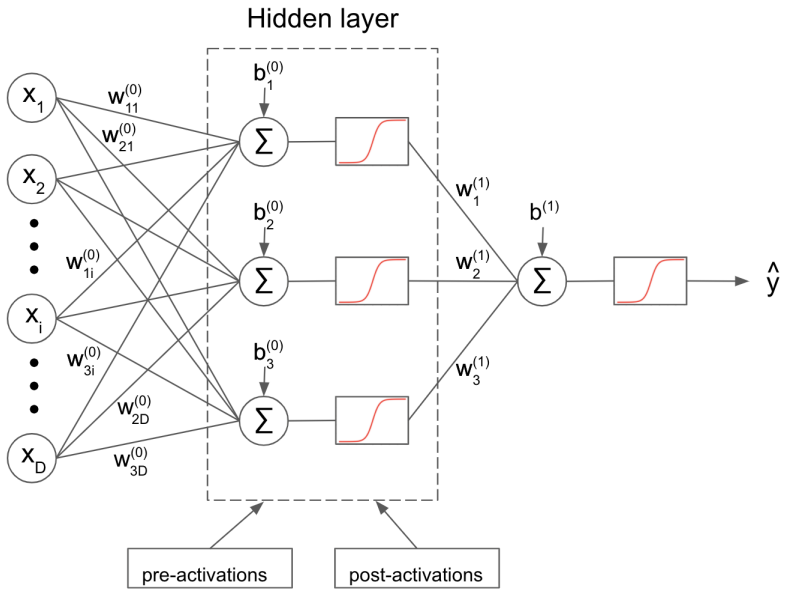


Figure 8.5. Multilayer perceptron with a single hidden layer consisting of three neurons.

We will usually write equations 8.2 and 8.3 in the more concise form:

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right), \tag{8.4}$$

$$\hat{y} = \sigma_{out}\left(\mathbf{w}^{(1)}\mathbf{h}^{(1)} + b^{(1)}\right), \tag{8.5}$$

where $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{W}^{(0)} \in \mathbb{R}^{n_h \times D}$, $\mathbf{b}^{(0)} \in \mathbb{R}^{n_h}$, $\mathbf{h}^{(1)} \in \mathbb{R}^{n_h}$, $\mathbf{w}^{(1)} \in \mathbb{R}^{1 \times n_h}$, $b^{(1)} \in \mathbb{R}$ and we overload notation with the activation functions $\sigma, \sigma_{out} : \mathbb{R} \mapsto \mathbb{R}$ by applying them 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.

3.2. MLP with multiple hidden layers. More generally, for an MLP with L hidden layers, we have

$$\mathbf{h}^{(0)} := \mathbf{x}, \tag{8.6}$$

$$\mathbf{h}^{(k)} = \sigma\left(\mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)}\right), \qquad k = 1, \ldots, L, \tag{8.7}$$

$$\hat{y} = \sigma_{out}\left(\mathbf{w}^{(L)}\mathbf{h}^{(L)} + b^{(L)}\right), \tag{8.8}$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{n_{k+1} \times n_k}$, $\mathbf{b}^{(k)} \in \mathbb{R}^{n_{k+1}}$, $\mathbf{h}^{(k)} \in \mathbb{R}^{n_k}$, and we have set $n_0 := D$, and n_k is the number of units in the k-th hidden layer. Figure 8.6 shows the MLP architecture with two 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 models [26].

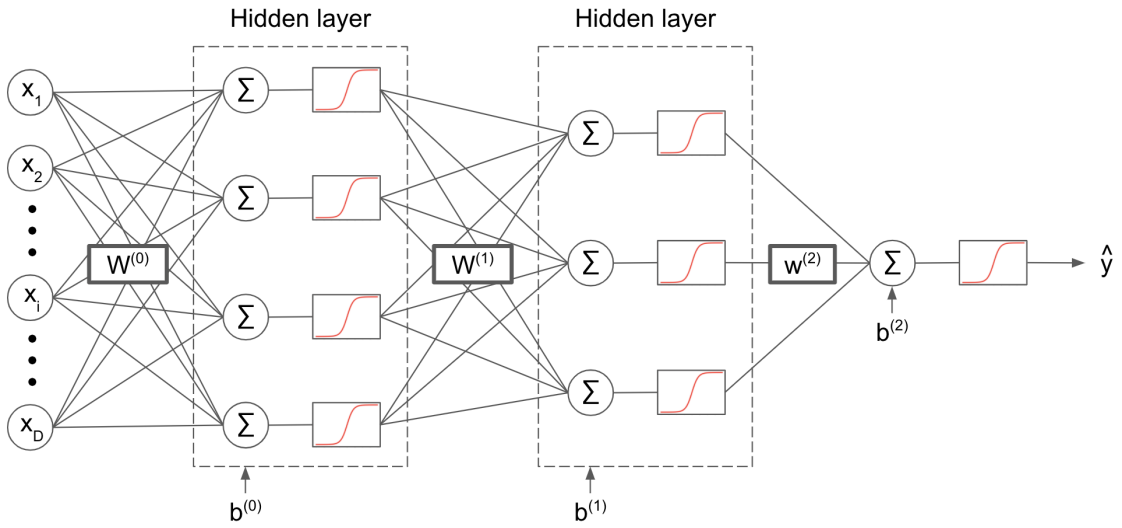


Figure 8.6. Multilayer perceptron with a two hidden layers.

3.3. 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 8.8 we have considered a single unit output y, which is produced by passing the pre-activation $a^{(L+1)} := \mathbf{w}^{(L)}\mathbf{h}^{(L)} + b^{(L)}$ through the activation function σ_{out}.

Note how linear regression (chapter 2) and logistic regression (chapter 4) 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 8.8 with $\hat{\mathbf{y}} = \sigma_{out}\left(\mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}\right)$, as can be seen in figure 8.7.

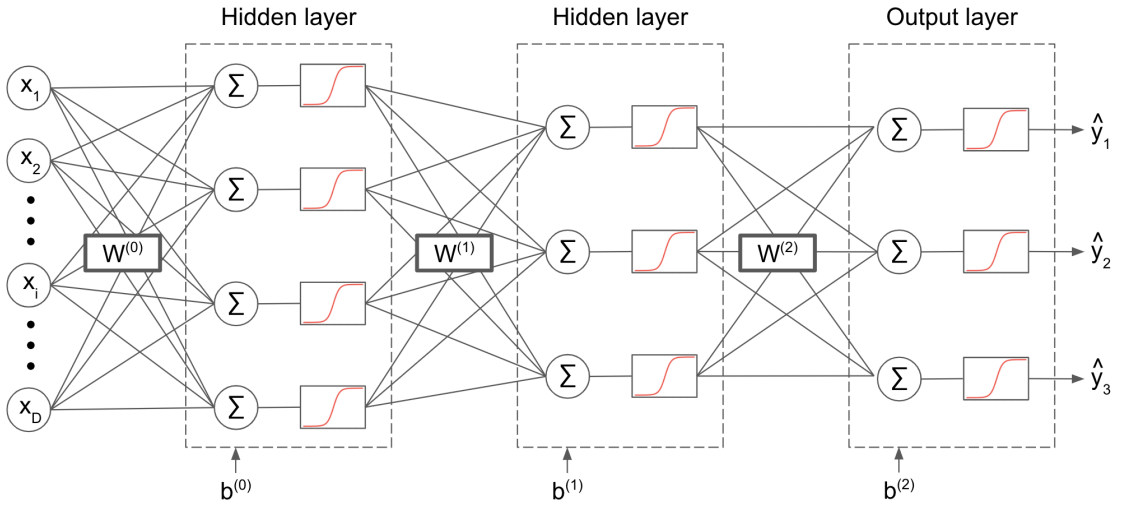


Figure 8.7. Multilayer perceptron with multiple outputs.

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.

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

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

with $\mathbf{W}^{(L)} \in \mathbb{R}^{C \times n_L}$, $\mathbf{b}^{(L)} \in \mathbb{R}^C$ where C is the number of classes, the softmax function is given by

$$\hat{\mathbf{y}}_j := \operatorname{softmax}(\mathbf{a}^{(L+1)})_j = \frac{\exp(a_j)}{\sum_i \exp(a_i)}.$$

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 8.8 for a schematic.

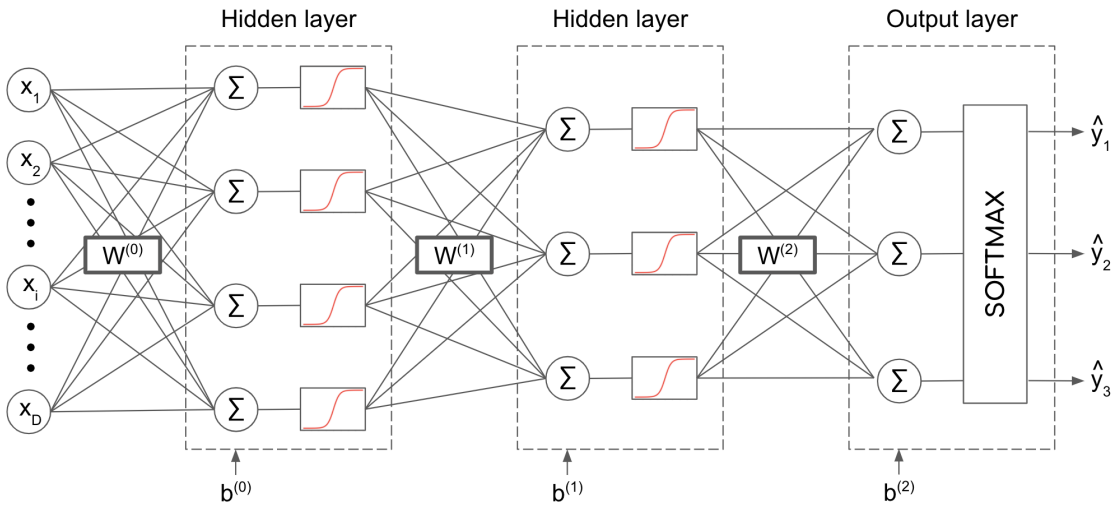


Figure 8.8. Multilayer perceptron with a softmax output layer.

4. Error backpropagation

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

We have already seen the parameter update rule according to stochastic gradient descent for a neural network model $f_\theta : \mathbb{R}^D \mapsto Y$, where Y is the target space (e.g. $\mathbb{R}^{n_{L+1}}$ or $[0,1]^{n_{L+1}}$):

$$\theta_{t+1} = \theta_t - \eta \nabla_t L(\theta_t; \mathcal{D}_m), \qquad t \in \mathbb{N}_0. \tag{8.9}$$

In the above equation, the minibatch loss $L(\theta_t; \mathcal{D}_m)$ is calculated on a randomly drawn sample of data points from the training set,

$$L(\theta_t; \mathcal{D}_m) = \frac{1}{M} \sum_{x_i, y_i \in \mathcal{D}_m} l(y_i, f_{\theta_t}(x_i)), \tag{8.10}$$

where $\mathcal{D}_m$ is the randomly sampled minibatch, $M = |\mathcal{D}_m| \ll |\mathcal{D}_{train}|$ is the size of the minibatch, and we will denote $L_i : Y \times Y \mapsto \mathbb{R}$, given by $L_i := l(y_i, f_\theta(x_i))$, as the per-example loss.

The update 8.9 requires computation of the term $\nabla_t L(\theta_t; \mathcal{D}_m)$ (from here we drop the $\mathcal{D}_m$ in this expression for brevity); that is, the gradient of the loss function with respect to all of the model parameters, evaluated at the current parameter settings θ_t.

The computation of the derivatives is done through applying the chain rule of differentiation. The algorithm for computing these derivatives in an efficient manner is known as **backpropagation**, and was popularised for use in neural network optimisation in Rumelhart et al 1986b [26] and Rumelhart et al 1986c [27], although the technique dates back earlier, see e.g. Werbos [31] which includes Paul Werbos' 1974 dissertation.

In this section, we will derive the important backpropagation algorithm for finding the loss function derivatives for a multilayer perceptron. First recall the layer transformations in the MLP:

$$\mathbf{h}^{(0)} := \mathbf{x}, \tag{8.11}$$

$$\mathbf{h}^{(k)} = \sigma\left(\mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)}\right), \qquad k = 1, \ldots, L, \tag{8.12}$$

$$\hat{\mathbf{y}} = \sigma_{out}\left(\mathbf{w}^{(L)}\mathbf{h}^{(L)} + b^{(L)}\right), \tag{8.13}$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{n_{k+1} \times n_k}$, $\mathbf{b}^{(k)} \in \mathbb{R}^{n_{k+1}}$, $\mathbf{h}^{(k)} \in \mathbb{R}^{n_k}$, $\hat{\mathbf{y}} \in Y$, $\sigma, \sigma_{out} : \mathbb{R} \mapsto \mathbb{R}$ are activation functions that are applied element-wise, $n_0 := D$, and n_k is the number of units in the k-th hidden layer.

Also recall that we define the **pre-activations**

$$\mathbf{a}^{(k)} = \mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)} \tag{8.14}$$

and **post-activations**

$$\mathbf{h}^{(k)} = \sigma(\mathbf{a}^{(k)}). \tag{8.15}$$

We will consider the gradient of the loss computed on a single data example $\nabla_t L_i(\theta_t)$, given the sum 8.10. We first compute the **forward pass** (8.11)-(8.13) and store the preactivations $\mathbf{a}^{(k)}$ and post-activations $\mathbf{h}^{(k)}$.

Consider the derivative of L_i with respect to $w_{pq}^{(k)}$ and $b_p^{(k)}$. We have:

$$\frac{\partial L_i}{\partial w_{pq}^{(k)}} = \frac{\partial L_i}{\partial a_p^{(k+1)}} \frac{\partial a_p^{(k+1)}}{\partial w_{pq}^{(k)}} \tag{8.16}$$

$$= \frac{\partial L_i}{\partial a_p^{(k+1)}} h_q^{(k)}, \tag{8.17}$$

where the second line follows from 8.14. Similarly,

$$\frac{\partial L_i}{\partial b_p^{(k)}} = \frac{\partial L_i}{\partial a_p^{(k+1)}} \frac{\partial a_p^{(k+1)}}{\partial b_p^{(k)}} \tag{8.18}$$

$$= \frac{\partial L_i}{\partial a_p^{(k+1)}}. \tag{8.19}$$

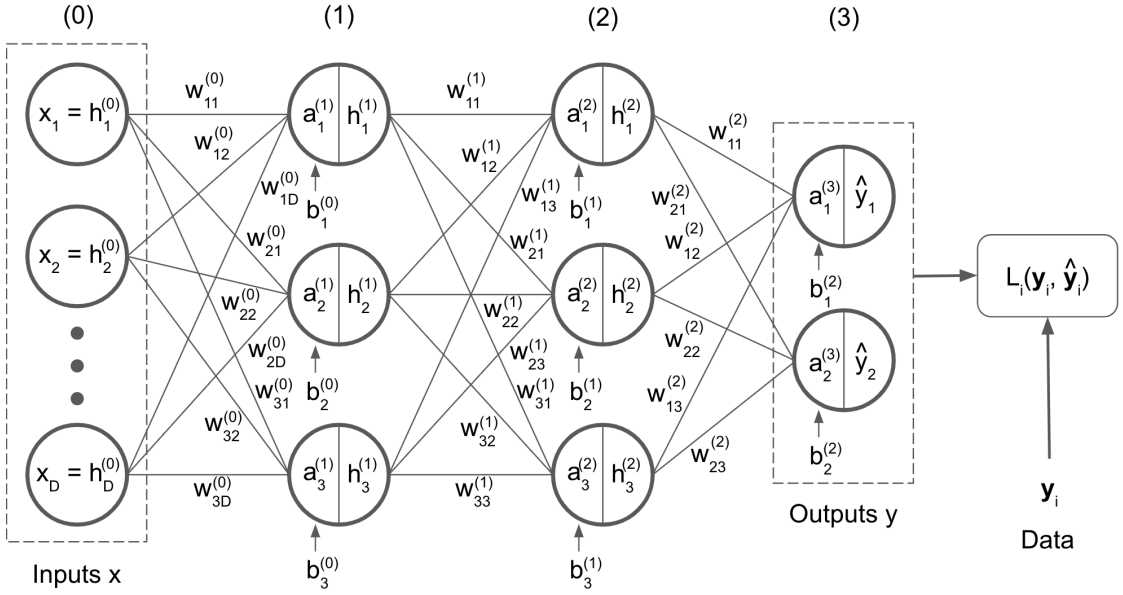


Figure 8.9. Pre-activations, post-activations, weights and biases in the forward pass

We introduce the notation $\delta_p^{(k)} := \frac{\partial L_i}{\partial a_p^{(k)}}$, called the **error**. We then write

$$\frac{\partial L_i}{\partial w_{pq}^{(k)}} = \delta_p^{(k+1)} h_q^{(k)} \tag{8.20}$$

$$\frac{\partial L_i}{\partial b_p^{(k)}} = \delta_p^{(k+1)}. \tag{8.21}$$

We therefore need to compute the quantity $\delta_p^{(k+1)}$ for each hidden and output unit in the network. Again using the chain rule, we have

$$\delta_p^{(k)} \equiv \frac{\partial L_i}{\partial a_p^{(k)}} = \sum_{j=1}^{n_{k+1}} \frac{\partial L_i}{\partial a_j^{(k+1)}} \frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} \tag{8.22}$$

$$= \sum_{j=1}^{n_{k+1}} \delta_j^{(k+1)} \frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} \tag{8.23}$$

Combining 8.14 and 8.15 we see that

$$a_j^{(k+1)} = \sum_{l=1}^{n_k} w_{jl}^{(k)} \sigma(a_l^{(k)}) + b_j^{(k)} \tag{8.24}$$

$$\frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} = w_{jp}^{(k)} \sigma'(a_p^{(k)}) \tag{8.25}$$

where σ' is the derivative of the activation function. So from the above equation and 8.23 we have

$$\delta_p^{(k)} \equiv \frac{\partial L_i}{\partial a_p^{(k)}} = \sum_{j=1}^{n_{k+1}} \delta_j^{(k+1)} \frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} \tag{8.26}$$

$$= \sigma'(a_p^{(k)}) \sum_{j=1}^{n_{k+1}} w_{jp}^{(k)} \delta_j^{(k+1)} \tag{8.27}$$

Equation 8.27 is analogous to 8.24, and describes the backpropagation of errors through the network. We can write it in the more concise form (analogous to 8.15):

$$\delta^{(k)} = \sigma'(\mathbf{a}^{(k)})(\mathbf{W}^{(k)})^T \delta^{(k+1)},$$

where $\sigma'(\mathbf{a}^{(k)}) = \text{diag}([\sigma'(a_p^{(k)})]_{p=1}^{n_k})$.

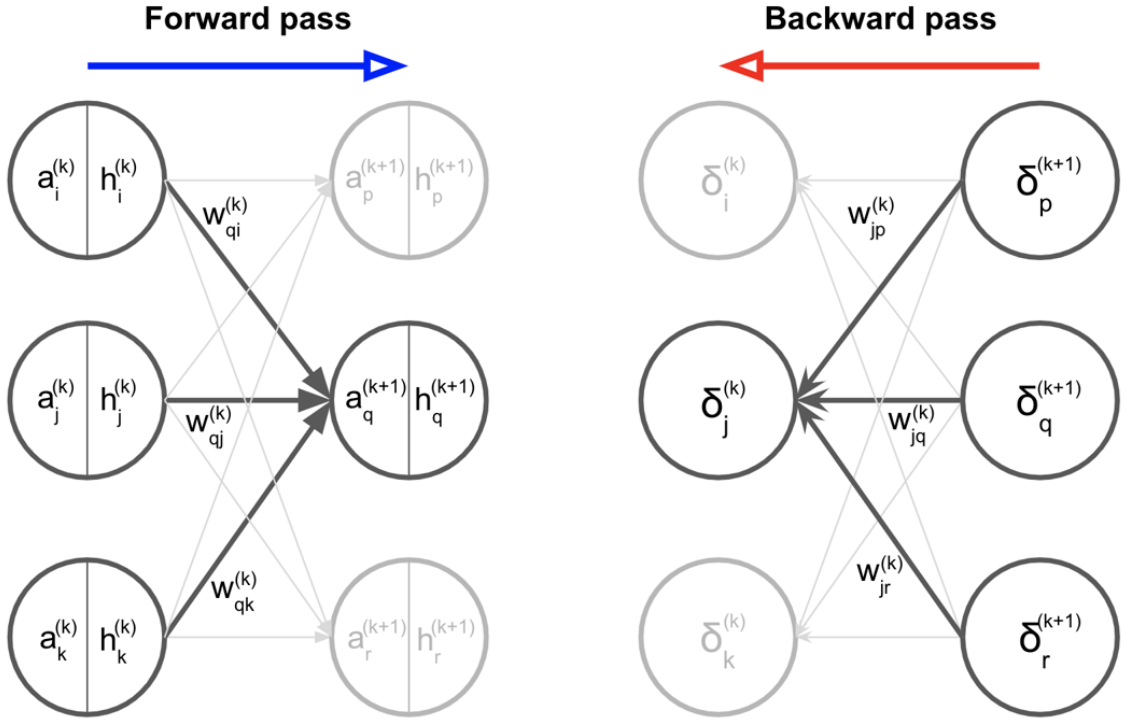


Figure 8.10. Forward and backward passes

Now we can summarise the backpropagation algorithm as follows:

(1) Propagate the signal forwards by passing an input vector x_i through the network and computing all pre-activations and post-activations using $\mathbf{a}^{(k)} = \mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)}$

(2) Evaluate $\delta^{(L+1)} = \frac{\partial L_i}{\partial \mathbf{a}^{(L+1)}}$ for the output neurons

(3) Backpropagate the errors to compute $\delta^{(k)}$ for each hidden unit using $\delta^{(k)} = \sigma'(\mathbf{a}^{(k)})(\mathbf{W}^{(k)})^T \delta^{(k+1)}$

(4) Obtain the derivatives of L_i with respect to the weights and biases using $\frac{\partial L_i}{\partial w_{pq}^{(k)}} = \delta_p^{(k+1)} h_q^{(k)}$, $\frac{\partial L_i}{\partial b_p^{(k)}} = \delta_p^{(k+1)}$

The backpropagation algorithm can easily be extended to apply to any directed acyclic graph, but we have presented it here in the case of MLPs for simplicity.

5. 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).

5.1. Stochastic gradient descent. We have already seen how stochastic gradient descent (SGD, Robbins & Monro 1951 [22]) can be applied to optimise neural network parameters.

Recall that SGD computes stochastic gradients by computing the loss on a minibatch of samples:

$$L(\theta_t; \mathcal{D}_m) = \frac{1}{M} \sum_{x_i, y_i \in \mathcal{D}_m} l(y_i, f_{\theta_t}(x_i)),$$

where $\mathcal{D}_m$ is a randomly sampled minibatch of training data points, $M = |\mathcal{D}_m|$ is the size of the minibatch (typically much smaller than $|\mathcal{D}_{train}|$). We then use the gradient $\nabla \tilde{L}(\theta_t)$ to update the parameters according to the SGD update rule

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} L(\theta_t; \mathcal{D}_m), \qquad t \in \mathbb{N}_0.$$

Stochastic gradient descent reduces redundancy in the gradient computation, and is faster than full (batch) gradient descent. However some challenges remain:

- Convergence can still be very slow with SGD
- Setting the learning correctly can be difficult, involving trial and error
- Different weights might operate on different scales, and require different rates of learning

Several optimisation algorithms have been proposed to help treat these problems.

5.2. Momentum. One common tweak to accelerate the slow convergence of SGD is to add momentum (Qian 1999 [20]):

$$\begin{aligned} \mathbf{g}_t &:= \nabla_\theta L(\theta_t; \mathcal{D}_m), \\ \mathbf{v}_{t+1} &= \beta \mathbf{v}_t + \eta \mathbf{g}_t \\ \theta_{t+1} &= \theta_t - \mathbf{v}_{t+1}, \end{aligned}$$

where $\beta \geq 0$ is the momentum term, and as before, $\eta > 0$ is the learning rate. When $\beta = 0$ then we recover plain SGD, but with $\beta > 0$ (a typical value is around 0.9), this gives the gradient a short term memory which often accelerates convergence.

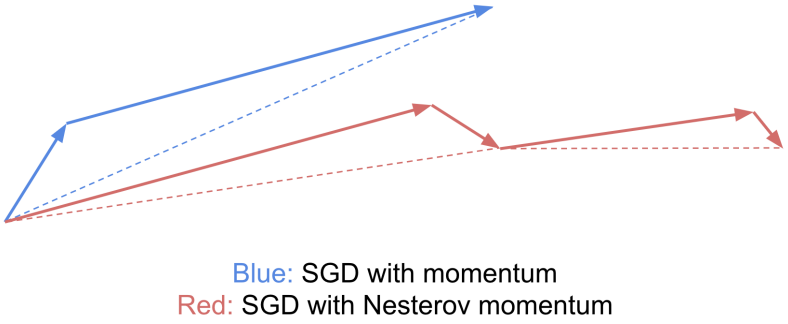


Figure 8.11. Nesterov momentum

5.3. Nesterov momentum. A common variant of momentum is to use Nesterov momentum (Nesterov 1983 [18]), which computes the gradient correction after the accumulated gradient, instead of before:

$$\begin{aligned}\mathbf{g}_t &= \nabla_\theta L(\theta_t - \beta \mathbf{v}_t; \mathcal{D}_m),\\ \mathbf{v}_{t+1} &= \beta \mathbf{v}_t + \eta \mathbf{g}_t\\ \theta_{t+1} &= \theta_t - \mathbf{v}_{t+1},\end{aligned}$$

The accumulated gradient approximates the next value of the parameters, and so by evaluating the gradient $\nabla_\theta \tilde{L}(\theta_t - \beta \mathbf{v}_t)$, this gives the optimiser a sense of 'look-ahead'.

5.4. Adagrad. The Adagrad optimiser (Duchi et al 2011 [4]) 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 [19] to train GloVe word embedding vectors.

The update rule is

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot \nabla_\theta L(\theta_t; \mathcal{D}_m),$$

where $G_t \in \mathbb{R}^{p \times p}$ 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 time step 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.

5.5. 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

$$\begin{aligned}\mathbb{E}[\mathbf{g}^2]_t &= \rho \mathbb{E}[\mathbf{g}^2]_{t-1} + (1-\rho)(\nabla_\theta L(\theta_t; \mathcal{D}_m))^2\\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[\mathbf{g}^2]_t + \varepsilon}} \odot \nabla_\theta L(\theta_t; \mathcal{D}_m)\end{aligned}$$

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.

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

5.6. Adam. The Adam optimiser (Kingma 2015 [13]) 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 $\varepsilon \approx 10^{-7}$.

6. 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. They are universal approximators (see e.g. Cybenko [3] for the large width case, or Lu et al [16] for the large depth case), and so overfitting can be a problem. When training neural networks, it is important to regularise them to prevent overfitting. As written above, there are several forms of regularisation, but in this section we will look at three in particular: weight regularisation, dropout and early stopping.

6.1. ℓ^2 and ℓ^1 regularisation. Recall that for a linear model of the form

$$f(\mathbf{x}) = \sum_j w_j \phi_j(\mathbf{x}),$$

a typical regularisation is to add a sum of squares penalty term to the loss term to discourage the weights w_j from growing too large. In this case, the regularised loss takes the form

$$L(\mathbf{w}, \alpha) = L_0(\mathbf{w}) + \alpha_2 \sum_i w_i^2,$$

where L_0 is the unconstrained loss function, and α_2 is a regularisation hyperparameter. This is ℓ^2 regularisation.

This form of regularisation is often referred to as **weight decay**, although the two terms are technically not the same. Weight decay (Hanson & Pratt [9]) is defined as a modification to the update rule, rather than to the loss function itself:

$$\theta_{t+1} \leftarrow (1-\lambda)\theta_t - \eta g_t,$$

where $\theta \in \mathbb{R}^p$ is the model parameters, λ, η are hyperparameters, and g_t is the t-th batch update. In the case of stochastic gradient descent, the update $g_t = \nabla_\theta L(\theta_t; \mathcal{D}_m)$ 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 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 encourages sparsity in the weights. Both ℓ^1 and ℓ^2 regularisation discourage the weights from growing too large, which restricts the capacity of the network.

It is also possible to add a weighted combination of both ℓ^2 and ℓ^1 regularisation to the loss function.

6.2. Dropout. Dropout was introduced by Srivastava et al [30] in 2014 as a regularisation technique for neural networks, that also has the effect of modifying the behaviour of neurons within a network.

The following is taken from the paper abstract:

> Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods.

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.

Suppose $\mathbf{W}^k \in \mathbb{R}^{n_{k+1} \times n_k}$ is a weight matrix mapping neurons in layer k to layer $k+1$:

$$\mathbf{h}^{(k+1)} = \sigma\left(\mathbf{W}^{(k)}\mathbf{h}^{(k)} + \mathbf{b}^{(k)}\right)$$

We can view dropout as randomly replacing each column of $\mathbf{W}^k$ with zeros with probability p_k. We can write this as applying a Bernoulli mask:

$$\begin{aligned} \mathbf{W}^k &\leftarrow \mathbf{W}^k \cdot \operatorname{diag}([\mathbf{z}_{k,j}]_{j=1}^{n_{k-1}}) \\ \mathbf{z}_{k,j} &\sim \text{Bernoulli}(p_k), \qquad k = 1, \ldots, L, \end{aligned}$$

with $\mathbf{W}^k \in \mathbb{R}^{n_{k+1} \times n_k}$. Figures 8.12 and 8.13 illustrate the effect of dropout on a neural network.

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 below (taken from the original paper [30]) 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

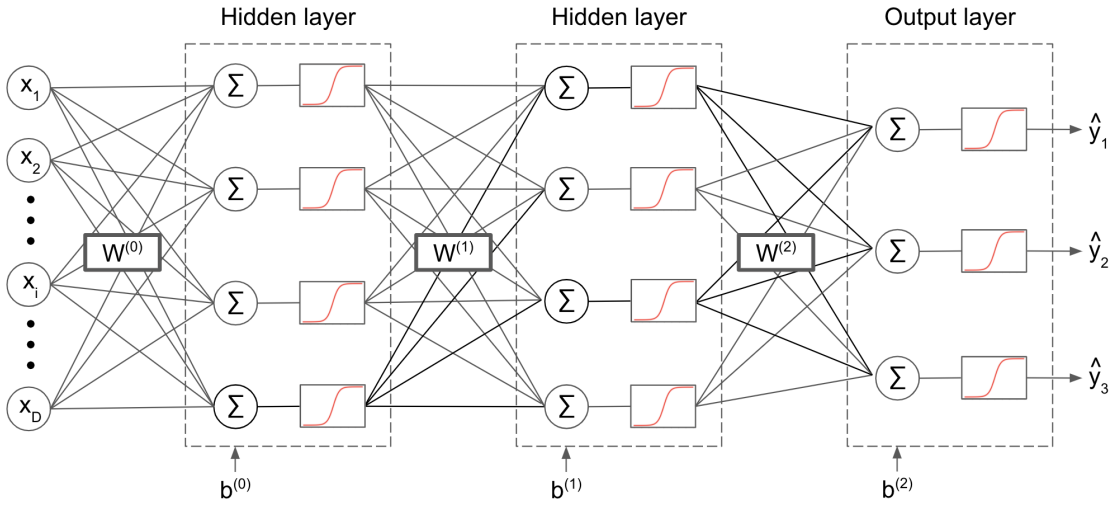


Figure 8.12. Neural network without dropout

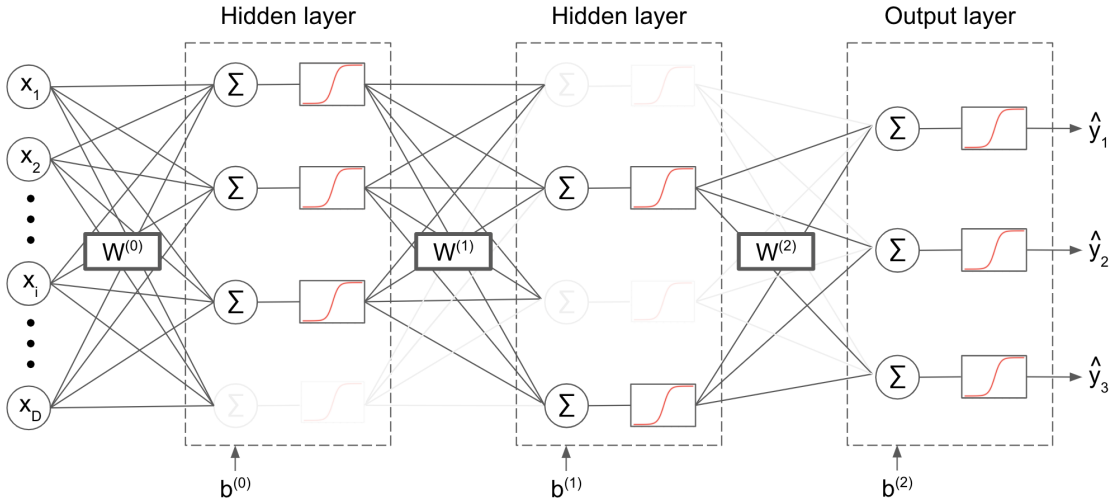


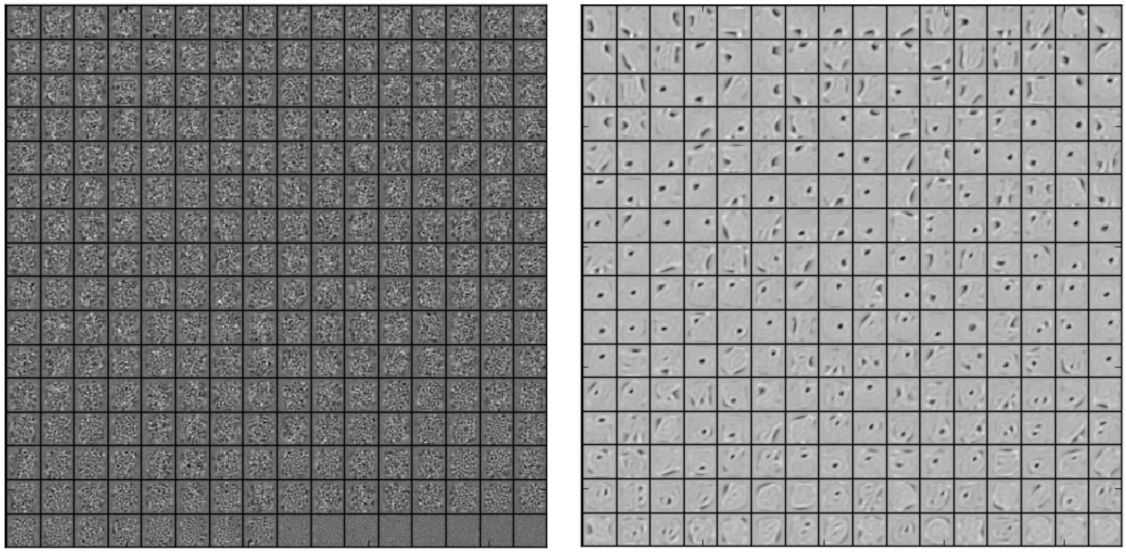
Figure 8.13. Neural network with dropout

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.

Typically, dropout is applied only in the training phase. When making predictions, all weight connections $\mathbf{W}^k$ are restored, but re-scaled by a factor of p_k to take account for the fact that fewer connections were present at training.

However, Gal & Ghahramani [7] describe a Bayesian interpretation of dropout, and proposed that dropout is also applied at test time in order to obtain a Bayesian predictive distribution.

6.3. Early stopping. You might have found in the tutorials that it is difficult to set a good number of epochs to train for ahead of time. In the simple MNIST example training is quick so it is not a problem to experiment, but in many cases training could take hours or days (or even longer!) and so this is not an option.



(a) Without dropout (b) Dropout with $p = 0.5$.

Figure 8.14. Learned features in a neural network trained without dropout (left) and with dropout (right). From Srivastava et al 2014

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 picture is similar to the balance between capacity and generalisation:

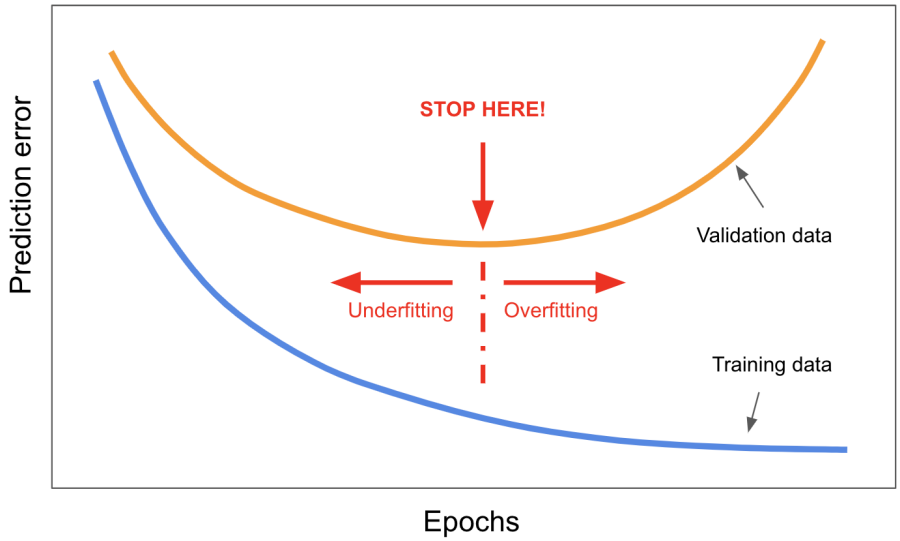


Figure 8.15. Prediction error vs number of training epochs

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.

The early stopping algorithm is outlined in pseudocode below. The inputs for this algorithms are: `val_metric`, `max_patience`

```
best_valid_loss = np.inf
patience = 0

for epoch in range(max_epochs):
    epoch_train_loss = train_model(train_data, train_loss)
    epoch_valid_loss = validate_model(valid_data, val_metric)
    if epoch_valid_loss < best_valid_loss:
        best_valid_loss = epoch_valid_loss
        patience = 0
    else:
        patience += 1

    save_model(epoch)

    if patience >= max_patience:
        break  # terminate training
```

It is also possible to validate the model using a measure that is different to the loss function used for training the model. Therefore `val_metric` is also an input to the early stopping algorithm above.

Of course, all of the regularisation techniques mentioned here (and more) can be used together in deep learning models (and they often are).

References for this chapter

[1] Chen, J. & Kyrillidis, A., (2019), "Decaying Momentum Helps Neural Network Training", arXiv preprint arXiv:1910.04952.

[2] 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.

[3] Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, **2** (4), 303–314.

[4] Duchi, J., Hazan, E., & Singer, Y. (2011), "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, **12**, 2121–2159.

[5] Dumoulin, V. & Visin, F. (2016), "A guide to convolution arithmetic for deep learning", arXiv preprint, abs/1603.07285.

[6] 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.

[7] Gal, Y. & Ghahramani, Z. (2016), "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning", Proceedings of The 33rd International Conference on Machine Learning, **48**, 1050-1059.

[8] Goodfellow, I., Bengio, Y., & Courville, A. (2016), "Deep learning", The MIT Press.

[9] 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.

[10] Hardt, M., Recht, B., & Singer, Y. (2015), "Train faster, generalize better: Stability of stochastic gradient descent", in *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, **48**, 1225-1234.

[11] 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.

[12] Keskar, N. S. & Socher, R. (2017), "Improving Generalization Performance by Switching from Adam to SGD", arXiv preprint, abs/1712.07628.

[13] Kingma, D. P. & Ba, J. L. (2015), "Adam: a Method for Stochastic Optimization", International Conference on Learning Representations, 1–13.

[14] Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017), "Self-Normalizing Neural Networks", *Neural Information Processing Systems (NIPS)*, 971-980.

[15] 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.

[16] 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.

[17] McCulloch, W. & Pitts, W. (1943), "A Logical Calculus of Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biophysics, **5**, 127-147.

[18] 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.

[19] Pennington, J., Socher, R. & Manning, C. D. (2014), "Glove: Global vectors for word representation", in *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*.

[20] 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.

[21] Ramachandran, P., Zoph, B. & Le, Q. V. (2018) "Searching for Activation Functions", arXiv preprint, abs/1710.05941.

[22] Robbins, H. and Monro, S. (1951), "A stochastic approximation method", *The annals of mathematical statistics*, 400–407.

[23] Rosenblatt, F. (1958), "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain", Psychological Review, 65-386.

[24] Rosenblatt, F. (1961), "Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms", Defense Technical Information Center.

[25] Rumelhart, D. E., McClelland, J. L. and the PDP Research Group (1986a), "Parallel Distributed Processing: Explorations in the Microstructure of Cognition", MIT Press, Cambridge.

[26] Rumelhart, D. E., Hinton, G., & Williams, R. (1986b), "Learning representations by back-propagating errors", Nature, **323**, 533-536.

[27] 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.

[28] Simonyan, K. & Zisserman, A. (2015), "Very Deep Convolutional Networks for Large-Scale Image Recognition", in *3rd International Conference on Learning Representations, (ICLR) 2015*, San Diego, CA, USA.

[29] Smith, L. N. (2015), "Cyclical Learning Rates for Training Neural Networks", arXiv preprint, abs/1506.01186.

[30] 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.

[31] Werbos, P. J. (1994), "The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting", New York:, John Wiley & Sons.

[32] 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

Chapter 9

Unsupervised learning. The clustering problem: k-means and hierarchical clustering

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 one (for simplicity, WLOG) output variable $y^{(i)}$:

$$\{\mathbf{x}^{(i)}; y^{(i)}\}_{i=1}^{N}, \qquad \text{where} \quad \mathbf{x}^{(i)} \in \mathbb{R}^p \quad \text{or} \quad \mathbf{x}^{(i)} \in \{c_1, \dots, c_k\}^p \tag{9.1}$$

$$\text{and} \begin{cases} y^{(i)} \in \mathbb{R} & \text{(regression)} \\ \qquad \text{or} & \\ y^{(i)} \in \{c_1, \dots, c_k\} & \text{(classification)} \end{cases}$$

In all the supervised methods, the set $\{\mathbf{x}^{(i)}; y^{(i)}\}_{i=1}^{N}$ is used as a training/validation set from which we learn the parameters (and hyper-parameters) $\boldsymbol{\theta}$ of the model $f(\mathbf{x}; \boldsymbol{\theta})$.

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):

$$\text{Given} \quad \mathbf{x}^{\text{in}}, \quad \text{predict} \quad \widehat{y} = \begin{cases} f(\mathbf{x}^{\text{in}}; \boldsymbol{\theta}) \in \mathbb{R} & \text{(regression)} \\ \qquad \text{or} & \\ f(\mathbf{x}^{\text{in}}, \boldsymbol{\theta}) \in \{c_1, \dots, c_k\} & \text{(classification)} \end{cases}$$

where the (hyper)parameters of the model $\boldsymbol{\theta}$ have been learnt from the training set (9.1), which we can view as our set of given examples with known outputs.

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 of course a data set 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 and observables. Our data set is now viewed only as a collection of (continuous or discrete) samples:

$$\{\mathbf{x}^{(i)}\}_{i=1}^{N}, \qquad \mathbf{x}^{(i)} \in \mathbb{R}^p \quad \text{or} \quad \mathbf{x}^{(i)} \in \{c_1, \ldots, c_k\}^p. \tag{9.2}$$

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 introduction indicates 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 data set in terms of the obtained groups (or 'clusters') in a concise manner. The clusters found can also drive the interpretation and discovery science from the data set through the investigation of the meaning of such groups of samples present in the data set.
- **Dimensionality reduction:** find a 'good' approximate representation of the original data set (9.2) in terms of lower dimensional variables $\mathbf{x}^{(i)} \in \mathbb{R}^k$ where $k \ll p$. 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 focusing on those variables that capture the properties of the data set.

In this chapter, we first introduce and define the pre-requisites common to all clustering algorithms, before showcasing these approaches with two of the most popular clustering methods, namely k-means clustering and hierarchical clustring.

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 data set $\mathbf{x} \in \mathbb{R}^p$, typical examples of *dissimilarity* include:

- the 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:
$$S(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \frac{\mathrm{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})}{\sqrt{\mathrm{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(i)}) \cdot \mathrm{cov}(\mathbf{x}^{(j)}, \mathbf{x}^{(j)})}}$$

Categorical or discrete data: If the variables in the data set are categorical, $\mathbf{x}^{(i)} \in \{c_1, \ldots, c_k\}^p$, then we can utilise a distance matrix based on pre-defined 'costs'. For example, given classes $\{c_1, \cdots, c_k\}$, we assign values to $D_{1,2}, D_{1,3}, \ldots$, which stand for the 'distances' between c_1 and c_2, c_1 and c_3, and so on, respectively. This can be summarised in a $k \times k$ matrix. A particular example of this is the *Hamming distance*, which is applied when there are only two classes $\mathbf{x}^{(i)} \in \{c_1, c_2\}^p =: \{0, 1\}^p$.

Ordinal data: If the data set is formed by ordinal variables that can be ranked, i.e. there exists some natural order, then we can transform this into 'pseudo-continuous' variables, e.g. given 4 classes ($k = 4$), we can assign $i = 1, 2, 3, 4$ to each class according to the ranking and transform this according to, e.g.,
$$j = \frac{i - \frac{1}{2}}{k}$$
This will yield values between 0 and 1, which can then be treated as a continuous numerical variable.

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 what clustering entails. Briefly, the clustering problem can be stated as:

> Given N samples $\left\{\mathbf{x}^{(i)}\right\}_{i=1}^{N}$ and a dissimilarity ('distance') $D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, 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 data set 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. If the samples are continuous real variables, and given some distance measure in p-dimensional space, we can think of the clustering problem as grouping points that are close to each other in $\mathbb{R}^p$. In other words, we are aiming to find groups in the data (if they exist) such that we minimise the total

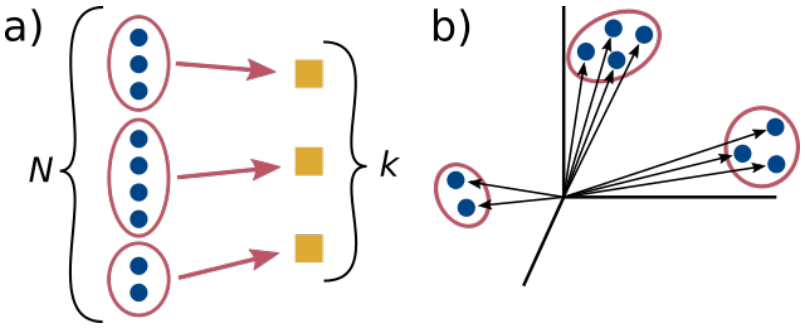


Figure 9.1. a) The clustering problem: Grouping N samples (blue dots) into k clusters (yellow boxes). **b)** If $\mathbf{x}^{(i)} \in \mathbb{R}^3$ (i.e. 3 variables), we can visualise the data as points in 3D space, which will be closer to each other within a cluster (red ellipses) as opposed to across clusters.

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).

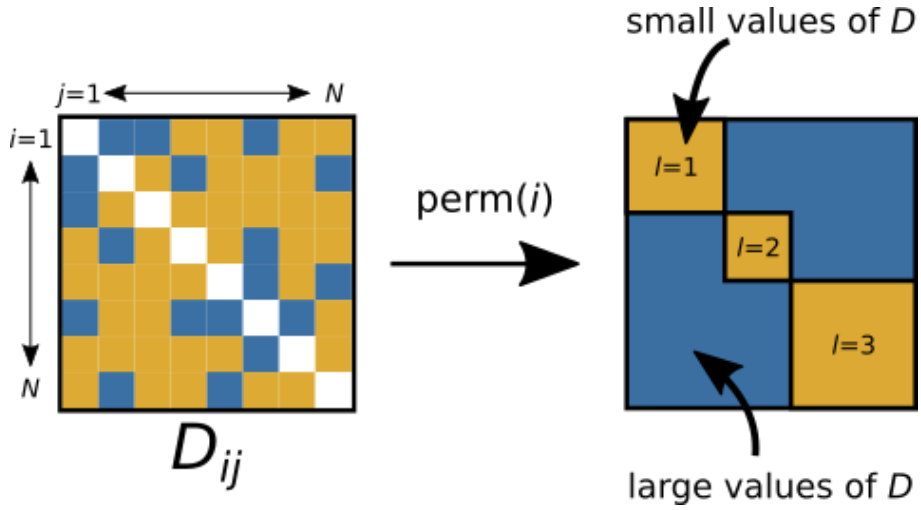


Figure 9.2. The clustering problem re-formulated as a matrix permutation problem. Starting from a distance matrix D, we find a permutation of the rows and columns, such that the resulting permuted matrix contains blocks of small values of D (in yellow) on the diagonal, whilst the off-diagonal sections 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 to 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 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. For a visual illustration, 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 distance W:

$$W(\mathcal{C}) = \frac{1}{2} \sum_{l=1}^{k} \sum_{i,j \in c_l} D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{9.3}$$

while maximimising the between distance B:

$$B(\mathcal{C}) = \underbrace{\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\left[D(\mathbf{x}^{(i)},\mathbf{x}^{(j)})\right]}_{\text{Total dissimilarity}} - W(\mathcal{C}) \tag{9.4}$$

Effectively, W is a sum of distances over the blocks on the diagonal.

This viewpoint shows that clustering can be seen as a combinatorial optimisation problem, i.e., the problem relies on finding some permutation of our samples into groups. As we have already discussed, 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. In many cases of importance, the only way of ensuring a globally optimal solution is to try all possible arrangements. Unfortunately, the number of clusterings that one has to check, explodes. For N samples and k clusters, the number of clusterings is given by:

$$S(N,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}(k-i)^N$$

This formula is known as the *Stirling numbers of the second kind*[1]. To give you some idea of how fast these numbers rise:

$$\begin{aligned} S(10,4) &= 34105 \\ S(19,4) &= 10^{10} \\ &\vdots \end{aligned}$$

This leads to what is called *combinatorial explosion*.

As can be seen, a complete enumeration of even small problems is unfeasible. Therefore, we need to come up with heuristics to optimise the problem. In many cases, one solves relaxations of the problem, or adopts an optimisation strategy that delivers 'optimised' solutions with no guarantees of global optimality. Another complication in some cases is the lack of a 'gradient' that can guide the optimisation in numerical algorithms.

4. The k-means algorithm for clustering

Now that we have stated the basic formulation of the clustering problem and established the pre-requisites of clustering algorithms, we introduce one of the most popular clustering algorithms in unsupervised learning: k-means.

In the basic k-means algorithm, we want to find a hard assignment (i.e., unique membership) between the N samples and k clusters. Firstly, we have to decide on a specific value of k. Our target then becomes to find the $N \times k$ *assignment matrix* which takes 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. Naturally, each row contains exactly one entry of 1, assigning that particular data

[1] https://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html

point to a particular cluster. Elsewhere, the row is simply filled with zeros. The outcome of the k-means algorithm is the assignment matrix H (or an equivalent computational representation).

Aside: The H matrix given above summarises a hard assignment to a clustering $\mathcal{C} = \{c_l\}_{l=1}^k$ 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_l for some $l = 1, \ldots, k$):

$$c_l \cap c_{l'} = \emptyset \qquad l \neq l'$$

$$\bigcup_{l=1}^{k} c_l = \{\mathbf{x}^{(i)}\}_{i=1}^N$$

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., $\mathbf{1}_N := \mathbf{1}_{N\times 1}$

Note that there are extensions of k-means that allow for orphan points as well as multi-class, probabilistic ('soft') assignments.

We now towards describing the underlying k means algorithm in more detail. As explained above, we start with the data points $\mathbf{x}^{(i)} \in \mathbb{R}^p$ and a distance measure (Euclidean in this case) $D(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2$. For a given clustering $\mathcal{C} = \{c_l\}_{l=1}^k$, we have an associated assignment matrix H. We then have the within distance:

$$W(\mathcal{C}) = \frac{1}{2}\sum_{l=1}^{k}\sum_{i,j\in c_l} \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \tag{9.5}$$

However, it is important to pay attention to unbalanced data sets, i.e. where the sizes of the subsets are expected to be quite different. Then one 'normalises' the above equation as follows (where $|c_l|$ is the cardinality of c_l):

$$W(\mathcal{C}) = \frac{1}{2}\sum_{l=1}^{k}\frac{1}{|c_l|}\sum_{i,j\in c_l} \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \tag{9.6}$$

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:

$$W(\mathcal{C}) = \frac{1}{2}\mathrm{Tr}\left[(H^TH)^{-1}\left[H_{k\times N}^T D_{N\times N} H_{N\times k}\right]\right] \tag{9.7}$$

Aside: In order to understand (9.7) a little better, we take each element of the equation above and explicitly write out the corresponding matrices. Due to the nature of H, we will see some useful properties of the constituent parts of the equation above. The insight is that the operation $H^T\,D\,H$ can be understood as a effecting a sum over the blocks implicitly defined by H.

Take $k = 3$, then we have:

$$H_{k\times N}^T D_{N\times N} H_{N\times k} = \begin{bmatrix} \clubsuit_1 & \square & \square \\ \square & \clubsuit_2 & \square \\ \square & \square & \clubsuit_3 \end{bmatrix},$$

where each $\clubsuit_l$ corresponds to the total 'within-cluster' distances for cluster c_l (which we are aiming to minimise):

$$\clubsuit_l = \sum_{i,j \in c_l} \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2$$

Furthermore, we have:

$$H^T H = \begin{bmatrix} \spadesuit_1 & 0 & 0 \\ 0 & \spadesuit_2 & 0 \\ 0 & 0 & \spadesuit_3 \end{bmatrix},$$

where each $\spadesuit_l$ corresponds exactly to the cardinality of each cluster c_l:

$$\spadesuit_l = |c_l|$$

Now we can see that (9.7) is indeed equivalent to (9.6).

Remember that the objective was 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 data set, 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 dissimilarity (9.4).

As discussed above, clustering algorithms proceed by optimisation heuristics. The heuristic 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:

$$\mathbf{m}_l = \frac{1}{|c_l|}\sum_{i \in c_l} \mathbf{x}^{(i)}, \quad l = 1, \ldots, k$$

- **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 $l_i^{(t)}$, and evaluate the following:

$$l_i^{(t+1)} = \underset{l}{\operatorname{argmin}} \|\mathbf{x}^{(i)} - \mathbf{m}_l\|^2$$

- Iterate steps 1 and 2 until convergence, i.e. $W(\mathcal{C})$ does not improve much (see below) or the assignments do not change.

Quick aside: The k-means optimisation algorithm makes sense.

Given a clustering $\mathcal{C} = \{c_l\}_{l=1}^k$, we write down the objective function:

$$W(\mathcal{C}) = \frac{1}{2}\sum_{l=1}^{k} \frac{1}{|c_l|} \sum_{i,j \in c_l} \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2,$$

which can be rewritten as:

$$W = \frac{1}{2}\sum_{l=1}^{k}\frac{1}{|c_l|}\sum_{i,j\in c_l}\|(\mathbf{x}^{(i)}-\mathbf{m}_l)-(\mathbf{x}^{(j)}-\mathbf{m}_l)\|^2$$

$$= \sum_{l=1}^{k}\frac{1}{2}\frac{1}{|c_l|}\sum_{i,j\in c_l}\left[\|\mathbf{x}^{(i)}-\mathbf{m}_l\|^2+\|\mathbf{x}^{(j)}-\mathbf{m}_l\|^2-2(\mathbf{x}^{(i)}-\mathbf{m}_l)(\mathbf{x}^{(j)}-\mathbf{m}_l)\right]$$

$$= \sum_{l=1}^{k}\sum_{i\in c_l}\left[\|\mathbf{x}^{(i)}-\mathbf{m}_l\|^2-\underbrace{(\mathbf{x}^{(i)}-\mathbf{m}_l)\frac{1}{|c_l|}\sum_{j\in c_l}(\mathbf{x}^{(i)}-\mathbf{m}_l)}_{=0}\right]$$

In summary:

$$W = \frac{1}{2}\sum_{l=1}^{k}\frac{1}{|c_l|}\sum_{i,j\in c_l}\|\mathbf{x}^{(i)}-\mathbf{x}^{(j)}\|^2 = \sum_{l=1}\sum_{i\in c_l}\|\mathbf{x}^{(i)}-\mathbf{m}_l\|^2$$

So, trying to minimise the distance to the centroids is indeed our objective.

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 of course 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 (at least), depending on the random initialisation (random assignment in Step 0). 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 data set. In many cases, one will then choose the optimal clustering (according to the cost function W) from this ensemble of optimised partitions obtained from the runs of k-means.

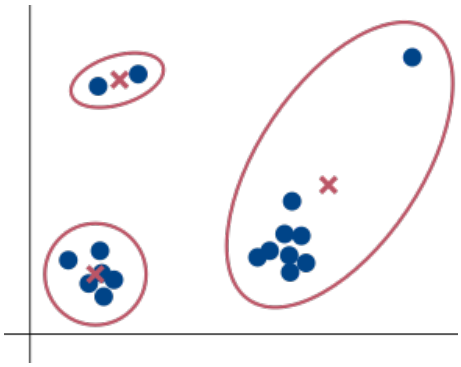


Figure 9.3. Cartoon depiction of the possible effect of outliers on k-means clustering. Note that due to an outlier being included in the right cluster, the centroid of this cluster is not representative of the data.

A couple of additional comments on k-means:

- Outliers can pose problems for the k-means algorithm as can be seen from Figure 9.3. 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.

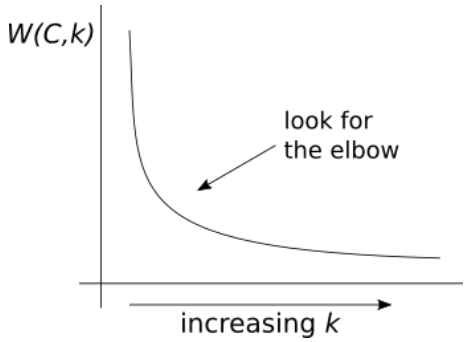


Figure 9.4. Optimisation of k: As k is increased, $W(\mathcal{C}, k)$ will decrease. The objective is to look for the 'elbow' of this curve, where increasing k further brings no significant further improvement.

- Secondly, and crucially, the question of choosing k remains. 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, this is sometime 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 data set, we want to find the smallest k that produces a small W. An illustration of this can be seen in Figure 9.4, 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

We now introduce *hierarchical* clustering, another popular clustering method.

Hierarchical clustering is intuitive and leads to nice visualisations, but has several shortcomings which will become apparent below. It is used extensively in fields such as bioinformatics, etc, as it produces a 'picture' 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 data set without having to choose *a priori* a number of clusters. Hence it is used broadly for data exploration.

We start with the usual description of our data set: $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$, and we also have some distance/similarity measure as defined above which produces our $N \times N$ matrix of distances between all points : $D_{ij} = d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, $i, j = 1, \ldots N$.

In hierarchical clustering analysis, the whole data set is eventually represented as a binary tree, which is usually called a *dendrogram*, where the N leaves at the bottom are samples and the root at the top is the whole data set. The binary tree represents how samples get agglomerated into larger groupings as we go up the dendrogram. Each bifurcation of the tree corresponds to a merger of two groups of samples.

A basic sketch of this idea can be seen in Figure 9.5. Note that a strict hierarchical structure of groupings is imposed on the data as a result of the requirement of binary splits. In other words, each binary split happens at a certain level of depth, leading to a

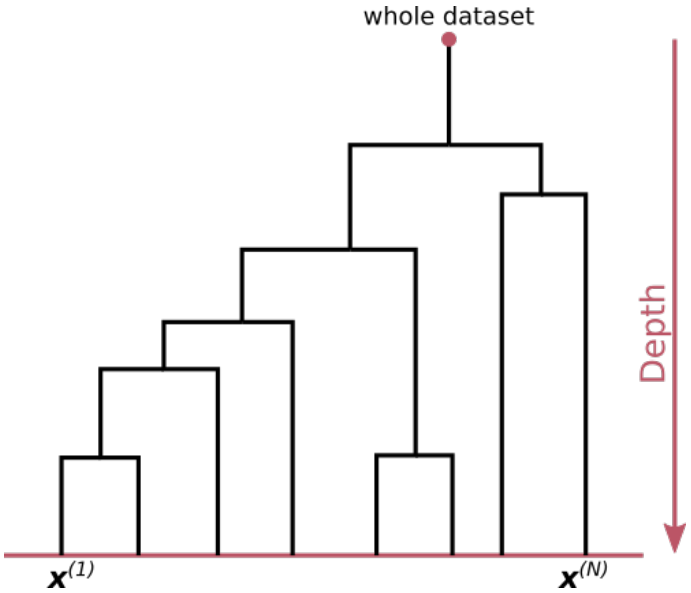


Figure 9.5. A typical dendogram, i.e., the binary tree summarising the groupings of the data samples obtained in hierarchical clustering as a function of the depth. Starting at the top, we have the whole data set. Moving down the tree, splits are made at various stages until finally, at the maximum depth, all individual data points are represented by a leaf of the tree each. Conversely, starting from the N leaves at the bottom that correspond to the N data points, we merge them into groups of increasing coarseness as go up towards the root.

strict hierarchy (i.e. no cross-over of data points into other clusters as depth progresses). That is, data points that have been separated into different two clusters will always remain split as depth progresses.

The tree is generated through a property that will keep a monotonic connection between the depth and the dissimilarity of samples, i.e. as depth *increases*, the total dissimilarity in the groups *decreases* (or similarity increases).

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 data set. 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 merger. 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 merger 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 and then looking for the minimal distance across all pairs of clusters present at a level of the dendrogram.

Group average (GA):. The optimisation function in this case is:

$$d_{\mathrm{GA}}(G,H) = \frac{1}{N_G N_H} \sum_{i \in G, j \in H} D_{ij}$$

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 merger.

Ward's criterion: Ward introduced another agglomerative method based on a general criterion, which has found broad use.

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.

An illustration of the dendrograms that can be reconstructed through these different criteria from real data is provided in *Hastie, Tibshirani, Friedman, The Elements of Statistical Learning*, Chap. 14, section 14.3.12.

Aside: A couple of comments about agglomerative schemes:

- The strictly hierarchical nature of the clusterings produced makes agglomerative methods prone to variability under noise, i.e., small amounts of noise in the data can lead to very different dendrograms.
- If just one partition is desirable, 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 mergers (or splits), i.e., the amount by which the distance function has increased in the merger. Usually, stable levels of the dendrogram are picked, signalled by large jumps in the depth of the dendrogram between levels.
- Note that better schemes do exist, but in general agglomerative algorithms are used extensively to get a first exploratory view of the structure of data sets and to produce visualisations.

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 $k = 2$. This will also yield a binary tree, *but*: (1) monotonicity is not preserved; and (2) it depends on the initialisation at every split. This can lead to quite variable results.

6. Other clustering methods:

Important remark: One of the most important types of clustering algorithms is *spectral clustering*. Although thematically spectral clustering belongs in this chapter about clustering, it is better understood once we introduce spectral methods in the next chapter. Hence we will cover this clustering later in the course and show that is indeed related to both spectral dimensionality reduction (like Principal Component Analysis, see section 1) and graph clusterings (see section 4).

7. Comparing clusterings

A brief note on the comparison of clusterings is in order.

In many cases, one is asked to compare how similar two clusterings are. This topic is related to our confusion and contingency matrices considered early on in classification, but here we do *not* have a ground truth against which we are comparing, and furthermore the clusterings can have different numbers of groups.

This is an area of open research but there are several well used measures and we will mention a few. More measures can be found (and should be investigated if necessary) in online resources.

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.

Two typical examples are:

7.1. Adjusted Rand index[2]. Given a set S of n elements, and two groupings or partitions (e.g. clusterings) of these elements, namely $X = \{X_1, X_2, \ldots, X_r\}$ and $Y = \{Y_1, Y_2, \ldots, Y_s\}$, 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_s	sums
X_1	n_{11}	n_{12}	$\cdots$	n_{1s}	a_1
X_2	n_{21}	n_{22}	$\cdots$	n_{2s}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
X_r	n_{r1}	n_{r2}	$\cdots$	n_{rs}	a_r
sums	b_1	b_2	$\cdots$	b_s	

The ARI using the Permutation Model is then defined by:

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2}\sum_j \binom{b_j}{2}\right]\Big/\binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2}\sum_j \binom{b_j}{2}\right]\Big/\binom{n}{2}}$$

7.2. Adjusted Mutual Information[3]. Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score 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.[4]

Given two clusterings U and V, the MI is defined as:

$$MI(U,V) = \sum_{i=1}^{R}\sum_{j=1}^{C} P_{UV}(i,j)\log\frac{P_{UV}(i,j)}{P_U(i)P_V(j)}$$

where $P_{UV}(i,j)$ denotes the probability that a point belongs to both the cluster U_i in U and cluster V_j in V:

$$P_{UV}(i,j) = \frac{|U_i \cap V_i|}{N}$$

[2] https://en.wikipedia.org/wiki/Rand_index#Adjusted_Rand_index

[3] https://en.wikipedia.org/wiki/Adjusted_mutual_information

[4] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_mutual_info_score.html

An adjustment for chance of this score then results in the AMI score:

$$AMI(U,V) = \frac{MI(U,V) - E\{MI(U,V)\}}{\max\{H(U), H(V)\} - E\{MI(U,V)\}}$$

Here, we have the following definitions:

$$H(U) = -\sum_{i=1}^{R} P_U(i) \log P_U(i)$$

$$E\{MI(U,V)\} = \sum_{i=1}^{R}\sum_{j=1}^{C} \sum_{n_{ij}=(a_i+b_j-N)^+}^{\min(a_i,b_j)} \frac{n_{ij}}{N} \log\left(\frac{N \cdot n_{ij}}{a_i b_j}\right) \times \frac{a_i! b_j! (N-a_i)! (N-b_j)!}{N! n_{ij}! (a_i - n_{ij})! (b_j - n_{ij})! (N - a_i - b_j + n_{ij})!}$$

Aside: Other notions of similarity based on mutual information are: the normalised variation of information and the normalised mutual information. Both are also broadly used.

Chapter 10

Probabilistic clustering: Mixtures and Hidden Markov Models

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, so called latent variable model, is applied. In brief, one considers a model

$$P(\boldsymbol{X}, Z), \tag{10.1}$$

where $\boldsymbol{X}$ are the observed continuous variables and Z are categorical latent variables corresponding the clusters. According to this model we can estimate the probability of a data point $\boldsymbol{x}_i$ belonging to the k-th cluster via

$$r_{ik} = P(Z = k|\boldsymbol{X} = \boldsymbol{x}_i). \tag{10.2}$$

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 section. Of course, any soft clustering can be converted to a hard one using

$$\operatorname*{argmax}_k P(Z = k|\boldsymbol{X} = \boldsymbol{x}_i). \tag{10.3}$$

From a probabilistic perspective (10.2) can be viewed as a posterior probability and (10.3) as a maximum posterior estimator. An advantage of probabilistic clusterings is that they involve a *generative model*, i.e., a model that can generate more data for each clusters.

1. Gaussian mixture models

Mixture models are generative models commonly used for clustering. The generative model consists of mixture components $P(\boldsymbol{X} = \boldsymbol{x}|Z = k) = p_k(\boldsymbol{x}|\boldsymbol{\theta})$ and mixture weights $P(Z = k) = \pi_k$ such that the marginal distribution of the data is a convex combination

$$P(\boldsymbol{X} = \boldsymbol{x}) = \sum_{k=1}^{K} \pi_k p_k(\boldsymbol{x}|\boldsymbol{\theta}). \tag{10.4}$$

The model involves parameters $\boldsymbol{\theta}$ and weights $\boldsymbol{w}$ that we need to choose. The cluster probabilities can then be computed as

$$\begin{aligned} r_{ik}(\boldsymbol{\theta}) &= P(Z = k|\boldsymbol{X} = \boldsymbol{x}_i, \boldsymbol{\theta}) \\ &= \frac{P(Z = k, \boldsymbol{X} = \boldsymbol{x}_i, \boldsymbol{\theta})}{P(\boldsymbol{X} = \boldsymbol{x}_i, \boldsymbol{\theta})} = \frac{\pi_k p_k(\boldsymbol{x}_i|\boldsymbol{\theta})}{\sum_{k'=1}^{K} \pi_{k'} p_{k'}(\boldsymbol{x}_i|\boldsymbol{\theta})}. \end{aligned} \tag{10.5}$$

The most popular mixture models are Gaussian Mixture Models for which

$$p_k(\boldsymbol{x}|\boldsymbol{\theta}) = (2\pi)^{-k/2} \det(\boldsymbol{\Sigma}_k)^{-1/2} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^{\mathsf{T}} {\boldsymbol{\Sigma}_k}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)\right), \tag{10.6}$$

In unsupervised learning we do not have access labels and therefore it is desired to all parameters $\boldsymbol{\theta} = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1,..,K}$ optimally such that the model approximates the data as close as possible. A common choice is to maximise the log-likelihood functions such that the generative closely matches the data, which will be discussed in the next section.

1.1. Expectation-Maximization algorithm (EM). In principle, the maximum likelihood estimate of the mixture is given by

$$\begin{aligned} \boldsymbol{\theta}_{\text{MLE}} = \operatorname*{argmax}_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) &= \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_i \log P(\boldsymbol{X} = \boldsymbol{x}_i|\boldsymbol{\theta}) \\ &= \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_i \log \sum_{k=1}^{K} \pi_k p_k(\boldsymbol{x}_i|\boldsymbol{\theta}), \end{aligned} \tag{10.7}$$

which in principle can be optimised using gradient descent. The theoretical difficulty is that the likelihood involves a sum over the states of the latent variables Z. EM exploits the fact that if both $\boldsymbol{X}$ and Z were fully observed, then the maximum likelihood estimate would be easy to compute. EM is an iterative algorithm which can be motivated as follows:

Assume we had access to the full data $(\boldsymbol{x}_i, Z_i)$, then "true" parameters of the mixture model are given by:

(10.8)

$$\pi_k(\boldsymbol{Z}) = \frac{1}{N}\sum_{i=1}^{N} \delta_{Z_i,k} \quad \boldsymbol{\mu}_k(\boldsymbol{Z}) = \frac{1}{N_k}\sum_{i=1}^{N} \delta_{Z_i,k}\boldsymbol{x}_i \quad \boldsymbol{\Sigma}_k(\boldsymbol{Z}) = \frac{1}{N_k}\sum_{i=1}^{N} \delta_{Z_i,k}(\boldsymbol{x}_i - \boldsymbol{\mu}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)^T$$

However, in practice, the clustering $\boldsymbol{Z}$ 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 average whose weights corresponding to the cluster probabilities.

Expectation step. If we can make a guess of the parameters $\boldsymbol{\theta}^{(n)} = \{\pi_k^{(n)}, \boldsymbol{\mu}^{(n)}, \boldsymbol{\Sigma}^{(n)}\}_{k=1,..,K}$, no matter how poor, we can use these to compute cluster probabilities $r_{ik}(\boldsymbol{\theta}^{(n)})$ and a set of weight vectors

$$w_{ik}(\boldsymbol{\theta}^{(n)}) = \frac{r_{ik}(\boldsymbol{\theta}^{(n)})}{\sum_{i'} r_{i'k}(\boldsymbol{\theta}^{(n)})}$$

. This is called the E-step.

Maximisation step. We can use these quantities to improve on our parameters by taking the expectation over these probabilities:

$$\pi_k^{(n+1)} = E_{\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{\theta}^{(n)}}[\pi_k(\boldsymbol{Z})] = \frac{1}{N}\sum_{i=1}^{N} E_{Z_i|\boldsymbol{X}_i,\boldsymbol{\theta}^{(n)}}[\delta_{Z_i,k}] = \frac{1}{N}\sum_{i=1}^{N} r_{ik}(\boldsymbol{\theta}^{(n)}). \tag{10.9}$$

To compute the mixture components, we use the weights w to obtain

$$\boldsymbol{\mu}_k^{(n+1)} = \sum_{i=1}^{N} w_{ik}(\boldsymbol{\theta}^{(n)})\boldsymbol{x}_i \tag{10.10}$$

$$\boldsymbol{\Sigma}_k^{(n+1)} = \sum_{i=1}^{N} w_{ik}(\boldsymbol{\theta}^{(n)})(\boldsymbol{x}_i - \boldsymbol{\mu}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)^T, \tag{10.11}$$

which are simply a weighted version of (10.8). This is called the M-step.

Iterating between E and M step, we obtain a sequence of paramters $(\boldsymbol{\theta}^{(n)})_{n=1,2,\ldots}$. The theoretical justification of this approach is that this sequence yields an increasing sequence marginal likelihoods

$$\ell(\boldsymbol{\theta}^{(0)}) \leq \ell(\boldsymbol{\theta}^{(1)}) \leq \ldots \leq \ell(\boldsymbol{\theta}^{(n)}) = \sum_i \log P(\boldsymbol{X} = \boldsymbol{x}_i|\boldsymbol{\theta}^{(n)}), \tag{10.12}$$

but we will not go into the details of the proof here.

1.2. MLE is not a convex optimisation problem. A caveat of the EM algorithm (and similarly gradient descent) is that it only guarantees convergence to a local minimum. Multiple minima almost always appear in applications. A common mitigation strategy is to start the algorithm multiple time from different initial guesses allows one to get closer to the global optimum.

1.3. Relation to k-means. You may have wondered whether about the relation between the EM-algorithm for Gaussian mixture models and k-means, which are in fact related. There are two important distinctions: (i) k-means algorithm uses equal centroids while the Gaussian mixture model uses different ellipsoids to model each cluster; (ii) k-means is a hard clustering while mixture model give a soft clustering.

Assume that $\boldsymbol{\Sigma} = \text{diag}(1)$ and $\pi_k = 1/K$ are fixed, so only the parameter to estimate is the cluster mean $\boldsymbol{\mu}_k$. We then enforce a hard clustering through taking the argmax of (10.5) via

$$r_{ik}^{\text{kMeans}}(\boldsymbol{\theta}) = \begin{cases} 1 & \text{if } k = \text{argmax}_j\, r_{ik}(\boldsymbol{\theta}) = \text{argmin}_j\, ||\boldsymbol{x}_i - \boldsymbol{\mu}_j||^2 \\ 0 & \text{otherwise} \end{cases} \tag{10.13}$$

reduces the EM iteration of the mean (10.10) to the k-means algorithm. This shows how EM-algorithm for Gaussian mixture models generalises k-means.

2. Clustering of sequential data: Hidden Markov Models

So far, we have dealt with models that, at least implicitly, assume that the order of data did not matter. In many instances, data may be involve sequences that can be modelled using Hidden Markov Models (HMMs). Common examples are time-series, speech recognition, or amino acid sequences. There are several clustering problems associated with temporal data depending on which information

Filtering means to compute the belief state $P(Z_t|\boldsymbol{X}_{1:t})$ recursively. This is called "filtering" because it reduces the uncertainty of estimating the hidden state by taking into account the present and past observations. In the context of HMMs, this is accomplished applying Bayes rule recursely, a procedure that is called the forward algorithm.

Smoothing computes $P(Z_t|\boldsymbol{X}_{1:T})$, given all the evidence: past, present and future data. This typically leads to a "smoother" sequence than using filtering since also future data

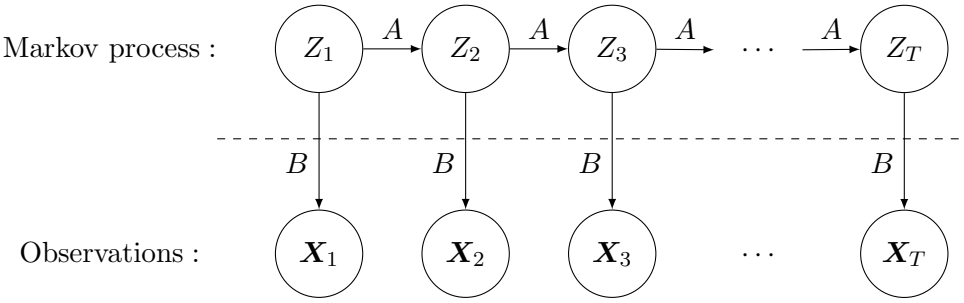
can be used. In the context of HMMs, this is accomplished using the forward-backward algorithm.

Maximum a posteriori decoding typically looks for the most probable state sequence over the posterior distribution of sequences given all the evidence. This means computing $\text{argmax}_{\boldsymbol{Z}_{1:T}} P(\boldsymbol{Z}_{1:T}|\boldsymbol{X}_{1:T})$. In the context of HMMs, the task is accomplished using the Viterbi algorithm.

These methods consider a distribution of clusters that is often referred to as the Bayesian posterior distribution. Most commonly, hyperparameters are optimised using maximum marginal likelihood, as we will see, leading to a point estimate of the parameters. They are thus not fully Bayesian in character.

The presentation of this chapter follows mostly the book of Kevin Murphy *Machine Learning: A Probabilistic Perspective* (2012).

2.1. Hidden Markov Models. HMMs are models of sequential data that is generated from an underlying discrete sequence. you will typically encounter a graphical representation like this one:



This is an example of a probabilistic graphical models. Here the process $\boldsymbol{Z}_{1:T} = (Z_1, Z_2, \ldots, Z_T)$ is assumed to take discrete values in $k = 1, 2, \ldots, K$. The assumption that $\boldsymbol{Z}_{1:T}$ is Markov chain allows us to write the probability of a path

$$P(\boldsymbol{Z}_{1:T}) = P(\boldsymbol{Z}_1)P(\boldsymbol{Z}_2|\boldsymbol{Z}_1)\ldots P(\boldsymbol{Z}_T|\boldsymbol{Z}_{T-1}) = \underbrace{P(Z_1)}_{\pi_{Z_1}} \prod_{t=2}^{T} \underbrace{P(Z_t|Z_{t-1})}_{A^T_{Z_t,Z_{t-1}}}, \tag{10.14}$$

where π is the initial distribution of the chain and A is the transition matrix, which is independent of time t. We then assume that, similarly to a mixture model, for each time-point the process is observed through a multivariate normal random variables $\boldsymbol{X}_{1:T}$ satisfying

$$P(\boldsymbol{X}_t = \boldsymbol{x}|Z_t = k) = (2\pi)^{-k/2} \det(\boldsymbol{\Sigma}_k)^{-1/2} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\intercal \boldsymbol{\Sigma}_k{}^{-1}(\mathbf{x} - \boldsymbol{\mu}_k)\right). \tag{10.15}$$

Specifically, this is a special case that is sometimes referred to as a Gaussian HMM. The above expression defines a matrix $B_{t,k}$ that can be computed from the data.

The full-data likelihood of the HMM satisfies

$$P(\boldsymbol{X}_{1:T} = \boldsymbol{x}_{1:T}, \boldsymbol{Z}_{1:T} = \boldsymbol{z}_{1:T}) = \pi_{z_1} \left(\prod_{t=2}^{T} A_{z_{t-1},z_t}\right) \left(\prod_{t=1}^{T} B_{t,z_t}\right). \tag{10.16}$$

However, generally the hidden states $\boldsymbol{z}_{1:T}$ are not accessible and we have to deal with the marginal likelihood of the observations $P(\boldsymbol{X}_{1:T} = \boldsymbol{x}_{1:T})$.

In summary, a HMM is fully specified by the initial distribution π, transition matrix A, and emission probability B:

$$\pi_k = P(Z_1 = k), \quad A_{kl} = P(Z_t = l|Z_{t-1} = k), \quad B_{tk} = P(\boldsymbol{X}_t|Z_t = k). \tag{10.17}$$

2.2. Forward-Backward algorithm. The main objective of this algorithm is to compute the smoothed cluster probabilities, which we will here denote by

$$\gamma_{tk} = P(Z_t = k|\boldsymbol{X}_{1:T} = \boldsymbol{x}_{1:T}). \tag{10.18}$$

The key observation relies on the fact that we can factor this matrix into two parts:

$$\gamma_{tk} \propto \alpha_{tk}\beta_{tk}, \tag{10.19}$$

where the prefactor can be found by normalisation. The matrices α_{tk} and β_{tk} are called the forward and backward probabilities, which can be efficiently computed using the recursion relations:

$$\alpha_{tk} = P(Z_t = k|\boldsymbol{X}_{1:t}) = \frac{1}{M_t} B_{tk} \sum_{j=1}^{K} A_{jk}\alpha_{t-1,j} \tag{10.20}$$

$$\beta_{tk} = P(\boldsymbol{X}_{t+1:T}|Z_t = k) = \sum_{j=1}^{K} \beta_{t+1,j} B_{t+1,j} A_{kj}, \tag{10.21}$$

where $\beta_{Tk} = 1$, which constitute the forwards and backwards parts of the algorithm. The forwards part involves the normalising constant Z_t, which is related to the likelihood via $P(\boldsymbol{X}_{1:T}) = P(\boldsymbol{X}_1)\prod_{t=2}^{T} M_t$ where $P(\boldsymbol{X}_1) = \sum_{k=1}^{K} B_{t,k}\pi_k$.

2.2.1. *Details of the derivation.* We only sketch only the basic idea here. The factorisation of the cluster probability follows from repeated application of Bayes theorem:

$$\begin{aligned}
\gamma_{tk} = P(Z_t = k|\boldsymbol{X}_{1:T}) =& P(Z_t = k, \boldsymbol{X}_{t+1:T}|\boldsymbol{X}_{1:T}) = P(Z_t = k, \boldsymbol{X}_{t+1:T}|\boldsymbol{X}_{1:t})P(\boldsymbol{X}_{1:t}|\boldsymbol{X}_{1:T}) \\
\propto& P(Z_t = k, \boldsymbol{X}_{t+1:T}|\boldsymbol{X}_{1:t}) = P(\boldsymbol{X}_{t+1:T}|Z_t = k, \boldsymbol{X}_{1:t}) \\
=& \underbrace{P(Z_t = k|\boldsymbol{X}_{1:t})}_{=\alpha_{tk}} \underbrace{P(\boldsymbol{X}_{t+1:T}|Z_t = k)}_{=\beta_{tk}},
\end{aligned} \tag{10.22}$$

up to a normalising factor. Note that in the last line we have used the Markov property.

The forwards part of the algorithm is to find an recursion relation that allows computing α_{tk}. This follows from

$$\begin{aligned}
\alpha_{tk} = P(Z_t = k|\boldsymbol{X}_{1:t}) = P(Z_t = k|\boldsymbol{X}_t, \boldsymbol{X}_{1:t-1}) = \frac{1}{M_t} P(\boldsymbol{X}_t|Z_t = k, \boldsymbol{X}_{1:t-1})P(Z_t = k|\boldsymbol{X}_{1:t-1}) \\
= \frac{1}{M_t} \underbrace{P(\boldsymbol{X}_t|Z_t = k)}_{=B_{tk}} \sum_j \underbrace{P(Z_t = k|P(Z_{t-1} = j)}_{=A^T_{kj}} \underbrace{P(Z_{t-1} = j|\boldsymbol{X}_{1:t-1})}_{\alpha_{t-1,j}} = \frac{1}{Z_t} B_{tk} \sum_j A_{jk}\alpha_{t-1,j},
\end{aligned} \tag{10.23}$$

where $M_t = P(\boldsymbol{X}_t|\boldsymbol{X}_{1:t-1})$ is a normalising factor.

The details of the backward part of the algorithm and the matrix ξ can be followed in the book of Kevin Murphy *Machine Learning: A Probabilistic Perspective* (Chapter 17).

2.3. EM algorithm for HMMs. In the expectation step, we essentially compute the clustering probabilities via:

$$\gamma_{tk}(\boldsymbol{\theta}^{(n)}) = P(Z_t = k|\boldsymbol{X}_{1:T} = \boldsymbol{x}_{1:T}, \boldsymbol{\theta}^{(n)}) \tag{10.24}$$

and the weights by

$$w_{tk}(\boldsymbol{\theta}^{(n)}) = \frac{\gamma_{tk}(\boldsymbol{\theta}^{(n)})}{\sum_{t=1}^{T}\gamma_{tk}(\boldsymbol{\theta}^{(n)})}. \tag{10.25}$$

Then the update equations for the parameter vector $\theta^{(n)}$ are given by

$$\pi_k^{(n+1)} = \gamma_{1k}(\boldsymbol{\theta}^{(n)}) \tag{10.26}$$

$$\boldsymbol{\mu}_k^{(n+1)} = \sum_{t=1}^{T} w_{tk}(\boldsymbol{\theta}^{(n)})\boldsymbol{x}_t \tag{10.27}$$

$$\boldsymbol{\Sigma}_k^{(n+1)} = \sum_{t=1}^{T} w_{tk}(\boldsymbol{\theta}^{(n)})(\boldsymbol{x}_t - \boldsymbol{\mu}_k)(\boldsymbol{x}_t - \boldsymbol{\mu}_k)^T. \tag{10.28}$$

These expressions follow analogously to the hidden Markov models where the cluster probabilities are now replaced with the smoothed version computed using the forward-backward algorithm.

The computation of the improved transition matrix $A^{(n+1)}$ is more involved since it involves the smoothed joint probability:

$$\xi_{t,i,j} = P(Z_t, Z_{t+1}|\boldsymbol{X}_{1:T}, \boldsymbol{\theta}^{(n)}), \tag{10.29}$$

which can be computed using the forward and backward probabilities:

$$A_{i,j}^{(n+1)} = \sum_{t=1}^{T-1}\xi_{t,i,j} = \frac{\alpha_{t,i}^{(n)} B_{t+1,j}^{(n)} \beta_{t+1,j}^{(n)} A_{ij}^{(n)}}{\sum_{j=1}^{K}\alpha_{t,i}^{(n)} B_{t+1,j}^{(n)} \beta_{t+1,j}^{(n)} A_{ij}^{(n)}}, \tag{10.30}$$

where the last equality follows from

$$\begin{aligned}
\xi_{t,i,j} &= P(Z_t, Z_{t+1}|\boldsymbol{X}_{1:T}, \boldsymbol{\theta}^{(n)}) \\
&= P(Z_t, |\boldsymbol{X}_{1:T}, \boldsymbol{\theta}^{(n)}) P(Z_{t+1}|Z_t, \boldsymbol{X}_{1:T}, \boldsymbol{\theta}^{(n)}) \\
&\propto P(Z_t|\boldsymbol{X}_{1:t}, \boldsymbol{\theta}^{(n)}) P(\boldsymbol{X}_{t+1:T}|Z_t, Z_{t+1}, \boldsymbol{\theta}^{(n)}) P(Z_{t+1}|Z_t, \boldsymbol{\theta}^{(n)}) \\
&\propto P(Z_t|\boldsymbol{X}_{1:t}, \boldsymbol{\theta}^{(n)}) P(\boldsymbol{X}_{t+1}|Z_{t+1}, \boldsymbol{\theta}^{(n)}) P(\boldsymbol{X}_{t+2:T}|Z_{t+1}, \boldsymbol{\theta}^{(n)}) P(Z_{t+1}|Z_t, \boldsymbol{\theta}^{(n)}) \\
&= \alpha_{t,i}^{(n)} B_{t+1,j}^{(n)} \beta_{t+1,j}^{(n)} A_{ij}^{(n)}
\end{aligned}$$

and using the fact that we need to have $\sum_j A_{ij}^{(n)} = 1$.

Iterating Eqs. (10.24)-(10.30) until convergence constitutes the EM algorithm for HMMs.

2.4. Viterbi algorithm. As noted in the book of Murphy, it is tempting to think that we can implement Viterbi by just replacing the sum-operator in the forward-backward algorithm with a max-operator. This is not always the case.

One considers the probability of the most likely sequence of states ending at state j at time t:

$$\delta_{tk} = \max_{\boldsymbol{Z}_{1:t-1}} P(\boldsymbol{Z}_{1:t-1}, Z_t = k|\boldsymbol{X}_{1:t}) \tag{10.31}$$

The key idea is that an optimal path $\boldsymbol{Z}_{1:T}^*$ contains inside it the sequence $\boldsymbol{Z}_{1:t}^*$, which is also an optimal path for $t < T$. This leads to

$$\delta_{tk} = \max_i \delta_{t-1,i} A_{ik} B_{tk}, \tag{10.32}$$

The most probable path $\boldsymbol{Z}^*_{1:T} = (z^*_t)_{t=1,2,\ldots,T}$ by backtracing the most probable state:

$$z^*_{t-1} = \underset{i}{\operatorname{argmax}}\, \delta_{t-1,i} A_{i,z^*_t} B_{t,z^*_t}, \tag{10.33}$$

with the final condition $z^*_T = \operatorname{argmax}_i \delta_{Ti}$.

Chapter 11

Dimensionality reduction: spectral methods

Continuing with the theme of unsupervised learning, this chapter introduces one of the key methods addressing the question of *dimensionality reduction*. As we will see shortly, these are *spectral methods* since 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. Much of what you will see until the end of the course exploit spectral properties of matrices associated with data.

> *Remark:* Although there exist clustering methods that employ approaches from spectral theory, which we will see later on in the next chapter, here we focus on the applications to *dimensionality reduction*.

1. Principal Component Analysis (PCA)

The classic tool for dimensionality reduction is*principal component analysis* (PCA). In its basic form, PCA is a *spectral linear method*, as we will see below.

As usual, we have the standard setup for unsupervised learning, with a data set

$$\{\mathbf{x}^{(i)}\}_{i=1}^{N} \qquad \mathbf{x}^{(i)} \in \mathbb{R}^p$$

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 (with no external 'ground truth'), guided by the data set 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 from modelling assumptions or insight into the data) that many of those variables are unnecessary or redundant (e.g., correlated variables, noise, etc), and the data set might be better described by fewer, more informative variables, i.e., we could try to represent the data in a lower dimension. In summary, the goal of dimensionality reduction is to find a description of the data set 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 data sets. By summarising the data through a reduction of dimensions, many models become more effective and computationally feasible, whilst losing next to nothing in terms of accuracy (or other quality measures).

There are many ways to formalise mathematically the task of dimensionality reduction. Some derivations are geometric; others are optimisation-based. With each of them one gets a particular perspective on this problem. We will present a couple of them below.

As discussed above, we aim for a description of every data point in terms of basis functions $\boldsymbol{\phi}_j$:

$$\mathbf{x}^{(i)} = \sum_{j=1}^{p} a_j^{(i)} \boldsymbol{\phi}_j, \text{ such that } \boldsymbol{\phi}_j \in \mathbb{R}^p, \tag{11.1}$$

and we search for an orthonormal basis such that

$$\boldsymbol{\phi}_j^T \cdot \boldsymbol{\phi}_k = \delta_{jk} := \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

From these definitions, it immediately follows that the coefficients in (11.1) are obtained as dot products (projections) on the basis:

$$a_j^{(i)} = \boldsymbol{\phi}_j^T \cdot \mathbf{x}^{(i)} = {\mathbf{x}^{(i)}}^T \cdot \boldsymbol{\phi}_j$$

One way to view dimensionality reduction is that we want to approximate $\mathbf{x}^{(i)}$ by:

$$\widehat{\mathbf{x}}_m^{(i)} = \sum_{j=1}^{m} a_j^{(i)} \boldsymbol{\phi}_j + \sum_{j=m+1}^{p} b_j \boldsymbol{\phi}_j \tag{11.2}$$

where the b_j are **not** dependent on i. This means, we want to find the $b_j, \boldsymbol{\phi}_j$ such that they apply to all samples. In doing so, we will describe each sample $\mathbf{x}^{(i)}$ only through the m coefficients $a_j^{(i)}$, whereas the sum over the b_j terms in (11.2) reflects the error (common to all samples) that is associated with the elements of the basis $\{\boldsymbol{\phi}_j\}_{j=m+1}^{p}$.

We can write down the error made from this assumption:

$$\Delta\mathbf{x}^{(i)} = \mathbf{x}^{(i)} - \widehat{\mathbf{x}}_m^{(i)} = \sum_{j=m+1}^{p} [a_j^{(i)} - b_j] \boldsymbol{\phi}_j$$

We now define our optimality criterion for this approximation to find the best description of the data. To this end, we will minimise the the mean squared error once more:

$$\begin{aligned} \text{MSE} &= \frac{1}{N} \sum_{i=1}^{N} \|\Delta\mathbf{x}^{(i)}\|^2 \\ &= \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{j,k=m+1}^{p} (a_j^{(i)} - b_j)(a_k^{(i)} - b_k) \boldsymbol{\phi}_j^T \cdot \boldsymbol{\phi}_k \right] \\ &= \frac{1}{N} \sum_{i=1}^{N} \sum_{j=m+1}^{p} (a_j^{(i)} - b_j)^2 \end{aligned} \tag{11.3}$$

As we are looking to optimise both b_j and $\boldsymbol{\phi}_j$, 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:

$$\frac{\partial \text{MSE}}{\partial b_j}\Big|_{b_j^\star} = 0 \quad \implies \quad b_j^\star = \frac{1}{N}\sum_{i=1}^{N} a_j^{(i)}$$

Now, we turn our attention to $\boldsymbol{\phi}_j$. Plugging the above expression into (11.3), we get:

$$\begin{aligned}
\text{MSE} &= \sum_{j=m+1}^{p} \frac{1}{N}\sum_{i=1}^{N} \Big(\underbrace{\mathbf{x}^{(i)T}\cdot\boldsymbol{\phi}_j}_{a_j^{(i)}} - \underbrace{\frac{1}{N}\sum_{i=1}^{N}\mathbf{x}^{(i)T}\cdot\boldsymbol{\phi}_j}_{b_j^\star} \Big)^2 \\
&= \sum_{j=m+1}^{p} \frac{1}{N}\sum_{i=1}^{N} \left[\left(\mathbf{x}^{(i)T} - \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}^{(i)T}\right)\cdot\boldsymbol{\phi}_j \right]^2 \\
&= \sum_{j=m+1}^{p} \boldsymbol{\phi}_j^T \cdot \underbrace{\frac{1}{N}\sum_{i=1}^{N} \left[\left(\mathbf{x}^{(i)} - \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}^{(i)}\right)\cdot\left(\mathbf{x}^{(i)} - \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}^{(i)}\right)^T \right]}_{\clubsuit} \cdot\boldsymbol{\phi}_j
\end{aligned}$$

Note that the 'clubsuit' block can be rewritten as follows:

$$\clubsuit = \mathbb{E}_i\Big[\,(\mathbf{x} - \mathbb{E}_i(\mathbf{x}))\cdot(\mathbf{x} - \mathbb{E}_i(\mathbf{x}))^T\,\Big],$$

which is the covariance matrix $(C_{\mathbf{x}})_{p\times p}$ of the data. Hence the MSE can be written compactly as:

$$\text{MSE} = \sum_{j=m+1}^{p} \boldsymbol{\phi}_j^T C_{\mathbf{x}}\, \boldsymbol{\phi}_j \tag{11.4}$$

To find the *orthonormal* $\boldsymbol{\phi}_j$, 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:

$$\mathcal{L} = \sum_{j=m+1}^{p} \boldsymbol{\phi}_j^T C_{\mathbf{x}} \boldsymbol{\phi}_j + \sum_{j=m+1}^{p} \lambda_j(1 - \boldsymbol{\phi}_j^T\boldsymbol{\phi}_j),$$

where the λ_j are Lagrange multipliers. In order to optimise this, we follow the same approach as before:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\phi}_j} = 2\boldsymbol{\phi}_j^T C_{\mathbf{x}} - 2\lambda_j \boldsymbol{\phi}_j^T$$

where the the derivative $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\phi}_j}$ is meant as the row vector $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\phi}_j} = \left(\frac{\partial \mathcal{L}}{\partial \phi_j^1}, ..., \frac{\partial \mathcal{L}}{\partial \phi_j^p}\right)$. The solution is then given as:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\phi}_j}\Big|_{\boldsymbol{\phi}_j^\star} = 0 \quad \implies \quad C_{\mathbf{x}}\boldsymbol{\phi}_j^\star = \lambda_j \boldsymbol{\phi}_j^\star \tag{11.5}$$

This is the main result. This solution shows that the Lagrange multipliers λ_j are simply the eigenvalues of $C_{\mathbf{x}}$ with corresponding eigenvectors $\boldsymbol{\phi}_j$. And therefore, the solution to our problem of finding a basis to expand is to use 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 in the way we defined previously. This can be written as:

$$\text{MSE} = \sum_{j=m+1}^{p} \boldsymbol{\phi}_j^{\star T} C_{\mathbf{x}} \boldsymbol{\phi}_j^{\star} = \sum_{j=m+1}^{p} \boldsymbol{\phi}_j^{\star T} \lambda_j \boldsymbol{\phi}_j^{\star} = \sum_{j=m+1}^{p} \lambda_j$$

This tells us that the error is in fact the sum of the eigenvalues that are discarded when reducing dimensions from p to m.

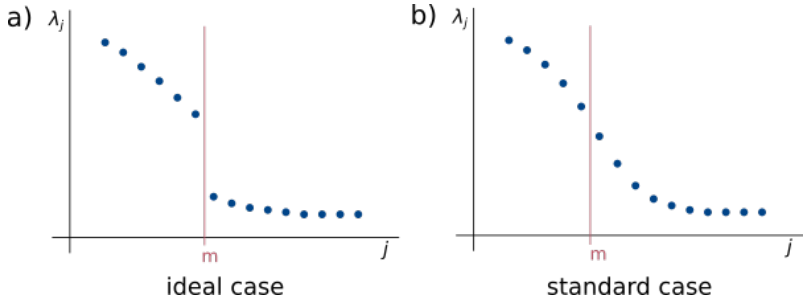


Figure 11.1. Finding an optimal m for dimensionality reduction is not always easy. In **a)**, we can see an ideal case scenario, where some eigenvalues are large, but at some point there is a drastic drop, leading to the remaining eigenvalues to be quite small. In this case, deciding on a good m is natural. 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 11.1. 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.

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%.

Given our chosen m, we can write down the approximated data as:

$$\widehat{\mathbf{x}}^{(i)} = \sum_{j=1}^{m} \left({\mathbf{x}^{(i)}}^T \cdot \boldsymbol{\phi}_j^{\star} \right) \boldsymbol{\phi}_j^{\star}$$

where $\boldsymbol{\phi}_j^{\star}$ are the eigenvectors of the covariance matrix, which are called the *principal components*, hence the name of the method.

1.1. Mathematical basis for PCA: Singular value decomposition (SVD). We have shown above that PCA has a direct interpretation in statistical terms, but we can also recast it in more general terms using matrix analysis. We now briefly show that PCA follows from the classic singular value decomposition (SVD) of a matrix. In words, the key SVD result applicable here states that given a matrix $A \in \mathbb{R}^{p \times p}$ of full rank p, then the best approximation $A_m \in \mathbb{R}^{p \times p}$ of rank m (in a precise sense to be defined below) is obtained by taking the first m singular values and singular vectors of A:

$$A \approx A_m = \sum_{j=1}^{m} \sigma_j \, \mathbf{u}_j \mathbf{v}_j^T,$$

where $\mathbf{u}_j$ are the right eigenvectors of A, $\mathbf{v}_j$ are the left eigenvectors of A, and σ_j are the singular values of A. In the remainder of this sub-section, we will go into further detail as to why this is the case.

SVD. The singular value decomposition is a matrix factorisation method, analogue to the diagonalisation of square matrices, but for rectangular matrices. Remember that the matrix diagonalisation for a diagonalisable $A_{n\times n}$ matrix is given by:

$$\left\{\begin{array}{l} AV = V\Lambda \\ V = \begin{pmatrix}\mathbf{v}_1 \dots \mathbf{v}_n\end{pmatrix} \\ \Lambda = \mathrm{diag}(\lambda_i) \\ \implies A = V\Lambda V^{-1} \end{array}\right\}$$

where the λ_i are the eigenvalues and v_i are the associated eigenvectors of A.

We now turn to the case of a rectangular matrix $A_{m\times n} \in \mathbb{R}^{m\times n}$. As an analogue to diagonalisation, consider the following expression:

$$A_{m\times n}V_{n\times n} = U_{m\times m}\Sigma_{m\times n}.$$

In the above equation, we have:

$$\begin{aligned} V &= \begin{pmatrix}\mathbf{v}_1 \dots \mathbf{v}_n\end{pmatrix} \\ U &= \begin{pmatrix}\mathbf{u}_1 \dots \mathbf{u}_m\end{pmatrix} \\ \Sigma &= \begin{pmatrix} \sigma_1 & & & & & \\ & \ddots & & & & 0 \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & 0 & & & & 0 \end{pmatrix} \end{aligned}$$

As mentioned before, the vectors $\mathbf{v}_j$ are usually known as the *right eigenvectors*, $\mathbf{u}_j$ are called the *left eigenvectors* and σ_j are the *singular values*. Note that the matrix Σ is a diagonal matrix, containing σ_j until r, after which there are only zeros. r is the rank of A and thus: $\min(m, n) \geq r$.

A more compact version of the SVD, also used in literature is the so-called *reduced form of the SVD*:

$$A = U\Sigma V^T = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j =: U_r \Sigma_r V_r^T$$

Here, U_r and V_r are truncated matrices, with only the first r vectors $\mathbf{u}_j$ and $\mathbf{v}_j$, respectively. Σ_r is then a square $r \times r$ diagonal matrix, containing all r (non-zero) singular values.

By construction, we require orthogonality of V and U, i.e., $VV^T = I$ as well as $UU^T = I$. It then follows that

$$\begin{aligned} \underbrace{A^T A}_{n\times n} &= V\Sigma^T U^T U\Sigma V^T = V(\Sigma^T\Sigma)V^T \\ \underbrace{AA^T}_{m\times m} &= U\Sigma^T V^T V\Sigma U^T = U(\Sigma^T\Sigma)U^T \end{aligned}$$

Therefore, $V_{n\times n}$ contains the eigenvectors of A^TA and $U_{m\times m}$ the eigenvectors of AA^T. Likewise, $\Sigma^T\Sigma = \Sigma\Sigma^T = \Sigma^2$ contains the non-zero eigenvalues of A^TA and AA^T on the diagonal. The singular values are thus simply the square root of the eigenvalues of A^TA and AA^T:

$$\left\{\begin{array}{l}(A^TA)\mathbf{v}_k = \sigma_k^2\mathbf{v}_k\\(AA^T)\mathbf{u}_k = \sigma_k^2\mathbf{u}_k\end{array}\right\}$$

This means that left and right singular vectors come in pairs with the *same* σ_k^2. It follows from the equality, that:

$$\mathbf{u}_k = \frac{A\mathbf{v}_k}{\sigma_k}$$

A quick check shows that this indeed makes sense:

$$AA^T\mathbf{u}_k = AA^T\frac{A\mathbf{v}_k}{\sigma_k} = \frac{A}{\sigma_k}A^TA\mathbf{v}_k = \frac{A}{\sigma_k}\sigma_k^2\mathbf{v}_k = \sigma_k^2\frac{A\mathbf{v}_k}{\sigma_k} = \sigma_k^2\mathbf{u}_k$$

Eckart and Young[1]proved the main SVD result, which is roughly stated as: 'Given A of rank k, find B with $\text{rank}(B) \leq k$ such that B is close to A, i.e. such that $\|A - B\|$ is small.'

In this case, we use $\|\cdot\|$ to denote a *matrix norm*, which can have a variety of definitions:

- **Vector induced norms: e.g. l_2 norm**

  $$\|A\|_2 = \max_{\mathbf{x}}\frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|},$$

  where the latter norms in the fraction are vector norms. The intuition of this type of matrix norm is that A operates on a vector space and the norm of the matrix is the maximal magnitude by which A expands the modulus of any vector in that space. It can be proven that this corresponds to the first and largest singular value: $\|A\|_2 = \sigma_1$.
- **Element-wise norms: e.g. Frobenius norm**

  $$\|A\|_F = \sqrt{\sum_{i,j}|a_{ij}|^2}$$

  Intuitively, this is essentially equivalent to vectorising the matrix A and computing the norm of the resulting vector. Again, this norm can be rewritten in terms of the singular values like this:

  $$\|A\|_F = \sqrt{\text{Tr}(A^TA)} = \sqrt{\text{Tr}(U\Sigma^2U^T)} = \sqrt{\text{Tr}(\Sigma^2)}$$

We can now reformulate the statement of Eckart-Young in terms of the norms introduced above.

Firstly, we consider the l_2 norm (aka spectral norm):

Statement. *If* $\text{rank}(B) \leq k$ *then*

$$\|A - B\| = \max_{\mathbf{x}}\frac{\|(A-B)\mathbf{x}\|}{\|\mathbf{x}\|} \geq \frac{\|(A-A_k)\mathbf{x}\|}{\|\mathbf{x}\|} = \|A - A_k\|$$

where $A_k = U_k\Sigma_kV_k$.

[1]Eckart, C., Young, G. The approximation of one matrix by another of lower rank. *Psychometrika* **1**, 211–218 (1936). https://doi.org/10.1007/BF02288367

Proof. By definition of the norm, we know that

$$\|A - A_k\| = \sigma_{k+1} \tag{11.6}$$

Then we have:

$$\exists \mathbf{x} \neq 0_v \text{ such that } B\mathbf{x} = 0_v \text{ and } \mathbf{x} = \sum_{j=1}^{k+1} c_j \mathbf{v}_j$$

given that $\dim[\ker B] \geq n - k$ which implies that:

$$\{\mathbf{v}_j\}_{j=1}^{k+1} \cap \ker B \neq \emptyset$$

From this, it follows that

$$\|(A - B)\mathbf{x}\|^2 = \|A\mathbf{x}\|^2 = \sum_{j=1}^{k+1} c_j^2 \sigma_j^2 \geq \left(\sum_{j=1}^{k+1} c_j^2 \right) \sigma_{k+1}^2 = \|\mathbf{x}\|^2 \sigma_{k+1}^2$$

In short:

$$\|(A - B)\mathbf{x}\|^2 \geq \|\mathbf{x}\|^2 \sigma_{k+1}^2 \tag{11.7}$$

where the $\{\sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_{k+1}, \ldots, \sigma_r\}$ are ordered in decreasing order. Hence, from (11.6) and (11.7), we can conclude that:

$$\frac{\|(A - B)\mathbf{x}\|^2}{\|\mathbf{x}\|^2} \geq \sigma_{k+1}^2 = \|A - A_k\|^2$$

□

Secondly, we consider Eckart-Young in terms of the Frobenius norm. Once again, we first reformulate this in terms of the norm and then give a short, constructive proof (some steps will be left as an exercise to the reader).

Statement. *For any B with $\text{rank}(B) \leq k$, we have:*

$$\|A - B\|_F \geq \|A - A_k\|_F$$

where A_k is as stated before.

Proof. In the following we give but a concise overview of the (constructive) proof.

Consider the problem

$$\min_B \|A - B\|_F^2,$$

i.e., finding some B of rank k or less such that the Frobenius norm $\|A - B\|_F$ is minimised. We can write any matrix B as a product of the following two matrices:

$$B = C_{m \times k} R_{k \times n} \quad \text{such that} C^T C = D \text{ and } RR^T = I.$$

We then define the 'error' of approximating A through the lower-ran matrix B):

$$E = \|A - CR\|_F^2$$

This has now become an optimisation problem and hence we compute the partial derivatives:

$$\left\{ \begin{array}{l} \dfrac{\partial E}{\partial C} = 2(CR - A)R^T \\ \dfrac{\partial E}{\partial R} = 2(R^T C^T - A^T)C \end{array} \right\}$$

At the minimum, we then have:

$$\left\{ \begin{array}{c} CRR^T = AR^T \\ R^T C^T C = A^T C \end{array} \right\}$$

Combining both results, we get:

$$\left\{ \begin{array}{lll} (A^T A)R^T & = R^T D & \Longrightarrow R^T = V_k \\ (AA^T)C & = CD & \Longrightarrow C = U_k \end{array} \right\}$$

From this we can see that the constructive proof has shown the SVD does indeed give the best CR and the error is then:

$$E = \|A - CT\|_F^2 = \sum_{j=k+1}^{r} \sigma_j^2$$

□

1.2. Connection of SVD with PCA. If we write our data set as a matrix, we can apply the singular value decomposition to effectively reduce the dimensions of our data, i.e. the number of variables for each data point.

To do so, let us compile all our samples into the data matrix:

$$X_{N\times p} = \begin{pmatrix} x_1^{(1)} & \dots & x_p^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \dots & x_p^{(N)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}^{(1)^T} \\ \vdots \\ \mathbf{x}^{(N)^T} \end{pmatrix}$$

where $\mathbf{x}^{(i)} \in \mathbb{R}^p$ are our data samples. Let us also define $\mathbf{1}_{N\times 1}$ to be a column vector of ones of length N. Then we have

$$\mathbf{1}^T X = \begin{pmatrix} \sum_{i=1}^N x_1^{(i)} & \dots & \sum_{i=1}^N x_p^{(i)} \end{pmatrix}$$

Then:

$$\frac{1}{N}\mathbf{1}(\mathbf{1}^T X) = \begin{pmatrix} \langle x_1 \rangle & \dots & \langle x_p \rangle \\ \vdots & \ddots & \vdots \\ \langle x_1 \rangle & \dots & \langle x_p \rangle \end{pmatrix}_{N\times p}$$

Here, we have denoted the average of a variable over the data set as $\langle x_j \rangle = \frac{1}{N}\sum_{i=1}^N x_j^{(i)}$. If we subtract this matrix of averages from our original data matrix we get:

$$\mathcal{X}_{N\times p} := X - \frac{1}{N}\mathbf{1}\mathbf{1}^T X = \underbrace{\left(I - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right)}_{\textit{centering matrix}} X,$$

which is the matrix of *centred data*. Remembering our previous section and in particular, equation (11.5), we consider the sample covariance matrix, which is indeed simply:

$$(\mathcal{X}^T\mathcal{X})_{p\times p} = N\, C_{\mathbf{x}} \tag{11.8}$$

Hence we easily conclude that PCA follows from the SVD of the centered data matrix $\mathcal{X}$, which is equivalent to obtaining the eigenvectors of the covariance matrix as shown in equation (11.8):

$$\mathcal{X}_{N\times p} = U_{N\times p}\Sigma_{p\times p}V_{p\times p}^T$$

From the Eckart-Young result, we can now write down our k rank approximation as:

$$\widehat{\mathcal{X}}_{N\times p} = U_k\Sigma_k V_k^T =: X_{\mathrm{PCA}}V_k^T$$

where the $N \times k$ matrix

$$X_{\text{PCA}} = U_k \Sigma_k, \tag{11.9}$$

expresses the approximation $\widehat{\mathcal{X}}$ in terms of the k singular vectors (known as the *principal components*) contained in V_k^T. Note that $k < p$ and Σ contains the *ordered* singular values. The matrix X_{PCA} thus contains the coordinates of each of the N data points (what we called $a_j^{(i)}$) in the k-dimensional space of the principal components, i.e., the vectors collected in V_k^T. In essence, this provides a concise representation or approximation of the data in a lower dimensional space ($k < p$) with error E quantified by the sum of squares of the discarded singular values.

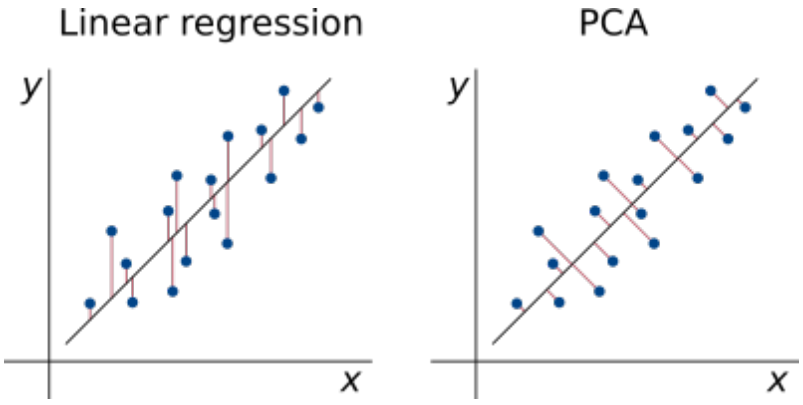


Figure 11.2. Geometric picture of PCA/SVD as opposed to linear regression.

1.3. Visual intuition of PCA. In Figure 11.2, we illustrate visually the difference between linear regression and PCA. Whilst in linear regression, the aim is to find a hyperplane that minimises the sum of the squared error (see Chapter 2), 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*. PCA (through SVD) is related to the methods of *total least squares* or *proper orthogonal decomposition*. 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 (hence the different definition of the MSE along the normal to the hyperplane).

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 three of them:

- how to deal with nonlinear data;
- how to produce sparse representations;
- how to produce non-negative representations.

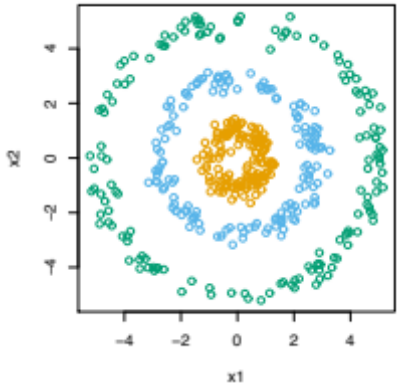


Figure 11.3. An example of non-linearly distributed data, that standard PCA will not be able to handle very well. Instead, we need to use kernel PCA methods to tackle such data.

2.1. Nonlinear extensions: Kernel PCA. PCA is a linear method, i.e., it finds linear projections onto a space of reduced dimensionality by minimising a quadratic. This begs the question of how to deal with nonlinear data that eludes such methods, e.g., see Figure 11.3. A first extension of PCA introduces the concept of kernels to dimensionality reduction. This should come as no surprise at this point of the course, since we have already introduced kernel functions as nonlinear extensions of SVMs in Chapter 7. The idea here is similar: substitute dot products by kernel functions (this is why we spent some time expressing our results above in terms of dot products).

(Note that these kernel functions should not be confused with the concept of kernel tensors used in Chapter 8 on Neural Networks.)

Remember that PCA was about the eigenvalues of $X^T X$ and/or XX^T, which 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:

$$\left(XX^T\right)_{ij} = {\mathbf{x}^{(i)}}^T \cdot \mathbf{x}^{(j)}.$$

where we are abusing notation and assuming that the original variables $\mathbf{x}^{(i)}$ were already centred. We can now instead consider a kernel function $K_{ij} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ (properly defined as described in Chapter 7), and proceed with the eigendecomposition of the kernel matrix K. For example, we take the radial basis (or Gaussian) kernel function:

$$K_{ij} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = e^{-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{c}} \tag{11.10}$$

More precisely, kernel PCA relies on the idea on introducing a non-linear, potentially high-dimensional mapping $\psi(\mathbf{x})$. While in PCA we were interested in the spectral decomposition of $C_{\mathbf{x}}$, in kernel PCA we look at the one of:

$$C_\psi = \frac{1}{N}\sum_{i=1}^{N} \psi(\mathbf{x}^{(i)})\psi(\mathbf{x}^{(i)})^T \tag{11.11}$$

which can be written in terms of its eigenvalues and eigenvectors:

$$C_\psi \mathbf{v}_j = \lambda_j \mathbf{v}_j \qquad j = 1, .., p \tag{11.12}$$

The eigenvectors $\mathbf{v}_j$ can be expressed as a function of ψ through a set of coefficients $a_j^{(i)}$:

$$\mathbf{v}_j = \sum_{i=1}^{N} a_j^{(i)} \psi(\mathbf{x}^{(i)}) \tag{11.13}$$

As we saw for SVMs (chapter 7), the introduction of kernels is convenient because we can work directly with them, without having to work out explicitly $\psi(\mathbf{x})$. In this setting, by plugging (11.13) into (11.12) and using the definition $K_{ij} = \psi(\mathbf{x}^{(i)}) \cdot \psi(\mathbf{x}^{(j)})$, we find:

$$\mathbf{K}\mathbf{a}_j = N\lambda_j \mathbf{a}_j \tag{11.14}$$

where $\mathbf{a}_j = \{a_j^{(i)}\}_{i=1}^N$. Hence, the coefficients $a_j^{(i)}$ in (11.13) can be found as eigenvectors of $\mathbf{K}$. As a result, for a given data point $\mathbf{x}$, its projection onto the principal components can be retrieved as:

$$\psi(\mathbf{x}) \cdot \mathbf{v}_j = \sum_{i=1}^{N} a_j^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) \tag{11.15}$$

where $j = 1, ..., m$, and m is chosen to be smaller than p. Note: in equation (11.11), we have assumed $\psi(\mathbf{x})$ to be already centered (i.e. with zero mean). This is not guaranteed to be the case even if $\mathbf{x}$ is centered, because $\psi(\mathbf{x})$ lives in a different space. If $\psi(\mathbf{x})$ is not centered, then one would need to work with:

$$\tilde{\psi}(\mathbf{x}) = \psi(\mathbf{x}) - \frac{1}{N} \sum_{i=1}^{N} \psi(\mathbf{x}^{(i)})$$

and the corresponding kernel $\tilde{K}_{ij} = \tilde{\psi}(\mathbf{x}^{(i)}) \cdot \tilde{\psi}(\mathbf{x}^{(j)})$. With some algebra, it can be shown that this normalised version of the kernel can be found as:

$$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_N \mathbf{K} - \mathbf{K}\mathbf{1}_N + \mathbf{1}_N \mathbf{K} \mathbf{1}_N \tag{11.16}$$

where $\mathbf{1}_N$ is a $N \times N$ matrix with all elements equal to $1/N$. (See section 12.3 in C.M. Bishop, *Pattern Recognition and Machine Learning* for more details on the calculations).

Kernel PCA would work well with the data shown in Figure 11.3, and Figure 11.4 shows the outcome of applying this kernel PCA.

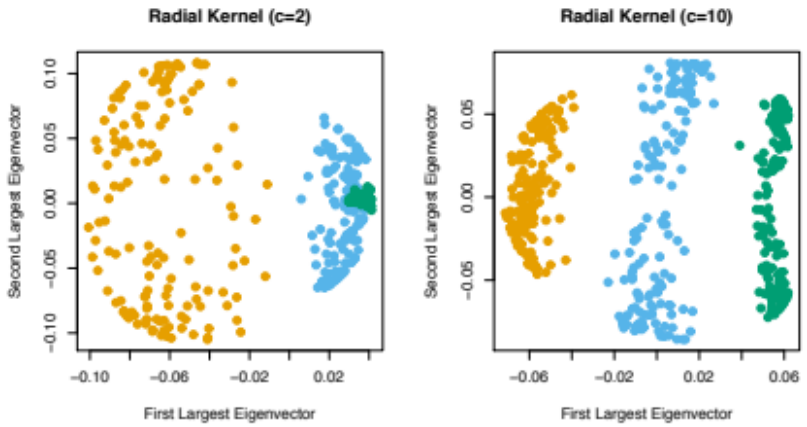


Figure 11.4. Kernel PCA as applied to data in figure 11.3, with two values of hyper-parameter for the radial kernel (equation (11.10).

2.2. Sparse PCA. In this second extension to 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 these methods 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 2.

First, recall our concise representation extracted from PCA in (11.9):

$$(X_{\text{PCA}})_{N\times k} = U_k \Sigma_k$$

whose rows contain the coordinates of each data point in terms of the m principal vectors $V_k = \big(\mathbf{v}_1 \dots \mathbf{v}_k\big)_{p\times k}$. By construction, V is a *non-sparse* basis in terms of the p original variables of the data set, which in turn leads to limited interpretability.

To increase the sparsity of the representation, we can consider penalty terms that reduce the magnitude of the vectors of the basis. An alternative way of writing down the original PCA is the following:

$$\max_{\mathbf{v}} \mathbf{v}^T(\mathcal{X}^T\mathcal{X})\mathbf{v}, \text{ where } \mathbf{v} \in \mathbb{R}^p \text{ such that } \mathbf{v}^T\mathbf{v} = 1$$

We can induce sparsity on the vectors $\mathbf{v}_i$ by adding extra constraints (see the later sections of chapter 2, especially LASSO, for similarities):

$$\sum_{j=1}^{p} |\mathbf{v}_j| \leq t$$

This will essentially lead to PCA attempting to find a basis that tries to minimise the quadratic error under sparsity constraints, which we can write down as:

$$\{\mathbf{v}_j^{\text{LASSO}}\}_{j=1}^{p}$$

and the $p \times k$ matrix V_k^{LASSO} will be sparser 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 coordinates indicating the fact that the sparse principal components can be expressed in terms of few of the original coordinates of the data.

2.3. Non-negative matrix factorisation (NMF). Whilst Kernel PCA was aiming to include non-linearity and Sparse PCA gave us a sparser, more interpretable basis, this 'extension' of PCA addresses 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 is non-negative, i.e., all the variables are restricted to be non-negative. An example of such data sets is images (where the description of the image is a grayscale value) or data sets 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).

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 k principal components is written:

$$\widehat{\mathcal{X}} = U_k \Sigma_k V_k^T =: X_{\text{PCA}} V_k^T$$

In NMF, this is generalised to the following matrix factorisation. Consider our usual $N \times p$ data matrix $\boldsymbol{X}$

$$X_{N\times p} = \begin{pmatrix} x_1^{(1)} & \dots & x_p^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \dots & x_p^{(N)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}^{(1)T} \\ \vdots \\ \mathbf{x}^{(N)T} \end{pmatrix} =: \begin{pmatrix} X_{11} & \dots & X_{1p} \\ \vdots & \ddots & \\ X_{N1} & & X_{Np} \end{pmatrix},$$

where we have introduced the notation X_{ij} for convenience.

The data matrix X is approximated by:

$$\boldsymbol{X} \approx \boldsymbol{W}\boldsymbol{H}, \quad \text{such that} \quad W_{ij} \geq 0, H_{ij} \geq 0$$

where we require a factorisation into matrices with non-negative elements, i.e. $W_{ij} \geq 0$ and $H_{ij} \geq 0$ for all i, j. Here $\boldsymbol{W}$ is $N \times r$ and $\boldsymbol{H}$ is $r \times p$, 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:

$$\mathbf{x}_{p\times 1}^{(i)} = \boldsymbol{H}^T \mathbf{w}_{r\times 1}^{(i)}, \quad i = 1, \dots, N.$$

In this formulation, $\boldsymbol{H}$ contains the non-negative equivalent of the principal components, and the matrix $\boldsymbol{W}$ contains the non-negative coefficients expressing the data $\mathbf{x}^{(i)}$ in the basis of vectors in $\boldsymbol{H}$.

This method was known from the 1970's and 80's but it was revitalised by Seung and Lee in 1999. It was shown that this optimisation is equivalent to the following optimisation problem:

$$\max_{\boldsymbol{W},\boldsymbol{H}} L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{i=1}^{N} \sum_{j=1}^{p} \left[X_{ij} \log(\boldsymbol{W}\boldsymbol{H})_{ij} - (\boldsymbol{W}\boldsymbol{H})_{ij} \right],$$

and, equivalently, X_{ij} is a Poisson distributed random variable with mean $(\boldsymbol{W}\boldsymbol{H})_{ij}$. Although the problem is non-convex, the following alternating algorithm (Lee and Seung, 2001) was shown to converge to a local maximum of $L(\boldsymbol{W}, \boldsymbol{H})$:

$$W_{ij} \leftarrow W_{ij} \frac{\sum_{k=1}^{p} H_{jk} X_{ik} / (\boldsymbol{W}\boldsymbol{H})_{ik}}{\sum_{k=1}^{K} H_{jk}} \tag{11.17}$$

$$H_{jk} \leftarrow H_{jk} \frac{\sum_{i=1}^{N} W_{ij} X_{ik} / (\boldsymbol{W}\boldsymbol{H})_{ik}}{\sum_{i=1}^{N} W_{ij}} \tag{11.18}$$

For more information about NMF, you can read the original paper by Seung and Lee: `https://www.nature.com/articles/44565`. That paper 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.

Graphs and 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.

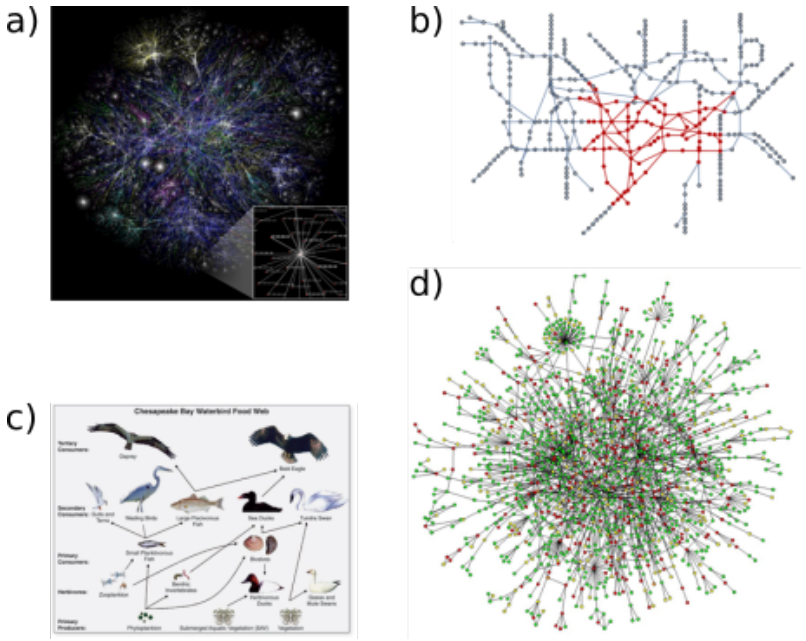


Figure 12.1. A few real-world examples of graphs. **a)** An illustration of the internet, where each node is a server and the edges are coloured according to traffic between them. **b)** A schematic of an Underground Map (unfortunately, the real map for the London underground is under heavy copyright 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.

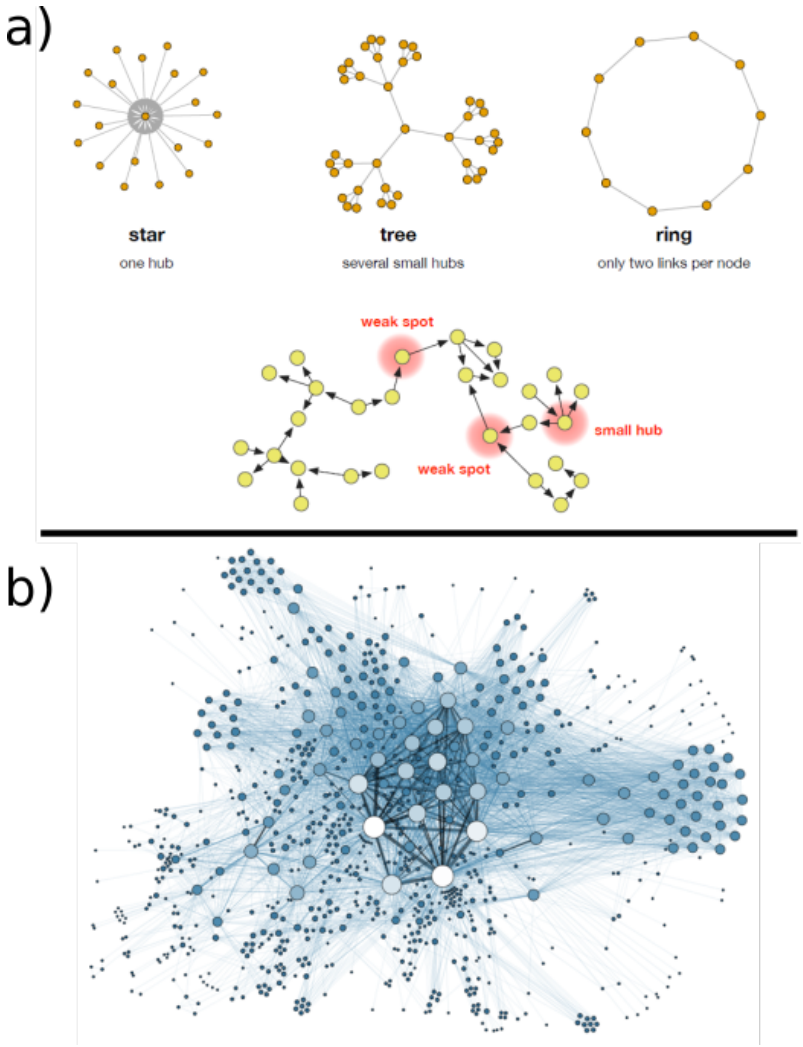


Figure 12.2. **a)** Small networks are intuitive,... **b)** but large ones are not (at all). This motivates the use of graph-based learning methods to understand highly complex data.

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 written in the usual format:

$$\{\mathbf{x}^{(i)}\}_{i=1}^{N} \qquad \mathbf{x}^{(i)} \in \mathbb{R}^p \qquad \mathbf{x}^{(i)} = \begin{pmatrix} x_1^{(i)} \\ \vdots \\ x_p^{(i)} \end{pmatrix}$$

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 other words, each data point can be represented by a vector in p-dimensional space. In the chapters above, 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 keeping most of the variability. 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:

$$\begin{aligned}
(\textit{cosine similarity}) \qquad & S_{ij} = \frac{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}}{\|\mathbf{x}^{(i)}\| \, \|\mathbf{x}^{(j)}\|} \\
(\textit{statistical similarity}) \qquad & S_{ij} = \frac{\mathrm{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})}{\sqrt{\mathrm{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(i)})}\sqrt{\mathrm{cov}(\mathbf{x}^{(j)}, \mathbf{x}^{(j)})}} \\
(\textit{Euclidean distance}) \qquad & D_{ij} = \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\| \\
(\textit{kernel matrix }) \qquad & D_{ij} = \exp\left[\frac{-\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{c}\right]
\end{aligned}$$

In most of the unsupervised learning methods, we had to define pairwise (dis)similarities between samples, which were the important determinants of the 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.)

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.

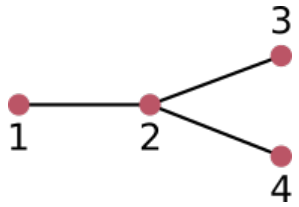


Figure 12.3. A simple example of a graph with $N = 4$.

In Figure 12.3, 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:

$$\begin{aligned}
V &= \{1, 2, 3, 4\} \\
E &= \{(1,2), (2,3), (2,4)\}
\end{aligned}$$

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 $A = A^T$.

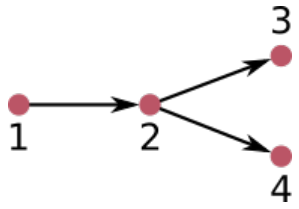


Figure 12.4. A simple example of a *directed* graph.

1.3. Directed graphs. For the case of a *directed graph*, this 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 12.4. 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:

$$\mathbf{d}_{\text{in}} = A\mathbf{1} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathbf{d}_{\text{out}} = A^T\mathbf{1} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

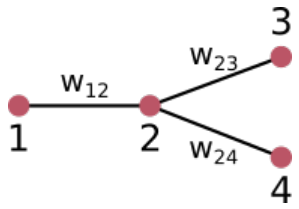


Figure 12.5. An example of a weighted graph.

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 12.5, the adjacency matrix would then look like the following:

$$A = \begin{bmatrix} 0 & w_{12} & 0 & 0 \\ w_{12} & 0 & w_{23} & w_{24} \\ 0 & w_{23} & 0 & 0 \\ 0 & w_{24} & 0 & 0 \end{bmatrix}$$

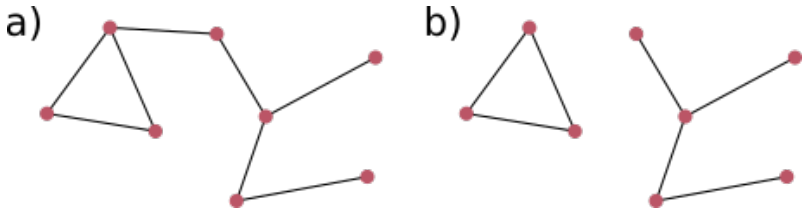


Figure 12.6. a) An example of a connected graph. **b)** An example of a disconnected graph with two components.

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 12.6 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 (i.e., 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 *graph distances*. 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.

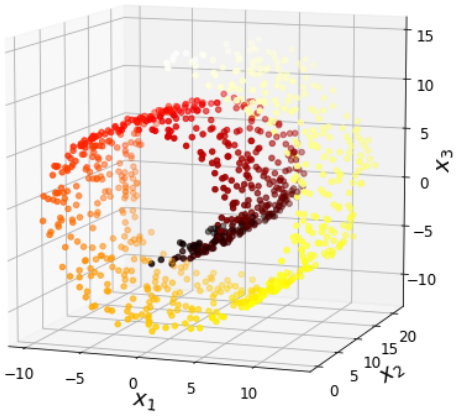


Figure 12.7. The 'Swiss roll' data set.

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 12.7. 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 12.8.

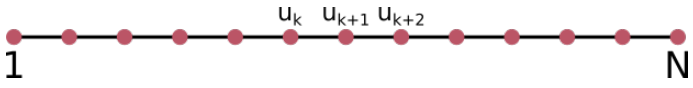


Figure 12.8. A discretisation of 1D space resulting in a graph consisting of a string of nodes connected to the next node by one edge each.

The heat equation on the discretised space can now be rewritten in terms of a tri-diagonal $N \times N$ matrix:

$$\frac{d\mathbf{u}}{dt} = \underbrace{\begin{pmatrix} -1 & 1 & & & & & \\ 1 & -2 & 1 & & & 0 & \\ & \ddots & \ddots & \ddots & & & \\ & & 1 & -2 & 1 & & \\ & & & \ddots & \ddots & \ddots & \\ & 0 & & & 1 & -2 & 1 \\ & & & & & 1 & -1 \end{pmatrix}}_{\text{Laplacian matrix: } -L} \begin{pmatrix} u_1 \\ \vdots \\ u_k \\ u_{k+1} \\ u_{k+2} \\ \vdots \\ u_N \end{pmatrix} \tag{12.1}$$

Note that the adjacency matrix of the graph in Figure 12.8 is given by:

$$A = \begin{pmatrix} 0 & 1 & & \\ 1 & 0 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 0 \end{pmatrix}$$

and the degree matrix is $D = \text{diag}(\mathbf{d}) = \text{diag}(A\mathbf{1})$.

It is then easy to see that the matrix in (12.1) is simply:

$$-L = A - D$$

L is called the *Laplacian matrix*. More specifically, L is commonly termed the *Combinatorial Graph Laplacian*.

The heat equation as defined above for a graph can be written compactly as:

$$\frac{d\mathbf{u}}{dt} = -L\mathbf{u} \tag{12.2}$$

This equation has some properties that follow immediately from the definitions. Recall that the solution of such a vector-matrix differential equation is given by the matrix exponential:

$$\mathbf{u}(t) = e^{-Lt}\mathbf{u}(0)$$

From this we know that the spectral decomposition of the Laplacian will be very important, since the evolution of this equation will be completely described by the eigenvalues and eigenvectors. (We will study this in more detail in the next section.) It is also important to notice from the definitions above that the vector of ones is an eigenvector of the Laplacian with eigenvalue zero:

$$L\mathbf{1} = (D - A)\mathbf{1} = \mathbf{d} - \mathbf{d} = 0$$

Therefore, $\mathbf{1}$ is a stationary point of the heat equation (12.2).

It can be shown that, for a connected graph, $\mathbf{1}$ is the only eigenvector with zero eigenvalue. Indeed, the number of zero eigenvalues of L is equal to the number of disconnected components. This can be shown fairly easily. If we have a graph with k disconnected components, the L can be rearranged into block-diagonal form. For instance, if $k = 3$, then we have the following direct sum:

$$L = L_1 \oplus L_2 \oplus L_3$$

Now, since not only L is a Laplacian matrix, but L_i are all independently Laplacian matrices too, we have not only $L\mathbf{1} = 0$, but also $L_i\mathbf{1} = 0$, $i = 1, 2, 3$. Therefore, by construction, the number of components corresponds to the number of zero eigenvalues.

1.7.1. *Positive semi-definiteness of the Laplacian and the incidence matrix.* Let us introduce another matrix that describes a given graph fully: the *incidence matrix*. In this matrix of dimensions $E \times N$, the rows represent individual edges of the graph and the columns correspond to the nodes. Then, each row will contain exactly two non-zero values (i.e. 1) at the columns that correspond to the end-points of the given edge. For the easy graph from Figure 12.3), the incidence matrix is:

$$B_{\text{inc}} := \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Note that each column is labelled by node number as given by Figure 12.3. This is one of the definitions of an incidence matrix, but it is commonplace to use the *oriented incidence matrix*. In this version, each edge is arbitrarily assigned a direction, i.e. one node acts as the tail and the other node is the head of each edge. In the case of a directed graph, the directions are naturally those of the graph. Once again, using the simple graph in Figure 12.3, we have:

$$B = B_{\text{ori}} := \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} \end{array}$$

If we then compute $B^T B$, it is easy to see that the combinatorial Laplacian can be obtained as the product of the incidence matrix by its transpose:

$$B^T B = \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} = L = D - A.$$

This is the discrete analogue of $\nabla^2(\cdot) = \nabla(\nabla(\cdot))$.

With this simple result, it is easy to show that L is positive semi-definite, i.e.,

$$\mathbf{u}^T L\mathbf{u} \geq 0, \quad \forall \mathbf{u} \in \mathbb{R}^N.$$

Note that the quadratic form can be rewritten as:

$$\mathbf{u}^T L\mathbf{u} = \mathbf{u}^T B^T B\mathbf{u} = \|B\mathbf{u}\|^2 \geq 0, \quad \forall \mathbf{u} \in \mathbb{R}^N.$$

Since L is positive semi-definite, then all eigenvalues of L are non-negative (easy to proof and a standard result in your course in linear algebra).

2. Diffusion on graphs

In the previous section, the (combinatorial) Laplacian matrix of a graph L was defined in analogy to the continuous Laplacian operator ∇^2. We introduced the connection between ∇^2 and L through a discretisation of continuous space (generating a lattice) but then extended the concept to any generic graph.

The Laplacian operator is fundamentally linked to conservation equations and spatio-temporal partial differential equations. In particular, we can now return to the heat (or diffusion) equation:

$$u = u(x,t), \quad \frac{\partial u}{\partial t} = \nabla^2 u$$

and write down its graph analogue:

$$\frac{d\mathbf{u}}{dt} = -L\mathbf{u}. \tag{12.3}$$

The solution of this linear equation is given by the matrix exponential:

$$\mathbf{u}(t) = e^{-tL}\,\mathbf{u}(0). \tag{12.4}$$

Using the eigendecomposition of L, we can write down the solution explicitly. The eigendecomposition of L follows from the eigenvalue problem :

$$L\mathbf{v}_i = \lambda_i \mathbf{v}_i,$$

which is equivalent to:

$$\begin{aligned} LV &= V\Lambda \\ \text{with } \Lambda &= \operatorname{diag}(\lambda_i) \\ \text{and } V &= \begin{pmatrix} \mathbf{v}_1 & \dots & \mathbf{v}_N \end{pmatrix}. \end{aligned}$$

Since we have an undirected graph, the Laplacian is symmetric ($L = L^T$) and it follows that the matrix V is orthogonal: $VV^T = I$. Hence we can rewrite the Laplacian as:

$$L = V\Lambda V^T$$

The matrix exponential is defined as

$$e^{-tL} = \sum_{k=0}^{\infty} \frac{1}{k!}(-tL)^k.$$

We then use $L = V\Lambda V^T$ to obtain

$$L^k = V\Lambda \underbrace{V^T V}_{=I} \Lambda V^T \dots V\Lambda V^T = V\Lambda^k V^T$$

so that the matrix exponential becomes:

$$\begin{aligned} e^{-tL} &= V\left[\sum_{k=0}^{\infty} \frac{(-t)^k}{k!}\Lambda^k\right] V^T \\ &= V \begin{pmatrix} \sum_{k=0}^{\infty} \frac{(-t)^k}{k!}\lambda_1^k & & 0 \\ & \ddots & \\ 0 & & \sum_{k=0}^{\infty} \frac{(-t)^k}{k!}\lambda_N^k \end{pmatrix} V^T = V \begin{pmatrix} e^{-\lambda_1 t} & & 0 \\ & \ddots & \\ 0 & & e^{-\lambda_N t} \end{pmatrix} V^T \\ &= V\,\mathrm{diag}\,(e^{-\lambda_i t})V^T = \sum_{i=1}^{N} e^{-\lambda_i t}\mathbf{v}_i\mathbf{v}_i^T \end{aligned}$$

Hence the eigenvectors and eigenvalues of the Laplacian fully describe the time evolution of the graph diffusion.

The solution (12.4) to the heat equation can then be written as:

$$\mathbf{u}(t) = e^{-tL}\mathbf{u}(0) = \sum_{i=1}^{N} [\mathbf{v}_i^T \cdot \mathbf{u}(0)]\, e^{-\lambda_i t}\mathbf{v}_i,$$

where the $[\mathbf{v}_i^T \cdot \mathbf{u}(0)]$ correspond to the projections of the initial condition on each of the eigenvectors of the Laplacian.

Remember that the Laplacian of a connected undirected graph is positive semidefinite and has a single zero eigenvalue associated with the vector of ones as eigenvector. Hence we can rewrite the solution as:

$$\mathbf{u}(t) = \big(\mathbf{1}^T \cdot \mathbf{u}(0)\big)\mathbf{1} + \sum_{i=2}^{N} [\mathbf{v}_i^T \cdot \mathbf{u}(0)] e^{-\lambda_i t}\mathbf{v}_i.$$

Since $\lambda_i > 0, \forall i \geq 2$, the diffusion converges to a unique value across all nodes as $t \to \infty$:

$$\lim_{t\to\infty} \mathbf{u}(t) = \left[\mathbf{1}^T \cdot \mathbf{u}(0)\right]\, \mathbf{1} = N\langle \mathbf{u}(0)\rangle\, \mathbf{1},$$

and the approach to this fixed point is dictated by the second eigenvalue asymptotically:

$$\text{i.e., for } t \gg \quad \mathbf{u}(t) - N\langle \mathbf{u}(0)\rangle \mathbf{1} \approx \left[\mathbf{v}_2^T \cdot \mathbf{u}(0)\right]\, e^{-\lambda_2 t}\mathbf{v}_2.$$

Here $\langle \mathbf{u}(0)\rangle$ corresponds to the average of the initial values across all nodes.

Therefore λ_2 and $\mathbf{v}_2$ contain crucial properties of the particular graph. If λ_2 is small, the approach to the fixed point will be slow, and the pattern on the graph encoded by $\mathbf{v}_2$ will survive the longest, as we converge to the uniform pattern on the graph given by $\mathbf{1}$.

Indeed, λ_2 is usually called the *spectral connectivity* of the graph, and $\mathbf{v}_2$ is called the *Fiedler eigenvector*. Both will appear again later in more detail when we will understand further their meaning in terms of graph partitioning.

3. Connection of graphs with random walks

The connection between graphs and random walks is well known. To remind ourselves of this, let us consider the discretisation of a 1D domain in Figure 12.8. On this space, we have a random walk, which is simply a discrete-time stochastic process in which a

walker jumps to neighbouring positions on the discretised line at random, i.e., it moves stochastically from node to node of the graph as allowed by the edges present. The state of the system can be written as $\mathbf{p}_t = (p_t^{(1)}, \ldots, p_t^{(N)})$, a $N \times 1$ vector with components $p_t^{(j)}$ describing the probability of the random walker being at node j and time t. Note that $\mathbf{p}_t$ is a probability vector, hence it is normalised:

$$\mathbf{1}^T \mathbf{p}_t = 1.$$

In the simplest case of a discrete 1D space, the random walker simply goes to the left or the right with equal probability, so that we have the following expression for node j:

$$p_{t+1}^{(j)} = \frac{1}{2}\left(p_t^{(j-1)} + p_t^{(j+1)}\right)$$

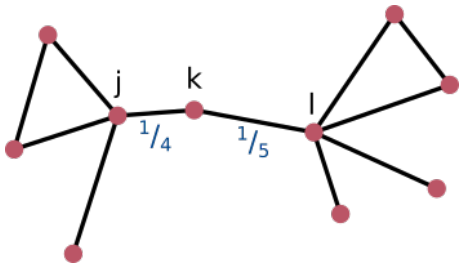


Figure 12.9. An example graph to illustrate a discrete random walker.

The above easy setup can of course be generalised and applied to any graph. For example, consider the graph given in Figure 12.9. For nodes k, we can write down its $t+1$'th probability as

$$p_{t+1}^{(k)} = \frac{1}{5} p_t^{(l)} + \frac{1}{4} p_t^{(j)}$$

Note that since j has 4 neighbours, its probability will be split into 4 and thus, $1/4$ ends up at k. Similarly, l has 5 neighbours, thus the term $\frac{1}{5}$ in the equation.

It is now obvious that the random walk has to do with the topology of the graph. This process is an example of a *discrete-time Markov Chain*, i.e., a stochastic process where the transitions happen in a discrete state space at discrete points in time. This process can be rewritten succinctly from a graph-theoretical perspective using the adjacency matrix A:

$$\mathbf{p}_{t+1} = (AD^{-1})\mathbf{p}_t =: M\mathbf{p}_t \tag{12.5}$$

where $D = \mathrm{diag}(\mathbf{d})$ contains the degree vector $\mathbf{d} = A\mathbf{1}$ on the diagonal. Including D^{-1} in (12.5) takes care of the 'equal split' of probabilities amongst all neighbouring nodes of a given node. Note also that in in (12.5) we have defined M, which is usually called the *transition matrix*. It is simply a 'normalised' version of the adjacency matrix, consistent with the conservation of probability in the process. The solution of this discrete stochastic system is:

$$\mathbf{p}_t = M^t \mathbf{p}_0,$$

and the asymptotics of this process (as $t \to \infty$) follow a similar set of considerations as those described at the end of Section 2 on graph diffusions but now focussing on the eigenvalues and eigenvectors of the matrix M. Random walks on graphs have been instrumental in tools for graph-based analysis, as we will see towards the end of this chapter.

Associated with a random walk, there is also a continuous-time process, the *continuous-time Markov Chain*, which can also be written in graph-theoretical terms as:

$$\frac{d\mathbf{p}(t)}{dt} = -\underbrace{(I - AD^{-1})}_{=:L_{\text{rw}}}\mathbf{p}_t.$$

(We do not go into the derivation of this process but it follows from going to the limit of small time increments between the transitions or 'jumps'.) The expression underscored as L_{rw} in the expression looks familiar, as it is indeed in essence a 'normalised' version of the Laplacian matrix, i.e.:

$$L_{\text{rw}} := I - AD^{-1} = L\,D^{-1}$$

This is called the *random walk Laplacian*. In many formulations and applications, rather than using the random-walk Laplacian, it is common to use another 'normalised' version of the Laplacian called the *symmetric Laplacian*, defined by:

$$\begin{aligned} L_{\text{sym}} :=& I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} \\ =& D^{-\frac{1}{2}}LD^{-\frac{1}{2}} = D^{-\frac{1}{2}}L_{\text{rw}}D^{\frac{1}{2}} \end{aligned}$$

Like the random-walk Laplacian, the symmetric Laplacian is also normalised, but it has the additional benefit of simplifying the eigendecompositions because of its symmetry $L_{\text{sym}} = L_{\text{sym}}^T$. Importantly, it remains isospectral with L_{rw}, i.e. L_{sym} and L_{rw} have the same eigenvalues, as they are just related by a similarity transformation, as shown above.

In Data Science, both the normalised Laplacians (symmetric or random-walk) and the standard (unnormalised) combinatorial Laplacian are used, although the use of the normalised Laplacians is more common as the interpretation of the results is usually more accessible due to their probabilistic basis, and because it has direct links with other heuristics proposed in Machine Learning.

4. 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 5 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 said partitions. If cutting a graph into separate subgraphs is 'easy', this is indicative of the presence of potentially good clusters.

4.1. 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 $\mathbf{x}^{(i)} \in \mathbb{R}^p, \quad i = 1, \ldots, N$. 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'. 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]
- **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[2] is a beautiful example of this approach.

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.

> As a quick insight left to the reader, consider the Swiss roll data set depicted in Figure 12.7 again and get an 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). The visual intuition is given in Figure 12.10 below. 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.

4.2. 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,

[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.

[2] *Graph Sparsification by Effective Resistances*, Daniel A. Spielman and Nikhil Srivastava, SIAM Journal on Computing 2011 40:6, 1913-1926

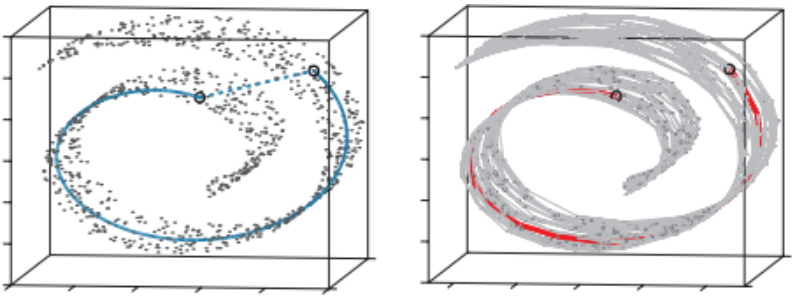


Figure 12.10. The Swiss roll data set with a geometric graph.

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:

$$C = \frac{1}{2} \sum_{i \in S_1, j \in S_2} A_{ij}$$

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:

$$C = \frac{1}{4} \sum_{i,j} \big(\underbrace{d_i \delta_{ij} - A_{ij}}_{L_{ij}} \big) s_i s_j = \frac{1}{4} \mathbf{s}^T L \mathbf{s} \tag{12.6}$$

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 C:

$$\min_{\mathbf{s}\in\{-1,+1\}^N} \mathbf{s}^T L\mathbf{s} \qquad \text{under the constraints } \begin{cases}\mathbf{s}^T\cdot\mathbf{s} & = n_1+n_2 = N\\ \mathbf{s}^T\cdot\mathbf{1} & = n_1-n_2\end{cases} \tag{12.7}$$

Note that $\mathbf{s}\in\{-1,+1\}^N$ by definition. Unfortunately, the optimisation 12.7 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 (12.7) 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:

$$L\mathbf{s}^\star = \lambda\mathbf{s}^\star + \mu\mathbf{1} \tag{12.8}$$

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:

$$\frac{\mu}{\lambda} = -\frac{n_1-n_2}{N} \tag{12.9}$$

We then solve equation (12.8) 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

$$\mathbf{v} := \mathbf{s}^\star + \frac{\mu}{\lambda}\mathbf{1}. \tag{12.10}$$

This is the main result. It tells us that, in fact, we are simply looking for an eigenvector of the Laplacian $\mathbf{v}$ and its associated eigenvalue λ. In words: the solution to this (relaxed) optimisation problem is given by an eigenvector as well as the corresponding eigenvalue of the Laplacian matrix.

The remaining question is now: *which eigenvector is it?* If we plug this result into the cost function (12.6), 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

$$\begin{aligned}\mathbf{v}^T\mathbf{v} &= \mathbf{s}^{\star T}\mathbf{s}^\star + 2\frac{\mu}{\lambda}\mathbf{1}^T\mathbf{s}^\star + \frac{\mu^2}{\lambda^2}\mathbf{1}^T\mathbf{1} \\ &= N - 2\frac{(n_1-n_2)^2}{N} + \frac{(n_1-n_2)^2}{N} = \frac{(n_1+n_2)^2-(n_1-n_2)^2}{N} = 4\frac{n_1 n_2}{N}.\end{aligned}$$

So 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 (12.10) 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 $\mathbf{v}_1$.

Hence the lowest value of $C^\star$ is achieved by choosing the second (i.e., the first non-trivial) eigenvector $\mathbf{v}_2$ with eigenvalue λ_2, and the minimal value of the cost function C is:

$$C^\star_{\min} = \lambda_2\frac{n_1 n_2}{N}$$

This is the classic result by Miroslav Fiedler, who obtained it in 1973[3].

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 $\lambda_2 \ll$ (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:

$$\mathbf{s}^\star = \mathbf{v}_2 + \frac{n_1-n_2}{N}\mathbf{1}, \quad \mathbf{s}^\star \in \mathbb{R}^N.$$

[3] Algebraic connectivity of graphs, Miroslav Fiedler, Czechoslovak Mathematical Journal 23, 298-305 (1973)

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 $\mathbf{s}^\star$:

$$\max_{\mathbf{s}\in\{-1,+1\}^N} \mathbf{s}^T \cdot \left(\mathbf{v}_2 + \frac{n_1-n_2}{N}\mathbf{1}\right) = \max_{\mathbf{s}\in\{-1,+1\}^N} \mathbf{s}^T\mathbf{v}_2 + \frac{(n_1-n_2)^2}{N}, \tag{12.11}$$

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 (12.11) of the discrete solution.

4.2.1. ***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.

Graph partitions based on the normalised Laplacians and spectral clustering:
It is important to mention here that although our derivation has been based on the *combinatorial* Laplacian L, similar algorithms based on the *normalised* Laplacians $L_{\text{rw}}, L_{\text{sym}}$ (defined in Section 3) can be trivially derived. 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} directly addresses the problem of graph partitioning that the combinatorial Laplacian L considers, but 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 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; see von Luxburg, *A tutorial on spectral clustering,* Statistics and Computing 17, 395–416 (2007) for further details.

4.3. 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 its essence, Modularity is a cost function that has a similar starting point to 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

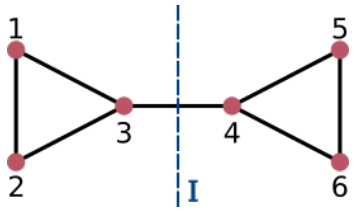


Figure 12.11. A simple example to illustrate Modularity. The blue dashed line is a good clustering split.

simple graph given in Figure 12.11 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 simply a cost function for a given graph clustering. In the simple graph in Figure 12.11, 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 12.11) 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.

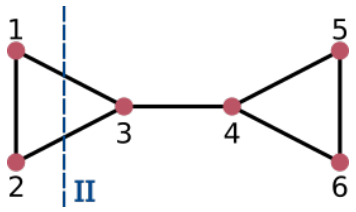


Figure 12.12. A simple example to illustrate Modularity. The blue dashed line is not a good clustering split.

In contrast, if we now consider a 'worse' split (denoted with the number II), shown by the blue dashed line in Figure 12.12, 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 $H^T A H$) will reduce the number of edges between clusters (off diagonal elements of $H^T A H$).

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* :

$$Q = \frac{1}{2E}\,\mathrm{Tr}\left[H^T \underbrace{\left[A - \frac{1}{2E}\mathbf{d}\mathbf{d}^T\right]}_{Z} H\right]$$

The matrix Z is typically known as the Modularity matrix. Now, the aim of the Modularity method is simply to maximise the modularity Q over all membership matrices H:

$$\max_{H} Q. \tag{12.12}$$

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 (12.12) 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* (it was developed at UC Louvain in Belgium) and has become the industry standard and used very broadly in many areas of data science. It was proposed by Vincent Blondel and his postdocs and students[4] in 2008.

[4]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).

5. Dimensionality reduction using graphs

sec:graph$_n$*ldr*

In Section 4 we saw how graph-based approximations of the dataset allow us to perform clustering in the unsupervised setting, especially when the clusters are not separated by linear boundaries. In this section we show how graphs enable us to perform the other common task in unsupervised learning — dimensionality reduction — in nonlinear settings.

In Chapter 11 we discussed one of the simplest methods of dimensionality reduction, namely principal component analysis (PCA), that attempts to project the (centered) data matrix $\mathcal{X}_{N\times p}$ using a projection matrix $W_{p\times k}$ onto a basis of dimensionality $k < p$, i.e. $\widehat{\mathcal{X}} := \mathcal{X}W$, such that the total reconstruction error upon re-projecting $\widehat{\mathcal{X}}$ to the original feature space $\mathcal{X}$, i.e. $\|\mathcal{X} - \widehat{\mathcal{X}}W^T\|_{\mathrm{F}}^2$ is minimised[5]. This optimisation problem yields the optimal solution of W to be the top-k eigenvectors of the covariance matrix of X stacked column-wise, denoted by V_k (see the unnumbered equation after (11.8)), and the optimal projection $\widehat{\mathcal{X}}$ to be the (scaled) eigenvectors of the (centered) Gram matrix of X (see (11.9)). Naturally, taking a linear projection is learning a linear transformation of the data, which is a limitation of PCA. We further saw that we can apply the kernel trick to extend PCA as a method of learning nonlinear transformations, by using eigenvectors of the (centered) kernel matrix $K_{N\times N}$ instead of the (centered) Gram matrix (see (11.14)), termed as kernel PCA. However, the success of kernel PCA relies heavily on making the right choice of the kernel function and its parameters.

We will now briefly 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 k, embedded in a high-dimensional space of dimensionality $p > k$ – 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 k, albeit the dimensionality of the "ambient space" in which the manifold lives can be $p > k$.

> The dimensionality k of a k-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 out of the data can approximate the structure of the manifold on which it lies. Consequently, we can use the techniques of graph construction from Section 4 to obtain a discrete approximation of the underlying manifold of the data. This is why performing nonlinear dimensionality reduction using the manifold assumption is also referred to as

[5]Recall that the centered data matrix can be obtained from the original data matrix X by pre-multiplication with a centering matrix C of size $N \times N$: $\mathcal{X} = CX$.

manifold learning. Figure 12.13 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.

5.1. 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

$$C := \frac{1}{2} \sum_{i,j} \|\hat{\mathbf{x}}^{(i)} - \hat{\mathbf{x}}^{(j)}\|_2^2 A_{ij},$$

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 4. 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 k 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) $\widehat{X}^T D \widehat{X} = \mathbf{I}$ (where 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**[6].

5.2. 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.

> 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

[6] Belkin, M., & Niyogi, P. (2003). Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation, 15(6)*, 1373-1396.

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 — 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.

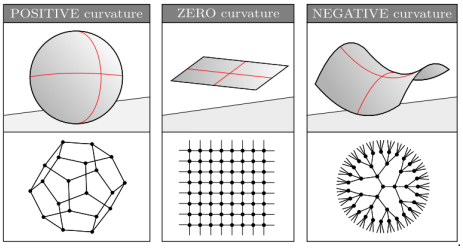


Figure from Devriendt, K., & Lambiotte, R. (2022). Discrete curvature on graphs from the effective resistance. *Journal of Physics: Complexity, 3(2)*, 025008.

In Chapter 11, 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-k eigenvectors of the covariance matrix encoding the total variance explained (see the discussion after Figure 11.1). 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[7], i.e. 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[8]. 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**[9]. One can summarise the full algorithm as follows:

(1) Construct a (weighted) graph G (using techniques in Section 4 like k-nearest neighbours).

(2) Compute all pairwise geodesic distances in G (using Dijkstra's algorithm for graphs with positive-valued edge weights).

[7]Torgerson, W. S. (1952). Multidimensional scaling: I. Theory and method. *Psychometrika, 17(4)*, 401-419.

[8]Ghojogh, B., Ghodsi, A., Karray, F., & Crowley, M. (2020). Multidimensional Scaling, Sammon Mapping, and Isomap: Tutorial and Survey. *arXiv preprint arXiv:2009.08136*.

[9]Tenenbaum, J. B., Silva, V. D., & Langford, J. C. (2000). A Global Geometric Gramework for Nonlinear Dimensionality Reduction. *Science, 290(5500)*, 2319-2323.

(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: $K_{ij} := -D_{ij}^2$)[10].

Figure 12.13 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.

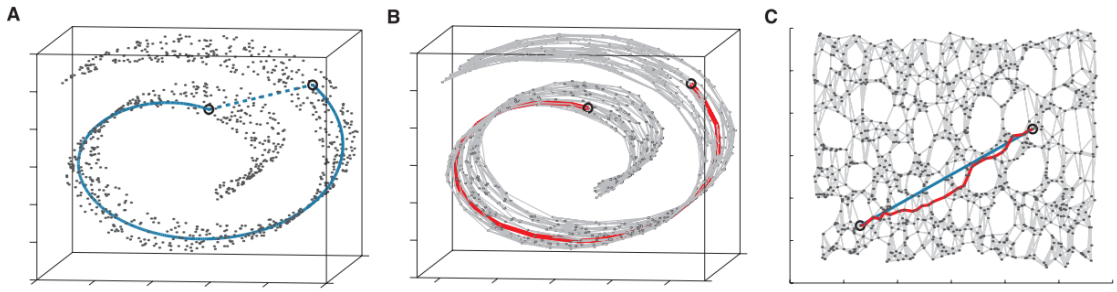


Figure 12.13. Using the Swiss roll dataset ($N = 1000$) to demonstrate the use of Isomap in performing nonlinear dimensionality reduction. Figure from Tenenbaum, Silva, & Langford (2000). (A) The Swiss roll dataset is not Euclidean as Euclidean distances do not capture the intrinsic non-linearity of 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.

6. 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:

[10]Recall from (11.16) that to "center" a kernel matrix it must be *doubly*-centered, i.e. by pre- *and* post-multiplication with a centering matrix C of size $N \times N$: $\mathcal{K} = CKC$.

- **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[11], 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

[11] Page, L., Brin, S., Motwani, R. and Winograd, T. (1999) "The PageRank Citation Ranking: Bringing Order to the Web". Technical Report. Stanford InfoLab.

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):

$$\mathbf{c}_{\mathrm{PR}\ t+1} = \alpha \left(AD^{-1}\right) \mathbf{c}_{\mathrm{PR}\ t} + (1-\alpha)\,\frac{1}{N}\mathbf{1} \tag{12.13}$$

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 (12.13), which is obtained by solving for $\mathbf{c}_{\mathrm{PR}}$ in:

$$\mathbf{c}_{\mathrm{PR}} = \alpha \left(AD^{-1}\right) \mathbf{c}_{\mathrm{PR}} + (1-\alpha)\,\frac{1}{N}\mathbf{1}, \quad \alpha \in [0,1]$$

As you can immediately identify, the process (12.13) corresponds to a weighted *random walk* on the graph (see Section 3 above) where the random walker follows the transition matrix of the graph $M = AD^{-1}$ 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.

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 4, 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 2.
- **Classification**: The task of predicting the class (or label) of a categorical variable (see chapter 1).
- **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 4, 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 3, 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 11.
- **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 8, section 6).
- **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 4, section 2).
- **Generalisation power (or generalisability)**: Model's ability to perform well on unseen data (see chapters 1 and 2).
- **Graph**: Representation of the relationships in a dataset given by edges connecting nodes (the data points), see chapter 12.

- **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 8).
- **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 2 and 8).
- **Likelihood**: Probability of the data under the model, see e.g. chapter 2, 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 2).
- **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 2.
- **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 unregularized parameters), see chapter 2.
- **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 2.
- **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 4, section 2).
- **Regression**: Task consisting in predicting the value of a quantitative variable, called outcome variable, given an input, called predictor variable (see chapter 1).
- **Regularization**: It refers generally to terms that help fix the issue of overfitting by keeping under control the values that the inferred parameters can assume (e.g. by imposing a penalty). Examples of regularization are the Ridge and Lasso regularization, which rely on penalty terms that, respectively, penalize large parameter values or pushes small values towards zero, see chapter 2.
- **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 1).
- **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 3, section 2).

- **Training set**: Part of the dataset that is used to *train* (i.e. learn) the model's parameters (see chapter 3, 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 4, 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 1).
- **Validation set**: Part of the dataset that is used in cross-validation to evaluate the model's performance to select the hyperparameters (see chapter 3, 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 2.