

PATTERN RECOGNITION AND NEURAL NETWORKS

Ludmila Kuncheva

Preface

I have tried everything – serious textbooks, not-so-serious textbooks, Powerpoint slides that sing and dance, live coding in front of the class, timed exercises, non-timed exercises, quizzes, even singing “*Gaudeamus igitur*” on the last lecture. I did!

Alas, lecture attendance dwindles exponentially every year. And when I mark the exam, I close my eyes and pretend that you call the ‘spade’ a ‘spade’ and not a ‘screwdriver’. I squint at your scribbles, tilt my head to one side, and – look! – it’s rather ‘spadiver’,... ‘spader’,...‘spade’, really! Hey, and the module evaluation forms filled by you at the end of the semester are supposed to sing me praises! Ha-ha-ha, she says wryly.

So, here goes my next attempt. I expect some knowledge of maths and MATLAB. The language is deliberately colloquial, trying to get closer to you. This is *not* how you write scientific papers and books! Work your way through the text and exercises, and you will pass the module!

Ludmila Kuncheva
Bangor, November 3, 2019

Contents

1	Introduction to Pattern Recognition	11
1.1	A little history	11
1.2	Classes, labels, and features	16
1.2.1	Classes and labels	16
1.2.2	Features	18
1.3	Data set	20
1.3.1	Definition and notations	20
1.3.2	The famous iris data set	22
1.3.3	Data set types	23
1.4	The pattern recognition cycle	25
2	Basics	27
2.1	Classifiers, Discriminant Functions, and Classification Regions	27
2.1.1	Classifier and discriminant functions	27
2.1.2	Classification regions	28
2.1.3	One-dimensional data	29
2.1.4	Binary classification	32
2.1.5	Plotting classification regions (MATLAB)	34
2.2	Evaluation of a classifier	36
2.2.1	The danger of overtraining	36

2.2.2	Training and testing protocols	37
2.2.3	The confusion matrix	41
2.3	ROC curves	47
2.3.1	A bit of history	47
2.3.2	False positives, false negatives and the two error types	47
2.3.3	The ROC curve	49
2.4	Imbalanced classes	54
2.4.1	What is class imbalance?	54
2.4.2	Classifier performance measures for imbalanced classes	55
2.5	A probabilistic view	59
3	Classifiers	63
3.1	The Nearest Mean Classifier (NMC)	64
3.1.1	How it works	64
3.1.2	Classification boundary of a 2-class NMC in 2D	65
3.1.3	Programming the NMC	68
3.1.4	Voronoi diagrams	71
3.2	The Linear Discriminant Classifier (LDC)	72
3.2.1	NMC is actually an LDC!	73
3.2.2	Linear boundaries	75
3.2.3	LDC in different dimensions	79
3.3	Rule-based classifiers	82
3.3.1	If-then classifiers	82
3.3.2	The ZeroR classifier	84
3.3.3	The OneR classifier	85
3.4	The Nearest Neighbour classifier (1-nn and k-nn)	90
3.4.1	Distances	91
3.4.2	1-nn and k-nn	92
3.4.3	k -nn in MATLAB	93
3.5	Decision tree classifiers	96

3.5.1	What is a decision tree classifier?	96
3.5.2	Why are decision trees good?	97
3.5.3	Training of a decision tree classifier	98
3.6	The Support Vector Machine (SVM) classifier	101
3.7	Classifier ensembles	104
3.7.1	Why will classifier ensembles work?	104
3.7.2	Bagging	105
3.7.3	Boosting	107
3.7.4	Random Subspace	108
3.7.5	Random Forest	110
4	Feature Selection	113
4.1	Redundant, irrelevant and useful features	113
4.2	A taxonomy of approaches	115
4.3	Feature extraction	116
4.4	Feature selection	117
4.4.1	Univariate feature selection	117
4.4.2	Multivariate feature selection	118
4.4.3	What criterion do we use to evaluate a given feature set?	122
5	Clustering	125
5.1	Introduction to clustering	125
5.2	Hierarchical clustering and the single linkage method	128
5.2.1	The generic agglomerative clustering algorithm	128
5.2.2	Single linkage	128
5.2.3	Determining the number of clusters for ag- glomerative clustering	131
5.2.4	Dendrograms	134
5.2.5	The chain effect of single linkage	136
5.2.6	Mean (centroid) linkage	136
5.2.7	MATLAB code and a caveat	139

5.3	Non-hierarchical clustering: k-means	142
5.3.1	Preliminaries	142
5.3.2	The famous k-means algorithm	143
5.3.3	The criterion function J_e	145
6	Neural Networks	155
6.1	A brief history of neural networks	155
6.1.1	The early ages	155
6.1.2	The second wave	156
6.1.3	The blossom of deep learning	157
6.2	Structure and elements of a NN	158
6.2.1	Neurons: real and artificial	158
6.2.2	The Threshold Logic Unit (TLU)	163
6.3	The Perceptron	165
6.3.1	A bit of history	165
6.3.2	The famous perceptron training algorithm	165
6.3.3	The perceptron convergence theorem	168
7	MLP, RBF, and SOM	171
7.1	Multi-Layer Perceptron (MLP)	171
7.1.1	Two perceptrons together	171
7.1.2	Structure of MLP	172
7.1.3	The error backpropagation algorithm	175
7.2	Radial basis functions networks (RBF)	179
7.2.1	The activation function	179
7.2.2	Structure and operation of RBF	181
7.2.3	Training of RBF	183
7.3	Self Organising Maps (SOM)	187
7.3.1	Definition and examples	187
7.3.2	Training of SOMs	190

8 Deep Learning NNs	201
8.1 Some definitions	201
8.2 Applications of DL	202
8.3 Structure and operation of DLs	209
8.4 Which CNN is the best?	215
Appendix A Maths you should know	223
Index	xxx

Chapter 1

Introduction to Pattern Recognition

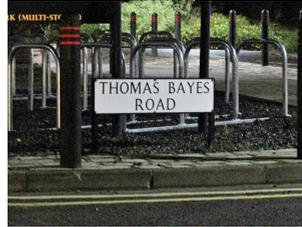
Pattern recognition is about assigning labels to objects. If the object is an image taken with a phone camera, the class label can be ‘cat’ or ‘not a cat’. Given enough training data, we can get a machine to assign labels to the objects by learning patterns which we - humans - can not break into steps, verbalise or explain completely. We rely on the machine to *learn* from the given examples and *generalise* on unseen objects. Magic!

1.1 A little history

Once upon a time, before there were pianos or fire extinguishers, a humble wizard lived in an archetypal Middle-England town. He made his living as a chapel minister. His name was Thomas Bayes. Little did he know that one of his two published works will become the most powerful spell of the Probability Kingdom, known as the

Bayes Theorem. Little did he know that, three centuries later, he will be proclaimed “more important than Marx and Einstein put together”.¹

Or that there will be a street
named after him...^a



^aPhoto courtesy of Tom Dabner,
CS graduate 2015.

Take the story of Mike Lynch, the Bayesian millionaire.



In 1991 Mike Lynch founded a company called “Autonomy”, originally to help Essex police force.

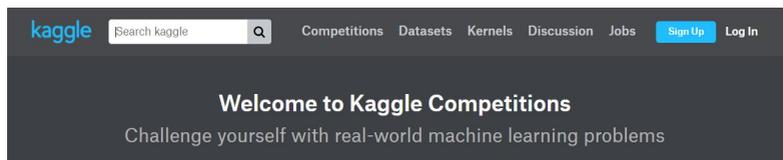
Quake in your boots, Essex criminals! Autonomy will read the number plate on your getaway car, and will promptly identify you by your fingerprint. Lo and behold, in 2001 Autonomy was estimated at £4.7 billion! And only a relative nanosecond later (in the grand scheme of things), in 2011, Autonomy was bought by Hewlett-Packard for \$11 billion.

Recall the NETFLIX contest on user preference prediction.² The first competitor whose prediction of user movie preferences reaches accuracy that is 10% or more higher than Netflix’s own algorithm will receive a prize of one million dollars. BellKor’s Pragmatic Chaos team received the prize in 2009 achieving improvement of 10.06%.

¹Telegraph Magazine, 3 February, 2001.

²<http://www.netflixprize.com/>

These days competitions and challenges abound. KAGGLE hosts one such site³. Some of the challenges carry monetary rewards, other only kudos, and there are some which are mere playground contests.



Historically, machine learning was mostly developed in the USA while pattern recognition was more of an European enterprise. At the times when there was no Internet, machine learning focused on the theory of inference while pattern recognition advanced in the area of image processing. As a result, even though we are solving (nearly) the same problem, we speak slightly different dialects of the same language (see Figure 1.1).

Take a look at Table 1.1 to see that the continents are somewhat geographically-protective of their respective ‘babies’. The leading conference on machine learning ICML (International Conference on Machine Learning)⁴ has been held 67% times in North America while the corresponding leading conference on pattern recognition ICPR (International Conference on Pattern Recognition)⁵ exhibits more ‘internationalism’. The majority of the editions of the conference were held in Europe (42%) but with more generous outreach to North America (33%) and elsewhere.

Another very high-calibre conference on machine learning which is even more territorially restricted to North America is the famous

³<https://www.kaggle.com/competitions>

⁴https://en.wikipedia.org/wiki/International_Conference_on_Machine_Learning

⁵<https://www.iapr.org/conferences/schedule.php>

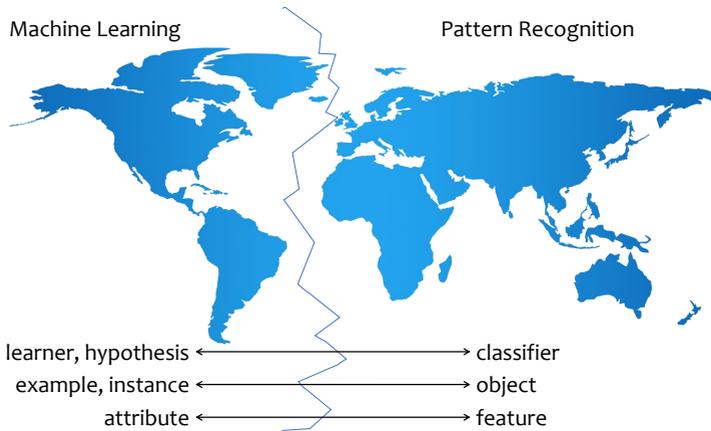


Figure 1.1: Language discrepancies between Pattern Recognition and Machine Learning.

NIPS conference (Neural Information Processing Systems).⁶ Of its 32 editions, NIPS has left North America twice, both times choosing Spain for its home-from-home.

Table 1.1: Locations of the major conferences ICML (machine learning) and ICPR (pattern recognition) over the years.

Year	ICML	ICPR
1973	–	 1 Washington D.C., USA
1974	–	 2 Copenhagen, Denmark
1976	–	 3 Coronado, USA
1978	–	 4 Kyoto, Japan
1980	 1 Pittsburgh, USA	 5 Miami, USA

⁶https://en.wikipedia.org/wiki/Conference_on_Neural_Information_Processing_Systems

1982	–	 6 Munich, Germany
1983	 2 Monticello, USA	–
1984	–	 7 Montreal, Canada
1985	 3 Skytop, USA	–
1986	–	 8 Paris, France
1987	 4 Irvine, USA	–
1988	 5 Ann Arbor, USA	 9 Rome, Italy
1989	 6 New York, USA	–
1990	 7 Austin, USA	 10 Atlantic City, USA
1991	 8 Illinois, USA	–
1992	 9 Aberdeen, UK	 11 Hague, Netherlands
1993	 10 Massachusetts, USA	–
1994	 11 New Jersey, USA	 12 Jerusalem, Israel
1995	 12 California, USA	–
1996	 13 Bari, Italy	 13 Vienna, Austria
1997	 14 Tennessee, USA	–
1998	 15 Wisconsin, USA	 14 Brisbane, Australia
1999	 16 Bled, Slovenia	–
2000	 17 Stanford, USA	 15 Barcelona, Spain
2001	 18 Massachusetts, USA	–
2002	 19 Sydney, Australia	 16 Quebec City Canada
2003	 20 Washington DC, USA	–
2004	 21 Banff, Canada	 17 Cambridge, UK
2005	 22 Bonn, Germany	–
2006	 23 Pittsburgh, USA	 18 Hong Kong
2007	 24 Corvallis, USA	–
2008	 25 Helsinki, Finland	 19 Tampa, USA
2009	 26 Montreal, Canada	–
2010	 27 Haifa, Israel	 20 Istanbul, Turkey
2011	 28 Bellevue, USA	–
2012	 29 Edinburgh, UK	 21 Tsukuba, Japan
2013	 30 Atlanta, USA	–
2014	 31 Beijing, China	 22 Stockholm, Sweden
2015	 32 Lille, France	–
2016	 33 New York, USA	 23 Cancun, Mexico
2017	 34 Sydney, Australia	–

2018		35 Stockholm, Sweden		24 Beijing, China
2019		36 California, USA	–	–

Europe	8/36	22%	10/24	42%
NA24	36	67%	8/24	33%
Other	4/36	11%	6/24	25%

But, all in all, in this era of intense communication and travel, science no longer observes geographic boundaries. The terminology dialects blend into a unified jargon, software is universally available, and researchers happily share ideas and resources. Systems for car number plate recognition might have been a revolution 40 years ago, but nowadays pattern recognition and machine learning are aiming much higher, demonstrating a remarkable level of human-like intelligence⁷.

What is pattern recognition used for nowadays?



So many things! Iris recognition, driverless cars (controversial!), facial emotion detection, mail sorting, network security, cancer diagnostics, and many more.

Pattern recognition is a close relative to at least the following disciplines: Machine Learning, Artificial Intelligence, Neuroscience, Data Mining, and Statistics.

1.2 Classes, labels, and features

1.2.1 Classes and labels

‘Classes’ in pattern recognition are the groups/types/categories of objects we wish to recognise. For example, a mobile photo may be

⁷<https://www.theatlantic.com/technology/archive/2016/03/the-invisible-opponent/475611/>

assigned to a class ‘cat’ or a class ‘no cat’. In this case we have a *binary* classification problem and the set of class labels is {cat, no cat}. Then *multi-class* classification is when we have more than two classes. For example, consider a photo with a single animal in it. We use the following set of possible labels: {cat, dog, rabbit, other}, where class ‘other’ contains every animal that is not cat, dog or rabbit.

Throughout this book we will assume that the classes are *mutually exclusive*. This means that one object can belong to one of the classes only. This is true for both the binary example above (cat, no cat) and the multi-class example. But there could be other situations where the classes are not mutually exclusive. For example, consider a single photo that contains more than one of the animals. If there are both a cat and a dog, the photo (the object) should be labelled in both classes. Pattern recognition problems in which the classes are not mutually exclusive are more complicated. They are termed *multi-label* problems. Note the subtle difference in terminology: multi-class versus multi-label.

The multi-class problems which we will be solving here will be not be multi-labelled; the classes will be mutually exclusive. For example consider recognition of handwritten digits. Each object is an image of size 28-by-28 pixels containing one digit. The class will be what the writer intended. Thus we can’t have an image that is both in class, say, 1 and 7. There may be significant doubt about the true class of the object as it may look very different from the intended digit. Nonetheless, there is only one ‘ground truth’. An example of 16 objects from the handwritten digit dataset MNIST is shown in Figure 1.2.⁸

Each object available to us within the given data set will have a label. The set of possible class labels will be denoted by capital

⁸<http://yann.lecun.com/exdb/mnist/>

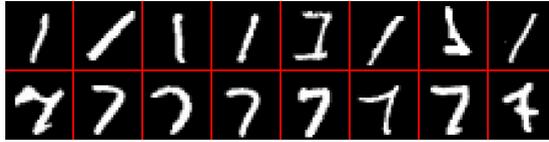


Figure 1.2: An example of two of the classes of handwritten digits from the MNIST dataset (top row 1, bottom row 7).

omega, and the labels within, by little omegas:

$$\Omega = \{\omega_1, \dots, \omega_c\}. \quad (1.1)$$

We will denote the number of classes by c .

Each object in the data set will be associated with a *true* label, an element of Ω .⁹

1.2.2 Features

Each object is described by features (also called attributes). The features may be measured in different ways; they may be continuous-valued, discrete, ordinal, nominal, etc. The set of features describing an object may contain a mixture of types of features. Take for example the arrhythmia data set from the UCI Machine Learning Repository¹⁰. [1] The task is to recognise one of 14 types of heart arrhythmia. Features include gender (nominal variable with two categories: male and female¹¹) as well as measurements from the electrocardiogram of the person (quantitative variables).

⁹‘Object’, ‘example’ and ‘instance’ are synonyms in the context of pattern recognition.

¹⁰<https://archive.ics.uci.edu/ml/index.php>

¹¹Yes, I know. But, come on, we are talking here about the biological gender, not the self identity.

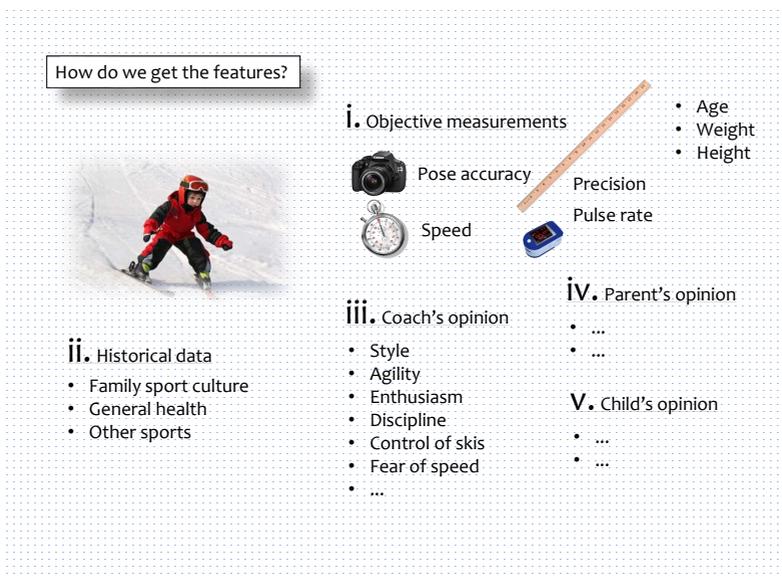


Figure 1.3: Thinking about the various possible types of features in the task of predicting whether a given young athlete will grow into an Olympic champion.

1.3 Data set

1.3.1 Definition and notations

A *data set* Z is a set of objects represented by their features, i.e.,

$$Z = \{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_N\} \subset \mathbb{R}^n. \quad (1.3)$$

We shall assume that the data set is arranged so that each row contains the feature values for a single object. If Z is a *labelled* data set, each object has a class label from the set Ω . The generic form

of a labelled data set is shown below

$$Z = \underbrace{\begin{bmatrix} z_{1,1} & z_{1,2} & \cdots & z_{1,n} \\ \vdots & \vdots & \cdots & \vdots \\ z_{j,1} & z_{j,2} & \cdots & z_{j,n} \\ \vdots & \vdots & \cdots & \vdots \\ z_{N,1} & z_{N,2} & \cdots & z_{N,n} \end{bmatrix}}_{\text{data}} \underbrace{\begin{bmatrix} y_1 \\ \vdots \\ y_j \\ \vdots \\ y_N \end{bmatrix}}_{\text{labels}} \quad (1.4)$$

Here $z_{j,k}$ is the value of feature k for object \mathbf{z}_j , and $y_j \in \Omega$ is the label of the object. An example of a two-dimensional data set is shown in Figure 1.4.

Data (Z)		Label
x_1	x_2	y
4	7	2
1	1	1
6	2	1
8	8	3
5	10	2
3	4	1
5	2	3
6	4	3

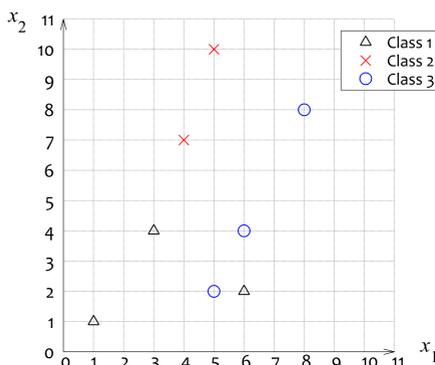


Figure 1.4: A labelled data set with $N = 8$ objects, $c = 3$ classes and $n = 2$ features.

The notations which will be used throughout this book for the number of classes, number of features and number of objects are:

C classes n features N objects.

1.3.2 The famous iris data set

Data sets coming from different domains can be of different sizes. But all is relative! In the 1970s, a data set with $N = 150$ objects was considered decently large, and dimensionality of $n = 150$ or so features was once termed “embarrassingly large”. [14] Today we laugh in the face of these numbers! The number of data points in modern data sets is in order of millions, and the number of features, in order of thousands. But we still like to be able to train and test our algorithms on toy or benchmark data sets.

The UCI Machine Learning repository contains toy data sets such as the famous *iris* data set [9, 3]. The iris data set was collected by the American botanist Edgar Anderson and subsequently analysed by the English geneticist and statistician Sir Ronald Aylmer Fisher in 1936 [9]. The iris data set has become one of the iconic hallmarks of pattern recognition and has been used in thousands of publications over the years.

The iris data still serves as a prime example of a ‘well behaved’ data set. There are three balanced classes, $c = 3$, each represented with a sample of 50 objects, hence $N = 150$. The classes are species of the iris flower: *setosa*, *versicolor* and *virginica*.

The four features describing an iris flower ($n = 4$) are sepal length, sepal width, petal length and petal width. The classes form neat elliptical clusters in the four-dimensional space. Class *setosa* is clearly distinguishable from the other two classes.



1.3.3 Data set types

Data sets are very different for different problems. In bioinformatics, for example, tens of thousands features are measured.



This is the case with gene expression data.

In such data sets the number of objects (patients suffering from some disease) is relatively small. This type of data sets are called *wide* data sets.

Compare this with *big* data sets. Big Data arise in areas such as computer vision, document classification, speech recognition, internet searches, and more. Big Data came into fashion recently, owing to the technological advances of the past few decades. Nowadays we have the means to source, store and process such data sets. An example is the famous ImageNet data set.¹²[6] ImageNet contains over 14 million images containing various objects. Using a clever crowdsourcing game developed by Luis von Ahn of Carnegie Mellon University, USA, each image has been labelled with the dominant objects within, and over a million of images have been hand-annotated with the corresponding bounding boxes. A sample of images from ImageNet is shown in Figure 1.5.

The number of classes c in ImageNet is over 20 thousand! A typical class contains several hundred images. Such a massive classification task is not in the realm of classical pattern recognition because the challenges are vastly different from those of the 20th century pattern recognition. Some of today's big data computer vision challenges are

- There is a colossal number of classes with intricate relationships between them.

¹²<https://en.wikipedia.org/wiki/ImageNet>



Figure 1.5: A sample from the ImageNet data set.

- The problems are computationally-hungry in terms of storage, processing power, and time.
- Understanding the relationship between the visual presentation of the image (pixels) and its semantic content is still missing and relegated to deep learning neural networks to figure out. Thus the classification models have to be complex, versatile, and trainable.

In the past, much effort has been devoted to finding ingenious ways to re-use the (small) data set so that the classifier is trained without overfitting. All those clever ways of error estimation, density modelling, feature selection... Whoosh! All goes out of the window. Enter parallel computing, GPUs, giant neural network architectures with beautiful and exotic processing elements!

1.4 The pattern recognition cycle

The ‘pattern recognition cycle’ starts with Real World and ends with Real World as shown in Figure 1.6.

The two major strands are Unsupervised Learning (Unsupervised Pattern Recognition) and Supervised Learning (Supervised Pattern Recognition). The task of unsupervised pattern recognition is to identify structure in the data. Usually this is done through applying a clustering method. The output is a partition of the data into groups, and this is what we return to the user. In supervised pattern recognition we use the data and the labels to build a classifier which can predict a class label for any previously unseen object. The classifier is the product which we return to the user.

Let’s see how it is done!

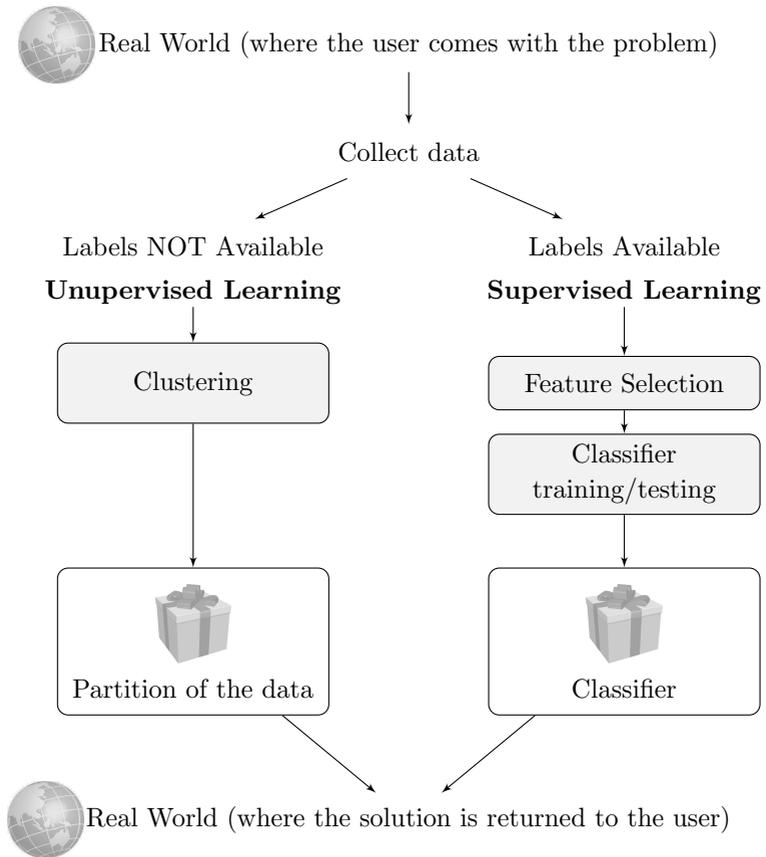


Figure 1.6: The Pattern Recognition Cycle: from real world to real world.

Chapter 2

Basics

2.1 Classifiers, Discriminant Functions, and Classification Regions

2.1.1 Classifier and discriminant functions

A *classifier* is any function, method or algorithm that assigns a class label to any given object. Formally, if $\mathbf{x} \in \mathbb{R}^n$ is an object, and $\Omega = \{\omega_1, \dots, \omega_c\}$ is the set of class labels, a classifier is a mapping:

$$D : \mathbb{R}^n \rightarrow \Omega. \quad (2.1)$$

In other words, D assigns an element of Ω to a given object \mathbf{x} .

A classifier can be specified equivalently with a set of *discriminant functions*. Think of these functions as ‘support’ for the classes. Each object \mathbf{x} will receive a collection of c values, one for each class. The class label assigned to \mathbf{x} should be the one with the largest support. These c values come from the discriminant functions:

$$g_i : \mathbb{R}^n \rightarrow \mathbb{R}, \quad i = 1, \dots, c.$$

⊕ ⊕ ⊕ **Example 2.1.1**

Consider a 2-dimensional space and $c = 3$ classes. Let the three discriminant functions be:

$$\begin{aligned} g_1(\mathbf{x}) &= 4, \\ g_2(\mathbf{x}) &= 3x_1^2 - x_2 + 5, \quad \text{and} \\ g_3(\mathbf{x}) &= x_1 x_2 + 7. \end{aligned}$$

What class label will this classifier assign to $\mathbf{x} = [-1, 2]^T$?

Solution: Calculate the three discriminant functions for the given \mathbf{x}

$$\begin{aligned} g_1(\mathbf{x}) &= 4. \\ g_2(\mathbf{x}) &= 3 \times (-1)^2 - 2 + 5 = 6. \\ g_3(\mathbf{x}) &= (-1) \times 2 + 7 = 5. \end{aligned}$$

Since the largest discriminant function is g_2 , the class label assigned to \mathbf{x} will be ω_2 .

⊖ ⊖ ⊖

2.1.2 Classification regions

The discriminant functions determine uniquely the *classification regions* of the classifier. The points within a region should be labelled in the corresponding class. These regions may have any shape and also may consist of disjoint parts of the space.

Figure 2.1 shows the classification regions for the classifier defined by the three discriminant functions in Example 2.1.1.

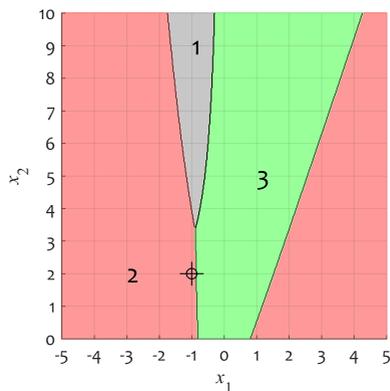


Figure 2.1: Classification regions for Example 2.1.1. The point $\mathbf{x} = [-1, 2]^T$ is marked with a target symbol.

The classification regions are defined in the space where the data lives, \mathbb{R}^n . For one-dimensional data, the classification regions are intervals on the real line \mathbb{R} .

2.1.3 One-dimensional data

When the data is one-dimensional, we can actually *see* the discriminant functions. They will be some functions of the feature $y = g(x)$. Then if there are c classes, we can determine the classification regions by comparing the c values $g_1(x), \dots, g_c(x)$. And we have one whole dimension to plot the functions! Take a look at Figure 2.2 and the example below.

⊕ ⊕ ⊕ Example 2.1.2

Consider a one-dimensional problem where the data come from

three classes. We have a classifier defined with the following discriminant functions:

$$g_1(x) = 3x - 2, \quad g_2(x) = 4, \quad g_3(x) = -2x^2 - 2x + 6.$$

Plot the discriminant functions for $x \in [-2, 4]$ and identify the classification regions.

Solution: Plot the functions against the values of x spanning the interval $[-2, 4]$. Observe which discriminant function is the largest (highest curve) for each value of x . To find the boundaries of the intervals which will be the classification regions, we must solve simultaneously the equations of the respective discriminant functions. For example, all points to the right of A' must be labelled as class 1 because g_1 will be the largest discriminant function for all those x s. Point A' is obtained by projecting the intersection point A onto x . (If you are wondering, this amounts to dropping the y -coordinate of A .) To find A' we must solve $g_1(x) = g_2(x)$:

$$3x - 2 = 4, \quad \text{hence} \quad x = 2.$$

Next, solve $g_2(x) = g_3(x)$:

$$-2x^2 - 2x + 6 = 4, \quad -2x^2 - 2x + 2 = 0, \quad x^2 + x - 1 = 0.$$

The discriminant of this quadratic equation is $D = 1^2 - 4(-1) = 5$, and the solutions are

$$s_1 = \frac{-1 + \sqrt{5}}{2} = 0.6180 \quad \text{and} \quad s_2 = \frac{-1 - \sqrt{5}}{2} = -1.6180.$$

Denote the respective points by B' and C' , respectively, and the boundaries of the classification regions by B and C (see the figure).

Solving the quadratic equation $g_1(x) = g_3(x)$ we obtain intersection points P' and Q'

$$3x - 2 = -2x^2 - 2x + 6, \quad -2x^2 - 5x + 8 = 0.$$

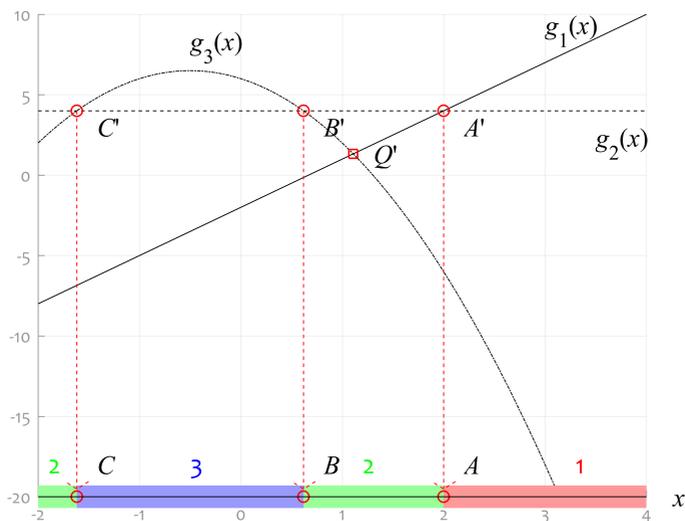


Figure 2.2: Classification regions for 1D data. The regions are obtained from the discriminant functions for the three classes as explained in Example 2.1.2.

The two solutions are

$$s_1 = \frac{5 + \sqrt{25 + 4 \times 2 \times 8}}{-4} = -3.6085, \quad s_2 = \frac{5 - \sqrt{89}}{-4} = 1.1085.$$

Then the intersection points are $P'(-3.6085, -12.8255)$ (duly fallen off the graph) and $Q'(1.1085, 1.3255)$. Spot Q' ! The regions are marked on the x -axis because this is where the data for this problem lives, on the real line \mathbb{R} . And Q' is beneath $g_2(x)$ which means that it will NOT define a classification boundary; it is simply not needed.

The thing is that, in the 1D example, through the magic of sight we can determine which intersection points are legitimate region boundaries and which are not. In higher dimensions, this will not be possible. ⊖ ⊖ ⊖

But this is not too bad because the classifier is specified *completely* by the discriminant functions, and we don't need to know or plot the classification regions. Any point that comes for classification will get its class label just by comparing the c discriminant functions.

2.1.4 Binary classification

When there are only two classes ($c = 2$, binary classification), things are even simpler! Instead of calculating and comparing two discriminant functions, we can get away with just one calculation. Let $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$ be the two discriminant functions.¹ Form

$$g(\mathbf{x}) = g_1(\mathbf{x}) - g_2(\mathbf{x}).$$

All objects for which the classifier should assign class 1 will have $g_1(\mathbf{x}) \geq g_2(\mathbf{x})$, therefore $g(\mathbf{x}) \geq 0$. And vice versa, all objects which the classifier puts in class 2 will have $g(\mathbf{x}) < 0$. So, just one discriminant function will suffice in this case. In theory, when $g(\mathbf{x}) = 0$, any of the two classes can be assigned. We have assumed here that in case of a tie, we will assign class 1.

Also, for such binary classification, there is no ambiguity as to which pair of discriminant functions determine the boundary of classification regions. Indeed, the boundary between the two classes is defined by the equation $g(\mathbf{x}) = 0$.

¹Note the boldface \mathbf{x} . This means that the object is represented by a vector with more than one dimension. That is, the number of features is $n > 1$. In the 1D example we used just x .

For a 2-dimensional binary classification problem we can calculate the equation of a linear discriminant function by choosing two suitable points in the 2D space.

⊕ ⊕ ⊕ **Example 2.1.3**

Figure 2.3 (a) shows a scatterplot of two classes in 2D. Propose a linear classification boundary which separates the two classes. (Well well, look, it is already there!) Explain how you will use this equation as a discriminant function. Plot the classification regions as in sub-plot (b).

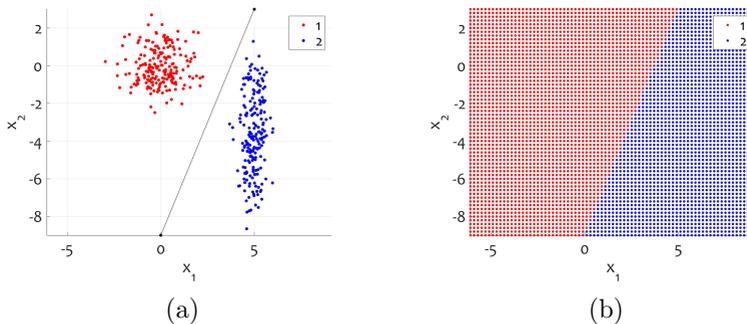


Figure 2.3: Classification boundary and classification regions for Example 2.1.3

Solution. Pick two points that define a line which separates (by eye) the two classes. Take, for example, $A(0, -9)$ and $B(5, 3)$. This is the segment plotted in sub-plot (a). Calculate the equation of the line:

$$\frac{x_1 - 0}{5 - 0} = \frac{x_2 - (-9)}{3 - (-9)}$$

$$12x_1 = 5x_2 + 45, \quad \text{hence} \quad 12x_1 - 5x_2 - 45 = 0.$$

The discriminant function will be

$$g(\mathbf{x}) = 12x_1 - 5x_2 - 45. \quad (2.2)$$

To determine which class corresponds to the ‘positive’ side of the line, pick any point in the space, for example $O(0, 0)$. Substitute the coordinates in the discriminant function (2.2). The sum is $-45 < 0$. Therefore, our classifier should assign class label 1 (red) to any point whose discriminant score is negative, and class 2 (blue) for a positive score. Any class label can be assigned for $g(\mathbf{x}) = 0$.

Now, for plotting the classification regions – see below! $\ominus \ominus \ominus$

2.1.5 Plotting classification regions (MATLAB)

$\oplus \oplus \oplus$ Example 2.1.4

Here comes the fun! Let’s plot classification regions in \mathbb{R}^2 using MATLAB. Consider a classifier given by the following discriminant functions:

$$\begin{aligned} g_1(\mathbf{x}) &= x_1^2 - x_2^2 - 6. \\ g_2(\mathbf{x}) &= 3 \sin(0.5x_1) x_2 \\ g_3(\mathbf{x}) &= 4x_1 - 2x_2 + 9. \end{aligned}$$

Plot the classification regions for this classifier for $x_1 \in [-30, 30]$ and $x_2 \in [-30, 30]$.

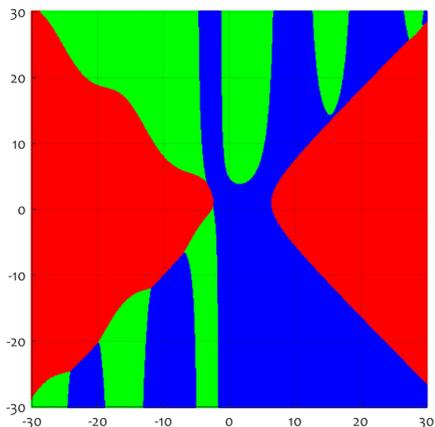
Solution.

```
1 % Generate the grid points
2 span_x = linspace(-30, 30, 500);
```

```

3 [x1,x2] = meshgrid(span_x,span_x);
4
5 % Apply the three discriminant functions to each point
6 g1 = x1.^2 - x2.^2 - 6;
7 g2 = 3*sin(x1/2).*x2;
8 g3 = 4 * x1 - 2 * x2 + 9;
9
10 % Prepare the figure
11 figure, hold on, grid on, axis equal tight
12 set(gca,'Layer','Top') % make the grid visible
13
14 % Plot the regions, class by class
15 region1 = g1 > g2 & g1 > g3; % logical variable for class 1
16 region2 = g2 > g1 & g2 > g3; % logical variable for class 2
17 region3 = g3 > g1 & g3 > g2; % logical variable for class 3
18
19 plot(x1(region1),x2(region1),'r.')
20 plot(x1(region2),x2(region2),'g.')
21 plot(x1(region3),x2(region3),'b.')

```



The output is shown on the left. These are the classification regions for the classifier in Example 2.1.4. Class 1 is plotted with red, class 2 with green and class 3, with blue. Weird and pretty, isn't it?

2.2 Evaluation of a classifier



How do we know whether our classifier is good? We need to train the classifier by many labelled examples and test it on unseen examples for which we (sneakily) know the labels.

2.2.1 The danger of overtraining

We must make sure that the testing data is not seen during training. Otherwise, we may encourage the classifier to learn all the noise and outliers in the data, and keep making mistakes on unseen data. This is termed *overtraining* or *overfitting the data*. Such overfitting is particularly likely when we have a small data set and a powerful classifier. An example is shown in Figure 2.4. Here the simple linear classification boundary is *optimal* in the sense that no other boundary will lead to more accurate labelling of unseen data from the distribution of this problem.

Over-fitting may come as a consequence of over-training. If the classifier has a parameter to tune, like a dial, we may over-tune it by getting the classifier to fit the training data better and better and better. Counter-intuitive as this may sound, sometimes it will be wise to stop the tuning early on. How early exactly? We should be looking for indications to tell us. An example of this strategy in operation is training decision trees. As we will see later, if we let the tree grow without a limit, it will classify perfectly all the training data. It is often better to prune the tree in order to ensure that the noise in the data will not affect its performance on unseen data.

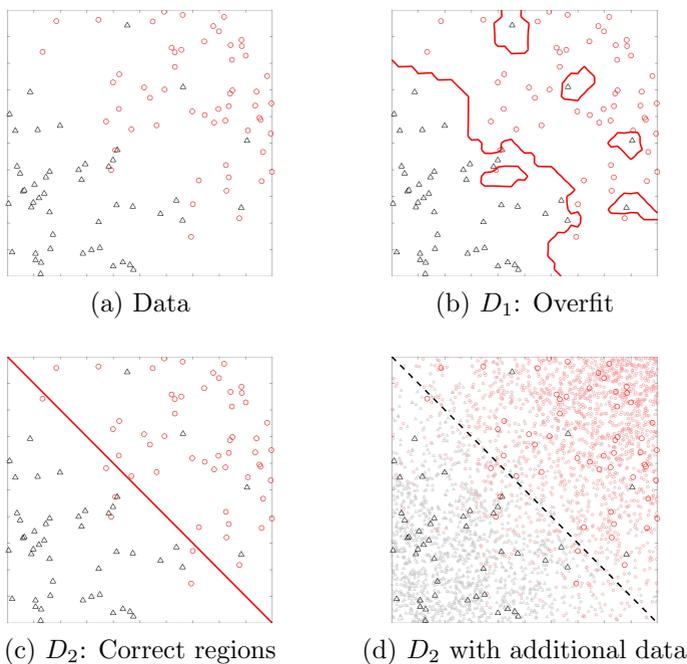


Figure 2.4: An example of a data set and two classifiers, D_1 and D_2 , shown by their classification regions.

2.2.2 Training and testing protocols

According to our pattern recognition cycle, we get the data, train a classifier, test it, and return it to the user along with the ‘accuracy certificate’. Only if it was that simple!...

A summary of training and testing protocols is shown in Figure 2.5 and detailed further on in this section.

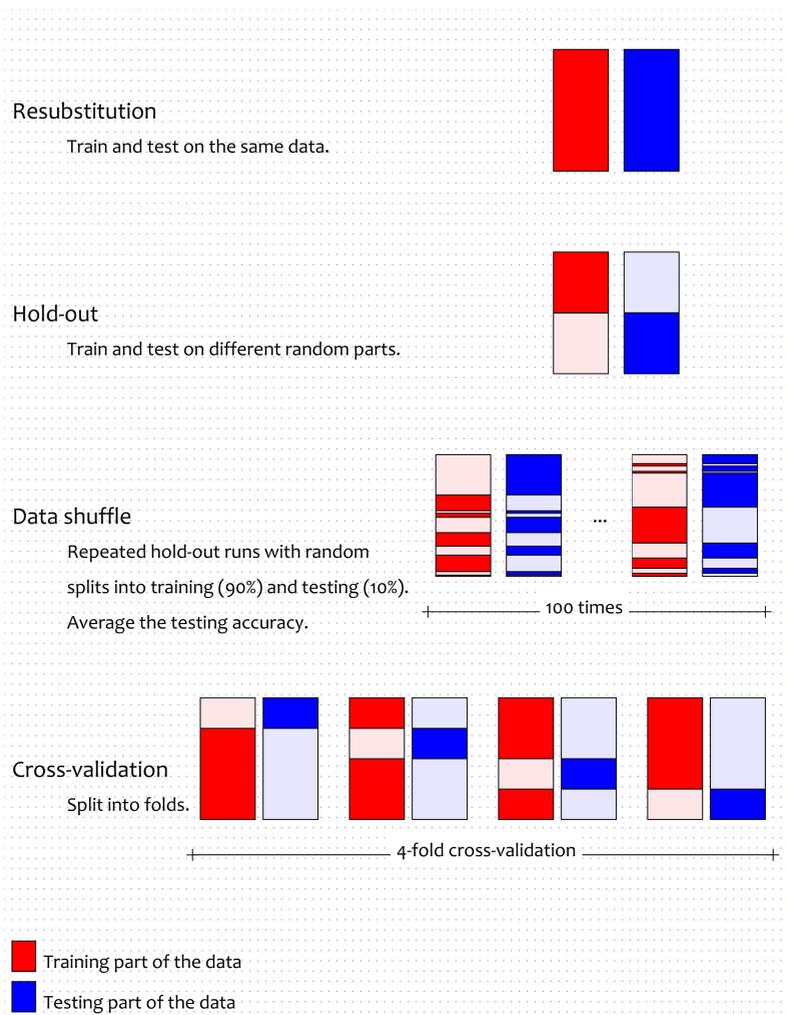


Figure 2.5: A visual summary of the most used training-testing protocols.

Resubstitution

What a long word! And the spell-checker will underline it relentlessly until you add it to the dictionary. But a very needed concept. *Resubstitution* training-testing protocol, or the R-method, means training and testing the classifier using the same data. Indeed, this hardly makes sense for small data sets. We can train the classifier to recognise perfectly every data point but then our ‘accuracy certificate’ will be a delusion of perfection or a speculation at best.

Hold-out

Logically, the more data we have, the better the classifier will be. With a large data set we may choose a complex classifier model and apply the most sophisticated training algorithm. Then we can test the classifier on previously unseen data, and obtain a faithful estimate of its accuracy. The assumption here is that we have so much labelled data that we can comfortably split it into training and testing parts. Adding more instances to the training part will not impact significantly the classifier. So, increasing the computational load will be just pointless. This method of splitting the data is called *hold-out*, or H-method.

Data shuffle

This method repeats the hold-out with different random splits into training and testing parts. Say, we carry out 100 runs (the commonly used default number of repetitions). For each run, the data is split randomly into a training part (90%) and a testing part (the remaining 10%). A classifier is trained on the training part and tested on the testing part. Remember to store the testing accuracy - this is why we are doing all these machinations! After the 100th run, we will have 100 estimates of accuracy, say A_1, \dots, A_{100} (those stored

values). Then the value which we return to the user as the ‘accuracy certificate’ will be the average of the A_s .

Cross-validation

In the Data shuffle approach above, the testing sets for the different runs will likely have some overlap. Ideally, we would like to test our classifier on non-intersecting (unseen) testing data. Enter cross-validation!

In this method we first decide on the number of folds, K . Then we split the data randomly into K folds of approximately equal size. One fold is left aside (say, fold i) as the testing part, and the remaining $K - 1$ folds are pooled to make the training part. A classifier is trained and tested using this partition, and the accuracy A_i is stored in our little piggy bank. This is repeated K times to guarantee that each of the K folds is used for testing once. Thus our testing set is exactly Z – the labelled data set we started off with! Typically, K is set to 10 or 5.

Leave-one-out (LOO)

And now comes one of the most famous cross-validation variants: the Leave-One-Out (LOO) method. For this method $K = N$, where N is the number of objects in the data (rows of Z). In other words, we train N classifiers by leaving aside ONE object for testing in each run. This one object is our testing data. The accuracy A_i can only be 0 (wrong label) or 1 (correct label) for the testing guy. At the end, after the N th run, we will have a collection of binary values A_1, \dots, A_N . The average will give us the estimate of the accuracy of the classifier.

LOO is a popular classifier evaluation method, especially useful for small data sets. Why for small data sets? Because in this case

we cannot afford to cut a sizeable testing part from the precious little data we have. If we do so, we may lose information about the structure of the data and possible ways to classify it. LOO is unbiased (or ever so slightly pessimistically biased), which is good! This means that, if we apply LOO to many many different samples Z from the big wide world, our ‘accuracy certificate’ will match the truth. But what good is that? We have only one data set Z , and we have no choice but trust the calculated value of the accuracy from the LOO for *this* Z . We can also return a measure of confidence, and it will not be very high for a small data set. Not ideal, ha? If your user wants a better classifier and stronger guarantees, they should collect more data. ‘Data’ is the magic word...

Notice that the classifier which we return to the user should be trained eventually on the entire data set Z . The methods described here serve to determine how accurate this classifier would be on unseen data, which is known as the *generalisation* ability of the classifier.

2.2.3 The confusion matrix

Cohorts of students believed that *I* have coined this term just to confuse them! I haven’t, honestly! This is a standard, well known term in pattern recognition and machine learning.

Table 2.2.3 shows a confusion matrix for a classifier D tested on a data set Z with N objects. The rows of the matrix correspond to the true labels while the columns, to the labels assigned by D . Entry a_{ij} in the confusion matrix is the number of objects from Z with true label ω_i , labelled by the classifier as ω_j . The confusion matrix can tell us which classes have been particularly difficult for the classifier, and where most of the mistakes occurred. The diagonal elements show the correctly labelled counts for each class. The off-diagonal elements show the number mislabelled objects.



Table 2.1: The confusion matrix for a data set Z of size N , labelled into c classes.

		Assigned labels						
		ω_1	ω_2	ω_j	...	ω_c
True class labels	ω_1	a_{11}	a_{12}	a_{1j}	...	a_{1c}
	ω_2	a_{21}	a_{22}	a_{2j}	...	a_{2c}
	\vdots							
	ω_i	a_{i1}	a_{i2}	...	a_{ii}	a_{ij}	...	a_{ic}
	\vdots							
	ω_c	a_{c1}	a_{c2}	a_{cj}	...	a_{cc}

Notes:

- a_{ij} is the number of objects from the data set with true label ω_i labelled by the classifier as ω_j .
- a_{ii} is the number of correctly labelled objects from ω_i , $i = 1, \dots, c$. (in the boxes)
- $\sum_{i,j} a_{ij} = N$.

Thus, the accuracy of the classifier can be estimated as

$$P_a = \frac{1}{N} \sum_{i=1}^c a_{ii},$$

where c is the number of classes. The error rate is, respectively

$$P_e = 1 - P_a = 1 - \frac{1}{N} \sum_{i=1}^c a_{ii}.$$

⊕ ⊕ ⊕ **Example 2.2.1**

Calculate and show the confusion matrix of the classifier represented with its classification regions in Figure 2.6. The four classes are represented with different markers and different colours. Each class region is shaded with a lighter version of the colour of the marker (for example, pink region corresponds to the red class (class 1), plotted with a red cross marker). Answer the following questions:

1. What is the total number of objects in the data set?
2. What is the total number of objects from class 3 in the data set?
3. What is the accuracy of the classifier for this data set?
4. Estimate the error of the classifier if it predicts class 2?

Solution: There are 4 classes, therefore the confusion matrix will be of size 4-by-4. The first row will represent class 1 (the red crosses). Four of the five red crosses are in the pink region, which means that the classifier has diligently labelled them in class 1. Success! None are in class 2, one poor thing is in the grey region labelled as class 3, and none of the reds is in the green region. Thus, the first row

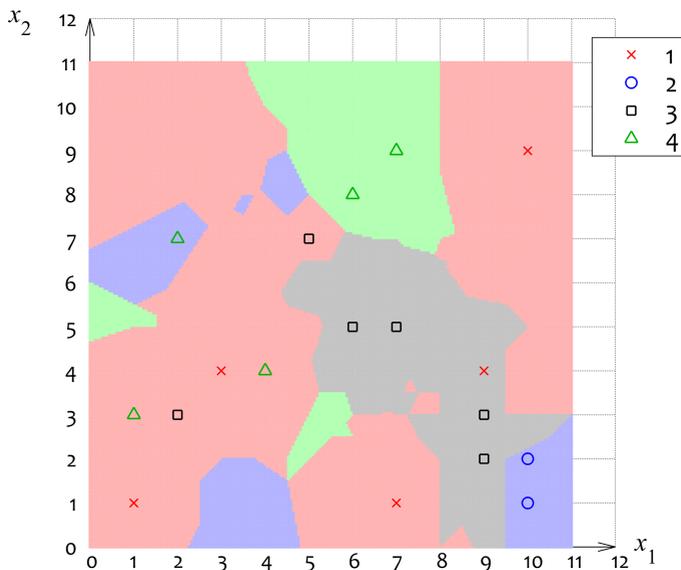


Figure 2.6: Classification region for classifier D and the data for preparing the confusion matrix.

of the table is $[4, 0, 1, 0]$. And so it goes next in constructing the second row of the table responsible for class 2 (blue circle): $[0, 2, 0, 0]$. Both circles are neatly in the blue region. Got it? Here is the confusion matrix itself:

		Assigned labels			
		class 1	class 2	class 3	class 4
True labels	class 1	4	0	1	0
	class 2	0	2	0	0
	class 3	2	0	4	0
	class 4	2	1	0	2

And now for the other questions:

1. What is the total number of objects in the data set? Each object falls in *one* cell of the table to contribute to the count there. Therefore $N = 4 + 1 + 2 + 2 + 4 + 2 + 1 + 2 = 18$.
2. What is the total number of objects from class 3 in the data set? All objects from class 3 are accounted for in row 3. Therefore $N_3 = 2 + 4 = 6$.
3. What is the accuracy of the classifier for this data set? This is the proportion of the objects in the cells on the main diagonal:

$$P_a = \frac{1}{18}(4 + 2 + 4 + 2) = \frac{12}{18} = 0.6667.$$

4. Estimate the error of the classifier if it predicts class 2? This time we look at column 2 responsible for the class 2 *predictions*. Out of the 3 predictions for this class, 2 were correct. Therefore, the *error* can be estimated as

$$P_e(\text{Prediction is class 2}) = \frac{1}{3} = 0.3333.$$

Not so confusing, eh?

⊖ ⊖ ⊖

The confusion matrix is specific for the given classifier D and the data set Z .

And here is our little MATLAB function which calculates a confusion matrix for a set of true labels and a set of assigned labels. (See how we make it universal, so that the labels don't have to be consecutive integers like 1, 2, 3, and so on, but can be integers like 135, 21, 67, and so on.)

```

1 function [cm,u] = confusion_matrix(true_labels, ...
    assigned_labels)
2 u = unique(true_labels); % returns a sorted column of
3   % unique labels
4 c = numel(u); % number of classes
5 cm = zeros(c); % c-by-c confusion matrix
6 for i = 1:c
7     for j = 1:c
8         cm(i,j) = sum(true_labels == u(i) & ...
9             assigned_labels == u(j));
10    end
11 end

```

And here is a little script to check that the function works:

```

1 clear, clc, close all
2 l1 = randi(4,100,1)*3+5; % generate 100 labels from the
3   % set {8, 11, 14, 17}
4 l2 = randi(4,100,1)*3+5; % generate 100 labels as the
5   % assigned labels
6 [cm, c1] = confusion_matrix(l1,l2)

```

The MATLAB output in the command window will be something like this:

```

1 cm =
2 4     8     5     4
3 5     10    7     7
4 7     5    12     9
5 7     4     3     3
6
7 c1 =
8 8
9 11
10 14
11 17

```

The label list will be the same if you run this script again, but

cm will change depending on the random labels generated for 11 and 12.

2.3 ROC curves



No, unfortunately it is not Rock'n'Roll all the way through ☹. ROC curves are another way to see how our classifier performs.

2.3.1 A bit of history

Let's decipher the acronym first: ROC stands for 'Receiver Operating Characteristic'. Huh? These curves were first devised during World War II following the attack on Pearl Harbor in 1941. They were applied to measure the ability of a radar receiver operator (human) to distinguish between Japanese aircraft and other objects from their radar signals. The name 'ROC curves' stuck since then.

2.3.2 False positives, false negatives and the two error types

In two-class (binary) classification problems, we may designate one of the classes as 'positive' and the other as 'negative'. The positive class is usually the class of special interest. Look at the curious reincarnation of the confusion matrix for this case in Table 2.2.

Table 2.2: Confusion matrix for two classes: positive and negative. Types of errors. False positives, false negatives, true positives and true negatives.

		Assigned labels	
		positive	negative
True labels	positive	True positive (TP) Convict the guilty	False negative (FN) Free the guilty! TYPE I ERROR
	negative	False positive (FP) Convict the innocent! TYPE II ERROR	True negative (TN) Free the innocent

A confusion matrix defines only a single point on the ROC curve. To calculate this point, we need two terribly important quantities, widely used in many areas, and mostly in medical data analysis:

Sensitivity of a test measures the proportion of correctly identified positive cases out of all positive cases, that is:

$$Sensitivity = \frac{TP}{TP + FN}.$$

Specificity of the test (a bit of tongue-twister, isn't it?) is the proportion of correctly identified negative cases out of all negative cases, that is:

$$Specificity = \frac{TN}{TN + FP}.$$

Ideally, both sensitivity and specificity should be 1. In this case, we have the ideal classifier with accuracy 100%.

2.3.3 The ROC curve

The ROC curve gives us the chance to pick a compromise between FP and FN for the classifier at hand. The general layout of a ROC curve is shown in Figure 2.7.

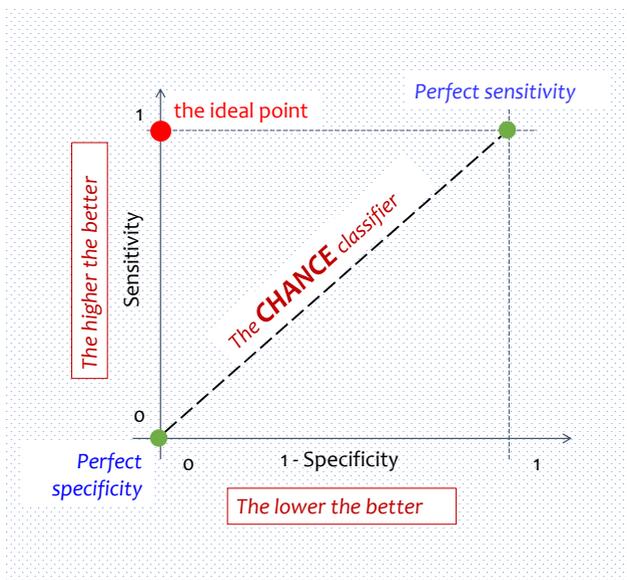


Figure 2.7: A general layout of a ROC curve.

On the x -axis, we plot $1 - \textit{Specificity}$ (proportion of false positives out of all negatives) and on the y -axis we plot $\textit{Sensitivity}$. Therefore, the ideal point is at $(0,1)$, corresponding to perfect sensitivity and specificity - top left corner on the graph. Only if we were able to reach it...

The y axis is the place of perfect specificity (whatever the sensitivity), and the horizontal line at $y = 1$ is the place of perfect

sensitivity (whatever the specificity). The bottom left corner is the point where we label everything as negative. Sensitivity is 0 but the specificity is perfect. Top right is the classifier which labels everything as positive, ensuring sensitivity of 1 and 0 specificity. Neither of these extreme points is desirable. We'd rather have the ideal point, please. But not every classifier can reach even close to it. Interestingly, if we are guessing the labels, the ROC curve will span the diagonal from $(0, 0)$ to $(1, 1)$, denoted 'The CHANCE classifier' in the graph. Any classifier better than chance will have a 'belly' towards the ideal point. Let's see how this happens.

A trained classifier is associated with a single confusion matrix, and therefore defines only *one* point on the ROC curve, called the *operational point*. An example is shown in Figure 2.8.

The operational point is calculated from the confusion matrix and plotted on the graph. But where will the other points come from (apart from the $(0,0)$ and $(1,1)$)? We assume that there is a parameter, something like a dial, which we can tweak in order to get different versions of the classifier and their corresponding points on the ROC curve. For example, suppose that the classifier outputs a value v between 0 and 1, and we threshold this value with some threshold θ to get the class label. Let's say that for $v \geq \theta$ we assign class positive, and for $v < \theta$, class negative. By sliding θ from 0 to 1, we will drive the operational point from point $(1, 1)$ (everything is labelled as the positive class) down to $(0, 0)$ (everything is labelled as the negative class).

⊕ ⊕ ⊕ **Example 2.3.1**

Given is the one-dimensional, 2-class data set depicted in Figure 2.9. Consider a classifier which uses a threshold θ . All data points to the left of θ are labelled as negative, and all points at θ and to the right are labelled as positive.

Build the ROC curve for this threshold classifier.

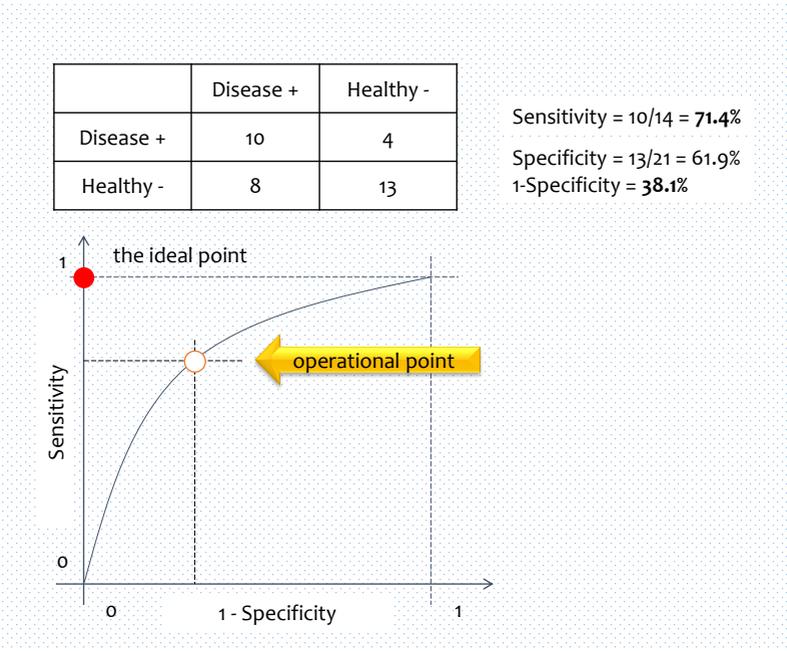


Figure 2.8: An example of calculating the operational point on the ROC curve from the confusion matrix.

Solution: Slide θ from 0 to 16 and calculate the confusion matrix for each threshold. Subsequently, calculate the operational point on the ROC curve for the current value of θ . Notice that the confusion matrix will change only when the threshold bypasses a point from the data set. Then we can choose threshold values between each consecutive pair of data points. Table 2.3 shows the calculated values.

Figure 2.10 shows the ROC curve for this threshold classifier estimated from the given data set.

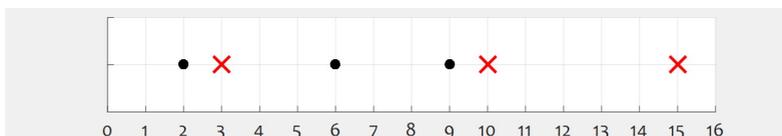


Figure 2.9: 1D dataset. Class positive is plotted with red crosses.

Table 2.3: Threshold values and the respective operational points on the ROC curves (in boldface) for the dataset in Figure 2.9.

θ	TP	FN	FP	TN	$Sens$	$Spec$	$1 - Spec$
1.0	3	3	0	0	1.0000	0.0000	1.0000
2.5	3	2	0	1	1.0000	0.3333	0.6667
4.5	2	2	1	1	0.6667	0.3333	0.6667
7.5	2	1	1	2	0.6667	0.6667	0.3333
9.5	2	0	1	3	0.6667	1.0000	0.0000
12.5	1	0	2	3	0.3333	1.0000	0.0000
15.5	0	0	3	3	0.0000	1.0000	0.0000

The ROC curve does not look very smooth, does it? But if we had 1000 points instead of 6, the picture would be different. $\ominus \ominus \ominus$

The MATLAB code for this example is shown below. Note that it uses the function `confusion_matrix` which we so-very-wisely prepared before.

```

1 clear, clc, close all
2 labels = [2 1 2 2 1 1]; % 1-class positive, 2-class negative
3 data = [2 3 6 9 10 15]; % 1D data
4 augmented_data = [0 data 16];

```

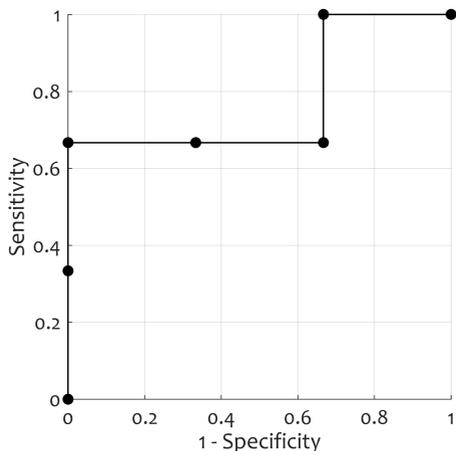


Figure 2.10: ROC curve for the data in Figure 2.9

```

5 thresholds = augmented_data + [diff(augmented_data)/2 0];
6
7 % Calculate the operational points for the ROC curve
8 for i = 1:numel(thresholds)-1
9     assigned_labels = (data < thresholds(i)) + 1;
10    % 1 for class +, 2 for class -
11    C = confusion_matrix(labels(:), assigned_labels(:));
12    x(i) = 1 - C(2,2)/(C(2,1)+C(2,2)); % 1- Specificity
13    y(i) = C(1,1)/(C(1,1) + C(1,2)); % Sensitivity
14 end
15
16 figure, hold on, grid on, axis([0 1 0 1]), axis square
17 plot(x,y, 'k.-', 'markersize', 12, 'linewidth', 1)
18 xlabel('1 - Specificity'), ylabel('Sensitivity')

```

A measure of performance of the classifier model is the area under the ROC curve. Area close to 1 will mean that the curve is ‘stretching’ to reach the ideal point (0,1), and the classifier is good.

Conversely, if the area is close to 0.5, the classifier is no different to random labelling. And if we should find that AUC is smaller than 0.5, Oh, Dear! We can reverse the labels! Call the black white, and the white black, and then the classifier will regain some dignity beyond random guessing. AUC has been an object of dispute in the past as a classifier performance measure [10] but it is still among the most popular metrics.

2.4 Imbalanced classes

2.4.1 What is class imbalance?

Classification accuracy is not always the best measure of the performance of a classifier. Imagine the case where we build a test to screen the population for a very rare disease. If 0.001% of people are affected by the disease, a classifier which predicts that everybody is unaffected will be almost perfect! It will have accuracy of 99.999%. Isn't this splendid? But this classifier will be of no use in recognising the rare disease of interest. Usually, it is not that easy to develop an accurate test. This means that in order to detect most cases of interest we 'pay' with a number of healthy people being picked out as affected. Further tests (more expensive and accurate) will filter those people out. But our initial screening test may have a lot smaller accuracy than the useless "all-healthy" classifier. (We shall call this classifier by several names: 'The Largest Prior' classifier, 'The Majority' classifier, and the 'trivial' classifier. All these amount to the same thing – always assign the predominant class label.)

Usually, in two-class problems, the class of interest is a lot smaller than the other class (rare disease versus healthy). This type of data is known as 'imbalanced' or 'unbalanced'. Classification accuracy is clearly not very useful for this type of problems. Instead, there are



other measures which give more weight to the minority class.

⊕ ⊕ ⊕ **Example 2.4.1**

An example of an imbalanced classification problem is identifying the square(s) containing a face in an image (Figure 2.11). Suppose that we slide a square with the correct dimension starting from the top left corner of the image, one pixel at a time, all the way across to the last possible position. Each of the squares we go through is an object in the data set. Then we return the square back to the left edge, move it a pixel down and repeat the sliding action. Out of many many squares, only a handful will contain a recognisable face image. There are 6 such squares in the image in Figure 2.11.

The image size is 79×118 pixels and the square size is 25×25 pixels. If there are 6 positive objects in the data set obtained through sliding the square over the whole image (as described above), how many negative objects will there be in the dataset?

Solution: In each row, we have only $118 - 25 + 1 = 94$ places to position the square. Also, the lowest vertical position will be at row number $79 - 25 + 1 = 55$. Therefore the total number of possible squares in the image is $94 \times 55 = 5170$. Out of these, Cutie-Patootie is in 6. Therefore the number of negatives is $5170 - 6 = 5164$. ⊖ ⊖ ⊖

2.4.2 Classifier performance measures for imbalanced classes

The most intuitive measure for imbalanced classes is the *Risk* or the *Loss*. Classification errors have naturally different costs. For example, in the justice system, the error of ‘freeing the guilty’ is not as heavy as ‘convicting the innocent’. Thus, we can assign numerical costs and measure the success of a classifier by the sum of the costs

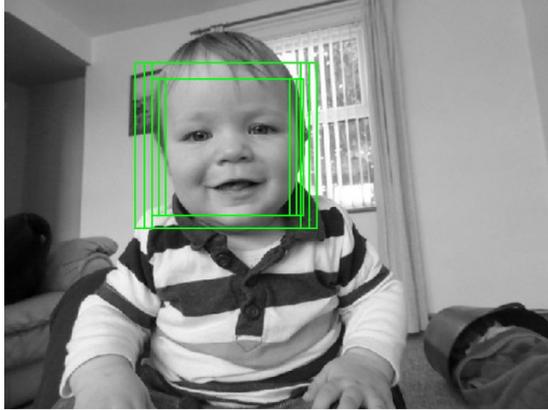


Figure 2.11: An example of an imbalanced classification problem. Identify the squares containing a face in an image.

incurred by the misclassifications. This brings the idea of a *loss matrix*. Remember the confusion matrix? Now let's populate it with *costs*, which we decide on. These costs will weigh the errors against one another. There is no rule as to how these costs should be assigned. It is up to use whether we should use integer values or fractions, whether they will sum up to a constant, and so on. The loss matrix is not associated with a classifier or a dataset. It is only meant to quantify our perception of how the misclassification errors are related to one another.

So, here are three measures of classifier performance that are useful for imbalanced classes:

- *Cost*. Denote by λ_{ij} the cost of labelling an object from true class ω_i into class ω_j . Correct classification incurs 0 penalty, so $\lambda_{ii} = 0, i = 1, \dots, c$. Thus, the cost of labelling dataset Z by classifier D can be obtained from the confusion matrix C (of Z and D) and the

loss matrix for the problem:

$$Cost = \sum_{i=1}^c \sum_{j=1}^c a_{ij} \times \lambda_{ij}.$$

The cost measure can only serve to compare two classifiers on the same data set.

- *GM measure.* To mitigate the effect of the imbalance in a two-class problem, we can take the geometric mean of the sensitivity and the specificity. This is known as the *GM* measure:

$$GM = \sqrt{Sensitivity \times Specificity}$$

Notice the the *GM* measure will give the same result even if we swap the positive and the negative labels. In other words, it does not matter which class we nominated as positive (the class of interest).

- *F-measure.* Finally, the *F* measure (or also F_1 measure) is all about predicting correctly the positive class. This measure is based on *Recall* and *Precision*. Recall is the same as sensitivity!

$$Recall = Sensitivity = \frac{TP}{TP + FN}.$$

Precision, on the other hand, tells us what proportion of the objects labelled as positive by our classifier are indeed positive:

$$Precision = \frac{TP}{TP + FP}.$$

Then

$$F = \frac{2 \times Recall \times Precision}{Recall + Precision}$$

F-measure is widely used in text recognition and retrieval systems. F varies between 0 and 1, with 0 meaning failure to recognise the

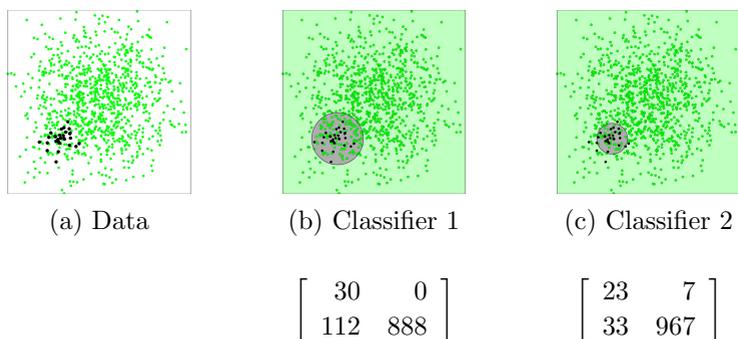


Figure 2.12: Imbalanced data and two classifiers. The respective confusion matrices are shown underneath the classifier scatterplots.

class of interest at all to 1 meaning perfect classification (all positives have been identified with no false positives).

⊕ ⊕ ⊕ **Example 2.4.2**

Well, there is no sensible way to compare measures. They measure classifier performance in different ways! They measure whatever they are designed to measure, and their values are not commensurable. Apples and oranges. But we can still illustrate the calculations.

Consider the imbalanced data set depicted in Figure 2.12 (a). The negative class (green) contains 1000 points (97%), and the positive class (black) contains 30 points (3%). Two classifiers are shown by their classification regions in Figure 2.12 (b) and (c). Which one is better?

The confusion matrices for the two classifiers are shown underneath the respective scatterplots. Classifier 1 labels all positive cases correctly at the expense of 112 false positives. Classifier 2, on the other hand, reduces the false positives to 33 but misses 7 of the pos-

itive cases. Table 2.4 shows the accuracy, GM and F for the two classifiers.

Table 2.4: Performance measures for the two classifiers for the imbalanced data.

	Sensitivity =Recall	Specificity	Precision	Accuracy	GM	F
C1	1.000	0.8880	0.2113	0.8913	0.9423	0.3488
C2	0.7667	0.9670	0.4107	0.9612	0.8610	0.5349

Clearly neither classifier dominates its rival on all measures. This means that we can choose based on how much we are inclined to tolerate errors of type I and type II. For example, if it is paramount to catch ALL positive cases, Classifier 1 is better. If, however, the cost for false positives is too great, we may prefer Classifier 2. Decisions, decisions... $\ominus \ominus \ominus$

2.5 A probabilistic view

Now this is a bit further from the easy-going maths-free exposition thus far. But the good news is that we will only sporadically refer to the probabilistic framework and only if you are striving to learn the ‘stratosphere’ of machine learning.

Think of the problem as some magical generator of data which spouts out an infinite stream of data, one data point at a time. The data point comes with a fixed but unknown label, which our classifier must predict.

The distribution of the classes *before* we measure anything about the data point at hand is called the *prior* distribution. The class label $\omega \in \Omega = \{\omega_1, \dots, \omega_c\}$ is a random variable, and its probability mass function is described by the *prior probabilities* for the classes (or the *a priori* probabilities) $P(\omega_1), \dots, P(\omega_c)$, $\sum_i P(\omega_i) = 1$.



All data points coming from the magical generator live in the n -dimensional space, $\mathbf{x} \in \mathbb{R}^n$. Each class is distributed according to the probability mass function $p(\mathbf{x}|\omega_i)$ (*class-conditional distribution*), $i = 1, \dots, c$. Using the Bayes theorem, we can calculate the *posterior* probabilities for each \mathbf{x}

$$P(\omega_i|\mathbf{x}) = \frac{P(\omega_i) p(\mathbf{x}|\omega_i)}{p(\mathbf{x})},$$

where $p(\mathbf{x})$ is the unconditional distribution of \mathbf{x}

$$p(\mathbf{x}) = \sum_{i=1}^c P(\omega_i) p(\mathbf{x}|\omega_i).$$

If all probability distributions were known, we will be able to build the ‘perfect’ classifier, called the Bayes classifier. We will incur the lowest error rate if we always assign \mathbf{x} to the most probable class, that is:

$$\text{Assigned Label } (\mathbf{x}) = \max_{i=1}^c \{P(\omega_i|\mathbf{x})\}.$$

Well, wouldn’t life be so much easier if we knew the probability distributions? Unfortunately, we never do! We have only Z to build the classifier. We may choose to approximate the probabilities but in most cases this is impractical. That is why we have the wonderful,

abundant, and ever-growing armoury of classifier methods which we will delve into next.

Chapter 3

Classifiers



Classifiers. There are so many classifiers! What do they do? How are they different from one another? How do we choose among them?

We will just lift the curtain a little bit in this chapter. There is a lot more out there but at least you will have a toy compass by the end of the chapter.

Here you will learn about:

- The Nearest Mean Classifier (NMC)
- The Linear discriminant classifier (LDC)
- Rule-based classifiers
- The k-Nearest Neighbour classifier (k-nn)
- The Decision Tree classifier
- The Support Vector Machine classifier (SVM)
- Classifier Ensembles

Some of the models are easy to program (like NMC) – great! You will have to program them for yourself. Others are a bit trickier

(SVM). But the good news is that they are all well-known classifier models and widely available in many programming languages.

3.1 The Nearest Mean Classifier (NMC)

3.1.1 How it works

Recall the definition of a classifier from Section 2.1.1: A *classifier* is any function, method or algorithm that assigns a class label to any given object. The nearest mean classifier does exactly what it says on the tin! Simple as this – find the means of all c classes in the n -dimensional space \mathbb{R}^n . For any $\mathbf{x} \in \mathbb{R}^n$, find the distances to the c means and assign to \mathbf{x} the label of the nearest class mean.

Figure 3.1 shows an example with 2 classes in 2D.

The data contains 40 points from each class. The point to be classified is shown at $(3, -1)$ as \mathbf{x} .

To apply NMC, we calculate the means of the classes (average the x_1 coordinates for all objects from the class, and then do the same for the x_2 coordinates). The means of the two classes are marked on the plot with a target marker: $\mathbf{m}_1 = [-0.5, 1.0]^T$ and $\mathbf{m}_2 = [2.2, 2.1]^T$. Next we calculate the distance between \mathbf{x} and the two centres¹

$$d(\mathbf{x}, \mathbf{m}_1) = \sqrt{(3 - (-0.5))^2 + (-1 - 1)^2} = \sqrt{12.25 + 4} \approx 4.03.$$

$$d(\mathbf{x}, \mathbf{m}_2) = \sqrt{(3 - 2.2)^2 + (-1 - 2.1)^2} = \sqrt{0.64 + 9.61} \approx 3.20.$$

As $d(\mathbf{x}, \mathbf{m}_2) < d(\mathbf{x}, \mathbf{m}_1)$, NMC assigns label ω_2 to object \mathbf{x} .

In summary, NMC operates like this: To train the classifier find and store the means of all classes. To run the classifier for a given data point, calculate the distance between the point and all class

¹Here we use Euclidean distance. See Appendix A.

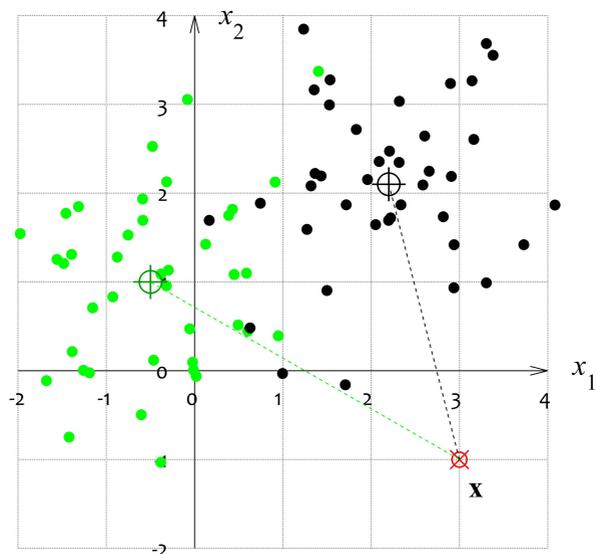


Figure 3.1: Illustration of the nearest mean classifier (NMC).

means. Assign to the point the label of the class with the closest mean.

3.1.2 Classification boundary of a 2-class NMC in 2D

For two classes, the classification boundary of NMC is determined by all points which are equidistant from the two class means. In an n -dimensional space, this boundary is a hyperplane. In a 2D space (lucky us!) it is a line. And we can calculate the equation of the

line using simple geometry.²

The line with equidistant points from the two means passes through the middle of the segment between the two means and is orthogonal to the segment. Figure 3.2 continues the example in Figure 3.1.

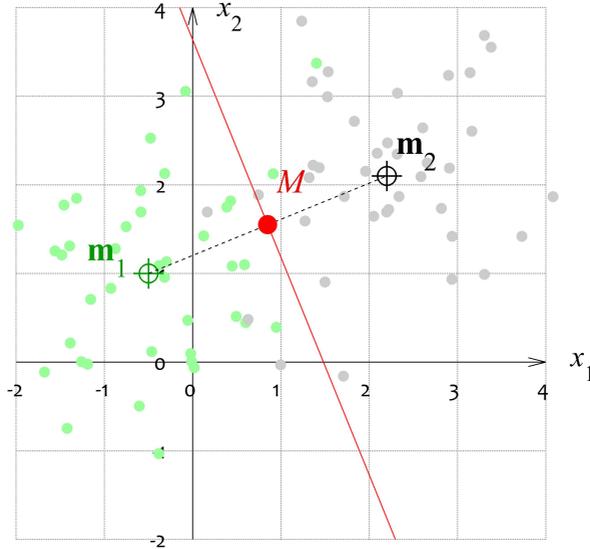


Figure 3.2: Illustration of the derivation of the classification boundary for NMC for a 2D, 2-class problem.

The two means are labelled on the plot and joined by a dashed line. The middle point of the segment is marked by M and is calculated as $(\mathbf{m}_1 + \mathbf{m}_2)/2$. Let's detail the coordinates a bit more. Since both means (vectors) are in \mathbb{R}^2 they have two components

²See Appendix A

each: $\mathbf{m}_1 = [m_{11}, m_{12}]^T$ and $\mathbf{m}_2 = [m_{21}, m_{22}]^T$. Then M can be expressed as the vector

$$M = [M_1, M_2]^T = \left[\frac{m_{11} + m_{21}}{2}, \frac{m_{12} + m_{22}}{2} \right]^T.$$

Form vector $\mathbf{v} = \mathbf{m}_1 - \mathbf{m}_2$ which will be a normal vector for the boundary

$$\mathbf{v} = [v_1, v_2]^T = [m_{11} - m_{21}, m_{12} - m_{22}]^T.$$

Use the components of \mathbf{v} as the coefficients in front of the unknowns in the equation of the boundary line:

$$v_1 x_1 + v_2 x_2 + c = 0.$$

Since M must lie on the line, substitute the coordinates of M in the equation and solve for c :

$$c = -v_1 M_1 - v_2 M_2,$$

to arrive at

$$v_1 x_1 + v_2 x_2 - v_1 M_1 - v_2 M_2 = 0.$$

For the grown-ups, we can rewrite the equation of the line in a vector form using only \mathbf{m}_1 and \mathbf{m}_2 :

$$(\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{x} - \frac{1}{2}(\mathbf{m}_1 - \mathbf{m}_2)^T (\mathbf{m}_1 + \mathbf{m}_2) = 0,$$

and shorten it further to

$$(\mathbf{m}_1 - \mathbf{m}_2)^T \left(\mathbf{x} - \frac{1}{2}(\mathbf{m}_1 + \mathbf{m}_2) \right) = 0.$$

This representation of the boundary demonstrates that NMC is, in fact, a linear classifier because the boundary is linear on the vector of unknowns \mathbf{x} . The boundaries are also linear for multiple dimensions and multiple classes.

3.1.3 Programming the NMC

Sweet, sweet MATLAB! Programming the NMC is easy! We will write one function for training and one for testing. The input to the training function is the labelled data set Z : **data** (numerical array of size $N \times n$) and **labels** (numerical array of size $N \times 1$). The output is the set of class means (numerical array of size $c \times n$, where c is the number of classes) and the corresponding class labels (numerical array of size $c \times 1$). Then the testing function takes the means, the labels, and the data to be labelled, and assigns to each point in the data the label of the nearest mean. Here we go:

```

1 function [class_means, mean_labels] = ...
    nmc_training(data, labels)
2 mean_labels = unique(labels); % the labels in ascending order
3 for i = 1:numel(mean_labels) % for each label
4     class_means(i, :) = mean(data(labels == ...
        mean_labels(i), :), 1);
5 end

1 function assigned_labels = nmc_test(ref_data, ref_labels, ...
    test_data)
2 c = numel(ref_labels); % number of classes
3 for i = 1:size(test_data, 1) % for each object to be labelled
4     x = test_data(i, :); % take the object
5     % Calculate the (squared) Euclidean distance to
6     % all reference points and find the index of the
7     % smallest distance:
8     [~, min_index(i)] = min(sum((ref_data - ...
        repmat(x, c, 1)).^2, 2));
9 end
10 % Recover the labels and transpose to return a column:
11 assigned_labels = ref_labels(min_index)';

```

⊕ ⊕ ⊕ **Example 3.1.1**

Remember how we plot classification regions? (Section 2.1.4) Let's try this on an example where we generate randomly data from 4 classes. The classes will be positioned in the unit square: class 1 – top left, class 2 – top right, class 3 – bottom left, and class 4 – bottom right. This is why we call the problem the '4-tile class problem'. We will use both functions from the listings above. It will be interesting to find out whether the size of the data (N) plays any significant role in determining the classification regions. (What do you think?)

Here is the code:

```

1 clear, clc, close all
2
3 cN = 200; % number of objects per class
4 class1 = [rand(cN,1)*0.5 rand(cN,1)*0.5+0.5];
5 class2 = [rand(cN,1)*0.5+0.5 rand(cN,1)*0.5+0.5];
6 class3 = [rand(cN,1)*0.5 rand(cN,1)*0.5];
7 class4 = [rand(cN,1)*0.5+0.5 rand(cN,1)*0.5];
8
9 % Create the dataset
10 D = [class1; class2; class3; class4];
11 L = [ones(cN,1); ones(cN,1)*2; ones(cN,1)*3; ones(cN,1)*4];
12
13 % Train the NMC
14 [M, ML] = nmc_training(D,L); % means and labels
15
16 % Create the meshgrid
17 [x,y] = meshgrid(0:0.005:1,0:0.005:1);
18 GD = [x(:) y(:)]; % format the grid into a dataset
19
20 % Label the grid points through NMC
21 GL = nmc_test(M, ML, GD);
22

```

```

23 % Plot the regions (labelled grid points)
24 figure, hold on, axis square off
25 plot(GD(GL == 1,1),GD(GL == 1,2),'k.','color',[1 1 1]*0.75)
26 plot(GD(GL == 2,1),GD(GL == 2,2),'k.','color',[1 0.8 0.8])
27 plot(GD(GL == 3,1),GD(GL == 3,2),'k.','color',[0.8 0.8 1])
28 plot(GD(GL == 4,1),GD(GL == 4,2),'k.','color',[0.8 1 0.8])
29
30 % Overlay the data
31 h(1) = plot(D(L == 1,1),D(L == 1,2),'k.','markersize',20);
32 h(2) = plot(D(L == 2,1),D(L == 2,2),'r.','markersize',20);
33 h(3) = plot(D(L == 3,1),D(L == 3,2),'b.','markersize',20);
34 h(4) = plot(D(L == 4,1),D(L == 4,2),'g.','markersize',20);
35 legend(h,'Class 1','Class 2','Class 3','Class 4')

```

Figure 3.3 shows the classification regions plotted by the code above for two sample sizes: $N = 20$ and $N = 800$.

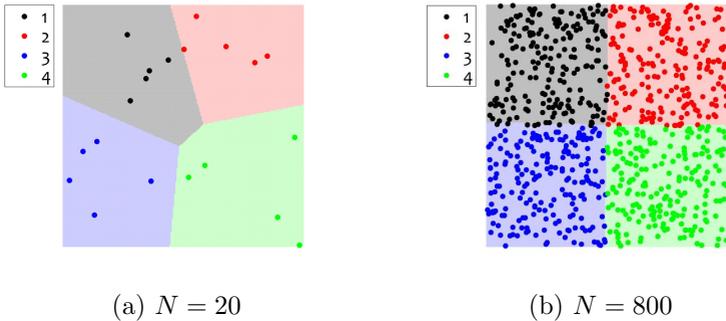


Figure 3.3: Classification regions for the NMC classifier for the 4-tile class problem in the unit square for two sample sizes N .

We observe from the figure that larger sample size leads to a lot more accurate classification regions (the 4 tiles). Notice also that the regions are bounded by piece-wise linear boundaries.

3.1.4 Voronoi diagrams



This is Georgy Feodosevich Voronoy (1868 — 1908 according to the almighty Wikipedia), a Russian mathematician credited with the invention of the so-called Voronoi diagram (notice the difference in the spelling).

Suppose that we have a data set $Z \subset \mathbb{R}^n$. A Voronoi cell of a point $\mathbf{x} \in Z$ is the region of \mathbb{R}^n where all points are closest to \mathbf{x} than to any other point in Z . Thus the space can be partitioned into Voronoi cells, called a *Voronoi diagram*. Some cells will be bounded and some may expand to infinity.

Now, consider a 2D space, and in particular the unit square. We can use our NMC testing code to plot beautiful Voronoi diagrams with random colours. Because, what is the Voronoi cell of \mathbf{x} ? Nothing else but the classification region for the class with mean \mathbf{x} ! All we need to do is generate the (pretend) means and plot the classification regions with lovely random colours as shown in Figure 3.4.

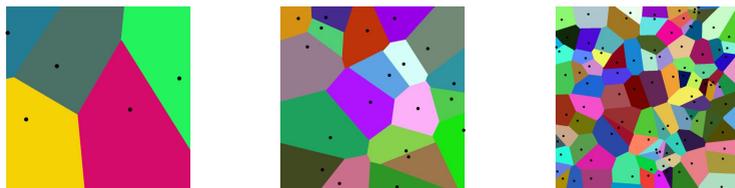


Figure 3.4: Examples of Voronoi diagrams in 2D.

If you are interested in the MATLAB code, here it is:

```
1 clear, clc, close all
2 N = 100; d = rand(N,2);
```

```

3 [M, ML] = nmc_training(d, (1:size(d,1))');
4 [x,y] = meshgrid(0:0.001:1,0:0.001:1);
5 GD = [x(:) y(:)];
6 GL = nmc_test(M, ML, GD);
7 figure, axes('Pos',[0 0 1 1]), hold on, axis square off
8 for i = 1:N
9     plot(GD(GL==i,1),GD(GL==i,2),'k.','color',rand(1,3))
10    plot(d(i,1),d(i,2),'k.','markersize',25)
11 end

```

Now, here are some questions to you: How will NMC work on the data plotted in Figure 3.5? Answer the following questions:

1. Where are the means of the two classes? Give a rough guess using the plot.
2. Where will the NMC classification boundary be? Draw it.
3. Can you give a rough estimate of the classification error rate?
4. Can you propose a better linear boundary between the classes?

3.2 The Linear Discriminant Classifier (LDC)



“Linear” means that each discriminant function is a linear combination of the features. Like this:

$$g_i(\mathbf{x}) = a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n + b_i,$$

where i means that this is the discriminant function for class ω_i , $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$ is the object that we want to classify described with its n features, $a_{i,j}$ are coefficients, and b_i are free terms. The first index of $a_{i,j}$ refers to the class, and the second, to the features.

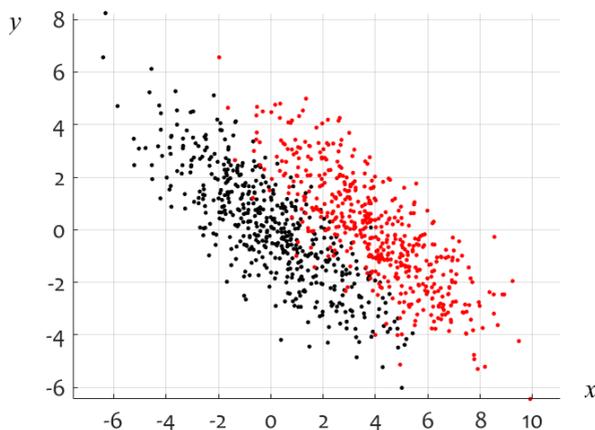


Figure 3.5: Scatterplot of two classes in 2D. How will NMC do for this data?

The coefficients can be any real numbers. (This includes 0, by the way, which means that some of the features may not participate in the discriminant function at all. Zilch input!)

3.2.1 NMC is actually an LDC!

But we have seen already a linear classifier! This was the NMC. It separates the classes with linear boundaries, right? True! NMC is a linear classifier indeed. This means that we can wring the equations that we had for the NMC so that they fit in the shape of the discriminant functions above.

And here is how the magic happens. Suppose we have some data point $\mathbf{x} \in \mathbb{R}^n$, and the c class means $\mathbf{m}_1, \dots, \mathbf{m}_c$, all points in \mathbb{R}^n . Calculate the c squared distances between \mathbf{x} and the means:

$$d_i(\mathbf{x}, \mathbf{m}_i)^2 = (\mathbf{x} - \mathbf{m}_i)^T (\mathbf{x} - \mathbf{m}_i) = \mathbf{x}^T \mathbf{x} - 2 \mathbf{m}_i^T \mathbf{x} + \mathbf{m}_i^T \mathbf{m}_i.$$

We don't need the square root, do we? Because if $\xi_1 > \xi_2$, then $\sqrt{\xi_1} > \sqrt{\xi_2}$ (taking the positive answer only as this is a distance after all). So, we will be comparing d_i with one another and taking the smallest one to give us the class label of \mathbf{x} . But look, in all of these we will have *the same thing* added to each sum: $\mathbf{x}^T \mathbf{x}$. Exactly the same quantity. If we remove it from all sums their ordering will stay the same. In other words, I can now compare some d'_i (not exactly the distances) which will identify for me the same class label as a properly calculated distance:

$$d'_i(\mathbf{x}, \mathbf{m}_i) = -2 \mathbf{m}_i^T \mathbf{x} + \mathbf{m}_i^T \mathbf{m}_i.$$

As we defined *discriminant functions* to be “the higher, the better”, and we want the opposite when we compare distances, we may as well maximise $-d'_i$. Agreed? Then we have our coveted discriminant functions:

$$g_i(\mathbf{x}) = -d'_i(\mathbf{x}, \mathbf{m}_i) = 2 \mathbf{m}_i^T \mathbf{x} - \mathbf{m}_i^T \mathbf{m}_i.$$

The index of the maximal $g_i(\mathbf{x})$ will give us the class label. Take a look at g_i . The second term is a scalar product of two numerical vectors (both \mathbf{m}), and is therefore a numerical constant. This is our disguised b_i . Writing out the first term, we come to:

$$g_i(\mathbf{x}) = \underbrace{2 \times m_{i,1}}_{a_{i,1}} x_1 + \underbrace{2 \times m_{i,2}}_{a_{i,2}} x_2 + \dots + \underbrace{2 \times m_{i,n}}_{a_{i,n}} x_n + \underbrace{(-\mathbf{m}_i^T \mathbf{m}_i)}_{b_i}.$$

There we go! Our exact LDC discriminant function! So, NMC is an LDC. But not a very good one. Check Figure 3.6 to see the difference. LDC gives a lot more accurate classification regions compared to NMC in the example in the figure.

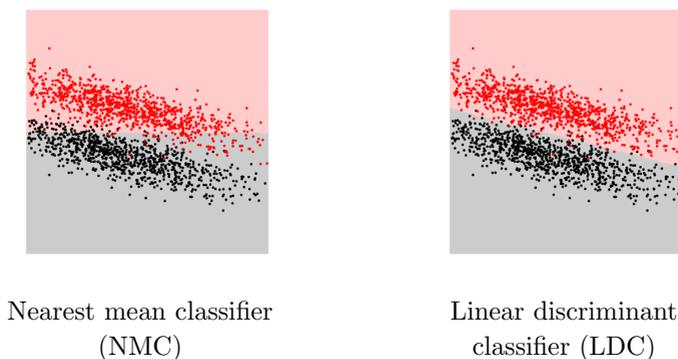


Figure 3.6: A data set and two sets of classification regions.

3.2.2 Linear boundaries

How do we train an LDC? Good question! There are many ways. Many. In the past, people focused a lot of effort on this [7]. But not anymore. Linear boundaries are not very interesting these days. Even the perfect linear boundary is limited. It cannot solve the “XOR” problem. The XOR problem is like this: Class 1 contains 2 objects described by 2 features $[0, 0]$ and $[1, 1]$. Class 2 contains two objects: $[0, 1]$ and $[1, 0]$. Picture them in your head – these are the 4 corners of the unit square; two opposite corners in each class. No line in the world can separate these two classes without an error. Luckily, we have a lot more versatile classifiers which can do that. But don’t dismiss yet the good old linear classifier. Chances are that most real-life problems can be solved very successfully with LDC! If nothing else, keep LDC in your toolbox as a baseline model.

OK but how do we train the LDC? That is, how do we find the coefficients and the free terms for the discriminant functions.

One possible way is given in Figure 3.7. Unfortunately, this

method does not always work. The problem is that the sometimes the pooled covariance matrix $\hat{\Sigma}$ is singular or very close to singular, and when we try to invert it in step 5, bad things will happen. Misbehaving Sigma! But we are not going to trouble ourselves with mischievous covariance matrices. Not in your assignments and the exam anyway ☺.

Now, this is what you should be able to do. Given a data with two classes in 2D, you should be able to plot the data and design *by hand* a linear boundary between the classes. Then, based on this boundary, you should be able to propose a classification rule and come up with two discriminant functions on top of this.

⊕ ⊕ ⊕ Example 3.2.1

Take the data in Figure 3.5. Propose a linear boundary, a classification rule and two discriminant functions.

Solution. Call the classes Black and Red instead 1 and 2. Pick two points A and B so that the line through them separates the classes as well as possible as shown in Figure 3.8. Yees, all is done by eye here. But we will beat the nearest mean classifier, hands down!

Suppose we picked A at $(8, -6)$ and B at $(-6, 8)$. Let's calculate the equation of the line through A and B . Remember your school maths? (If not, look in Appendix A.)

$$\frac{x - 8}{-6 - 8} = \frac{y + 6}{8 + 6}$$

$$14(x - 8) = -14(y + 6)$$

$$x + y - 2 = 0.$$

The classification rule can be very simple. For any $\mathbf{x} \in \mathbb{R}^2$, substitute the coordinates in the left-hand-side and classify according to the sign of the sum. Positives versus negatives. The question now is

TRAINING OF LDC

1. Estimate the prior probabilities for the classes. Let N_i be the number of objects in the data set \mathbf{Z} from class ω_i , $i = 1, \dots, c$, and $y_j \in \Omega$ be the class label of $\mathbf{z}_j \in \mathbf{Z}$. Then

$$\hat{P}(\omega_i) = \frac{1}{N_i}, \quad i = 1, \dots, c. \quad (3.1)$$

2. Calculate estimates of the class means from the data

$$\hat{\boldsymbol{\mu}}_i = \frac{1}{N_i} \sum_{y_j = \omega_i} \mathbf{z}_j. \quad (3.2)$$

3. Calculate the estimates of the covariance matrices for the classes, by

$$\hat{\Sigma}_i = \frac{1}{N_i} \sum_{y_j = \omega_i} (\mathbf{z}_j - \hat{\boldsymbol{\mu}}_i)(\mathbf{z}_j - \hat{\boldsymbol{\mu}}_i)^T. \quad (3.3)$$

4. Calculate the common covariance matrix for LDC as the weighted average of the class-conditional covariance matrices

$$\hat{\Sigma} = \frac{1}{N} \sum_{i=1}^c N_i \hat{\Sigma}_i. \quad (3.4)$$

5. Calculate the coefficients and the free terms of the c discriminant functions

$$\mathbf{w}_i = \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_i, \quad w_{i0} = \log(\hat{P}(\omega_i)) - \frac{1}{2} \hat{\boldsymbol{\mu}}_i^T \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_i, \quad (3.5)$$

6. The discriminant functions are

$$g_i(\mathbf{x}) = \mathbf{w}_i \mathbf{x} + w_{i0}, \quad i = 1, \dots, c. \quad (3.6)$$

Figure 3.7: Training of LDC (for wizards).

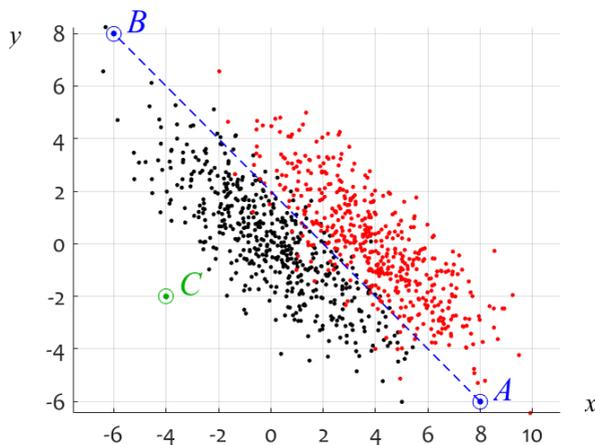


Figure 3.8: Constructing an LDC by hand

which ones are the positives? The blacks or the reds? Pick a point on the plane. Any point (like a card trick ☺). And... we pick $C(-4, -2)$, just because it is far enough to see on the graph in Figure 3.8.³ Substitute:

$$-4 - 2 - 2 = -8 < 0,$$

therefore the blacks are on the negative side of the line. The classification rule is: ‘assign class Black to $[x, y]^T$ if $x + y - 2 < 0$ and class Red, otherwise’.

We can stop here as the classifier is trained. All done. However, I still want you to be able to represent this linear classifier in the

³By the way, it is a lot simpler if we pick $C(0, 0)$ – fewer calculations.

form of two discriminant functions, however trivial. Here is one way:

$$\begin{aligned}g_{\text{Black}}(\mathbf{x}) &= 0, \\g_{\text{Red}}(\mathbf{x}) &= x + y - 2.\end{aligned}$$

Check it! First, both functions are linear functions of x and y . Observe that g_{Black} does not actually include any of the two variables, so we have $a_{\text{Black},1} = 0$ and $a_{\text{Black},2} = 0$. The free term also happens to be 0, that is, $b_{\text{Black}} = 0$. For any point (x, y) , we calculate the two discriminant functions. If $g_{\text{Red}}(\mathbf{x})$ is positive, given that $g_{\text{Black}}(\mathbf{x}) = 0$, we will always assign class Red. If $g_{\text{Red}}(\mathbf{x})$ is negative, given that $g_{\text{Black}}(\mathbf{x}) = 0$, we will always assign class Black. Works, doesn't it?

Well, there is one small question left. What if $g_{\text{Red}}(\mathbf{x}) = 0$? Then the two functions have the same value. This means that the point has been unfortunate enough to sit *exactly* on the border. Then any of the two classes may be assigned. No right or wrong decision. The best kind. $\ominus \ominus \ominus$

3.2.3 LDC in different dimensions

In \mathbb{R}^1 , a linear classifier is a threshold on $x \in \mathbb{R}^1$. Here is an example.

$\oplus \oplus \oplus$ Example 3.2.2



The training data set is a class of 40 students. The only feature x is a test score; a number between 0 and 100. Some good students

may have scored low on the test and some not so good ones may have fluked a high score. Based on their overall performance thus far in the course, the students in the class are labelled into three classes: (1) mediocre, (2) good, and (3) excellent. Figure 3.9 shows the labelled data.

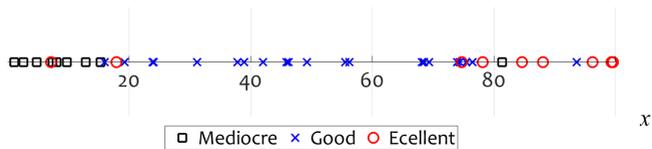


Figure 3.9: The 1D student score data.

Let's build an LDC which predicts the student class label based on their test score x . Following the algorithm in Figure 3.7, we obtain the following discriminant functions:

$$g_1(x) = 0.0216x - 1.5426$$

$$g_2(x) = 0.0773x - 2.6440$$

$$g_3(x) = 0.1071x - 5.3370$$

Figure 3.10 plots the three discriminant functions of x . The largest of the three functions will determine the label of the respective x . The intersection points will be candidates for the threshold values or, equivalently, the class boundaries. These are found by setting $g_1(x) = g_2(x)$, $g_1(x) = g_3(x)$, and $g_2(x) = g_3(x)$, and solving for x .

$$g_1(x) = g_2(x) \rightarrow x = 19.79$$

$$g_1(x) = g_3(x) \rightarrow x = 44.36$$

$$g_2(x) = g_3(x) \rightarrow x = 90.13$$

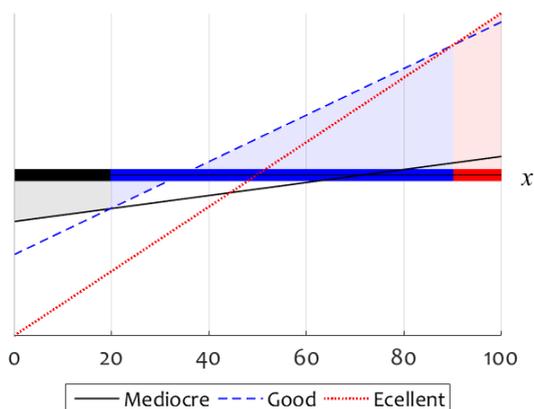


Figure 3.10: Discriminant functions and classification regions for the 1D student score data.

Inspecting the discriminant functions in Figure 3.10, we observe that the intersection between g_1 (Mediocre) and g_3 (Excellent) is under g_2 . This means that this point will not qualify as boundary as the dominant function is above it. Hence, the boundaries are $x = 19.79$ and $x = 90.13$. The classification regions are marked on the figure. Notice that these regions are *only* on the x -axis because this is our feature space. This is where all the points live.

The classification boundaries and the discriminant functions above determine one and the same classifier. $\ominus \ominus \ominus$

In 1D, the classification regions are intervals on the x -axis, one for each class. In 2D, the regions are convex, bounded or unbounded, separated by line segments (Voronoi cells), one per class. In 3D, the regions are convex volumes separated by planes. And in \mathbb{R}^n , we have hypervolumes, one per class, separated by hyperplanes (fancy that!). Thankfully, we don't need to draw classification regions in spaces beyond \mathbb{R}^2 . Fair enough, we don't need to draw them in \mathbb{R}^2 either, but I have to examine you on something!

3.3 Rule-based classifiers

Generally speaking, any classifier is based on at least one rule. An example is *the maximum membership rule*: 'Select the class with the highest value of the discriminant function'. Here we talk about other rules such as 'if feature x_1 is less than 6 and feature x_2 is more than 9, assign class ω_3 '.

3.3.1 If-then classifiers

The so-called if-then classifiers were in high fashion in the 1980s. They were the heart and the soul of the Expert Systems 'movement', hugely popular during that time. Expert Systems were the face of Artificial Intelligence at the time, and everybody was building them. In such a system, you draw a conclusion about a class label by traversing a large number of interlinked if-then rules and aggregating "the evidence". A mighty example of such a system was MYCINE. This was a famous expert system developed during the 1970 by the Stanford University, USA [18]. Its task was to identify bacteria

causing severe infections and to recommend antibiotics.

An expert systems can be developed using just knowledge from textbooks. For example, a diagnosis of a certain condition is done through a series of questions and tests. This knowledge is usually accumulated over many years and formulated as a set of rules. In this approach, there is no machine learning as such because raw data are not involved. All the steps from entering \mathbf{x} to receiving the class label for it are interpretable. A human can make sense of them. This was the biggest advantage of expert systems over the black-box approach where the classifier was built from data. There main drawback of such expert systems was their *brittleness*. Brittleness means that a complex system (many features and many rules) may not be able to handle all possible cases. Some parts of the feature space would not be covered by the rules, and the decision in those parts would be arbitrary.

In our definition, a classifier should be able to produce a class label for any \mathbf{x} coming as the input. Consider the data in Figure 3.11.

For this 2D example, we can devise a rule-based classifier by eye. The rules for the green and the blue classes are marked with red windows. Then the rule-based classifier operates according to the following rules:

- | | | | |
|---------------|--|------|-------------|
| (1) If | $x \geq 0.25$ and $x < 0.5$ and $y \geq 0.2$ | then | class green |
| (2) else if | $x \geq 0.55$ and $y < 0.35$ | then | class blue |
| (3) otherwise | | | class grey. |

Such a classifier is not that difficult to build when the dimensionality of the problem is small. There are possibly over hundred methods for rule induction from data. Most such methods are well equipped to accommodate the uncertainty (probabilistic or not) associated with if-then rules.

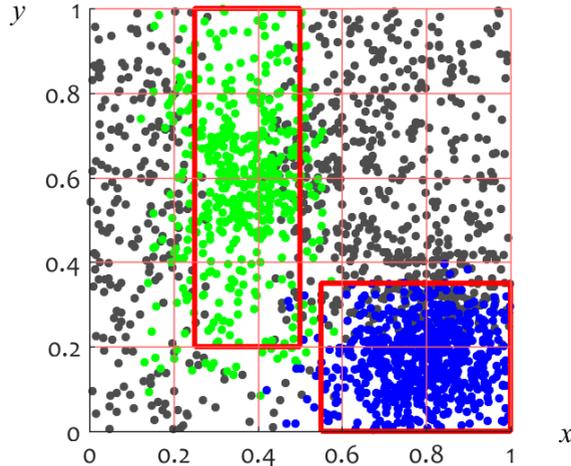


Figure 3.11: Data labelled in three classes and two windows defining classification if-then rules.

3.3.2 The ZeroR classifier

This classifier always assigns the label of the prevalent class. Simple as that! Regardless of what you measure and submit as the input \mathbf{x} . Everybody is in the same class. And how useless is this? ZeroR is also called *the Majority Classifier* or *the Largest-Prior Classifier*.

In the example in Figure 3.11, there are 831 grey points (42%), 480 green points (24%), and 689 blue points (34%). Answer this question: what would be the classification error of the ZeroR classifier for this data set?

Did you get it? If everything is classified as the largest class (grey), then all the green points (24%) and all the blue points (34%) will be mislabelled as grey. The total error will be $24+34 = 58\%$. We

can reach the same conclusion by calculating the error as $100 - 42\% = 58\%$.

3.3.3 The OneR classifier

The OneR classifier has one rule. This rule involves only one feature, and is typically a threshold-based rule. This means that you are allowed to assign only two class labels: one label if the feature value is lower than the threshold and the other label otherwise. What if we have three classes? Well, just two “privileged” classes will be given at the output of our OneR classifier.

How do we train OneR? Here are the steps:

1. For each feature x_i ($i = 1, \dots, n$):
 - (a) Create a set number of increasing thresholds between $\min(x_i)$ and $\max(x_i)$ as identified from data.
 - (b) For each threshold t_k , identify the largest class among all objects for which $x_i < t_k$. Store this class label (y_{Lk}) and the number of misclassified objects (L_k) assuming that all objects in this group are assigned to the largest class (ZeroR classifier on this part of the data set).
 - (c) Apply the previous step to all objects such that $x_i \geq t_k$. Record the class label (y_{Rk}) and the number of objects misclassified by ZeroR (R_k).
 - (d) Store the threshold value, the two class labels, and the total number of misclassifications: $\langle t_k, y_{Lk}, y_{Rk}, T_k \rangle$, where $T_k = L_k + R_k$.

Note that at this step you have an instance of the OneR classifier that uses feature x_i , threshold t_k , and outputs: class y_{Lk} if $x_i < t_k$ and class y_{Rk} otherwise.

- (e) Find the index of the best threshold for feature x_i , that is

$$i^* = \arg \min_{k=1}^K \{T_k\}.$$

Denote the winning combination by $\langle t_i^*, y_{L_i}^*, y_{R_i}^*, T_i^* \rangle$ and save it.

At this point you have *the optimal* OneR classifier, should feature x_i be selected as the final single rule. Along with the knowledge of which feature it is, we have stored the optimal threshold, class labels for the left and the right regions, as well as the total misclassifications for this choice.

2. Identify the feature with the smallest total number of errors:

$$j = \arg \min_{i=1}^n \{T_i^*\}.$$

3. Return the trained OneR classifier:

- Feature: x_j
- Threshold: t_j^*
- Class labels: All objects (points) for which $x_j < t_j^*$ should be labelled as $y_{L_j}^*$, and all objects (points) for which $x_j \geq t_j^*$ should be labelled as $y_{R_j}^*$.
- Errors: The value of T_j^*

The sequence of steps only looks complicated. It is pretty simple, really. Check all features one-by-one and all possible thresholds for each feature by applying ZeroR for the data on each side of the threshold for that feature. Eventually, select the feature and its threshold that gave you the lowest number of misclassifications.

Here is my wonderful example, which will help you understand the OneR classifier (fingers crossed!).

⊕ ⊕ ⊕ **Example 3.3.1**

The data set is shown in Figure 3.12. There are Humans and Martians.



Figure 3.12: The “Martians” data set for Example 3.3.1. (The Martians are fondly depicted as cats, blatantly disregarding the issue of political correctness ☹.)

We all know that Martians are little green men. Hm, yeah, I couldn’t find a decent silhouette icon for a Martian, so my Martians in the data set suspiciously resemble cats. Like everything else in the world, not all people are equal. So I have a big Martian (left) and a green Human (the lady in the middle).

You are an OHIGA (oh-so-important governmental agent) and your task is to build an OneR classifier to recognise Martians from Humans. You can only measure two features: colour and height. But remember, you are only allowed to use one of your features in the OneR classifier. Luckily, your features have only two values each: colour $\in \{\text{green, black}\}$, and height $\in \{\text{short, tall}\}$, so the classifier will be easy to train. Here goes:

(1) Take feature $x_1 = \text{‘colour’}$ first. As there are two values, there is only one dividing point (playing the role of the threshold). Let’s say left is ‘green’. Take only the green individuals. There are 3 Martians and 1 Human. ZeroR will say that they are all Martians. So we store label $y_L = \text{‘Martian’}$ and $L = 1$ (one error, the poor green lady). Notice that there is no index k because we have only one threshold.

For the right value, ‘black’, ZeroR will correctly label all 8 humans as humans. Then we store $y_R = \text{‘Human’}$ and $R = 0$. For the single possible threshold of feature x_1 , we have the following rule: Label all greens as Martians and all blacks as Humans. This classification will result in 1 error for the given data (Figure 3.13 (a)).

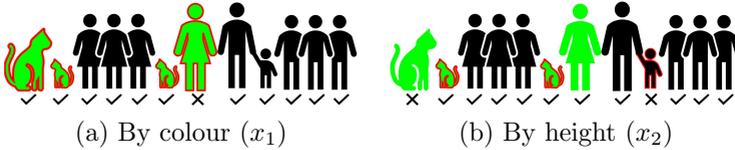


Figure 3.13: The two instances of OneR classifier for the Martians data set.

(2) Take now feature $x_2 = \text{‘height’}$. As there are two values, there is only one dividing point again. Let’s say left is ‘short’. Take only the short individuals. There are 2 Martians and 1 Human. ZeroR will say that they are all Martians. So we store label $y_L = \text{‘Martian’}$ and $L = 1$ (one error, the child). For the right value, ‘tall’, ZeroR will correctly label the 8 humans as humans but will mislabel the tall Martian as Human. Then we store $y_R = \text{‘Human’}$ and $R = 1$. For the single possible threshold of feature x_2 , we have the following rule: Label all shorties as Martians and all tall ones as Humans. This classification will result in 2 errors for the given data.

(3) It is time to choose the one feature to take forward. As x_1 (colour) gives fewer errors, we return the respective OneR classifier: Label all greens as Martians and all blacks as Humans. Job done!

Of course there are other features which we could have used. For example, “if it has a tail, it is most definitely a Martian”. However, in real life, when feature representation is being decided upon, it is not clear what variables, parameters, tests, questions, etc., are

important. The domain expert usually has a good idea about what to measure. Sometimes this is a very difficult problem, and many features are included in the data just in case. $\ominus \ominus \ominus$

Remember the data in Figure 3.11? Recall that there are 831 grey points (42%), 480 green points (24%), and 689 blue points (34%). Answer this question: what is the *minimum* possible error rate for the OneR classifier for this data set? And no, I am not giving you the answer yet.

Figure 3.14 shows the trained OneR classifier for this data set. The best feature is y and the best threshold for y is $t_y^* = 0.2626$. The OneR classifier works as follows:

Assign label ‘blue’ if $y < 0.2626$ and label ‘grey’ if $y \geq 0.2626$.

The error rate for this data set is 0.3385 as shown in the figure.

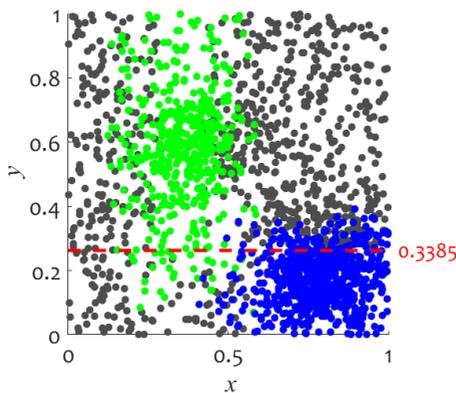


Figure 3.14: The trained OneR classifier for the data set in Figure 3.11.

Are you ready to answer that question yet? About the minimum possible error rate. Alright. Look at the example. The trained OneR does not give a damn about the poor green class. All those green point will be labelled as either grey or blue. But this is the least of evils because any other choice of feature/threshold will lead to higher error. OneR's fault. The model is too simple. So back to the question, if we have more than two classes, all classes but two will be misclassified. So, to minimise the error, OneR should choose the two most populous classes. Suppose there is a feature which discriminates perfectly between these two classes, regardless of the distribution of the points from the other $c - 2$ classes. Then this is the best that OneR can offer. The smallest possible error will be therefore the sum of the proportions of all smaller classes. In the 3-class example in Figure 3.11, the minimum possible error will be 0.24 (the proportion of the green class).

I am sure you can program OneR in a flash! But it may not be worth the effort. Well, unless I ask you to do so for your assignment ☺.

3.4 The Nearest Neighbour classifier (1-nn and k-nn)

The nearest neighbour classifier is just wonderful. It is intuitive, elegant, and, quite often, amazingly accurate. Its philosophy is like this. Assign the new object to the class of the object most similar to it. Thus, if we have an object \mathbf{x} to classify, and a set of labelled past examples, we look through them for the closest match to \mathbf{x} (called the 'nearest neighbour') and label \mathbf{x} in the same class.



3.4.1 Distances

‘Closest match’ requires a measurable concept of similarity or distance between a pair of objects. The most popular choices for a distance in \mathbb{R}^n are the Euclidean distance and the Hamming distance. Given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, the Euclidean distance is calculated as follows (see Appendix A)

$$d_E(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2} = \sum_{i=1}^n (x_i - y_i)^2,$$

where x_i and y_i are the individual features of objects \mathbf{x} and \mathbf{y} , respectively.

The Hamming distance has several different guises. You may come across it as ‘Manhattan distance’, ‘city block distance’, ‘Minkowski distance’, or ‘ L_1 norm’. Wha-ha! All the same. The Hamming distance is calculated as

$$d_H(\mathbf{x}, \mathbf{y}) = |x_1 - y_1| + \cdots + |x_n - y_n| = \sum_{i=1}^n |x_i - y_i|.$$



Curiously, the names ‘Manhattan’ and ‘city block’ come from the cosmopolitan New York district! Imagine that the streets are orthogonal to the avenues and all blocks are perfect squares.

To make things simple, suppose that the avenues (x -axis) go from left to right (not exactly true) and the streets (y -axis) go from bottom to top. If you want to go from the corner of 5th Avenue and 34th Street (point $A(5, 34)$) to the corner 8th Avenue and 36th

Street (point $B(8, 36)$) you have to walk 3 blocks right and 2 blocks up (well, unless you have a superpower to go through walls). So, you will have to walk $3 + 2 = 5$ blocks at least. Even if you pick another (shortest) way, say, 1 right, 1 up, 1 right, 1 up, 1 right, you still have to walk 5 blocks. Indeed, the Manhattan (Hamming) distance is

$$d_H(A, B) = |5 - 8| + |34 - 36| = 3 + 2 = 5.$$

3.4.2 1-nn and k-nn

The k -nearest neighbour algorithm (k-nn) is described formally in Figure 3.15. For $k = 1$, this becomes just the nearest neighbour algorithm, 1-nn.

THE K-NN ALGORITHM

Input: A labelled data set Z , a distance, the desired number of neighbours k , and a query point \mathbf{x} (to be labelled).

Training: Store the labelled data set Z (the reference set).

Operation:

1. Ignoring the class labels, identify the k points from Z closest to \mathbf{x} in terms of the chosen distance.
2. Retrieve the labels of the k points found at Step 1.
3. Identify the majority^a class among the retrieved labels and return it as the label of \mathbf{x} .

^a‘plurality’ is a more accurate word here

Figure 3.15: Training and operation of the k -nn algorithm.

The training of k -nn is minimal. Just store the data. Training done! The operation is also pretty simple. Identify the k nearest points to the query point \mathbf{x} and take the most represented label amongst the k labels to be the label of \mathbf{x} .

The following example illustrates 1-nn and k -nn in \mathbb{R}^2 .

⊕ ⊕ ⊕ Example 3.4.1

We have three classes shown with green, black and blue markers in Figure 3.16, and an alien, the yellow triangle, which desperately yearns to belong somewhere. Which class label shall we give him? According to the single nearest neighbour (Figure 3.16 (a)) the alien should be in the green class.

But if our yellow alien looks a bit further afield, and checks its five nearest neighbours (Figure 3.16 (b)), he should rather think he belongs to class black.

So, where does the truth lie? Should we colour our alien green or black? The truth is that k is a parameter of the k -nn classifier and should be tuned using some cross-validation protocol.

⊖ ⊖ ⊖

3.4.3 k -nn in MATLAB

How do we code k -nn? If you are a proud owner of the Statistics Toolbox of MATLAB, you can just use:

```
1 knn_classifier = fitcknn(training_data,training_labels,...
2   'NumNeighbors',3);
3 assigned_labels = predict(knn_classifier,training_data);
```

But we want to be able to program k -nn ourselves! Or at least 1-nn. Let's follow the Algorithm in Figure 3.15. Assume Euclidean

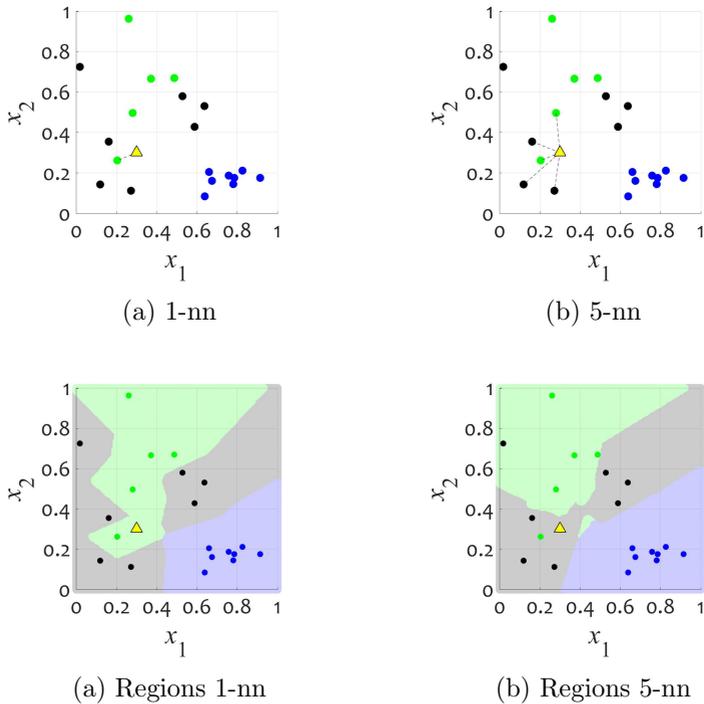


Figure 3.16: An example of 1-nn and 5-nn on a small random 2D dataset.

distance and a set of points to label rather than a single query point \mathbf{x} . One possible MATLAB implementation is given below:

```

1 function assigned_labels = knn_classifier(training_data,...
2 training_labels, k, testing_data)

```

```

3 % Returns a column of class labels for the testing points.
4
5 N = size(training_data,1); % number of training points
6 M = size(testing_data,1); % number of testing points
7
8 for i = 1:M % for every testing point
9
10     x = testing_data(i,:); % construct x
11
12     d_E = sqrt(sum((training_data - repmat(x,N,1)).^2,2));
13     % d_E is a column with the Euclidean distances ...
14     % and the points in the training data
15
16     [~,index] = sort(d_E);
17     % index contains the neighbours (all of the ...
18     % arranged from nearest (top) to farthest (bottom)
19
20     assigned_labels(i,1) = mode(training_labels(index(1:k)));
21     % assign the label most represented (hence "mode") among
22     % the top k neighbours
23
24 end

```

Sooo, easy! ☺

k -nn is a lovely and accurate classifier but would require a lot of memory resources for a big reference set Z . And, of course, it will need a lot of computing power to calculate the distances. Hence, much effort has been devoted over the years to developing ingenious ways for quickly finding the nearest neighbours, either exactly or approximately.

Here is my little k -nn challenge for you. Answer the following questions: If your data contains N points, N_i from class i , $i = 1, \dots, c$,

1. What is the resubstitution error of 1nn?

2. What is the resubstitution error of N -nn?
3. What is the resubstitution error of $(N - 1)$ -nn? (assuming that the largest class contains at least two more points than the second largest class)

3.5 Decision tree classifiers

3.5.1 What is a decision tree classifier?

The decision tree classifier takes \mathbf{x} at the root and passes it through a series of nodes until it reaches a leaf. At the leaf, \mathbf{x} gets its class label. The nodes between the root and the leaf are called ‘intermediate nodes’, and the decision there is which branch to follow next. An example is shown in Figure 3.17. Shown in the figure is a decision tree for distinguishing between three types of adult bears.⁴ The three classes are



The features are coloration, size and attack on humans. (Curiously, almost white individuals of American Black Bear could be found in the north-western region of North America).

A special case of the decision tree classifier is the one-node tree, aptly called a ‘decision stump’.

In the standard version of a binary decision tree classifier, each non-leaf node has two children nodes – left and right. At the parent node, one feature, say x_k is compared with a threshold, say θ . If $x_k \leq$

⁴Based upon information found at <http://www.bears-bears.org/>



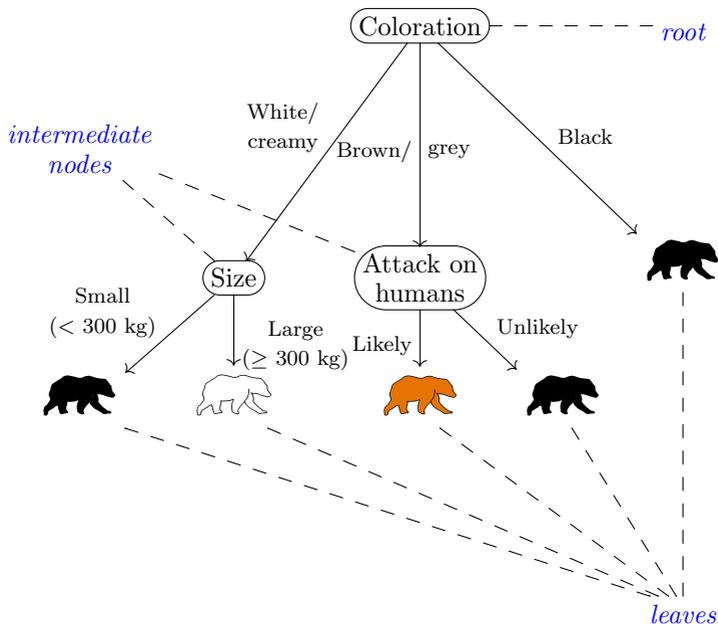


Figure 3.17: An illustration of a decision tree and related terminology.

θ , go left, otherwise, go right. Training of the decision tree amounts to identifying the tree structure (very automatically!), which feature should be used at each node (also determined through the training process), and what label should be assigned at each leaf.

3.5.2 Why are decision trees good?

What is so good about decision tree classifiers?

1. They can handle irrelevant and redundant features. Each split uses a single best feature, hence irrelevant features may never be picked during training.
2. Continuous-valued, discrete and categorical features can be handled together; there is no need to convert one type into another.
3. Scaling of the features does not matter. Since each feature is handled separately to find a bespoke threshold, there is no need to normalise or re-scale the data to fit into a given interval. because of this, decision trees are called *non-metric methods* for classification [7].
4. If all the objects are distinguishable, that is, there are no identical elements of Z with different class labels, then we can build a tree classifier with zero resubstitution error.
5. Tree classifiers are intuitive because the decision process can be traced as a sequence of simple choices. Tree structures can capture a knowledge base in a hierarchical arrangement, most pronounced examples of which are botany, zoology, and medical diagnosis.
6. Training is reasonably fast while operation can be extremely fast.

3.5.3 Training of a decision tree classifier

The training of the decision tree classifier is similar to the training of the OneR classifier. However, it is applied recursively. A recursive algorithm is shown in Figure 3.18.

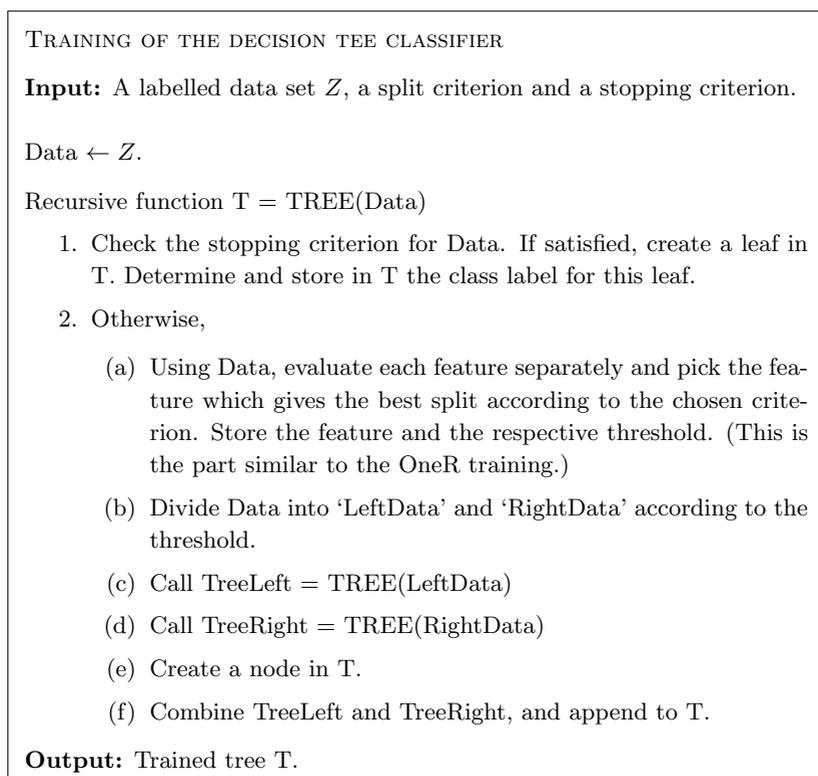


Figure 3.18: A recursive algorithm for training a decision tree classifier. And, no, I don’t want you to program it yourself.

⊕ ⊕ ⊕ Example 3.5.1

A small 2-class 2D data sets is shown in Figure 3.19.

The tree for this data set is shown in Figure 3.20. Assume that the stopping criterion is achieving ‘pure nodes’, that is the labels of all data points coming to the node is the same (all points are red or

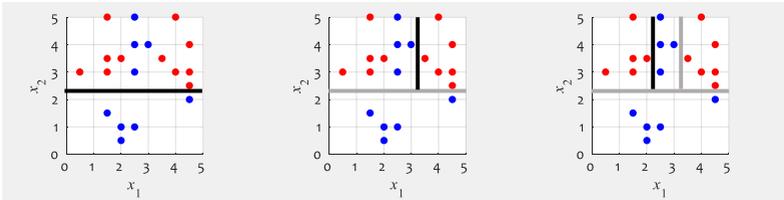


Figure 3.19: Stages of training the decision tree classifier.

all points are blue).

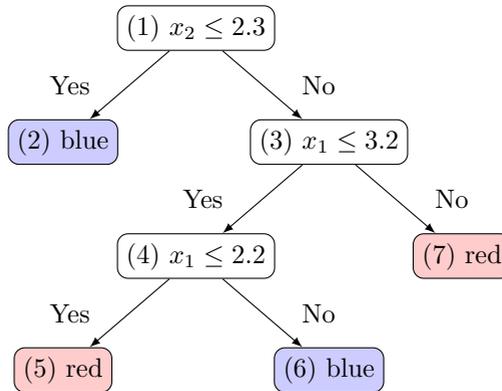


Figure 3.20: Decision tree for the 2D data. The construction steps are depicted in Figure 3.19.

⊖ ⊖ ⊖

The training algorithm starts at node (1) and splits the data on x_2 . It will create a leaf with label ‘blue’ for the left child node of the root for $x_2 \leq 2.3$ (node (2)). It will next pass the top part of the data to the right child node (node (3)). The best split for this part of the data is shown in the middle plot of Figure 3.19. The data

is split again into left and right parts according to $x_1 \leq 3.2$. The left part will contain both red and blue labels (node (4)), hence the algorithm is called again and the data is split into two pure nodes (leaves, (5) and (6)). The right data from the split $x_1 \leq 3.2$ have only red labels, hence leaf (7) is created with label “red”.

Notice something interesting! Since only one feature is used at each node, the classification regions will always be separated from one another with sets of lines parallel to the coordinate axes.

3.6 The Support Vector Machine (SVM) classifier

Ever since its conception in the 1990s, the Support Vector Machine classifier (SVM) has been a prominent landmark in statistical learning theory. The success of SVM can be attributed to two ideas (Figure 3.21): (1) a transformation of the original space into a very high-dimensional new space and (2) identifying a ‘large margin’ linear boundary in the new space.

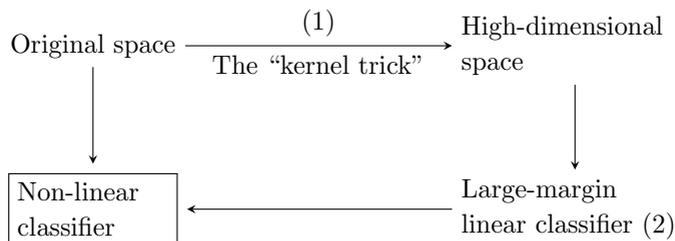


Figure 3.21: Illustration of the idea of the SVM classifier. The desired classifier is in the original space (shown in the box).

To explain why the first idea works, consider the one-dimensional

two-class data set shown with two types of markers, both classes in grey, on the x -axis in Figure 3.22.

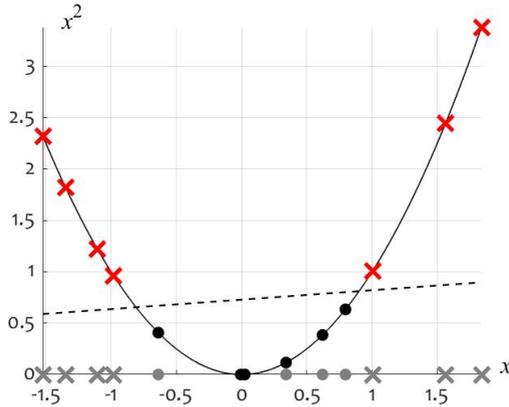


Figure 3.22: One-dimensional data set, which becomes linearly separable in the space (x, x^2) .

The two classes are not linearly separable in the one-dimensional space x . However, by adding a second dimension, calculated as x^2 , the classes become linearly separable. The linear classification boundary in \mathbb{R}^2 is shown in the figure with a dashed line.

The second idea, also a hallmark of SVM, is the large-margin classifier. Figure 3.23 illustrates the margin concept.

Shown in the figure are two identical data sets and two linear classifiers separating perfectly the two classes. In the left plot, the distances between the points closest to the boundary (from both classes) are smaller than the respective distances in the right plot. If we choose the classifier with the smaller distances (left), a small fluctuation of a coordinate may shift the point to the other side of the border, and therefore place it in the wrong class. A much larger

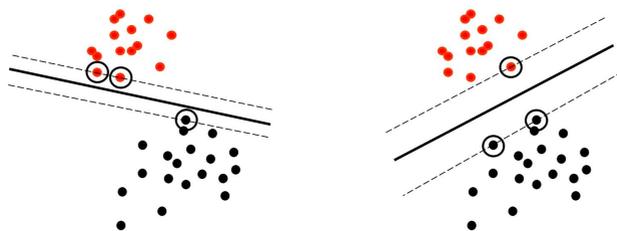


Figure 3.23: Margins of two SVM classifiers for the same data set.

fluctuation will be needed for the same effect for the classifier in the plot to the right. This is why the classifier with large margins is deemed to be more robust and therefore should be preferred. SVM has a cute algorithm for identifying the boundary that is farthest away from the nearest points from both classes. The points closest to the boundary (circled in both plots) are called the “support vectors”, hence the name Support Vector Machines.

Now, the example in Figure 3.23 is in 2D, but this is our ultra-simplified illustration. All this happens in the VERY-high-dimensional feature space and the line that separates the classes is, in fact a hyperplane. Not to worry! The SVM algorithm knows exactly how to calculate it.

The presumption here is that after the transformation to the new very-high-dimensional space, the classes will be linearly separable there. This is not always the case. Therefore SVM has a regularisation parameter, often denoted by C which, in a way, softens the separability restriction.

SVM is tricky to program but there are plenty of software packages for machine learning and pattern recognition which can be used. SVM is included in the MATLAB Statistics Toolbox. Originally SVM was developed for two classes. Most modern packages will

have multi-class extensions.

SVM was the reigning champion for a long time before being unceremoniously dethroned by Deep Learning Neural Networks, whom we will meet later.

3.7 Classifier ensembles



Classifier ensembles are all about teamwork! Why use one classifier if we can consult several of them and make a more informed decision about the class label of the point of interest?

This is my forte! I can talk about classifier ensembles until your ears fall off. But here we will only have a tiny slice of the pizza.

3.7.1 Why will classifier ensembles work?

This is something like the wisdom of the crowd. There is an old story about gauging the weight of an ox in a country fair in the early 1900. A lot of participants entered their guesses. Some participants underestimated the weight while other overestimated it, maybe by little or maybe by much. It turned out that the average of all the guesses was extremely close to the true weight of the ox. This is the idea behind classifier ensembles too. If we train each individual classifier to learn a slightly different perspective of the data, when we combine their class label predictions, we may rectify individual mistakes, and achieve a more accurate overall decision.

Any classifier model can be used in the ensemble. The individual classifiers are called *base classifiers*. The key to the success of classifier ensembles is to create individually *accurate* and *diverse* base classifiers.

The base classifiers can be of the same type, for example decision trees, which makes the ensemble *homogenous*. If the base classifiers are of different types, for example, some are decision trees while others are 1-nn, the ensemble is called *heterogeneous*.

To enforce diversity, the base classifier model has to be *unstable*. This means that the classifier should be sensitive to the training data. Small changes in the training data should lead to significant changes in the classifier. The decision tree classifier is an example of this type. If we make a small alteration of the training data, the splits may be different, and so will be the classification regions. This does not mean that the classifier will be less accurate! It will just follow a different training path.

Figure 3.24 illustrates the ensemble idea with an artificial example. The two classes are pink (background) and blue (a 2D torus shape) and the features are the x and y coordinates of the points. The figure demonstrates that the ensemble regions are more accurate than those of the individual classifiers.

3.7.2 Bagging

Bagging is the most intuitive classifier ensemble method. It is rather an approach which can be perfected further. In bagging, each classifier is trained on a *bootstrap sample* of the training data. A bootstrap sample is obtained by sampling with replacement. Therefore, we may have more than one copy of an object from the training data in the sample.

In MATLAB, we can create a bootstrap sample by generating random numbers between 1 and N (number of objects in the data),

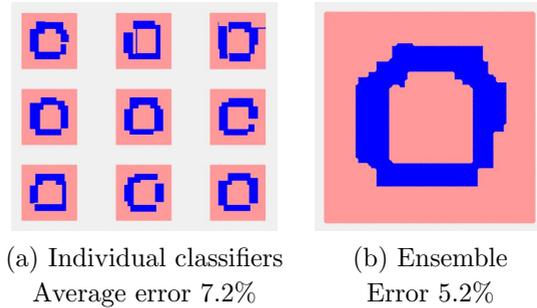


Figure 3.24: Illustration of the classification regions of individual classifiers (subplot (a)) and the regions of the ensemble (subplot (b)).

without guarding against repeated numbers. We can then use these numbers as indices, and pull out the respective rows of the data set as our new training data. This new training data will be used for one of the classifiers in the ensemble.

Denote the training data set by Z and the respective vector-column with labels by Y (see Section 1.3). Z will have N rows (objects) and n columns (features). Denote by ZZ the new data set and by YY the corresponding labels. Here is the code for obtaining a bootstrap sample (ZZ, YY) from (Z, Y) :

```

1 N = size(Z,1); % number of objects
2 index = randi(N,1,N); % indices
3 ZZ = Z(index,:); % bootstrap sample (objects)
4 YY = Y(index); % bootstrap sample (labels)

```

We need to decide how many classifiers we want in the ensemble (say, L). Depending on the task, we may choose $L = 3, 100, 2000$, and so on. Then we sample a different bootstrap sample for each

classifier. The ensemble is ready for operation after all classifiers are trained on their respective training data. In the operation phase, \mathbf{x} is submitted to all classifiers, and each classifier predicts a class label. Take majority (plurality) vote to assign one final label to \mathbf{x} .

We will use a simple mnemonic notation to remind us about what each ensemble method does. For bagging, we take *independent* bootstrap samples. This is illustrated in Figure 3.25.

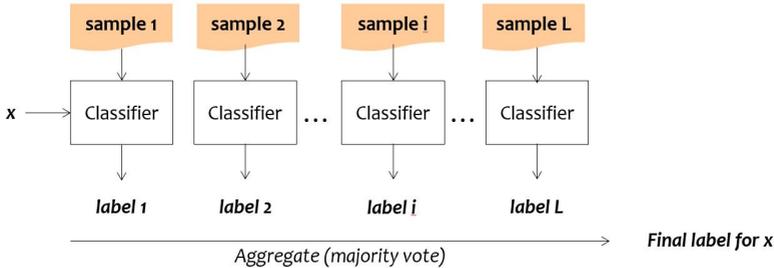


Figure 3.25: An illustration of Bagging.

The Bagging algorithm is shown more formally in Figure 3.26.

3.7.3 Boosting

In boosting, the training datasets for the base classifiers are not sampled independently. The rationale is as follows. The objects which are more difficult to classify correctly should be picked more often in the training data. The first classifier picks a standard bootstrap sample to train on. When sampling for the next base classifier, however, we increase the chances of objects misclassified by the first classifier to be picked. Now we have an ensemble of two classifiers. The training sample for the third classifier will contain more copies of objects which were misclassified by the ensemble so far, and so

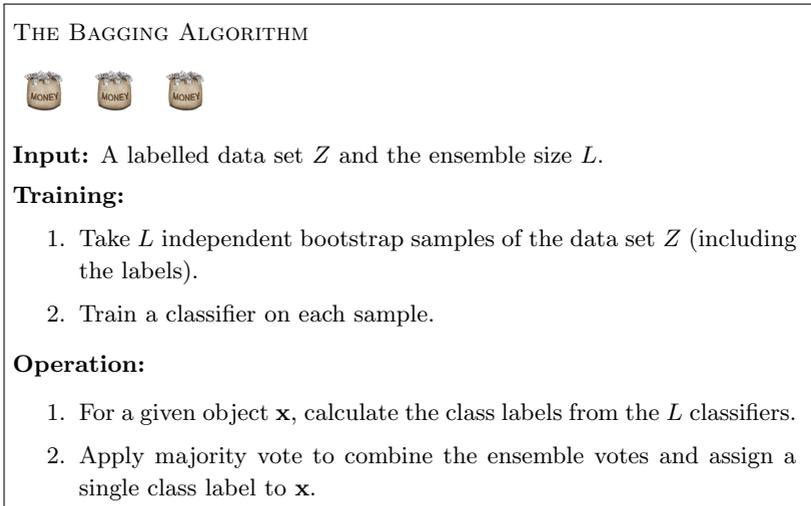


Figure 3.26: Training and operation of the Bagging classifier ensemble algorithm.

on. If this sounds too complicated, just remember that the samples are dependent. This is illustrated in Figure 3.27.

The most notable example of a boosting classifier ensemble method is AdaBoost [20]. And the most famous application of AdaBoost is in the Viola-Jones method for detecting faces in images. You have AdaBoost working tirelessly in your phone – that little yellow rectangle around the face in an image when you point the camera at one or more people is the Viola-Jones’s AdaBoost in operation.

3.7.4 Random Subspace

This time we sample from the features. Each classifier gets a random subsample of k columns of Z and all of its rows (all objects). This

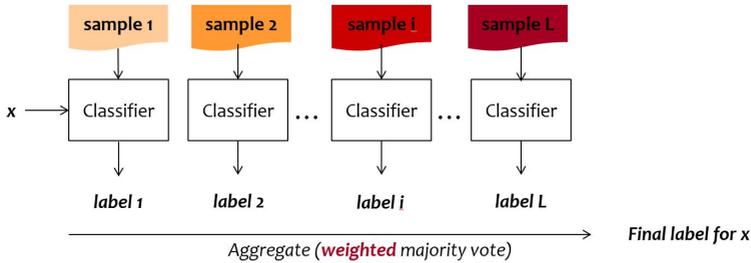


Figure 3.27: An illustration of Boosting. Different colours indicate that the samples for the different classifiers in the ensemble are dependent.

is why the samples are rotated at 90 degrees in Figure 3.28, which illustrates the Random Subspace ensemble method. The subsamples are taken independently of one another.

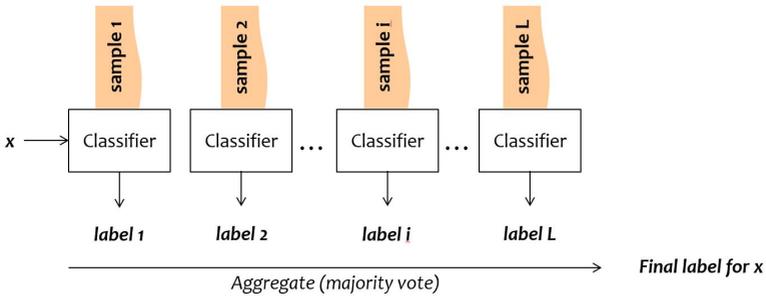


Figure 3.28: An illustration of Random Subspace.

Like Bagging and unlike AdaBoost, Random Subspace is easy to implement. The algorithm is very similar to Bagging in Figure 3.26. The only difference is in training step 1. Instead of a bootstrap

sample from the data, we take a subsample of the columns of Z . The number of features to sample, k is a parameter of the algorithm and should be specified as one of the inputs. You can try to write your own MATLAB code.

3.7.5 Random Forest

Random Forest [4] (Figure 3.29) is one of the most successful classifier ensemble methods. Delgado et al. [8] carried out a massive experimental comparison of classifiers in an attempt to answer the provocative question: Do we need hundreds of classifiers to solve real-world classification problems? A staggering 179 classifiers from 17 families were compared on 121 data sets. And the authors' answer is no! We don't need hundreds of classifiers. The current favourites are Random Forest [4] and the SVM [5]. Ah, wait, but there is a new kid on the block! Rotation Forest [16] beats them all according to a more recent study by Bagnall et al. [2]. (I am quite proud of this, actually, as I have a little contribution myself to the Rotation Forest ensemble method.)

The beauty of Random forest is in its simplicity. In fact, this is just Bagging as in Figure 3.26! The twist is that the base classifiers used in the ensemble are *random trees*.

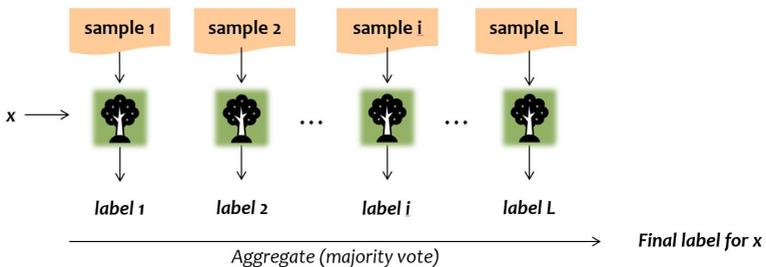


Figure 3.29: An illustration of Random Forest.

The difference between the decision tree which you saw in Section 3.5 and a random tree is subtle, but very powerful in creating a diverse ensemble. Just genius! Suppose we have reached node x in the training process of the tree. Instead of checking all possible features so as to pick the best split, we subsample M of the n features and only select among the sampled ones. A fresh sample of M features is drawn for each intermediate node. At the testing stage, a random tree is exactly the same as a standard tree.

There are many more classifier ensemble methods; you can take my word for it! But enough is enough! You are now well prepared to face the world of classification ☺.





Chapter 4

Feature Selection

Feature selection is a huge topic in pattern recognition. Over 25,000 papers were published in the past 10 years containing “feature selection” in their title or abstract. Who is going to read all these papers, I wonder. What is feature selection and why is it so important?

Suppose that you are working with a bioinformatics team who are analysing microarray data. They wish to know which collection of genes out of a set of 7,000 genes are the most relevant in distinguishing between a healthy tissue and a cancerous tissue. Feature selection is about answering this question.

4.1 Redundant, irrelevant and useful features

A feature x is a good feature if we can predict the class labels using its value. In real data, a single feature is almost invariably insufficient for a good class prediction, which is why we have a set of features X . It is important to make sure that the set of features

which we select from X is good *collectively*. Such a set, ideally, will not contain any redundant features or irrelevant features.

Redundant features are the ones which do not contribute any new discriminative information to the subset already selected. For example, if there is one feature that predicts the class labels perfectly, all other features are redundant. *Irrelevant* features are those which are not related to the class label variable. This means that we cannot predict the class label by observing the feature value. If a feature is irrelevant, it is redundant too because it would not contribute any new discriminative information.

Figure 4.1 demonstrates the difference between redundant, irrelevant and useful features. Suppose that we have only two features, x and y . Then the data lives in a 2D space. The classes are shown with different colours in the figure.

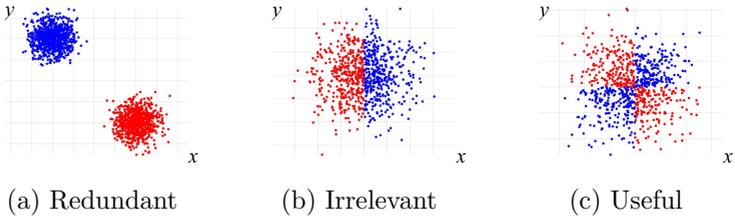


Figure 4.1: An example of redundant, irrelevant and useful features.

Either x or y in subplot (a) is redundant because the labels (red and blue) can be predicted without an error by the other feature. In subplot (b), feature x is perfect but feature y contains zero discriminative information. If we know the value of y only, we will have no way of saying with any certainty whether the point is red or black. y is both irrelevant and redundant. Finally, look at subplot (c). Both x and y are irrelevant. If we know only x or only y , we will be none the wiser which class label to assign. However, when we have x and

y together, we can label the point without any error.

This brings us to the most important point in feature selection. Some features which are irrelevant individually may be invaluable in a group with other features.



This is like mixing ingredients to achieve a certain property. Taken one by one, none of the ingredients alone will give us the desired property. We need a certain *combination* of ingredients which interact with one another.

Some of the ingredients together may give us a close enough result, some other combinations may be worse, regardless of whether we remove an ingredient or put a new one. And the worst thing is that we can't predict how the ingredients will behave together. For example, we can't assume that two individually good features will make a good pair. All depends on what happens when they are mixed!

So, the million-dollar question is: How do we identify those features that are relevant together?

4.2 A taxonomy of approaches

Before we look at the fun algorithms for feature selection, I am giving you an elaborate diagram (Figure 4.2) with a lot of unfamiliar words in it. Bear with me, please. We'll see what these are in a minute.

Feature selection is a branch of what is known as *dimensionality reduction*. We start with n features but wish to build our classifier on m features, where $m < n$. Sometimes, 'feature selection' is used as a synonym of 'dimensionality reduction'. Weird, right?

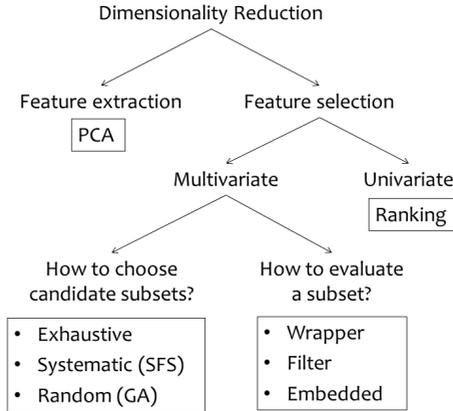


Figure 4.2: The dimensionality reduction (feature selection) taxonomy.

4.3 Feature extraction

The two branches are *features extraction* and *feature selection*. In feature extraction, we take the whole feature space and transform it into a new space of a smaller dimension. This can be done, for example, by multiplying the vector $\mathbf{x} \in \mathbb{R}^n$ by a transformation matrix T of size $m \times n$, that is $\mathbf{x}' = T\mathbf{x}$, and $\mathbf{x}' \in \mathbb{R}^m$. This is a *linear* transformation because the components of \mathbf{x}' are obtained by linear combinations of the components of \mathbf{x} .

A prime example of feature extraction is the Principal Component Analysis (PCA) method. PCA calculates T using a dataset (without labels). This method rotates the original space in such a way, that the components of \mathbf{x}' are arranged by how much they explain the variability of the data. The most important components

are at the front. Last components are thought to be irrelevant because the data has almost the same value for each such component. Initially T is an $n \times n$ matrix and \mathbf{x}' has n components itself. (But what use is that? We wanted to *reduce* n !) The solution is to keep the first m components and discard the remaining (less important) $n - m$ components. This means that we take only the first m rows of T to make sure that $x' \in \mathfrak{R}^m$.

PCA is often used to project the data in 2D so that we can *look* at it. In other words, we cut and keep only the first two components and plot the data in the space they span. If we also include the labels in the plot, we can see how the classes are behaving; whether they are easily distinguishable or not.

Notice two things. First, we do not reduce the original feature set! We will still need all the features in \mathbf{x} in order to calculate \mathbf{x}' . Second, we didn't mention class labels in the calculation of T . This means that PCA does not care whether the features are relevant or redundant in regard to classification. It only cares about preserving as well as possible the variability of the data, so that we don't lose much information when using the new feature space.

4.4 Feature selection

Feature selection, on the other hand, reduces the original set of features. It branches into two strands in our diagram: Univariate and Multivariate.

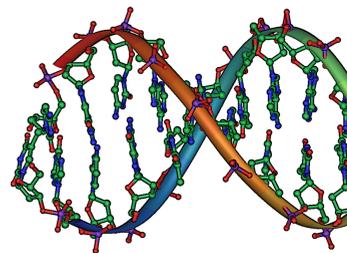
4.4.1 Univariate feature selection

The reason for this distinction is that there is an abundance of univariate methods which is dictated by the specifics of real life problems. In univariate feature selection, we evaluate each feature sepa-

rately and then choose the individually best m features (Ranking). I know, I know, we were just saying how important it is to consider the features together! The problem is that, in many real-life datasets, the number of objects N is too small compared to the number of features n , and any judgement on the merit of feature combinations is unreliable. Remember the overtraining phenomenon – where the classifier learns the data very well but cannot generalise to new data. Something like this happens in datasets where $N \ll n$. Let's return to the example of selecting genes for discriminating between two classes of tissues. Usually the number of objects is less than hundred (for example, patients in a medical trial) while the number of genes is several thousands. If you picture the matrix of this dataset ($N \times n$), you will understand why such datasets are called “wide”.

4.4.2 Multivariate feature selection

If we have enough data, we can do multivariate feature selection. There are two questions to answer: (1) How do we pick the combinations which we want to evaluate? and (2) What criterion do we use to evaluate a given feature set? In answering these questions, researchers have come up with a wonderful variety of methods. Look at the beautiful and exotic examples which I selected for you in (Figure 4.3), which all take inspiration from nature. How do animals look for prey, camouflage themselves, organise their community, locate relevant objects, coordinate their foraging? Other feature selection algorithms follow processes and phenomena from physics, for example particle swarm movement.



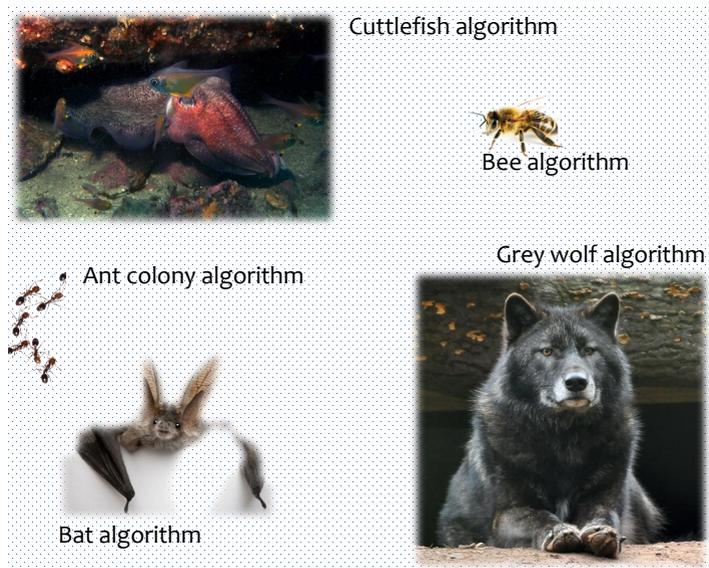


Figure 4.3: What is in fashion in feature selection today?

How do we pick feature combinations to evaluate?

There are n features. How many possible subsets are there? Create n potential positions in the feature set, one for each feature. Store a 0 in position i if feature i is not chosen, and 1 if it is. Then you have 2^n possible subsets. This number includes the combination of all 0s. But all zeros is an empty subset! Therefore, we have $2^n - 1$ possible feature sets to traverse through. And now imagine that you have a bioinformatics dataset with 1000 features. Hmm, the super-hero computer that can handle going through all feature subsets has not been born yet!

Checking each possible subset is called *exhaustive search*. Indeed, this approach is applicable only for relatively small feature spaces.

What can we do instead?

One possible solution is *Sequential Forward Selection* (SFS). The SFS algorithm is shown in Figure 4.4. We start with an empty set and add one feature at a time; the one which makes the best combination with the already selected features. The algorithm stops when we reach the desired number of features.

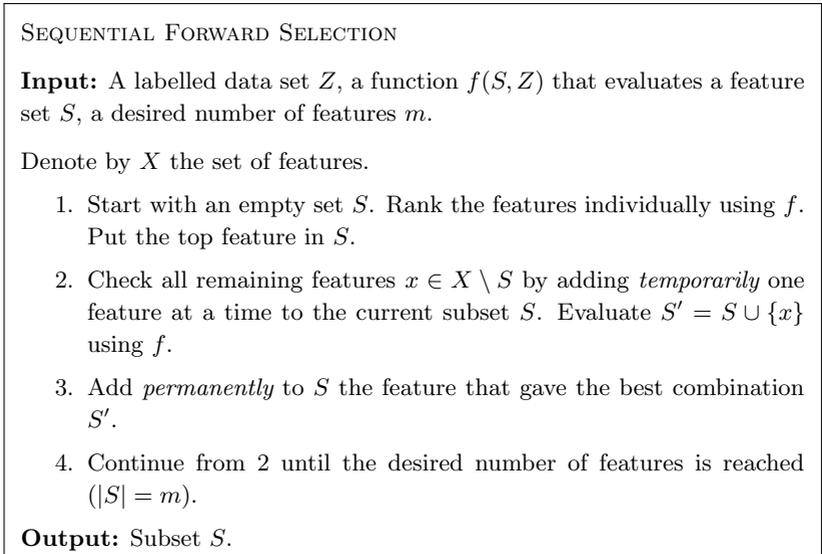


Figure 4.4: The Sequential Forward Selection (SFS) algorithm.

Does the algorithm look a bit confusing with all the notations? Alright, here is a toy example.

⊕ ⊕ ⊕ **Example 4.4.1**

Suppose that you have 10 features numbered from 1 to 10 and want to select a set of 3. I am choosing a fake criterion function $f(S)$

(nothing to do with any data) to evaluate a subset S calculated in the following way. $f(S)$ is a sum of the feature numbers in S taken with sign plus if the cardinality of S is even, and with sign minus if the cardinality is odd. For example, $f(S)$ where $S = \{1, 4, 7\}$ is

$$f(S) = -(1 + 4 + 7) = -12.$$

Notice the minus sign. This is because there are 3 features in S . If S was $S = \{1, 4\}$, we would have $f(S) = 1 + 4 = 5$. Without loss of generality we will assume that higher values of f are better.

Start SFS! First, we evaluate all features individually and rank them.

feature #	1	2	3	4	5	6	7	8	9	10
f	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10

The best feature will be 1, hence $S = \{1\}$. At step 2, we add one feature at a time

feature #	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10
f	3	4	5	6	7	8	9	10	11

The best pair is $S = \{1, 10\}$. At step 3,

feature #	1,10,2	1,10,3	1,10,4	1,10,5	1,10,6	1,10,7	1,10,8	1,10,9
f	-13	-14	-15	-16	-17	-18	-19	-20

Thus, the final set with three features is $S^* = \{1, 2, 10\}$ with value $f(S^*) = -13$. Not ideal. With cardinality 3 we have a better subset $S^{**} = \{1, 2, 3\}$ with value $f(S^{**}) = -6$. But, to be fair, SFS was never claimed to guarantee finding the best subset.

⊖ ⊖ ⊖

A question to you: How many evaluations of the criterion function f do we need in order to select 5 features out of 20? What proportion is this of the total number of feature subsets?

Usually the evaluation function f is so peculiar, with multiple minima and maxima, and plenty of irregularities. SFS is not guaranteed to return the best subset of features in real life. It only checks a relatively small number of subsets. The only approach that guarantees finding the best subset is the exhaustive search.

4.4.3 What criterion do we use to evaluate a given feature set?

The approaches here are: Filter, Wrapper and Embedded.



The wrapper approach is the simplest – you train a classifier on the data using only the features in the candidate subset, and the error estimate of the classifier shows how good this candidate subset is. That is *wrapper*, not rapper ☺. It is called wrapped because the classifier is wrapped within the error estimate.

The filter approach, on the other hand, bypasses the whole training and testing of a classifier and replaces this with a simpler calculation. For example, we can calculate some form of scaled distance between the centres of the classes in the spaces spanned by the features in the candidate set. The larger the distance, the better. This evaluation can be done in a fraction of the time needed to train and test a classifier. When we are planning to evaluate millions of candidate subsets, a filter approach is preferable.

In the embedded approach, the classifier selects features anyway, because feature selection is part of the training algorithm. The prime example in this category is the decision tree classifier. Remember, only features chosen at the nodes of the tree are then used to label a new object. Then we just don't need the other features.

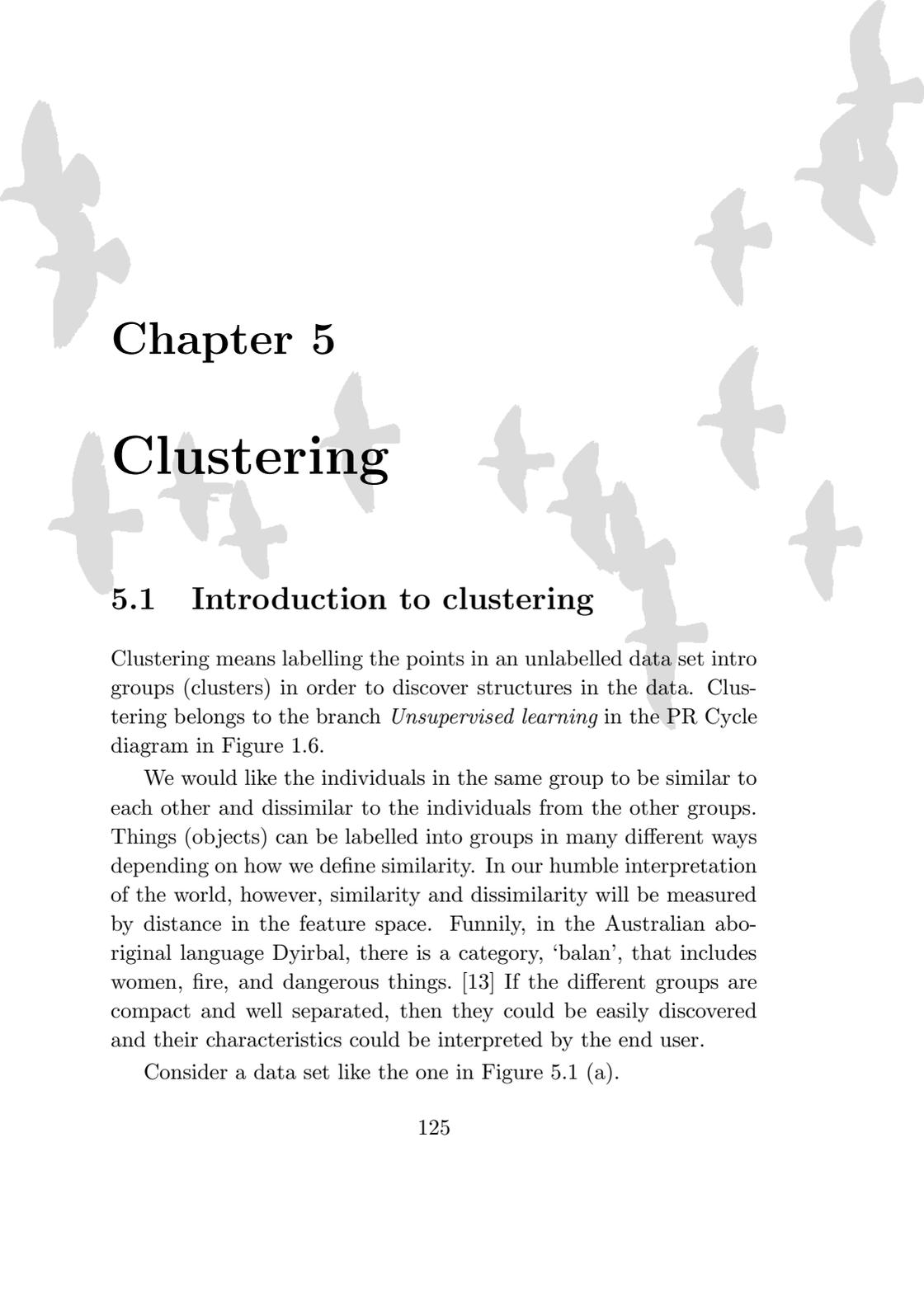
Some answers and solutions

- How many evaluations of the criterion function f do we need (in SFS) in order to select 5 features out of 20?

To choose the first feature, we need 20 evaluations. Then, there are 19 features left which can be taken one at a time to check which the best pair is. Therefore, the total number of evaluations for selecting 5 features would be: $20 + 19 + 18 + 17 + 16 = 90$ evaluations.

- What proportion is this of the total number of feature subsets?

The total number of possible feature sets is $2^{20} - 1 = 1,048,575$. Then the proportion of evaluations to select 5 features through SFS is 8.6×10^{-5} .



Chapter 5

Clustering

5.1 Introduction to clustering

Clustering means labelling the points in an unlabelled data set into groups (clusters) in order to discover structures in the data. Clustering belongs to the branch *Unsupervised learning* in the PR Cycle diagram in Figure 1.6.

We would like the individuals in the same group to be similar to each other and dissimilar to the individuals from the other groups. Things (objects) can be labelled into groups in many different ways depending on how we define similarity. In our humble interpretation of the world, however, similarity and dissimilarity will be measured by distance in the feature space. Funnily, in the Australian aboriginal language Dyirbal, there is a category, ‘balan’, that includes women, fire, and dangerous things. [13] If the different groups are compact and well separated, then they could be easily discovered and their characteristics could be interpreted by the end user.

Consider a data set like the one in Figure 5.1 (a).

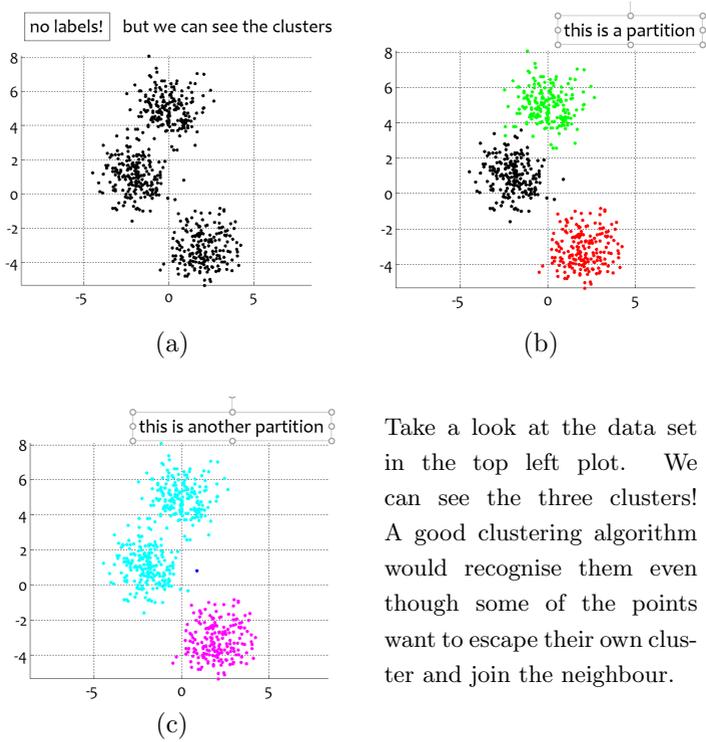


Figure 5.1: An examples of a data set and two partitions of it into three clusters.

A partition is *any* labelling of the data. Any! A good partition would be one which corresponds to our (human) perception of grouping. An example of a good partition is shown in plot (b). Another partition is shown in plot (c), also into three clusters. Notice that little dark blue loner in the middle of plot (c)? That is your third cluster! The algorithm deemed it to be too far away from its neighbours and therefore it is a cluster of its own.

In two dimensions, it is easy! We can see the result and agree or disagree with the clustering algorithm. We can cluster the points ourselves, come to that. But in 3 and more dimensions, things are dramatically different. There is no easy way to see the clusters. The only possible validation of our results lies with the end user. Their data – their verdict. The user may discover that the grouping we return to them makes sense and use the result in their further work. For example, if the data were cancer patients, different treatment plans could be tried with different groups discovered by our cluster analysis. But there is no single number that can tell us how good our result is. Recall Supervised learning. There we have class labels which we need to match. We have the classification error as a measure of success. No such luck with unsupervised learning.

Then how can we design our algorithms? By choosing what we want them to do: put similar points together in the same cluster and dissimilar points in different clusters. Easier said than done! How do we define similar and dissimilar? How do we go about changing the point labels without compromising the meaning of the cluster? How do we make sure that our algorithms converge and not loop forever?

Here we will study two archetypal clustering approaches and their leading representative methods: hierarchical clustering with the *single linkage* method and non-hierarchical clustering with the *k-means* method.

5.2 Hierarchical clustering and the single linkage method

5.2.1 The generic agglomerative clustering algorithm

Hierarchical clustering grows or splits the clusters step by step. There are two sub-approaches:

- Agglomerative clustering: Start with every object being a cluster of its own. Group clusters successively until all objects fall into one single cluster.
- Divisive: Start with all objects in one single cluster. Split clusters successively until every object becomes a cluster of its own.

Then we decide on the number of clusters that are most likely to be present in the data, and return to the user the labels at the respective step. The vast majority of clustering algorithms in this category are agglomerative. Figure 5.2 shows the generic agglomerative clustering algorithm. The variants differ only by the definition of the function $d(p, q)$ that measures the distance between two clusters of points, p and q .

5.2.2 Single linkage

The prime example of an agglomerative clustering algorithm is Single Linkage. For this variant of the generic algorithm, the distance between two clusters is defined as the distance between the closest points, one from each cluster

$$d(C_i, C_j) = \min_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \{d_E(\mathbf{x}, \mathbf{y})\}, \quad (5.1)$$

THE GENERIC AGGLOMERATIVE CLUSTERING ALGORITHM

Input: An unlabelled data set Z and a distance function $d(C_i, C_j)$ between clusters of points C_i and C_j .

1. Start with clusters C_1, \dots, C_N , each containing a single object $\mathbf{z}_j \in Z$.
2. Find the nearest pair of distinct clusters according to d , say C_i and C_j , merge them, delete C_j and decrease the number of clusters by 1.
3. If the number of clusters is 1, then STOP, else continue from 2.
4. Determine the likely number of clusters in the data and return those cluster labels.

Output: Cluster labels.

Figure 5.2: The generic agglomerative clustering algorithm

where C_i and C_j are clusters of points and d_E is Euclidean distance. Any other distance between two points \mathbf{x} and \mathbf{y} can be used too (see Section 3.4.1).

This distance between clusters is also called the *nearest neighbour distance*. Hence, sometimes single linkage is called nearest neighbour clustering.

⊕ ⊕ ⊕ Example 5.2.1

Take a look at the one-dimensional dataset in Figure 5.3. It consists of six 1D points (one feature only): $Z = \{-1, 2, 6, 20, 22, 26\}$. Let's apply single linkage to this data set. This example will explain only how we run steps 1–3 in the algorithm in Figure 5.2. Step 4 is trickier and deserves an example of its own.

At Step 1, each point is a cluster itself, therefore we have six

clusters. Let's call them (a) , (b) , (c) , (d) , (e) and (f) as in the figure. This will be our Iteration #1. At Step 2, we identify the two *nearest* clusters. To do so, we must calculate the distances between all 15 pairs of clusters. Well, we can see from the figure that clusters (d) and (e) are the closest. As instructed, we merge them into (d, e) . Now cluster (e) does not exist on its own, and we have five remaining clusters: (a) , (b) , (c) , (d, e) , and (f) . This is Iteration #2 done.



Figure 5.3: The iterations in applying single linkage to a 1D dataset.

According to Step 3, we should check whether the cluster number is 1. As it is 5, we continue from Step 2 with the new cluster structure. Before we continue though, record the distance at which the clusters were merged. In our case, this distance is $22 - 20 = 2$. This will be a criterion value, say J . So, for our Iteration #2, we can record $J(2) = 2$. Just for completeness, we can also record $J(1) = 0$.

For Iteration #3, we identify the next nearest pair of clusters. These are (a) and (b) at distance 3. Join them into (a, b) , record $J(3) = 3$, and end up with four clusters: (a, b) , (c) , (d, e) , and (f) . At the next iteration, we have a choice. Both pair of clusters: (a, b) and (c) , as well as (d, e) and (f) are at the same distance of 4. Any of the two pairs can be taken at Iteration #4. I have chosen (a, b) and (c) in the example. Hence, at Iteration #4 we have clusters

(a, b, c) , (d, e) , and (f) , and $J(4) = 4$. It is only fair that we join (d, e) and (f) next, therefore at Iteration #5, there are two clusters: (a, b, c) and (d, e, f) , and $J(5) = 4$.

Still not done! There are two clusters, so, as instructed by Step 3, we need one more iteration. Iteration #6 sees us to one cluster (a, b, c, d, e, f) with $J(6) = 20 - 6 = 14$. Now we are done and can proceed to the scary and exciting Step 4. $\ominus \ominus \ominus$

I will certainly ask you to run single linkage by hand, either for your homework assignment or for the exam. How do we document the iterations? Table 5.1 shows you how to do this for the example above.

Table 5.1: Documenting the iterations of running Single Linkage by hand for Example 5.2.1.

Iteration #	Clusters	Number	J
1	$(a), (b), (c), (d), (e), (f)$	6	0
2	$(a), (b), (c), (d, e), (f)$	5	2
3	$(a, b), (c), (d, e), (f)$	4	3
4	$(a, b, c), (d, e), (f)$	3	4
5	$(a, b, c), (d, e, f)$	2	4
6	(a, b, c, d, e, f)	1	14

5.2.3 Determining the number of clusters for agglomerative clustering

This is Step 4 from the generic agglomerative clustering algorithm shown in Figure 5.2. This step is the same for any variant of the agglomerative algorithm, not just single linkage.

Now that we have the table with the iteration records, we should

decide how many clusters are likely in the data. Looking at Figure 5.3, what is your estimate? How many clusters?

The idea behind this method of determining the likely number of clusters is simple and intuitive. Think of the distance criterion J as ‘force’ which is needed to join the two clusters. Obviously, the further we go with the iterations, the more force we will need as the clusters closer together would have been joined already. So, we look at the *increase* of the force at each iteration (call this a ‘jump’). If the force increases by the same amount, the jumps will be the same from iteration to iteration. But if we come across a particularly large jump, this will indicate that a lot more force is required to join the two clusters compared to that on the previous iteration. In other words, these two clusters strive to stay apart. Therefore, we calculate all the jumps and choose the number of clusters to be that before the largest jump. For the above example, we add a last column in the table with the value of the jump as in Table 5.2. For Iteration $\#i$, the jump is $J(i) - J(i - 1)$. We don’t have a jump value for Iteration $\#1$.

Table 5.2: Determining the likely number of clusters by the largest jump for Example 5.2.1.

Iteration #	Clusters	Number	J	Jump
1	$(a), (b), (c), (d), (e), (f)$	6	0	–
2	$(a), (b), (c), (d, e), (f)$	5	2	2
3	$(a, b), (c), (d, e), (f)$	4	3	1
4	$(a, b, c), (d, e), (f)$	3	4	1
5	$(a, b, c), (d, e, f)$	2	4	0
6	(a, b, c, d, e, f)	1	14	10

The largest jump is for Iteration $\#6$ (hence the line in the table), which tells us not to make the move from 2 clusters to 1 cluster.

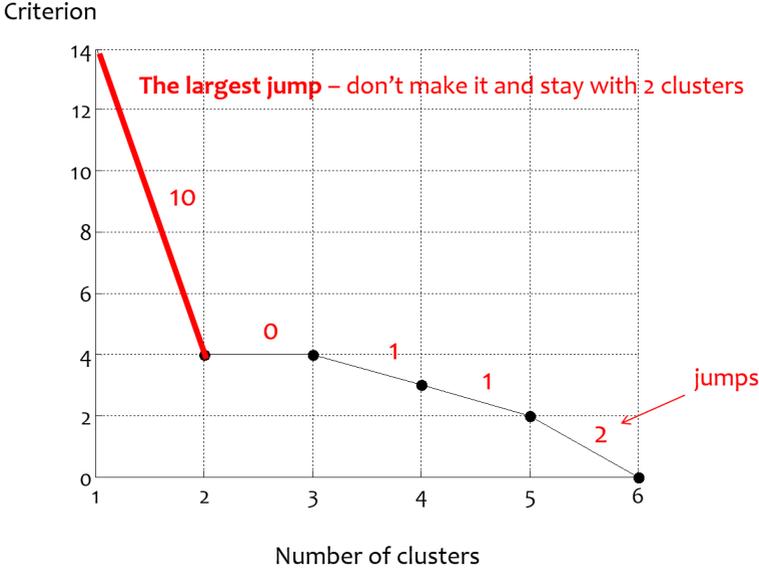


Figure 5.4: Illustration of the process of determining the likely number of clusters through analysing the jumps of the criterion J .

Therefore we recommend 2 clusters for this data. Figure 5.4 plots the criterion value, J , as a function of the number of clusters, just to give a graphical illustration of the process of determining the likely number of clusters. The two clusters that we return can be reproduced from the table: (a, b, c) and (d, e, f) .

Test yourself on this problem: Figure 5.5 shows the scatterplot of an unlabelled data set Z with 8 objects. The objects are numbered as shown in the figure.

Apply the single linkage clustering algorithm to this data. Give the steps in the table format explained above. Subsequently, recommend the number of clusters for this data set, and show which points

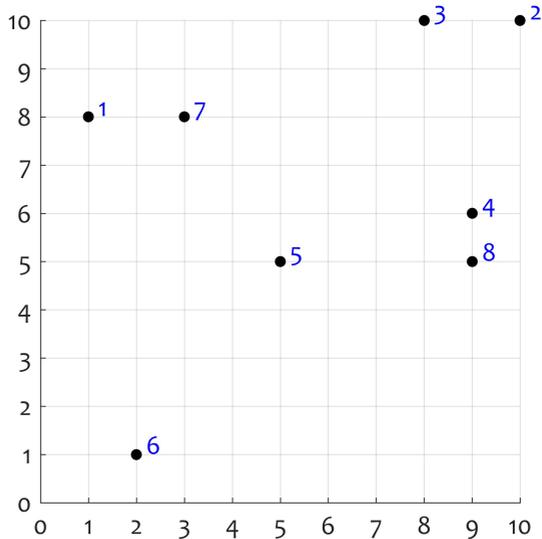


Figure 5.5: Scatterplot of a 2D data set.

are included in each cluster. (And if you are VERY impatient, skip to the end of this chapter to see the solution.)

5.2.4 Dendrograms

A dendrogram is a diagram that shows the hierarchical relationship between objects, usually as a result of hierarchical clustering. The name dendrogram derives from the ancient Greek word ‘dendron’ meaning ‘tree’. The dendrogram of the clustering result for Example 5.2.1 is shown in Figure 5.6.

The iterations are drawn by joining the points at the respective value of the distance J . For example, Iteration #2 is carried out

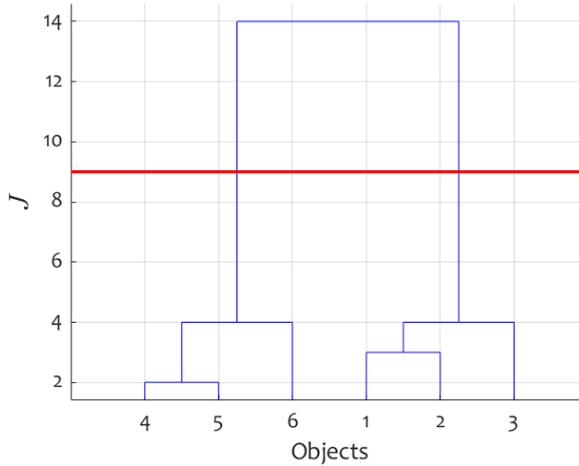


Figure 5.6: Dendrogram of the clustering result for Example 5.2.1. The red line shows the ‘cut’ into two clusters.

at distance $J(2) = 2$, and the points that are joined are 4 and 5. The next pair of points are 1 and 2, at distance $J(3) = 3$, and so on. The ordering of the points on the x-axis is immaterial as these points represent objects. The dendrogram is drawn to scale only on the y-axis to show the distance between the objects and clusters.

This dendrogram helps seeing straight away where the largest jump of J is. In this case, it is between 4 and 14. The red horizontal line shows where the split is done. The two clusters resulting from the cluster analysis can also be read from the dendrogram. These are the subtrees from the red line down to the x-axis. The two clusters are (4, 5, 6) and (1, 2, 3). Note that it does not matter which cluster we will call cluster #1 and cluster #2. The important part is to recognise that there are two groups and to give the members

of these groups different labels.

5.2.5 The chain effect of single linkage

When the clusters are well-separable, single linkage will identify them without a problem, as will any other reasonable clustering algorithm. But when there is noise, single linkage might struggle. Even a single outlier in the data could be catastrophic. The algorithm will group everything else in one cluster and declare the outlier a cluster of its own. Single linkage works well when the clusters are shaped like strings. Many other clustering algorithms may not succeed in this case.

Figure 5.7 shows an example of two data sets with different cluster configurations. Single linkage fails to identify the spherical clusters. It singles out an outlier as one of the clusters (circled point) and labels all other points as the other cluster (grey crosses). For the data with elliptical clusters, single linkage separates the two clusters perfectly. But notice that the leftmost and the rightmost points of each of these clusters are far apart from one another, suggesting that points in the same cluster are not similar at all. They are labelled into the same clusters because their nearest neighbour from the whole data is from that cluster. This behaviour of single linkage is called the *chain effect*.

5.2.6 Mean (centroid) linkage

Single linkage is very useful when the clusters have intricate configuration but at the same time are well separable. On the other hand, when there is noise in the data, the *mean linkage* may be more suitable.

The difference between single linkage and mean linkage is *only* in the way we define the distance between clusters of points. In single

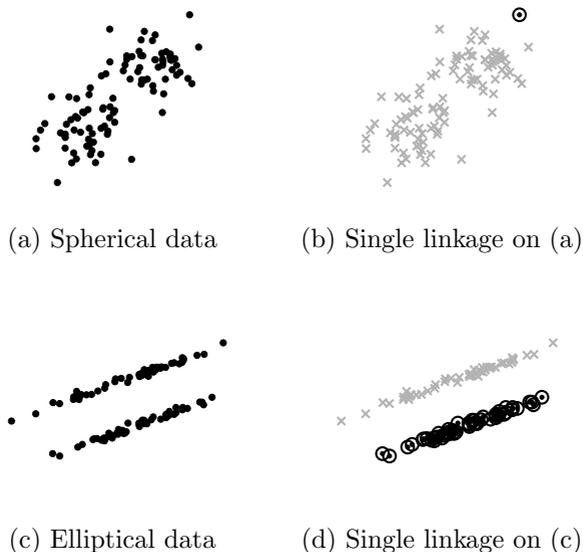


Figure 5.7: For which type of data is single linkage useful?

linkage, this distance was the point-wise distance between the two nearest points, one from each cluster (Equation (5.2)). In mean linkage, the distance between two clusters is the Euclidean distance between their centroids:

$$d(C_i, C_j) = d_E \left(\underbrace{\frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}}_{\text{mean of } C_i}, \underbrace{\frac{1}{|C_j|} \sum_{\mathbf{y} \in C_j} \mathbf{y}}_{\text{mean of } C_j} \right), \quad (5.2)$$

where C_i and C_j are clusters of points, d_E is Euclidean distance, and $|\cdot|$ denotes cardinality. Any distance between the means (points in the feature space) can be used too (see Section 3.4.1).

Figure 5.8 mirrors Figure 5.7 in that the same data sets are used. But this time, we apply the mean linkage. Look how different the clusters are! Mean linkage does not mind the noise and the outliers, and identifies the two clusters in (a) (circled points versus grey crosses), while it fails to distinguish the string-like clusters in (c).

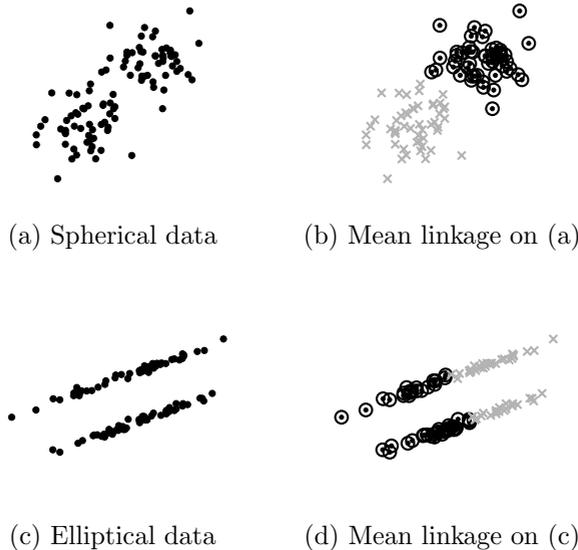


Figure 5.8: For which type of data is mean linkage useful?

And again, we would not know in advance which type of data we have in the n -dimensional space. How do we know which method is suitable? The end user has the final word on this. Hopefully some of the solutions which we offer them will make good sense!

5.2.7 MATLAB code and a caveat

MATLAB offers a function `linkage` which can calculate the criterion value. Then, after determining the likely number of clusters (by ourselves, that is!), function `cluster` can be used to retrieve the cluster labels. An example is shown below.

⊕ ⊕ ⊕ Example 5.2.2

For this example we retrieved the data set ‘zoo’ from the famous UCI Machine Learning Repository¹. To simplify the example, we cut out only a part of the data as shown below:

```
antelope,1,0,0,1,0,0,0,1,1,1,0,0,4,1,0,1
bear,1,0,0,1,0,0,1,1,1,1,0,0,4,0,0,1
buffalo,1,0,0,1,0,0,0,1,1,1,0,0,4,1,0,1
cheetah,1,0,0,1,0,0,1,1,1,1,0,0,4,1,0,1
dolphin,0,0,0,1,0,1,1,1,1,1,0,1,0,1,0,1
frog,0,0,1,0,0,1,1,1,1,1,0,0,4,0,0,0
giraffe,1,0,0,1,0,0,0,1,1,1,0,0,4,1,0,1
haddock,0,0,1,0,0,1,0,1,1,0,0,1,0,1,0,0
honeybee,1,0,1,0,1,0,0,0,0,1,1,0,6,0,1,0
housefly,1,0,1,0,1,0,0,0,0,1,0,0,6,0,0,0
leopard,1,0,0,1,0,0,1,1,1,1,0,0,4,1,0,1
lion,1,0,0,1,0,0,1,1,1,1,0,0,4,1,0,1
newt,0,0,1,0,0,1,1,1,1,1,0,0,4,1,0,0
piranha,0,0,1,0,0,1,1,1,1,0,0,1,0,1,0,0
seal,1,0,0,1,0,1,1,1,1,1,0,1,0,0,0,1
stingray,0,0,1,0,0,1,1,1,1,0,1,1,0,1,0,1
toad,0,0,1,0,0,1,0,1,1,1,0,0,4,0,0,0
tuna,0,0,1,0,0,1,1,1,1,0,0,1,0,1,0,1
```

The 16 features (corresponding to the columns) are as follows:

- | | |
|-------------|---------|
| 1. hair | Boolean |
| 2. feathers | Boolean |
| 3. eggs | Boolean |

¹<https://archive.ics.uci.edu/ml/machine-learning-databases/zoo/>

4. milk	Boolean
5. airborne	Boolean
6. aquatic	Boolean
7. predator	Boolean
8. toothed	Boolean
9. backbone	Boolean
10. breathes	Boolean
11. venomous	Boolean
12. fins	Boolean
13. legs	Numeric (0,2,4,5,6,8)
14. tail	Boolean
15. domestic	Boolean
16. catsize	Boolean

The following piece of code applies single linkage to this data. File `zoo_short_nolabels.txt` contains the data.

```

1 clear, clc, close all
2 t = readtable('zoo_short_nolabels.txt');
3 data = table2array(t(:,2:end));
4 N = size(data,1); % number of objects to cluster
5 names = table2array(t(:,1));
6
7 z = linkage(data); % apply single linkage
8
9 % Determine the number of clusters
10 J = z(:,3); % distances (Iterations 2 to N)
11 [~,index_largest_jump] = max(diff(J));
12 number_of_clusters = N - index_largest_jump;
13
14 % Plot the dendrogram and the cut-off line
15 dendrogram(z, 'Labels', names)
16 set(gca, 'XTickLabelRotation', 90)
17 hold on
18 t = (J(index_largest_jump) + J(index_largest_jump+1))/2;
19 plot([0 N+1],[t t], 'r-', 'linewidth', 1.5)
20

```

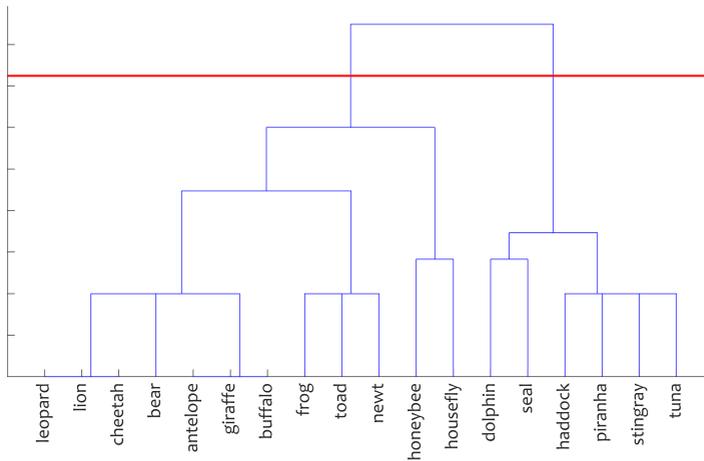


Figure 5.9: Dendrogram for the zoo data and single linkage clustering.

```

21 % Find the cluster labels
22 labels = cluster(z, 'MaxClust', number_of_clusters);
23
24 % List the animals
25 for i = 1:number_of_clusters
26     fprintf('\nCluster %i:\n', i)
27     in_cluster = find(labels == i);
28     for j = 1:numel(in_cluster)
29         fprintf('    %s\n', names{in_cluster(j)})
30     end
31 end

```

The code displays the dendrogram and the cut-off line (Figure 5.9) so the clusters can be read from the plot.

In addition, the code lists the names of the animals in each cluster:



Cluster 1: dolphin, haddock, piranha, seal, stingray, tuna.

Cluster 2: antelope, bear, buffalo, cheetah, frog, giraffe, honeybee, housefly, leopard, lion, newt, toad.



Where is the caveat? The result from the clustering depends critically on the chosen distance and not only! Rescaling the features may have a dramatic effect on the result too. Feature # 13 in the zoo data is the only non-Boolean feature. What will happen if we rescale it between 0 and 1 so that it does not dominate all other features in calculating the distances. Why should the number of legs be eight times more important than whether or not the animal produces milk? As soon as we apply this rescaling, the clustering results changes. Instead of two clusters, with this tiny modification, single linkage returns 14 clusters:

Cluster 1: stingray; Cluster 2: tuna; Cluster 3: piranha; Cluster 4: haddock; Cluster 5: frog; Cluster 6: toad; Cluster 7: newt; Cluster 8: bear; Cluster 9: cheetah, leopard, lion; Cluster 10: antelope, buffalo, giraffe; Cluster 11: dolphin; Cluster 12: seal; Cluster 13: honeybee; Cluster 14: housefly.

Which result is better? I am leaning more towards the two-cluster result. But there is no right or wrong in clustering. All is in the eye of the beholder, especially if that beholder is the domain expert whose data we have been analysing.

5.3 Non-hierarchical clustering: k-means

5.3.1 Preliminaries

Non-hierarchical clustering is different from hierarchical clustering in that the partitions in subsequent iterations are not nested. In single

linkage, if two points are labelled in the same cluster at iteration i , they will stay together in the same cluster until the end of the algorithm. This is not guaranteed for non-hierarchical algorithms, where the points can be reassigned to any cluster at the subsequent iterations.

In non-hierarchical clustering, the number of clusters is chosen in advance. This number is usually chosen by the end user who knows/suspects/hypothesises the likely number of clusters.

5.3.2 The famous *k*-means algorithm

k-means is the most well-known clustering algorithm. It dates back to the 1950s. In this algorithm (Figure 5.10), the data is split initially into k clusters (a number specified by the user), and at each iteration points are moved to another cluster if they are closer to its current centroid (the mean). The algorithm stops when there is no possible point movement.

The initial means can be chosen in many different ways. The common practice is to pick randomly k points from the dataset Z for this purpose. These means can be generated as small random numbers or as numbers “equally spread” in the part of the space \mathbb{R}^n occupied by the data. (Tricky, that last one!)

In our examples we will stick to choosing the means among Z .

⊕ ⊕ ⊕ Example 5.3.1

We will apply *k*-means on a small, toy dataset with two features, x and y and 5 points (Figure 5.11).

Let’s cluster the points into $k = 2$ clusters. Choose points 1 and 3 to be the initial means, $\mathbf{m}_1 = [2, 0]^T$ and $\mathbf{m}_2 = [2, 3]^T$. At Iteration #1, we label the points to the closest mean. Thus point 1 will be in cluster 1 while all the other points will be in cluster 2. Moving to the next step of the algorithm, we recalculate the means.

THE K-MEANS ALGORITHM

Input: An unlabelled data set Z and a distance function $d(\mathbf{x}, \mathbf{y})$ between points $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^n$ and a number of clusters k .

1. Pick the k initial means as points in \mathbb{R}^n .
2. Classify all the objects in Z according to the nearest mean using distance d .
3. Recalculate the means.
4. If there is a change in any of the means, then continue from step 2. Otherwise stop, return the cluster labels.

Output: Cluster labels.

Figure 5.10: The k-means clustering algorithm

The mean for cluster 1 is point 1 itself, that is $\mathbf{m}'_1 = [2, 0]^T$, which is the same as the initial mean. For cluster 2, whose old mean was the chosen point 3, we now have

$$\mathbf{m}'_2 = \left[\frac{1 + 2 + 4 + 5}{4}, \frac{2 + 3 + 4 + 4}{4} \right]^T = [3, 3.25]^T.$$

Check whether there has been a difference in any of the old and new means. We have $\mathbf{m}'_1 = \mathbf{m}_1$, but $\mathbf{m}'_2 \neq \mathbf{m}_2$. Therefore we should proceed with Iteration #2. Reassign the means so that the old means are replaced by the new means, that is $\mathbf{m}_1 \leftarrow \mathbf{m}'_1$ and $\mathbf{m}_2 \leftarrow \mathbf{m}'_2$.

At Iteration #2, we first re-cluster the data. This time, point 2 will be labelled in cluster 1. The new means are:

$$\mathbf{m}'_1 = \left[\frac{2 + 1}{2}, \frac{0 + 2}{2} \right]^T = [1.5, 1]^T.$$

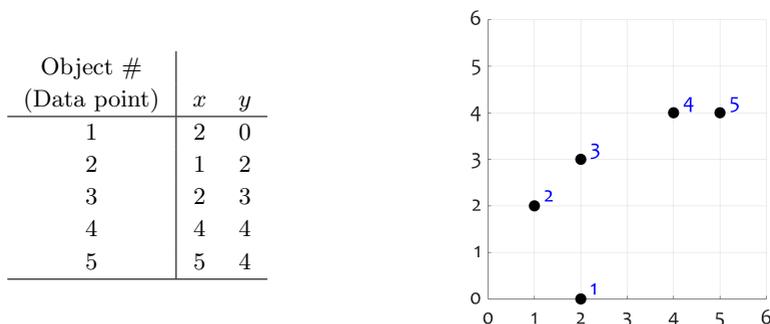


Figure 5.11: Data used in Example 5.3.1

$$\mathbf{m}'_2 = \left[\frac{2 + 4 + 5}{3}, \frac{3 + 4 + 4}{3} \right]^T = [3.67, 3.67]^T.$$

Noticing that $\mathbf{m}'_1 \neq \mathbf{m}_1$ and $\mathbf{m}'_2 \neq \mathbf{m}_2$, we carry out Iteration #3.

At Iteration #3, the points are relabelled in the same clusters! Therefore the means will be the same. The stopping criterion kicks in, and we return the cluster labels: points 1 and 2 are in cluster 1, and points 3, 4, and 5 are in cluster 2.

The migration of the means and the respective clusters are shown in Figure 5.12.

⊖ ⊖ ⊖

How shall we record the iterations? For small 2D datasets, we should store the means of all clusters for each iteration. Table 5.3 shows the record for the *k*-means run in Example 5.3.1.

5.3.3 The criterion function J_e

Let's delve deeper into *k*-means. Why does it work? Typically, non-hierarchical clustering methods optimise a certain criterion which

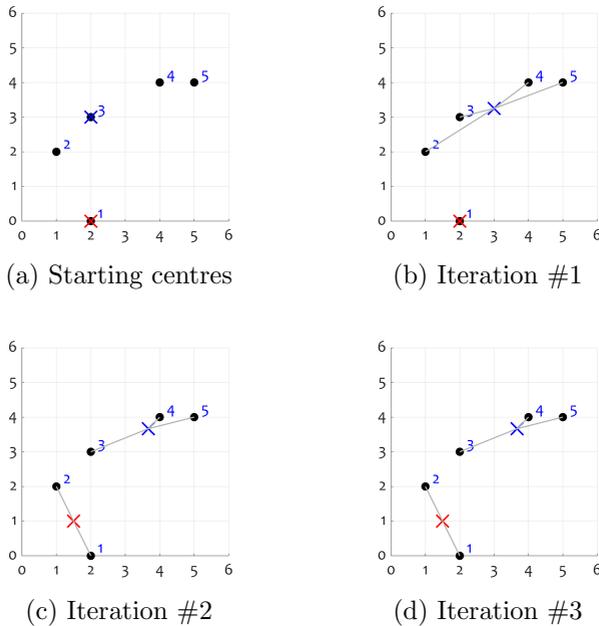


Figure 5.12: Migration of the cluster centres. The final two clusters are shown in subplot(d).

ensures that points in the same cluster are similar to one another and dissimilar to points from different clusters. For k -means, this criterion is called J_e . This criterion evaluates how good a certain partition is. Smaller values of J_e mean better partitions. The value is not comparable with any pre-defined constant but can be used to compare two different partitions of the same data. See the example in Figure 5.13.

If the partition reflects well the cluster structure in the data, the value of J_e is small. The mean linkage algorithm gives a better result

Table 5.3: An example of recording the *k*-means iterations.

Iteration 1

Old means: (2.00, 0.00) (2.00, 3.00)

Clusters: (1);(2, 3, 4, 5);

New means: (2.00,0.00)(3.00,3.25)

Iteration 2

Old means: (2.00, 0.00) (3.00, 3.25)

Clusters: (1, 2);(3, 4, 5);

New means: (1.50,1.00)(3.67,3.67)

Iteration 3

Old means: (1.50, 1.00) (3.67, 3.67)

Clusters: (1, 2);(3, 4, 5);

New means: (1.50,1.00)(3.67,3.67)

Returned clusters: (1, 2) (3, 4, 5)

than single linkage for this data set. Indeed, the clusters identified by mean linkage are more “reasonable”.

Suppose that the data has been partitioned into c clusters C_1, \dots, C_c . Denote by \mathbf{m}_i the mean of cluster C_i . The criterion function is calculated as:

$$J_e = \sum_{i=1}^c J_e(i),$$

where

$$J_e(i) = \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \mathbf{m}_i\|^2. \quad (5.3)$$

Let’s calculate J_e for the partition of the 5-point data shown Figure 5.12, subplots (c) and (d). First, the means of the two clusters

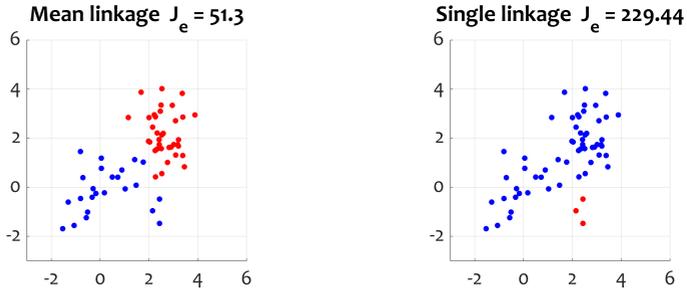


Figure 5.13: Two partitions of the same dataset and the respective values of the J_e criterion function.

are: $\mathbf{m}_1 = [1.5, 1]^T$ and $\mathbf{m}_1 = [3.67, 3.67]^T$, calculated already in Example 5.3.1. Then

$$J_e(1) = \underbrace{(2 - 1.5)^2 + (0 - 1)^2}_{\|\mathbf{x}_1 - \mathbf{m}_1\|^2} + \underbrace{(1 - 1.5)^2 + (2 - 1)^2}_{\|\mathbf{x}_2 - \mathbf{m}_1\|^2} = 2.5.$$

and

$$\begin{aligned} J_e(2) &= \underbrace{(2 - 3.67)^2 + (3 - 3.67)^2}_{\|\mathbf{x}_3 - \mathbf{m}_2\|^2} + \underbrace{(4 - 3.67)^2 + (4 - 3.67)^2}_{\|\mathbf{x}_4 - \mathbf{m}_2\|^2} \\ &\quad \underbrace{(5 - 3.67)^2 + (4 - 3.67)^2}_{\|\mathbf{x}_5 - \mathbf{m}_2\|^2} = 5.33. \end{aligned}$$

Finally

$$J_e = J_e(1) + J_e(2) = 2.5 + 5.33 = 7.83.$$

Not that the choice of the initial means is random. Different initial means can lead to different final results! How is this possible? Unfortunately, J_e could be multi-modal, which means that there

may be numerous local minima achieved by different partitions. For example, if we chose data points 2 and 5 as the initial means in Example 5.3.1, we would end up with clusters (1, 2, 3) and (4, 5). One of these solution is better than the other in terms of J_e but for some initialisations, *k*-means will be trapped into the local minimum. For the 5-point example, the two partitions are the only minima. There are 10 possible pairs of points to initialise the means. Four of these initialisations will lead to partition (1, 2), (3, 4, 5) with $J_e = 7.83$ and the other six initialisations will lead to partition (1, 2, 3), (4,5) with $J_e = 5.83$. This dependency on the initialisation, underpins the standard practice of using *k*-means. We run *k*-means a set number of times (say, 10), and then pick the result with the smallest J_e . There is no guarantee that we have hit the global minimum of J_e but more attempts give us a better chance.

K-means implementation in MATLAB is child's play! You can do it easily. But I will be a bit lazy here and will use the ready-made function `kmeans` of MATLAB's Statistics Toolbox.

⊕ ⊕ ⊕ **Example 5.3.2**

This example is an illustration of *k*-means on a slightly more grown-up problem. Suppose that you have an image and you want to represent it in the most faithful way with k colours. The image chosen here is 'peppers', one of the standard MATLAB images. The code below formats the red, green and blue channels of the image as the three features in the dataset, and then clusters them into 10 clusters.

```
1 clear, clc, close all
2 a = imread('peppers.png'); % upload image
3 s = size(a);
4
```

```

5 % Create data
6 b_red = double(a(:,:,1));
7 b_green = double(a(:,:,2));
8 b_blue = double(a(:,:,3));
9 data = [b_red(:) b_green(:) b_blue(:)];
10
11 number_of_clusters = 10; % Choose the number of clusters
12
13 % Determine the labels and centroids (colours)
14 [labels,co] = kmeans(data,number_of_clusters);
15 im = ind2rgb(reshape(labels,size(a,1),size(a,2)),co/255);
16
17 % Show the result
18 figure, set(gca,'Pos',[0 0 1 1])
19 imshow(im,'InitialMagnification','fit')
20 figure, set(gca,'Pos',[0 0 1 1]), imshow(a)

```

Figure 5.14 shows the original image and result from the clustering. The original image has 99,059 colours while the clustered image has only 10. And you have to agree, the resulting colours (cluster centres) are pretty close to the original! Long live k-means! ☺

☺ ☺ ☺

Finally, here is a problem for you to try. (It will take time but is a nice exercise!)

Consider the 1D (unlabelled) data set

$$Z = [6, -10, 1, 0, -2, 18, -14, 3, -6]^T.$$

1. Run k-means to cluster the data set into 3 clusters using the first three points as the initial means. Show the final clusters.
2. Calculate the J_e criterion for each iteration.

See the answers at the end of the chapter.

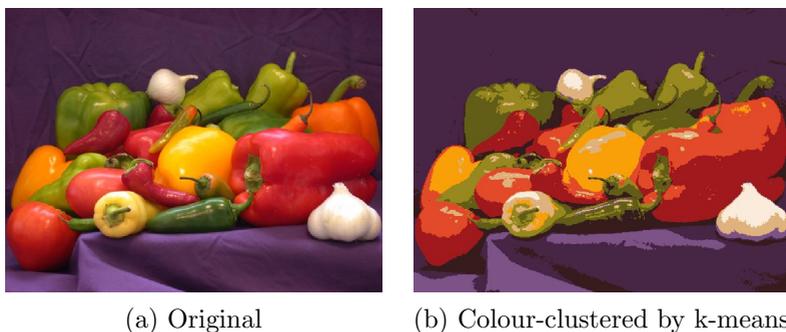


Figure 5.14: An example of *k*-means applied to cluster the 99,059 colours in the original image into 10 clusters. Each pixel in subplot (b) is shown with its cluster colour.

The promised answers and solutions

The solution of the Single Linkage problem from page 133 is shown below. Table 5.4 gives the record of the iterations and Figure 5.15 shows the clusters.

Table 5.4: Iterations of SL for the problem from page 133.

Iteration #	Clusters	Number	J	Jump
1	1,2,3,4,5,6,7,8	8	0	–
2	48,1,2,3,5,6,7	7	1.000	1.000
3	17,48,2,3,5,6	6	2.000	1.000
4	23,17,48,5,6	5	2.000	0.000
5	157,23,48,6	4	3.606	1.606
6	14578,23,6	3	4.000	0.394
7	14578,236	2	4.123	0.123
8	12345678	1	5.000	0.877

Clusters returned: (2,3), (1,7), (4,8), (5), (6).

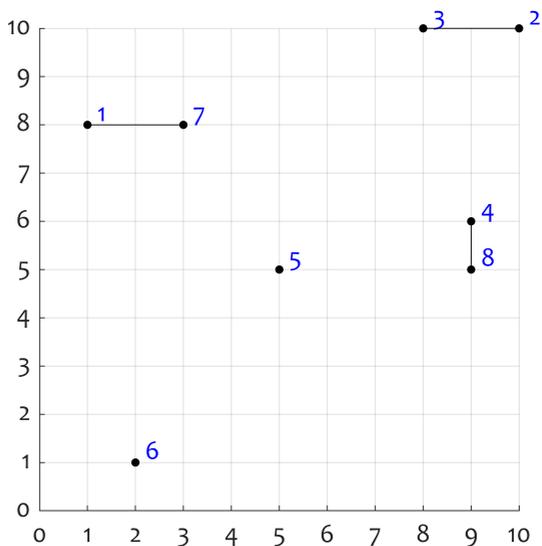


Figure 5.15: The five clusters returned for the problem from page 133.

The solution of the k-means problem from page 150 is shown in Table 5.5 and Figure 5.16

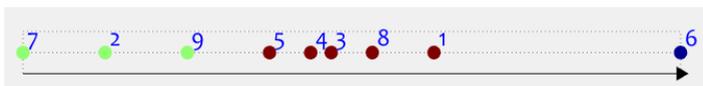


Figure 5.16: Clusters found for the problem from page 150.

Table 5.5: Iteration record for the k-means problem from page 150.

Iteration 1

Old means: (6.00) (-10.00) (1.00)

Clusters: (1, 6);(2, 7, 9);(3, 4, 5, 8);

New means: (12.00)(-10.00)(0.50)

Iteration 2

Old means: (12.00) (-10.00) (0.50)

Clusters: (6);(2, 7, 9);(1, 3, 4, 5, 8);

New means: (18.00)(-10.00)(1.60)

Iteration 3

Old means: (18.00) (-10.00) (1.60)

Clusters: (6);(2, 7, 9);(1, 3, 4, 5, 8);

New means: (18.00)(-10.00)(1.60)

Clusters returned: (7, 2, 9), (5, 4, 3, 8, 1), (6)

The values of J_e for the three iterations are: Iteration #1, $J_e = 190$; Iteration #2, $J_e = 111.25$; Iteration #3, $J_e = 69.2$.

Chapter 6

Neural Networks

6.1 A brief history of neural networks

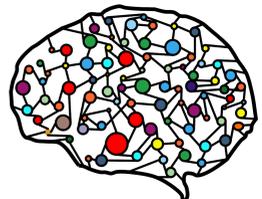
Since the dawn of time humans have been fascinated by how the brain works. Ancient Egyptians believed that human brain is the ‘seat of intelligence’. The breakthrough in neuroscience came in the 20th century with a series of hypes of enthusiasm followed by troughs of disillusionment. Below is a brief story of Neural Networks.¹

6.1.1 The early ages



- 1943. Neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons in the brain might work. They modelled a simple neural network using electrical circuits assuming that neurons can perform logical operations like ‘and’, ‘or’, and ‘not’.

¹They were called before ‘neuronal networks’, ‘neural nets’, and more recently ‘Artificial Neural Networks’. We drop the ‘A’ as most people do these days.





- 1949. Canadian psychologist Donald Hebb introduced the concept of reinforcement learning. According to this concept, neural pathways are strengthened each time they are used and weakened if they are not used for a prolonged amount of time.



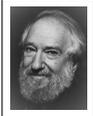
- 1951. Using symbolic reasoning, cognitive scientist Marvin Minsky created the first neural network that solved a problem from the real world: finding the best way out of a labyrinth.



- 1958. American psychologist Frank Rosenblatt invented the famous ‘perceptron’. Drawing upon biological principles, he built an electronic device and showed its ability to learn. These so called perceptrons are the basis of today’s (artificial) neural networks.



- 1969. Marvin Minsky and Seymour Papert published their study: “Perceptrons – An introduction to computational geometry”. Heavily criticising the ability of neural networks built using perceptrons, this study is often thought to have caused a decline in neural networks research in the 1970s and early 1980s. During this period, researchers developed smaller projects outside the mainstream, while symbolic AI research saw explosive growth.



6.1.2 The second wave

The problem was that, in those times, there was no suitable algorithm to train a reasonably sized neural network. Seeds of the famous *error backpropagation* algorithm existed long before that time. Similar algorithms were used for solving problems in different areas. It was only in the 1970 when this algorithm was applied for training neural networks. Two similar versions were independently developed

by Paul Werbos (in his 1974 dissertation) and by David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams and James McClelland (1986). The authors of the latter study claimed to have overcome the problems presented by Minsky and Papert, and that “their pessimism about learning in multilayer machines was misplaced”. This heralded a Renaissance in the neural networks research.

In 1989, Kurt Hornik, Maxwell Stinchcombe and Halbert White published a study to ascertain that multilayer feedforward networks are universal approximators. In other words, however twisted and difficult the class distribution is in \mathbb{R}^n , there is a feedforward neural network with a finite structure which is able to approximate the classification regions with any given, fixed precision.

The applications boomed, and all the other classifiers you have seen in Chapter 3 were deemed inferior to King Neural Network.

Well, this is not strictly true... Brewing since the 1960, the *support vector machine* classifier (SVM) rose to power in the early 1990s. By the early 2000s, SVM dominated the research scene overshadowing the neural networks. During those times, you could hardly publish a paper in a renown journal or conference if there weren't any SVMs or other type of kernel classifiers in it. But don't despair! Neural networks returned with a vengeance!

6.1.3 The blossom of deep learning

Deep learning neural networks were born from the giant advancements in technology. By year 2000, we were ready to process large amount of information in a very short time, and people started experimenting with massive neural network structures, with many layers of thousands of neurons on each layer. A strong fundament of the NN area had been laid already, and the path of deep learning was clear.



In March 2019, Yoshua Bengio, Geoffrey Hinton and Yann LeCun were awarded the prestigious Turing Award, generally recognised as the highest distinction in computer science, something like and the “Nobel Prize of computing”. They received it for conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing.

Nowadays deep learning neural networks (DL) dominate the landscape of artificial intelligence. They have won numerous challenges and competitions, and have found their use in many applications in areas such as computer vision, speech recognition, natural language processing, bioinformatics, drug design, medical image analysis, and board game play. We will learn more about their exploits later in the book.

6.2 Structure and elements of a NN

Neural Networks (NN) are engineering/mathematical models mimicking the structure and functioning of the brain.²

Neural networks may solve many different problems but most such problems boil down to:

- Clustering (unsupervised learning)
- Classification (supervised learning)
- Regression (function approximation and prediction)

6.2.1 Neurons: real and artificial

Since the seminal model by McCulloch and Pitts, many models of neurons have been proposed; some quite sophisticated and elaborate,

²Notice that we are still talking about *Artificial* Neural Networks.

some beautifully simple. A diagram of the current favourite model is shown in Figure 6.1.

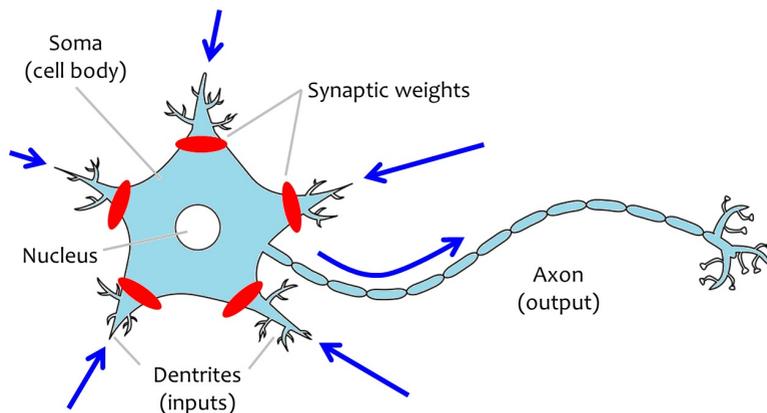


Figure 6.1: A diagram of the biological neuron.

Why is this model the favourite? Because it translates into the standard common element that most NNs are made of (eh, well, apart from the posh deep learning NNs, but they are a different species altogether as we shall see later). The computational version of the diagram in Figure 6.1 is shown in Figure 6.2.

The circle in this figure represents the soma. It takes inputs from the dendrites, where each such input u_i is multiplied by the respective semantic weight w_i , $i = 0, \dots, q$. Ah, and look at the sneaky pair u_0 and w_0 ! They are called the *bias input* and the *bias weight*. The bias input u_0 has a fixed value of 1. Always, always. It is needed so that we have the chance to add a constant (w_0) to the sum of weighted inputs. The inputs and the weights can be organised into vectors: $\mathbf{u} = [u_1, u_2, \dots, u_q]^T$ and $\mathbf{w} = [w_1, w_2, \dots, w_q]^T$.

This diagram shows what is happening in the neuron and how

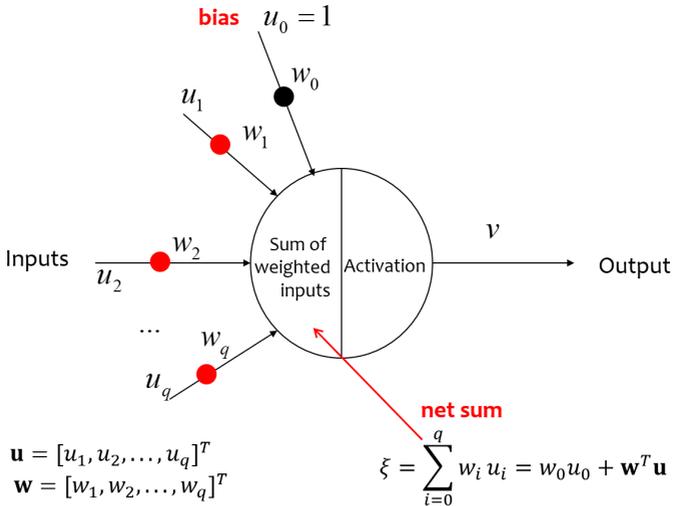


Figure 6.2: A diagram of the computational neuron.

the output is calculated from the inputs and the synaptic weights. First, we calculate the *net sum*, which we will denote by the adorably funny, worm-like, Greek letter ‘xi’ (that is ‘ ξ ’ in \LaTeX):

$$\xi = w_0 u_0 + w_1 u_1 + \dots + w_q u_q.$$

And, clever as we are, we can use the summation notation, and even scalar product of two vectors (recall your first year maths) to represent this same quantity:

$$\xi = \sum_{i=0}^q w_i u_i = w_0 u_0 + \mathbf{w}^T \mathbf{u}.$$

The net sum represents the ‘excitement’ that reaches this neuron through its inputs (the dendrites). According to biology, if there is a

lot of this excitement at a given time, the neuron gets excited itself and propagates the joy through its output (the axon) to the neurons connected to it. To model this process, the net sum ξ is compared with a threshold (typically zero). If $\xi \geq 0$, the neuron gets activated, or *fires*, yielding a 1 at its output. Else ($\xi < 0$), the neuron outputs a value of zero. This simple threshold rule is sometimes insufficient. Take a look at Figure 6.3.

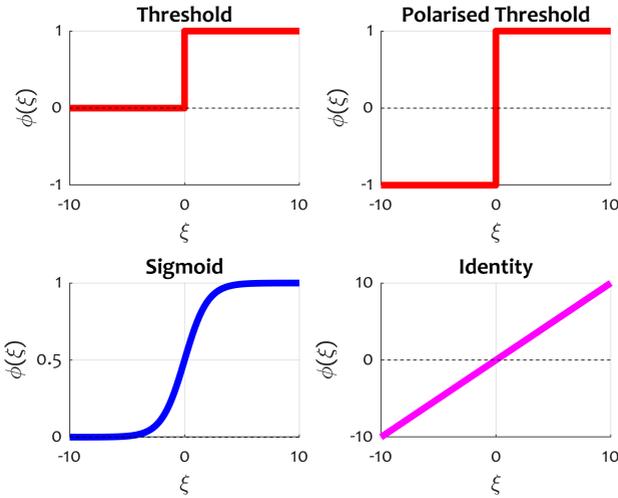


Figure 6.3: Four widely used activation functions.

There are four widely used activation functions which define how the output v depends on the net sum ξ . Notice that, even though the neuron has q inputs (plus the sneaky bias), the net sum is just one number: the weighted sum of the inputs plus the bias weight. Formally the activation functions are defined as follows:

- *Threshold.*

$$v = \begin{cases} 0, & \text{when } \xi < 0 \\ 1, & \text{when } \xi \geq 0 \end{cases} .$$

- *Polarised Threshold.*

$$v = \begin{cases} -1, & \text{when } \xi < 0 \\ 1, & \text{when } \xi \geq 0 \end{cases} .$$

- *Sigmoid.*

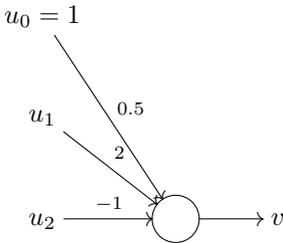
$$v = \frac{1}{1 + \exp(-\xi)} .$$

- *Identity.*

$$v = \xi .$$

Notice the case where ξ is exactly 0. We shall assume that the threshold activation function jumps to 1, and returns $v = 1$.

⊕ ⊕ ⊕ **Example 6.2.1**



Consider the neuron shown on the left. Calculate the net sum ξ and the output v for input $\mathbf{u} = [0.9, 1.5]^T$ using the threshold activation function.

According to the above equations,

$$\xi = 0.5 \times 1 + 2u_1 - u_2 .$$

That is, $\xi = 0.5 \times 1 + 2 \times 0.9 - 1.5 = 0.8$. Since $\xi \geq 0$, the output is $v_{\text{threshold}} = 1$.

Now, let's calculate the output v for the sigmoid activation function:

$$v_{\text{sigmoid}} = \frac{1}{1 + \exp(-0.8)} \approx 0.69 .$$

⊖ ⊖ ⊖

6.2.2 The Threshold Logic Unit (TLU)

A *Threshold Logic Unit (TLU)* is a neuron with a threshold activation function (either the standard or the polarised version). TLU was the first widely used (and widely criticised) model of neuron. The modern version which almost completely replaced the TLU uses the sigmoid activation function because, unlike the threshold activation, the sigmoid is differentiable. A huge advantage! Nonetheless, due to the historical significance of TLU, we will be studying it in more detail.

How can we use TLU for classification? Recall the neuron from Example 6.2.1. Notice that if we set the net sum to zero, $\xi = w_0 + w_1u_1 + w_2u_2 = 0$, we have an equation of a line in 2D. For this TLU, the line is $2u_1 - u_2 + 0.5 = 0$. The line is shown in Figure 6.4.

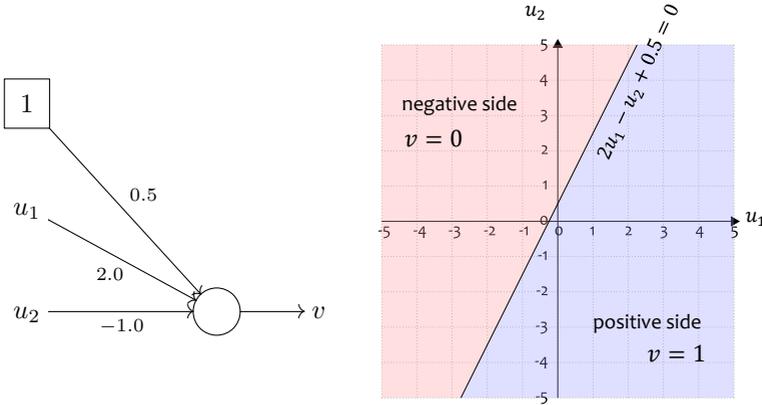


Figure 6.4: Geometric interpretation of TLU as a line splitting the space into two half-spaces corresponding to the two outputs $v = 0$ and $v = 1$.



Hm, this is not an easy thing to do – to figure out how the damn thing on the left in Figure 6.4 all of a sudden becomes a line in 2D! Grab a cup of coffee, sit down and think about this until you get it. This will give you power and enlightenment, I promise!

TLU in 2D splits the plane in two half-planes. All points on the one side of the line will return positive net sums $\xi = 2u_1 - u_2 + 0.5$, and will make the TLU fire ($v = 1$) while all points on the other side will return negative net sums and leave the TLU dormant ($v = 0$).

Now, assume that we put directly features x_1 and x_2 as the TLU inputs u_1 and u_2 . This is how TLU turns into a classifier! We say that $v = 0$ will correspond to class 1 and $v = 1$, to class 2. Voila! We have a linear two-class classifier. In other words, every TLU is a linear classifier. And this is not only in 2D; the same holds for \mathbb{R}^n for any n . In the n -dimensional case, we will have n inputs and $n + 1$ weights but the output is always one of the class labels: $v = 0$ or $v = 1$.

Here is a set of problems and questions for you (see the answers at the back of the chapter):

1. Calculate the output v for a TLU with weights $w_0 = -2$, $w_1 = 4$, $w_2 = -6$, and $w_3 = 5$ for input $[-1, -2, -1]^T$.
2. Calculate the output v for a neuron with the same weights as in the previous question but with a sigmoid activation function.
3. Draw the classification regions for a TLU with weights $w_0 = 3$, $w_1 = 1$, and $w_2 = 0$.
4. How will the classification regions for a polarised TLU differ from the ones for the standard TLU?

6.3 The Perceptron

6.3.1 A bit of history

In 1958, Rosenblatt stirred a controversy among the budding AI community by announcing his *perceptron*. Speculations flourished. New York Times reported that the perceptron is the “embryo of an electronic computer” which will be capable of mimicking numerous human activities such as walking, talking, seeing, writing, and even reproducing itself. Not only that, but this computer will be conscious of its existence! Wow!

Rosenblatt constructed his Perceptron automaton, named Mark I, which occupied 6 racks of electronic equipment. Mark I was meant to be a visual pattern classifier. The success of this early-days physical model led to the current fame and glory of the perceptron.



The Institute of Electrical and Electronics Engineers (IEEE), the world’s largest professional association dedicated to advancing technological innovation and excellence for the benefit of humanity, named its annual award in honour of Frank Rosenblatt.

Technically speaking, the perceptron we use today, is *exactly* a TLU! Agreed, in the current era of challenging and complex classification problems, the perceptron’s practical application as a two-class linear classifier is somewhat limited. But we cannot progress in this field without honouring this highly influential model by diligently studying its details.

6.3.2 The famous perceptron training algorithm

Here comes one way of training a linear classifier in the n -dimensional space. But beware! This will work **ONLY** for two classes, and only

if the classes are completely separable by a linear function.

The algorithm is shown in Figure 6.5.

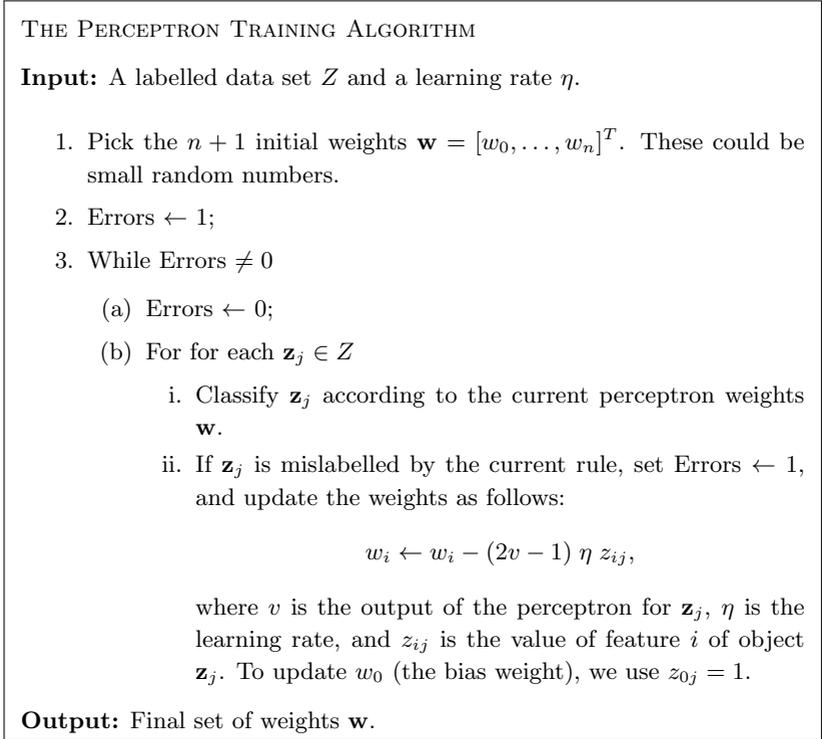


Figure 6.5: The perceptron training algorithm

The learning rate η is a positive constant that determines how quickly or slow the discriminant function will be found. As we shall see later, the convergence of the algorithm does not depend on the value of η but only on whether the classes are linearly separable.

⊕ ⊕ ⊕ **Example 6.3.1**

In this example, we will carry the steps of the perceptron training algorithm for two consecutive elements of Z (two steps of loop 3(b))

Let the current weights be $\mathbf{w} = [3, -2, -1]^T$. Figure 6.6 shows the perceptron line (solid black line) and the two respective class regions: Red and Blue. For class Blue, $v = 0$, and for class Red, $v = 1$. Two points from Z are also plotted, $\mathbf{z}_1 = [4, -2]^T$ and $\mathbf{z}_2 = [-3, 3]^T$, both from class Blue.

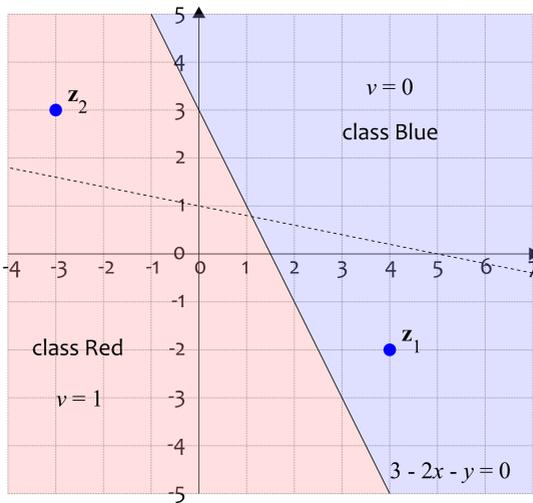


Figure 6.6: A step in the perceptron training algorithm.

Assume that the given learning rate is $\eta = 0.5$. Points \mathbf{z}_1 and \mathbf{z}_2 are submitted in this order to the perceptron training algorithm. Let's calculate the perceptron output for \mathbf{z}_1

$$\xi(\mathbf{z}_1) = 3 - 2 \times 4 - (-2) = -3.$$

As $\xi(\mathbf{z}_1) < 0$, $v = 0$, and we label \mathbf{z}_1 in class Blue. So far, so good. Point \mathbf{z}_1 is in its own classification region, so we don't have a reason to change the weights. For \mathbf{z}_2 , we have

$$\xi(\mathbf{z}_2) = 3 - 2 \times (-3) - 3 = 6.$$

Then $v = 1$, and \mathbf{z}_2 is labelled in class red. Wrong class! Step 3(b)ii in the algorithm tells us that the weight should be recalculated (recall that $\eta = 0.5$ and the current values of the weights are $\mathbf{w} = [3, -2, -1]^T$):

$$\begin{aligned} w_0 &= 3 - (2 \times 1 - 1) \times 0.5 \times 1 = 2.5 \\ w_1 &= -2 - (2 \times 1 - 1) \times 0.5 \times (-3) = -0.5 \\ w_2 &= -1 - (2 \times 1 - 1) \times 0.5 \times 3 = -2.5 \end{aligned}$$

We plot the new boundary in Figure 6.6 with a dashed line. Look what happened! Now \mathbf{z}_2 will be on the right side of the boundary but \mathbf{z}_1 escaped to the wrong side! This is why we have to make passes through the algorithm over and over and update the weights until there are no points outside their own regions (Errors = 0).

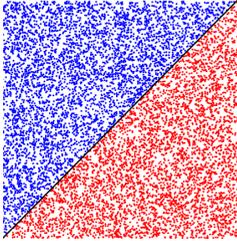
⊖ ⊖ ⊖

6.3.3 The perceptron convergence theorem

Will the algorithm ever converge? The theorem has two clauses (you must know them both):

1. If the two classes are linearly separable in \mathbb{R}^n , then the algorithm always converges in a finite number of steps to a linear discriminant function with zero resubstitution errors, for any learning rate η .

2. If the two classes are not linearly separable in \mathbb{R}^n , then the algorithm will loop infinitely through Z , and never converge for any learning rate η .



The plot on the left shows an example in 2D where 10,000 points are generated in the unit square, labelled into two classes so that the true boundary is the diagonal from (0,0) to (1,1). Drawn in the figure is the final boundary returned by the perceptron training algorithm. How good is that!

The perceptron algorithm is not that difficult to program. You can do it! Start with this:

```
1 function w = perceptron_training(d,l,e)
```

Here \mathbf{d} is the dataset, \mathbf{l} is the vector with the labels, and \mathbf{e} is the learning rate η . The output is the final weight vector. And I can tell you that I have the rest of the function in 106 characters (without counting the white spaces). This count also includes all semicolons to suppress unwanted output in the MATLAB command window. Can you beat that?

The promised answers and solutions

Answers to the questions from page 164:

1. Calculate the output v for a TLU with weights $w_0 = -2$, $w_1 = 4$, $w_2 = -6$, and $w_3 = 5$ for input $[-1, -2, -1]^T$.

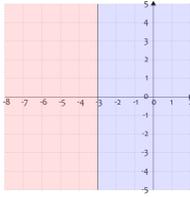
$$\xi = -2 + 4(-1) - 6(-2) + 5(-1) = 1.$$

Since $\xi \geq 0$, $v_{\text{threshold}} = 1$.

2. Calculate the output v for a neuron with the same weights as in the previous question but with a sigmoid activation function.

$$v_{\text{sigmoid}} = \frac{1}{1 + \exp(-1)} = 0.7311.$$

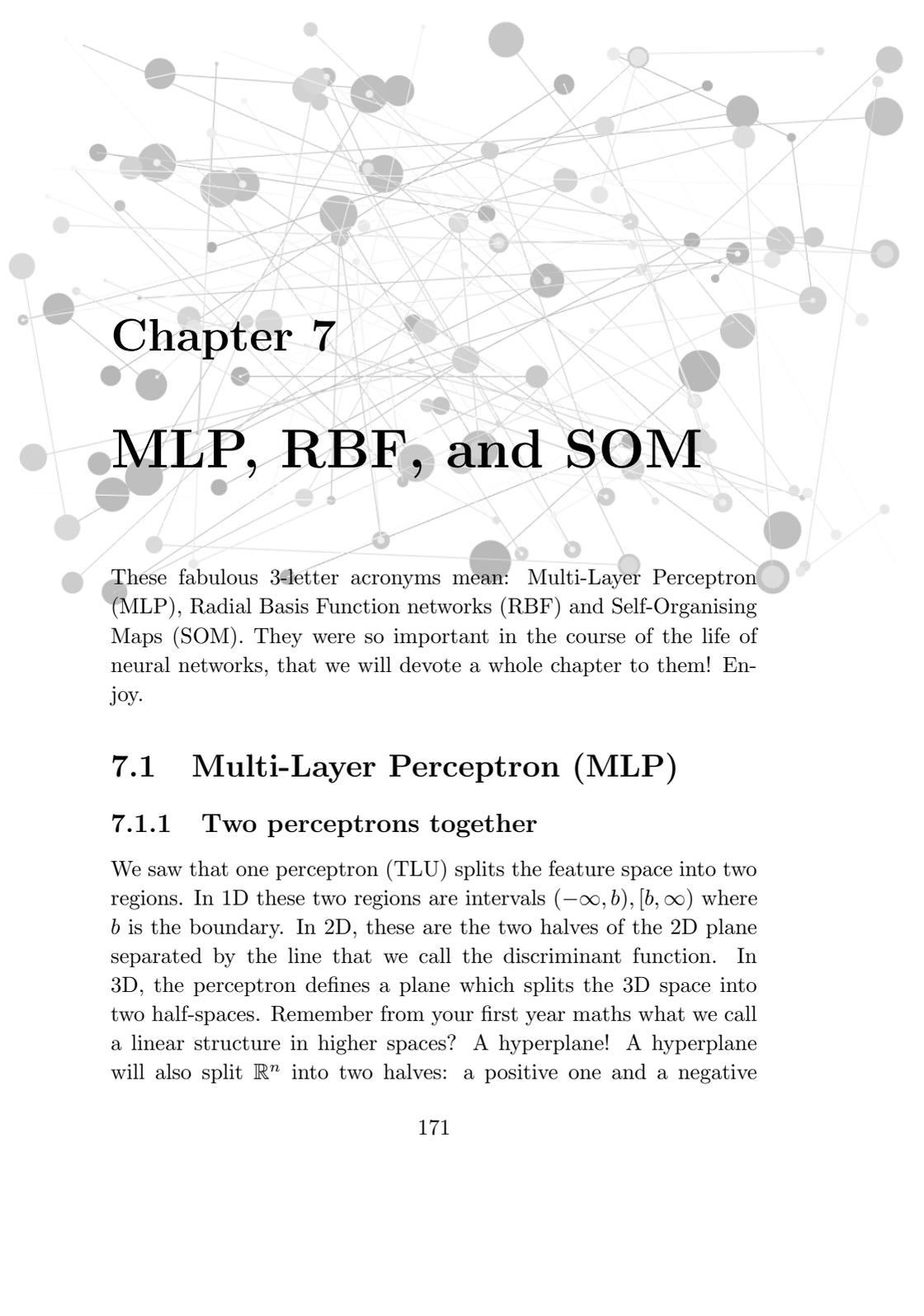
3. Draw the classification regions for a TLU with weights $w_0 = 3$, $w_1 = 1$, and $w_2 = 0$.



The equation of the line is $\xi = 3 + x = 0$ or $x = -3$. The regions are illustrated on the left. The blue region corresponds to $v = 1$, and the pink region, to $v = 0$.

4. How will the classification regions for a polarised TLU differ from the ones for the standard TLU?

They would not differ. The only difference between the two activation function is the value of v . In the standard threshold activation, the labels of the classes would be 0 and 1, and for the polarised activation, -1 and 1.



Chapter 7

MLP, RBF, and SOM

These fabulous 3-letter acronyms mean: Multi-Layer Perceptron (MLP), Radial Basis Function networks (RBF) and Self-Organising Maps (SOM). They were so important in the course of the life of neural networks, that we will devote a whole chapter to them! Enjoy.

7.1 Multi-Layer Perceptron (MLP)

7.1.1 Two perceptrons together

We saw that one perceptron (TLU) splits the feature space into two regions. In 1D these two regions are intervals $(-\infty, b)$, $[b, \infty)$ where b is the boundary. In 2D, these are the two halves of the 2D plane separated by the line that we call the discriminant function. In 3D, the perceptron defines a plane which splits the 3D space into two half-spaces. Remember from your first year maths what we call a linear structure in higher spaces? A hyperplane! A hyperplane will also split \mathbb{R}^n into two halves: a positive one and a negative

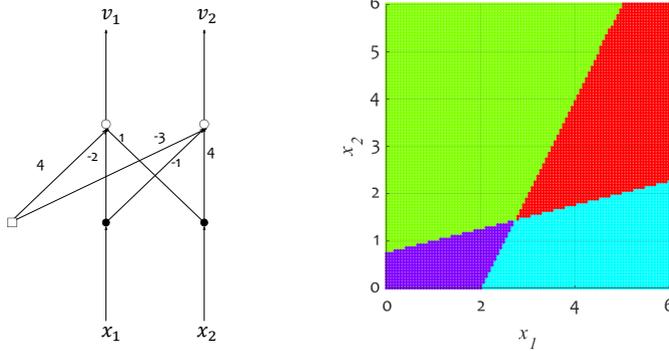


Figure 7.1: An example of four regions defined by an NN with 2 perceptrons.

one. What will happen if we put two perceptrons together? See for yourself in Figure 7.1.

As each perceptron splits the space into two halves, two perceptrons with different weights will define 4 regions. We can therefore distinguish between four classes: $(v_1 = 0, v_2 = 0)$, $(v_1 = 0, v_2 = 1)$, $(v_1 = 1, v_2 = 0)$, and $(v_1 = 1, v_2 = 1)$.



And just imagine what this NN will be able to recognise! A lot more complicated and intertwined class regions.

7.1.2 Structure of MLP

Multi-layer perceptron is called so because it consists of multiple layers of perceptrons. It has an input layer, a set number of hidden

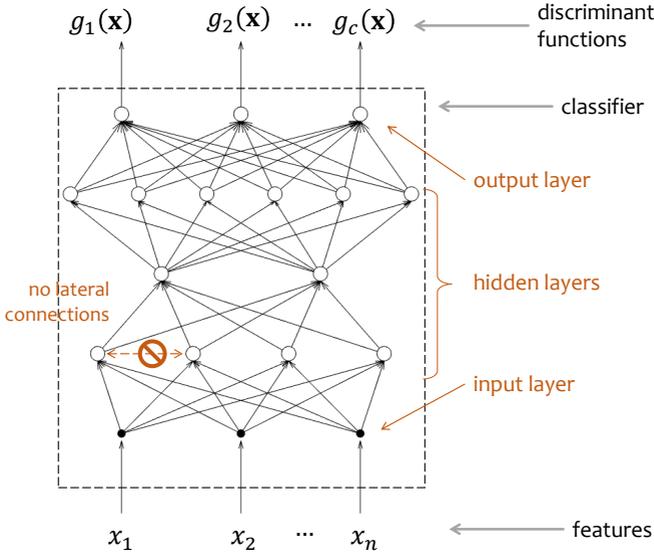


Figure 7.2: Structure of a multi-layer perceptron and its use as a classifier.

layers, and an output layer as shown in Figure 7.2. The features of an object to be classified are submitted at the input layer. The black dots in Figure 7.1 represent the input layer nodes. Their purpose is just to transmit the features to all neurons in the the first hidden layer. According to our categorisation by activation function (Figure 6.3), the input neurons are of type *identity*: they directly put through the net sum value, which in this case is the feature value itself.

Notice that there are no lateral connections at any of the layers, nor are there any backward connections. This type of networks are

called *feedforward* NNs.

The number of hidden layers is not restricted and nor is the number of neurons at each hidden layer. However, if we are to use our NN as a classifier, the number of inputs must be the same as the number of features (n) and the number of outputs must be the same as the number of classes (c). Each output is associated with a class. It will give us the value of the respective discriminant function $g_i(\mathbf{x})$. As always, the final class label is chosen as the tag of the largest discriminant function. And, as discussed before, if we have only two classes, one discriminant function would suffice. We can assign the class labels based on its value. In NNs, typically, the output value is between 0 and 1, and a threshold of 0.5 determines the cut-off points between the classes.

Now, you can argue, my clever clever reader, that we can do exactly the same for multiple classes! Why don't we cut *one* discriminant function (one output) into c intervals and assign class labels according to which interval the output falls in? True, we can do that. But training the network to learn this output pattern becomes awkward. Besides, we will lose important information! If we have all c outputs, we will know how much each of the classes is supported for the given input \mathbf{x} . This will give us a chance to offer 'the top k classes' to the user, which could be very important if we are dealing with a large number of classes.

⊕ ⊕ ⊕ **Example 7.1.1**



This is, actually, something for you to do. Suppose that you have an MLP with two inputs, two hidden layers with 3 and 2 neurons, respectively, and three outputs. The neurons at the hidden layers and the output layer have *identity* activation. So, try this:

1. If you are to use the MLP as a classifier (as explained above),

what is the dimensionality of the feature space, and how many classes are there?

2. Sketch the MLP.
3. Write MATLAB code to calculate the output of this MLP using the weight vectors below (recall that the first weight is always w_0 , the bias weight):

Layer	Neuron #1	Neuron #2	Neuron #3
Hidden 1	$[-2, 1, -1]^T$	$[4, -3, 3]^T$	$[4, -2, -3]^T$
Hidden 2	$[-3, 2, 2, 1]^T$	$[-2, 2, 0, -1]^T$	
Output	$[2, -2, 3]^T$	$[-2, -1, -3]^T$	$[2, 2, 4]^T$

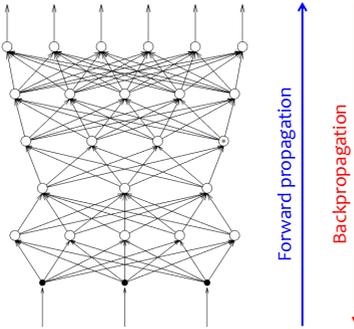
4. Suppose that the outputs of the MLP correspond to classes 1, 2, and 3, respectively. Use your code to calculate the output of the network for objects: $\mathbf{x}_1 = [-2, 6]^T$, $\mathbf{x}_2 = [3, 5]^T$, and $\mathbf{x}_3 = [-1, 8]^T$. Based on the output, determine what class label will be assigned to each of these objects.

Look for the solution at the end of this chapter. But do try first, please!



7.1.3 The error backpropagation algorithm

The *error backpropagation algorithm* gained popularity for training NNs in the 1980s. This is an iterative algorithm which takes labelled data, object by object, and tweaks the weights of the NN to gradually improve the classification accuracy of the NN.



The NN is initialised with small random weights. Each iteration of the algorithm consists of two steps. In the *forward propagation* step, an object is submitted at the input of the NN and the output is calculated. The output is compared to the desired output and an error vector is calculated.

In the *backpropagation* step, the error vector is propagated backwards by calculating the hypothetical error at the previous layer. The weights of the neurons at this layer are changed to minimise the error. The change is similar to that in the perceptron training algorithm. When all the layers are visited, the next object from the training data is submitted at the input. The algorithm finishes when either a limit number of iterations is reached or the training error of the NN drops below a given threshold.

Let's have a look at how the error at the top layer is calculated. Suppose that we use the NN as a classifier into 4 classes. Thus, there will be 4 outputs: $\mathbf{o} = [o_1, o_2, o_3, o_4]^T$. Suppose that the object submitted at the input, \mathbf{x} , is from class 3. The ideal NN output for this object would be $\mathbf{o} = [0, 0, 1, 0]^T$. Now, suppose that our network gave output $\hat{\mathbf{o}} = [0.3, 0.6, 0.4, 0.1]^T$. The error vector is (square error loss)

$$\begin{aligned} \mathbf{e} &= [(\hat{o}_1 - o_1)^2, (\hat{o}_2 - o_2)^2, (\hat{o}_3 - o_3)^2, (\hat{o}_4 - o_4)^2] \\ &= [0.3^2, 0.6^2, 0.6^2, 0.1^2] = [0.09, 0.36, 0.36, 0.01]. \end{aligned}$$

The elements of these vector will be used to change the weights of the four output neurons. The error will be recalculated for the neurons

at the penultimate layer, and propagated in the same way down to the first hidden layer.

There are two versions of the backpropagation algorithm: online and batch.

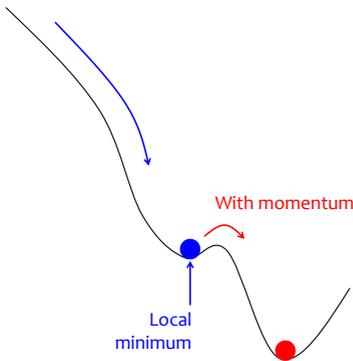
On-line backpropagation training. In the on-line version, the weights of the NN are modified after the presentation of each object, as explained above. The objects in the training data are arranged in random order. A presentation of the whole data set is called *an epoch*. After an epoch, the data in the training set is shuffled randomly, and a new epoch is started. The stopping criterion of the algorithm is often a set number of epochs.

Batch backpropagation training. In the batch version, the weights of the NN are modified after presentation of the whole data set. The objects are presented during the forward pass but there is no backpropagation step until the last object in the data set is presented. The error is accumulated, and used in a single backpropagation step. Thus, the concept of epoch here does work. All of the data set is seen before the backpropagation step.

Remember the learning rate η in the perceptron training algorithm? We found out that it is not too important because the algorithm will always converge if the classes are linearly separable, for any η . The learning rate will only determine how slow or how quickly this will happen. This is not the case for backpropagation. Unfortunately, we have no guarantees that the network can separate the classes perfectly. At each step the algorithm minimises the error but it can easily slip into a local minimum of the error function. In other words, there may be another set of weights which gives a lower training error but the algorithm was “unlucky” to get trapped into the local minimum. Backpropagation is a *stochastic* algorithm. There are several random elements which will govern the algorithm

and eventually determine where it finishes. These random elements are the initial random weights and the ordering of the data points in the training data (for the on-line version). To minimise the chance of getting trapped in a local minimum, researchers have come up with a clever idea: modify the learning rate to ‘dislodge’ the NN from a possible local minimum of the error.

To get a fine-tuned solution, we need to start with a large learning rate, where the NN learns fast, and then decrease the learning rate when we approach the minimum of the error. To avoid getting stuck in a local minimum, we can suddenly shoot the learning rate up again and let the NN follow a different path to another minimum. If this new minimum happens to be less good than the previous one, we can always return and pick the version of the weights which gave us the best NN. The number of these peaks of the learning rate is not limited. If we plot the the learning rate over the number of iterations, the graph may look like saw teeth.



Another interesting idea is to add a *momentum* to the learning rate. This means that once the weights start moving in a particular direction in the weight space, they tend to continue moving in that direction. The benefit of this idea can be illustrated by imagining a ball rolling down a hill.

With the standard learning rate pattern, gradually declining along the iteration count, the ball may get stuck in a little trough on the slope. But if the ball has enough momentum, it will be able to jump out of the trough and continue rolling down the hill.

Choosing the pattern of change of the learning rate and the momentum is the single most important thing in training a NN, including the oh-so-fashionable deep learning NNs.

There are many studies on how to tune these parameters, the most influential recent one being about the Adam optimiser [11].

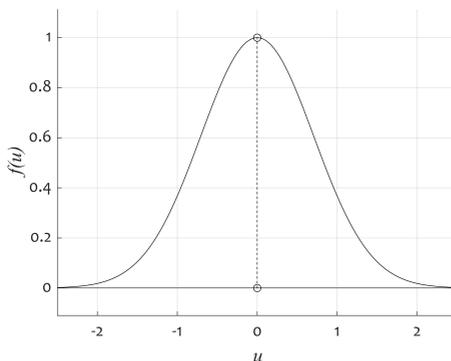


Here are a few questions for you to test your understanding:

1. Describe the principle of the backpropagation algorithm and its two versions.
2. Describe the type of classification regions which a TLU can produce, and which an MLP can produce.
3. Explain how you may use an MLP neural network as a classifier.

7.2 Radial basis functions networks (RBF)

7.2.1 The activation function

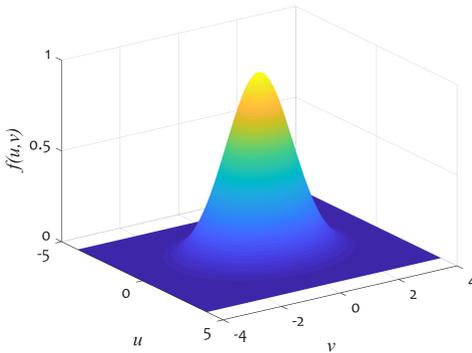


Radial basis function networks take their name from the activation function of their neurons. The shape of the function for a single input u is shown on the left.

The equation is

$$f(u) = \exp\left(-\frac{(u-c)^2}{2s^2}\right),$$

where $c \in \mathbb{R}$ is a centre and s is a positive constant called the *spread* of the function. In the example in the figure, $c = 0$ and $s = 1$. Note that the function is symmetric about c .



In two dimensions, the function looks like a pointy sombrero or a witch's hat.

The function is radially symmetric about the centre, hence RBF. The equation in two dimensions is

$$f(\mathbf{w}) = \exp\left(-\frac{(u-c_u)^2 + (v-c_v)^2}{2s^2}\right),$$

where $\mathbf{w} = [u, v] \in \mathbb{R}^2$ is the vector with the inputs to the neuron, s is again a scalar constant determining the spread of the function, and $\mathbf{c} = [c_u, c_v] \in \mathbb{R}^2$ is the centre.

And here we stop with the picture of RBF, because we can't draw in \mathbb{R}^n . But we can calculate the value of the activation function for n inputs. In this case \mathbf{x} and the centre \mathbf{c} both live in \mathbb{R}^n , that is $\mathbf{x} = [x_1, x_2, \dots, x_n]^T \in \mathbb{R}^n$ and $\mathbf{c} = [c_1, c_2, \dots, c_n]^T \in \mathbb{R}^n$. The activation function returns a single value (the output of the neuron)

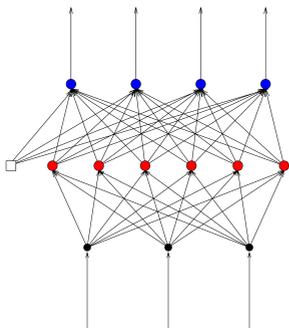
calculated as

$$f(\mathbf{x}) = \exp \left(- \frac{\overbrace{(x_1 - c_1)^2 + \cdots + (x_n - c_n)^2}^{\text{squared Euclidean distance}}}{2s^2} \right).$$

And look at the numerator of the fraction in the exponent! This is the squared Euclidean distance between \mathbf{x} and \mathbf{c} ! This means that all points that are at the same distance from \mathbf{c} (equidistant) will have the same value of $f(\mathbf{x})$. There is the radial symmetry we were talking about!

Intuitively, points which are closer to the centre (in any dimension) will activate the neuron more, compared to more distant points. The highest activation is achieved if the input hits exactly the centre \mathbf{c} .

7.2.2 Structure and operation of RBF



A typical RBF NN consists of an input layer, a single hidden layer with RBF neurons on it, and a single output layer, usually with identity neurons. For each hidden neuron, we need to know the centre vector \mathbf{c} with as many elements as the number of inputs, and the spread s .

For each output neuron, we need $k + 1$ weights, where k is the number of hidden neurons.

The example below shows you how to calculate the output of an RBF NN for a given input.

⊕ ⊕ ⊕ **Example 7.2.1**

An RBF is shown in Figure 7.3. Calculate the output of the network for $x_1 = 2$. (This could easily be an exam question.)

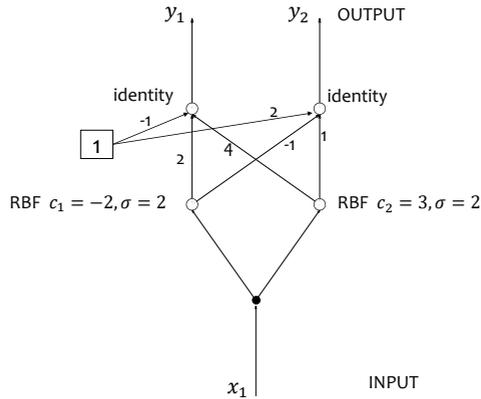


Figure 7.3: An example of an RBF NN.

Solution. Notice that we have a 1D space, hence the centres c_1 and c_2 contain only a single value. The spread is denoted here by σ and has value two for both neurons.

Calculate first the output of hidden neuron 1 (left) for $x_1 = 2$

$$h_1 = \exp\left(-\frac{(2 - (-2))^2}{2 \times 2^2}\right) = \exp(-2) = 0.1353.$$

For the second hidden neuron,

$$h_2 = \exp\left(-\frac{(2 - 3)^2}{2 \times 2^2}\right) = \exp\left(-\frac{1}{8}\right) = 0.8825.$$

Next we calculate the outputs:

$$y_1 = -1 \times 1 + 2 \times 0.1353 + 4 \times 0.8825 = 2.8006.$$

$$y_2 = 2 \times 1 + (-1) \times 0.1353 + 1 \times 0.8825 = 2.7472.$$

If we were using this NN as a classifier with two classes, and treat the outputs as values of the discriminant functions for the two classes, we should assign class 1 to $x_1 = 2$ as $y_1 > y_2$.

⊖ ⊖ ⊖

A question to you: How many parameters does an RBF NN have if it has 5 inputs, 10 nodes at the hidden layer and 3 output nodes? (Assume that all hidden nodes have the same fixed spread $s = 1$)

Solution. $n = 5$ (5-dimensional input space), therefore each centre (one per hidden node) will be an n -dimensional vector too. There are 10 hidden nodes, therefore there will be $10 \times 5 = 50$ parameters for the hidden nodes. There are 3 output nodes, each will have 10 weights plus a bias weight. Therefore, there will be $3 \times (10 + 1) = 33$ parameters for the output nodes. Then the total is $50 + 33 = 83$ parameters.

7.2.3 Training of RBF

Training an RBF amounts to finding all its parameters so that we can label an input. Among many possible training methods, I chose for you the following two:

Method 1: Random centres + regression. Pick the centres randomly from the training set. Then train the weights of the output nodes using a regression technique.

Figure 7.4 shows the results of this method of training for a 2-class data set.

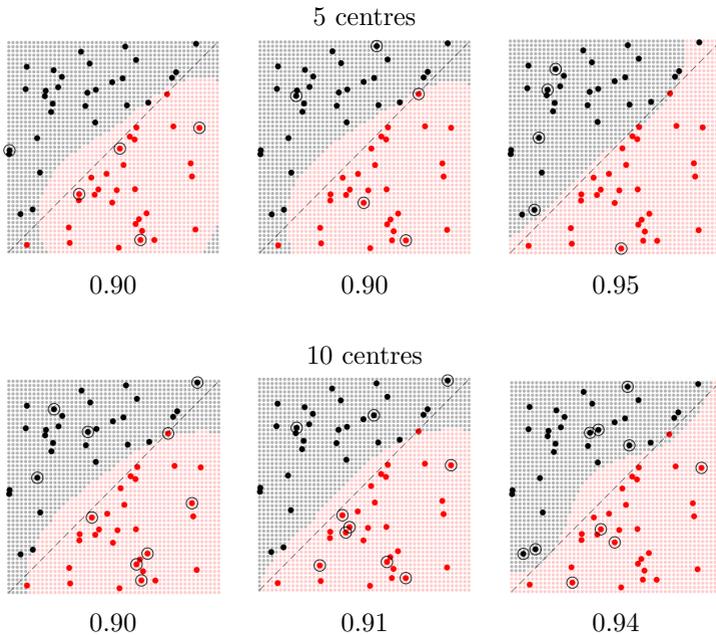


Figure 7.4: Results from training RBF NNs on the same data set using Method 1: Random centres + regression. The testing accuracy is shown under each plot.

The classes are plotted with black and red dots. The region for the black class discovered by the RBF is shown in grey, and the region for the red class, in pink. The actual classification boundary used to label the points originally is the diagonal plotted with the black dashed line. These regions are what we want to approximate. But we have to be fair to these RBF guys, they don't know that! They only have the data in the scatter plot.

We trained six RBFs for the same data set; three with 5 cen-

tres and 3 with 10 centres (sigma was set at 0.5 in all runs). The data points randomly picked as centres are circled. The testing accuracy of the RBF classifier is given under the respective subplot. All RBF versions have actually done a pretty good job in identifying the regions, haven't they?

Method 2: Trained centres + regression. This training method is a little more elaborate than Method 1. It goes through the following steps:

1. Set a target threshold ϵ for the training error. Start with an empty set of centres.
2. Add temporarily one centre at a time from the available points in the data set. Calculate the weights using a regression technique. Calculate the RBF NN error with the added centre.
3. Choose the centre with the smallest error (E) and add it permanently to the set with centres.
4. If E is larger than ϵ , then repeat from the Step 2. Else, return the trained RBF (centres and weights).

We can set an arbitrarily small ϵ but then we are risking over-training. If we continue the training process until all data points become centres, then the training error (resubstitution error) will be equal to zero. What is then a good error rate threshold? We could use a validation set to gauge this. We can cut some of our (precious!) training set to serve as validation, and keep training while the error of the RBF on the validation set goes down. As soon as it starts picking up, we should stop the training and return the RBF version corresponding to the lowest error on the validation set.

Figure 7.5 shows the progression of the training of an RBF using Method 2. Starting with one centre, the training proceeds to selecting 13 out of the 60 centres. At this iteration, the training error

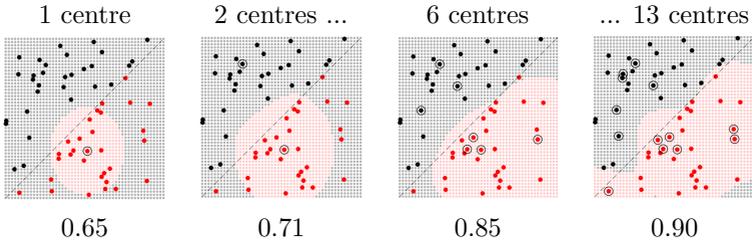


Figure 7.5: Progression of the RBF training using Method 2: Trained centres + regression. The testing accuracy is shown under each plot.

dropped under the threshold $\epsilon = 0.001$. The Figure shows iterations 1, 2, 6 and 13, and the respective *testing* error under each subplot.

Figure 7.6 shows the training and testing errors as the training progresses from 1 to 13 centres. The training error goes down at each iteration while the testing error levels off. The best testing error is achieved at 7 centres while the training error achieves minimum at 13 centres. Overfitting danger!

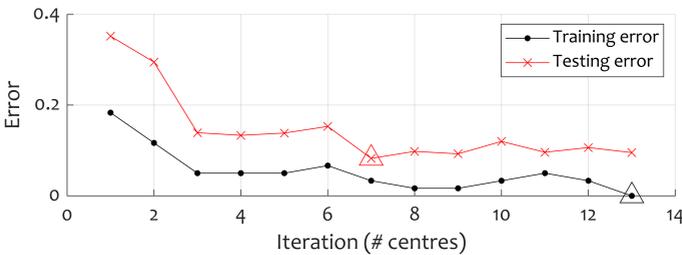


Figure 7.6: Training and testing error of RBF training Method 2 as a function of iterations (number of centres). The best values are indicated with triangle markers.

They are good! RBF neural networks are quite good. The problem, as with almost any other classifier, is picking the right combination of parameters: number of centres, the spread sigma, the training method (and the error threshold, if applicable). A large training data would let us set aside a validation set, and tune parameters on unseen data.

7.3 Self Organising Maps (SOM)



Finish scientist Teuvo Kohonen has been credited with the creation of the foundation of Self Organising Maps (SOM).

The SOM algorithm grew out of early models of associative memory and adaptive learning.

7.3.1 Definition and examples

Self-organizing maps (SOMs) are a data visualisation technique which reduces the data dimensionality. The main idea is to produce a ‘map’ of usually one or two dimensions. Each node on the map is responsible for a subset of data points. Note that these data points live in \mathbb{R}^n even though their representation is in 1D or 2D. Ideally, similar data points in \mathbb{R}^n will be represented by the same node or by neighbouring nodes in the SOM.

What can SOMs be useful for? A neat and convincing example of the wonders of SOM is developed by Weller et al. [21]. Palynology is the study of pollen grains and other spores (palynofacies), especially in geological deposits. The existence, type and density of such spores can be indicators of oil deposits! Moneeeeeey! Under the microscope they look as the lovely duvet pattern in Figure 7.7.

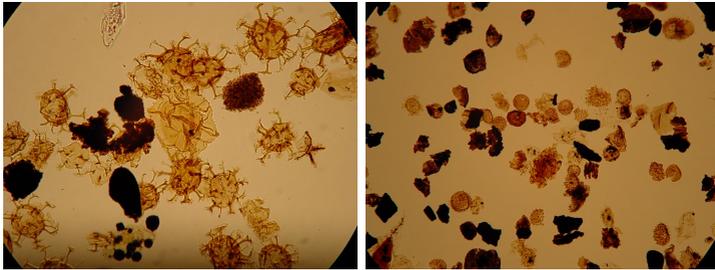


Figure 7.7: Examples of microscope images containing palynofacies.

Suppose we have a data set with many images. How can we catalogue the species within? Let us dream on and assume that we have a nice piece of software which will extract the single creatures from each image, and will take the most important collection of features from each creature. Now train a SOM network to place all these creatures in a 2D table. Similar creatures go in the same cell. Figure 7.8 is reproduced from the study by Weller et al. [21]. Lo and behold, the creatures have found their ways into the table!

Compare the top left cell with the bottom right cell! They can't be more different. And, indeed, they are farthest apart in the map. Neighbouring cells contain similar creatures. This is all done automatically, by training the SOM. How cool is that?

Here is another example. Suppose that we are interested in the content of an array of over 3,000,000 colours. Think of each stored colour as a pixel in an image. The array is pictured in Figure 7.9 (a).

Figure 7.9 (b) is the initial SOM where each cell of the map is initialised with a random colour. By the end of the training, the map looks as shown in Figure 7.9 (c). A different initialisation will lead to a different final map but the palette of colours will be similar. Examples of five outputs started with different SOM initialisations are shown in Figure 7.10.

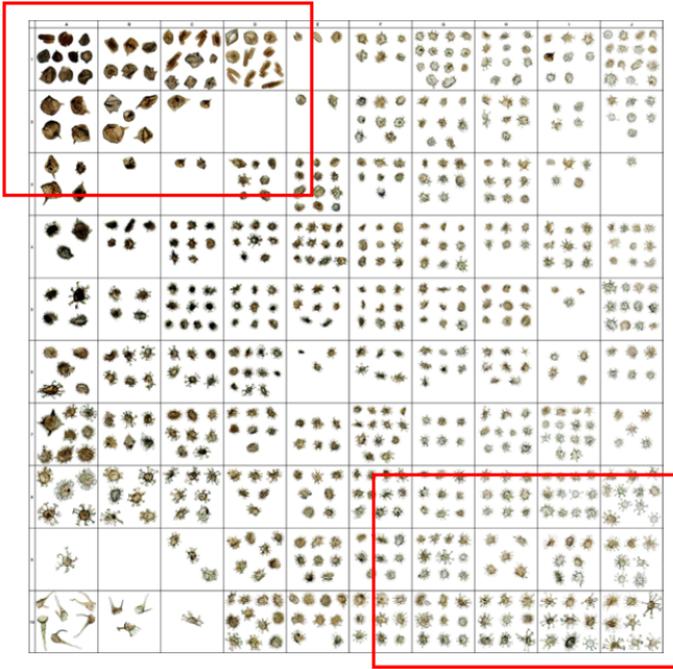


Figure 7.8: A SOM with categorised palynofacies (after [21]).

Are you curious now to see where these colours come from? The image used for training the SOM is shown in Figure 7.11. It is a beautiful view of Menai Strait photographed from the Bangor side. The pixels were scrambled and fed to the training algorithm of the SOM. The colours in the photo are represented well in the SOM outputs even though the arrangements are different.

For comparison, we we ran the SOM training for another image with a different colour palette. The results are shown in Figure 7.12.

Don't get me wrong! SOM are not used exclusively for revealing

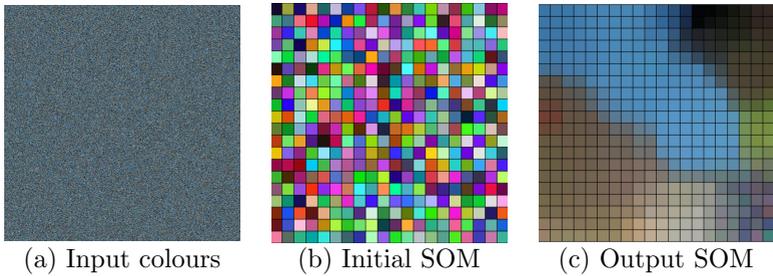


Figure 7.9: Illustration of SOM.

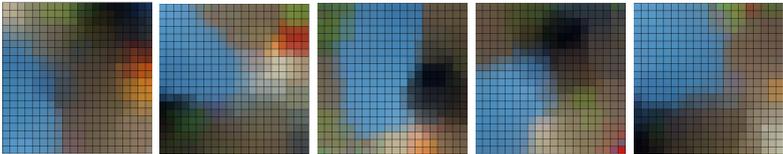


Figure 7.10: Different SOM outputs for the input array of colours shown in Figure 7.9 (a).

colours in arrays. Arrays of colours make a nice example of how data in the original 3D space (red-green-blue) is summarised in a 2D table.

7.3.2 Training of SOMs

Hebbian learning

Training of SOMs draws upon Hebbian learning (Donald Hebb, 1949) which postulates that:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously), then the strength of that synapse is selectively increased.



Figure 7.11: Original images for the SOM in Figures 7.9 and 7.10.

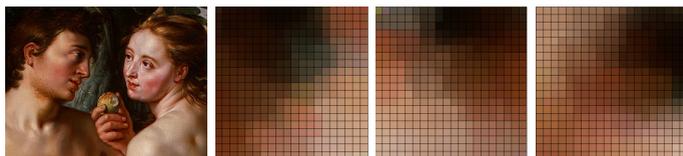


Figure 7.12: Input image (colours) and three SOM outputs.

2. If two neurons on either side of the synapse are activated asynchronously, then this synapse is selectively weakened or eliminated.

Understanding the SOM activation pattern

Training of SOM is based on the concept of activation pattern. Consider a 2D SOM where each cell of the map is a node of the network.

Each node on the SOM stores n values and can be associated with a point in \mathbb{R}^n . Denote those points by $\mathbf{s}_{i,j}$, where i is the row

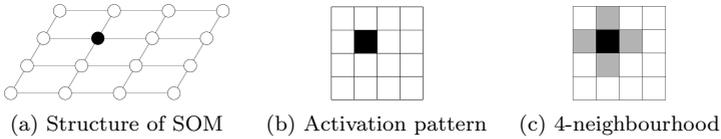


Figure 7.13: Structure of SOM, an activation pattern, and a 4-neighbourhood pattern.

of the node and j is its column. During the SOM training, objects from the training set are submitted in random order. For each object, the distances to all $\mathbf{s}_{i,j}$ are calculated, and the nearest node is identified. This node gets activated, or ‘fires’, while the remaining nodes remain dormant. Subplot (a) in Figure 7.13 highlights in black a hypothetical activated node, say (i^*, j^*) . Subplot (b) shows the *activation pattern* of the SOM, where the chosen cell of the map is shown in black. In addition, we consider the *neighbourhood* of the activated cell. These are the cells on the SOM closest to the activated cell, and NOT the cells whose $\mathbf{s}_{i,j}$ are closest to \mathbf{s}_{i^*,j^*} . The 4-neighbourhood of the activated cell is shown in subplot (c). Sometimes 8-neighbourhood is used instead of 4-neighbourhood.

The SOM training algorithm

The SOM training algorithm is shown in Figure 7.14.

The algorithm runs T times through the data set Z (as before, each pass through Z is called an epoch) receiving one element $\mathbf{z} \in Z$ at a time, and updating the SOM accordingly. The winner node and its neighbours are identified. Their weights $\mathbf{s}_{i,j}$ are updated using equations (7.1) and (7.2), respectively. The updates will ‘pull’ the node in \mathbb{R}^n towards \mathbf{z} . The difference in the update is only in that the winner is pulled by α , and the neighbours, by α^2 . Since $\alpha < 1$, $\alpha^2 < \alpha$, which means that the winner will be pulled closer than the

THE SOM TRAINING ALGORITHM

Input: A labelled data set Z , the SOM size ($M \times K$), a learning rate $\alpha \in (0, 1)$, and a limit number of epochs T .

1. *Initialisation.* Initialise the weights $\mathbf{s}_{i,j}$ on the $M \times K$ grid with small random numbers, $i = 1, \dots, M, j = 1, \dots, K$. Shuffle Z .
2. *Competition.* A new data point $\mathbf{z} \in Z$ is presented to the algorithm. The nodes on the grid compete for it, and the closest node (in the input space \mathbb{R}^n) is declared the winner.
3. *Cooperation.* The neighbours of the winner are identified.
4. *Update.* The weights of the winner and the neighbour nodes are updated, so that the nodes become ‘more like’ the input. (Self-amplification inspired by Hebbian learning.) The following equation is used for the winner node:

$$w_{\text{new},q} \leftarrow w_{\text{old},q} + \alpha(z_q - w_{\text{old},q}), \quad q = 1, \dots, n. \quad (7.1)$$

where $w_{\text{old},q}$ is the current value of the q -th element of the winner’s $\mathbf{s}_{i,j}$ and z_q is the q -th element of \mathbf{z} . The neighbours are updated as

$$w_{\text{new},q} \leftarrow w_{\text{old},q} + \alpha^2(z_q - w_{\text{old},q}), \quad q = 1, \dots, n. \quad (7.2)$$

5. *Repetition.* If the number of epochs (passes through the whole of Z) is not reached, shuffle Z and continue from step 2.

Output: SOM network with $\mathbf{s}_{i,j}, i = 1, \dots, M, j = 1, \dots, K$.

Figure 7.14: The SOM training algorithm.

neighbours.

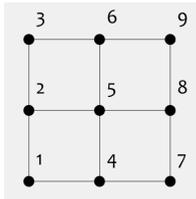
And notice another thing – the pulling only happens in \mathbb{R}^n . The SOM doesn’t move an inch! It is the same grid from start to finish. Only the correspondence of the nodes with \mathbb{R}^n will change; the nodes

will move until they are settled. (In the examples above, the initial random colours will gradually move towards colours represented in the colour array so that similar colours in \mathbb{R}^3 stay close in the SOM grid.)

There is another little detail which I missed in the algorithm. On purpose. It may complicate the matter a little but, in real applications, it is essential. If we use the same α throughout, the algorithm may not converge to a precise solution. It may oscillate about the solution without going close. This is why we need yet another parameter which will dictate how α decreases with iterations. We may choose $\gamma < 1$ so that at each iteration we decrease α by $\alpha \leftarrow \alpha \times \gamma$. The problem with such algorithms is that the success of the algorithm depends critically on the right tuning of the parameters. In other words, we need to run experiments to determine useful values of: M , K , α and γ .

⊕ ⊕ ⊕ Example 7.3.1

Here we will carry out only two tiny steps of the SOM training algorithm to see how things work.



Consider a SOM network as a 3-by-3 grid as shown on the left. The neighbourhood of a node is defined as all the nodes immediately connected to it. At the beginning of the training process, the weights of the nodes are random pairs of values as shown in Table 7.1.

Table 7.1: Initial weights of the SOM in Example 7.3.1.

node #	1	2	3	4	5	6	7	8	9
w_1	-10	11	10	-5	-7	-8	12	-8	-8
w_2	3	-6	-5	-5	7	12	13	-12	-2

Object $\mathbf{z}_1 = [7, 10]^T$ has been submitted for training of the SOM, followed by object $\mathbf{z}_2 = [3, 0]^T$. Let's see how the SOM will change after each training step. We shall assume that the learning rate is $\alpha = 0.6$ and will forget about γ for now.

Notice that there are only 2 weights for each node of the SOM. So, our data lives in \mathbb{R}^2 . We love \mathbb{R}^2 ! We can plot things there. So we will start by plotting weights and \mathbf{z}_1 to find the winning node. Figure 7.15 shows this plot.

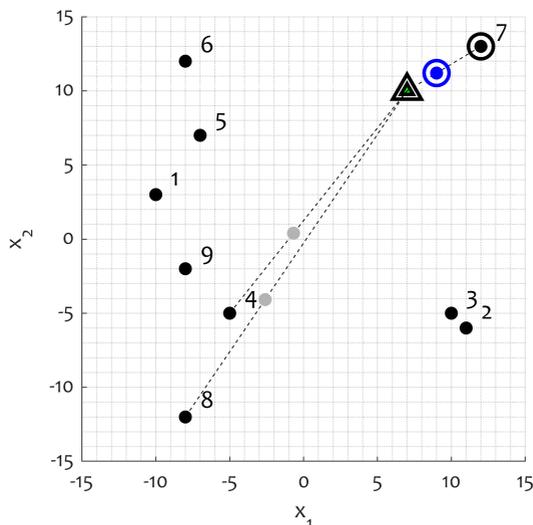


Figure 7.15: Representation of the SOM nodes from Example 7.3.1 in the original 2D space and the effect of training with object \mathbf{z}_1 (shown with a triangle).

Notice that, at the start of the training, neighbourhood in the SOM grid does not translate into neighbourhood in the original

space. When we change the weights, we change the positions of the nodes in \mathbb{R}^2 but not on the SOM grid. Thus the points will float around until they represent the data that is used to train the SOM. This data set is NOT shown in the figure! The data points $\mathbf{z} \in Z$ will be coming one by one and make the nodes float about until they assume positions representing the data in the best possible way. For example, the nodes may nestle in the centres of clusters. This will happen as the of points from a cluster will be close to one another and will thus activate the same winner node on the SOM.

The data point of interest, $\mathbf{z}_1 = [z_{11}, z_{12}]^T = [7, 10]^T$, is shown with a fancy triangle in the figure. Its closest node from the SOM (in the original space!) is node 7. Node 7 is the winner, marked with a black circle. In the cooperation step, we identify the neighbours of node 7. Here is the catch! These neighbours are NOT in the original \mathbb{R}^2 space; they are the neighbours in the SOM lattice! Node 7 is connected with nodes 4 and 8, therefore they will be updated too. At the update phase, according to equation (7.1), we have:

$$w_{7,1} \leftarrow w_{7,1} + \alpha(z_{11} - w_{7,1}) = 12 + 0.6 \times (7 - 12) = 9$$

and

$$w_{7,2} \leftarrow w_{7,2} + \alpha(z_{12} - w_{7,2}) = 13 + 0.6 \times (10 - 13) = 11.2.$$

The new position of node 7 is shown with a circled blue dot. Observe that node 7 is pulled at $\alpha = 0.6$ of the way from 7 to \mathbf{z}_1 . For the neighbour nodes 4 and 8, the updates are similar but we use α^2 instead of α , which will pull them only 0.36 of the way from the node to \mathbf{z}_1 . The new positions of these nodes are marked with grey dots in the figure. Their new weights are respectively $\mathbf{w}_4 = [-0.68, 0.4]^T$, $\mathbf{w}_8 = [-2.6, -4.08]^T$.

You can complete this example on your own! The starting position of the nodes in \mathfrak{R}^2 will be the position *after* the change resulting

from seeing \mathbf{z}_1 . Go ahead, have a try. The answer is at the back of this chapter. ☹ ☹ ☹

And guess what! You can program the SOM in MATLAB, even the fancy version with both alpha and gamma. The trickiest part would be determining the neighbourhood on the SOM grid. Best of luck!

SOM had a few close relatives over the years: Vector Quantisation (VQ) and Learning Vector Quantisation (LVQ). VQ was the unsupervised version and LVQ was the supervised one. Both VQ and LVQ use training algorithms very similar to SOM's, mimicking Hebbian learning. VQ has been used for clustering and LVQ for classification. But, somehow, they have run their course and have given way to newer, more effective classification methods. Will SOM survive the quest of time? Who knows? Keep it in your piggy bank as a quirky visualisation tool.

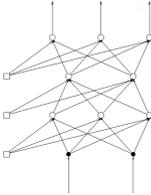
A final remark

There are many more types and models of NNs, and we haven't even touched the elephant in the room, the Deep Learning! I used to teach Hopfield NNs in this module, not because they are very widely used but because they are part of the 'classics'. The field is really very large and there is no space here to indulge in reinforcement learning, Boltzmann machines and more. But now you know the basics and you can build upon that foundation. ☺

The promised answers and solutions

Solution of Example 7.1.1

1. Dimensionality of the feature space is $n = 2$. The number of classes is $c = 3$ because there are three outputs.



2. The sketch of the MLP is shown on the left.

3. MATLAB code to calculate the output of the MLP for a random input $\mathbf{x} = [1, 2]^T$. The code displays at the end the three outputs of the MLP for the given input.

```

1 clear, clc, close all
2
3 % Weights of the first hidden layer H1
4 w1 = [-2, 1, -1; 4, -3, 3; 4, -2, -3];
5
6 % Weights of the second hidden layer H2
7 w2 = [-3, 2, 2, 1; -2, 2, 0, -1];
8
9 % Weights of the output layer
10 wo = [2, -2, 3; -2, -1, -3; 2, 2, 4];
11
12 x = [1, 2]; % input
13 xa = [1 x]; % augmented x with u_0 = 1 (the bias input)
14
15 h1 = w1 * xa'; % the three outputs of H1
16 ah1 = [1 h1']; % augmented output of H1
17
18 h2 = w2 * ah1'; % the two outputs of H2
19 oh2 = [1 h2']; % augmented output of H2

```

```

20
21 o = wo * oh2'; % the output of the MLP
22 disp(o) % display the MLP output

```

The output of this code is $o_1 = -12, o_2 = 9, o_3 = -12$. This output spells class 2 for the given $\mathbf{x} = [1, 2]^T$ because $o_2 > o_1$ and $o_2 > o_3$.

4. Now just replace line 12 with $\mathbf{x} = [-2, 6]$; and read the output: $o_1 = -80, o_2 = 11, o_3 = 0$, which assigns class label 2 to $\mathbf{x}_1 = [-2, 6]^T$. For $\mathbf{x}_2 = [3, 5]^T$, we get $[39, -15, 14]$, which places it in class 1, and for $\mathbf{x}_3 = [-1, 8]^T$, the output is $[-54, -3, 16]$, which places it in class 3.

Solution of the second part of Example 7.3.1

Figure 7.16 illustrates the movement of the nodes of the SOM after submitting $\mathbf{z}_2 = [z_{21}, z_{22}]^T = [3, 0]^T$.

The closest node in R^2 will be the (new) node 4, and the neighbours (on the SOM!) will be nodes 1, 5 and 7. The calculations are shown below:

$$w_{4,1} \leftarrow w_{4,1} + \alpha(z_{21} - w_{4,1}) = -0.68 + 0.6 \times (3 - (-0.68)) = 1.5280$$

and

$$w_{4,2} \leftarrow w_{4,2} + \alpha(z_{22} - w_{4,2}) = 0.4 + 0.6 \times (0 - 0.4) = 0.16.$$

For the neighbours,

$$w_{1,1} \leftarrow w_{1,1} + \alpha^2(z_{21} - w_{1,1}) = -10 + 0.36 \times (3 - (-10)) = -5.32$$

$$w_{1,2} \leftarrow w_{1,2} + \alpha^2(z_{22} - w_{1,2}) = 3 + 0.36 \times (0 - 3) = 1.92$$

$$w_{5,1} \leftarrow w_{5,1} + \alpha^2(z_{21} - w_{5,1}) = -7 + 0.36 \times (3 - (-7)) = -3.4$$

$$w_{5,2} \leftarrow w_{5,2} + \alpha^2(z_{22} - w_{5,2}) = 7 + 0.36 \times (0 - 7) = 4.48$$

$$w_{7,1} \leftarrow w_{7,1} + \alpha^2(z_{21} - w_{7,1}) = 9 + 0.36 \times (3 - 9) = 6.84$$

$$w_{7,2} \leftarrow w_{7,2} + \alpha^2(z_{22} - w_{7,2}) = 11.2 + 0.36 \times (0 - 11.2) = 7.168.$$

The new position of 4 is marked with a circled blue dot, and the new positions of 1, 5 and 7, with grey dots.

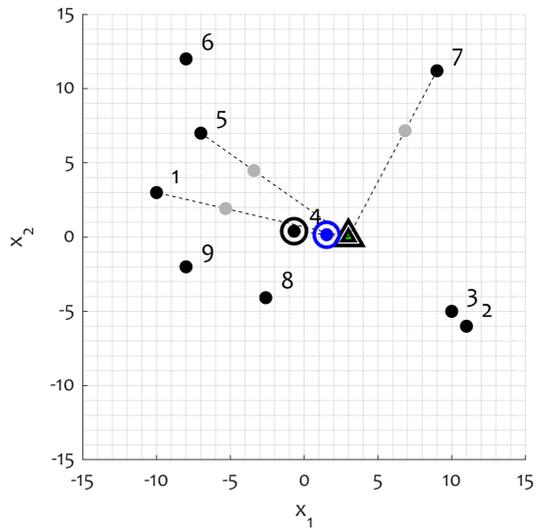
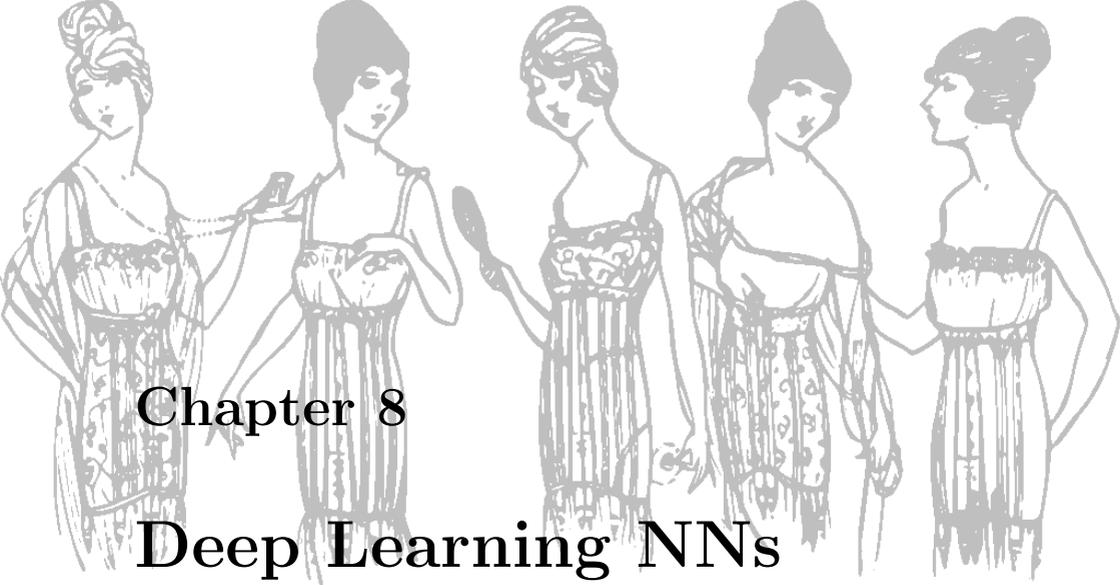


Figure 7.16: Representation of the SOM nodes upon presentation of training object z_2 (triangle) *after* training with z_1 .



Chapter 8

Deep Learning NNs

Deep learning neural networks (DL) are top fashion at the moment. There could be a whole module, and more than one, come to that, about DL. We will only have a little glance in this module.

8.1 Some definitions

A deep-learning neural network is a large-scale neural network with multiple hidden layers of units. A DL can be viewed as a structure with multiple levels of representations of the data where higher level features are derived from lower level features to form a hierarchical representation. The final representation layer can be used as input to a standard classifier. In a way, DL can be thought of as giant feature extractors.

Unlike standard NNs, DL have bespoke and elaborate training algorithms and protocols. Two essential requirements for using DL are:

- *Parallelism.* Multiple graphics processing units (GPUs), mul-

multiple machines, computer clusters are needed for the training of a DL.

- *Large training data.* The number of data points in the training data set must be large: thousands, sometimes millions. If the data set is not sufficiently large, it is often augmented with artificial data points.

There is no consensus at the moment about the exact birthday of DL or the people who conceived it (maybe there will be by the time you read this book) [17]. DL area grew gradually on the foundation of NNs, and keeps moving forward with massive leaps owing to scientists like Geoffrey Hinton, Andrew Ng, Yann LeCun, Andrej Karpathy and many more.

8.2 Applications of DL

Jason Brownlee keeps an enlightening website with curious applications of DL¹. His list includes eight applications:

1. Colourisation of Black and White Images.
2. Adding Sounds To Silent Movies.
3. Automatic Machine Translation.
4. Object Classification in Photographs.
5. Automatic Handwriting Generation.
6. Character Text Generation.
7. Image Caption Generation.
8. Automatic Game Playing.

Image colourisation with DL is amazing! See the results published by Zhang et al. [22] in Figure 8.1.

¹<https://machinelearningmastery.com/inspirational-applications-deep-learning/>

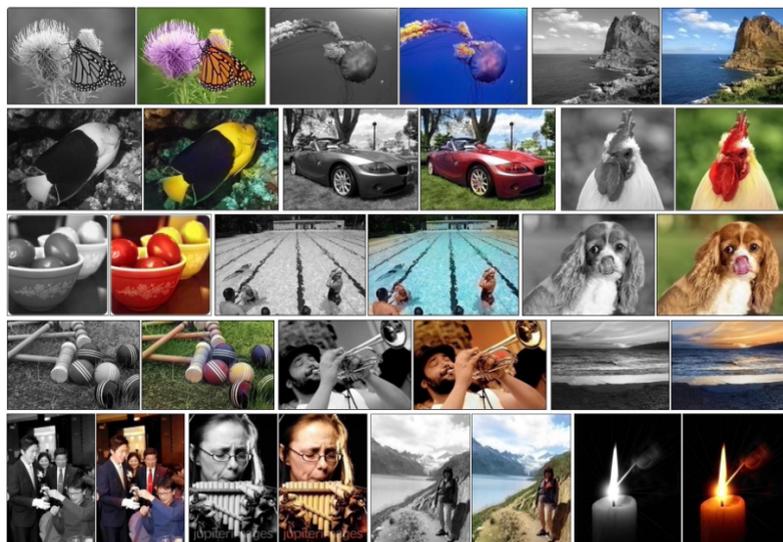


Figure 8.1: Example of image colourisation with DL (Zhang et al. [22]).

Plus, I checked out their cool demo.² See the results in Figure 8.2. Compare the result with the original which I converted into grey to submit to the demo. How good is that? Even Pikachu is yellow! I can't shake off the thought that there was somebody on the other end of the demo who quickly laid the colours by hand. (Kidding...)

This colourisation has been applied to colourise old black and white movies.

The second application in Brownlee's list – adding sound to silent movies – is also fabulous. One day you will have books with sound, but not today. You'll have to watch the little video on YouTube.³

²<https://demos.algorithmia.com/colorize-photos>

³https://www.youtube.com/watch?time_continue=12&v=0FW99AQmMc8

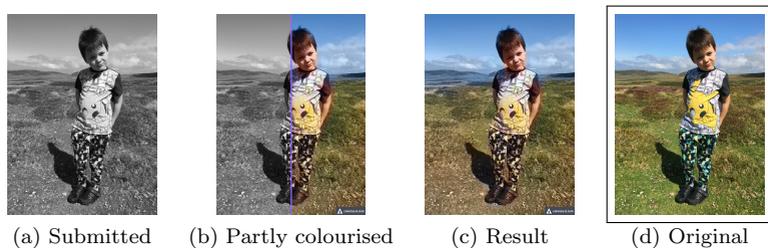


Figure 8.2: Results from the grey image colourisation using DL.

Figure 8.3 is reproduced from the website of Google Translate app announced in July 2015.⁴ It illustrates the success of DL in translating text in images.

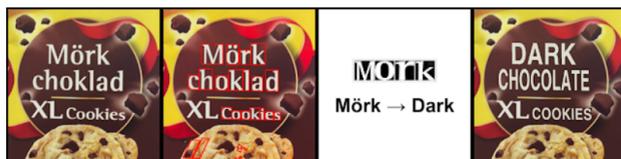


Figure 8.3: An example of the stages of image text translation by DL.

Automatic object classification in photographs has been a precious AI dream for a long time. Recall the ImageNet data mentioned in the introduction chapter (Figure 1.5)? This collection has been a treasure for training and evaluating DLs. Figure 8.4 shows an image as well as the tags and labels proposed by two image labelling systems: Imagga⁵ and Vision AI⁶.

Vision AI even outlines the boxes with the detected objects.

What is even more interesting, is that DL can learn form specific,

⁴<https://ai.googleblog.com/2015/07/how-google-translate-squeezes-deep>.

Imagga



100.00%	ski
82.57%	snow
73.37%	skier
72.50%	mountain
64.06%	winter
58.76%	cold
42.18%	sport
40.90%	mountains
36.73%	slope
32.70%	extreme

Vision AI



92%	Person
91%	Person
79%	Person
76%	Ski
68%	Pants
61%	Outerwear
52%	Luggage & bags

Figure 8.4: Two examples of labelling an image by a DL.

complex concepts from data without knowing any labels! This is related to the idea of a ‘grandmother cell’, proposed in the 1960 and quickly dismissed by psychologists at the time.

html

⁵<https://imagga.com/auto-tagging-demo>

⁶<https://cloud.google.com/vision/>



The grandmother cell is a hypothetical neuron that represents a complex but specific concept or object.

Suppose that the concept/object is the person's grandmother. The neuron will activate upon seeing Grandma, hearing her name or talking about her. Curiously, people have also named this neuron the 'Jennifer Aniston Neuron'. So, I can rest assured that, in my brain, I have neuron that lights up like a Christmas tree every time when I see Jennifer Anniston on the screen, or even now as I am typing! I love this!



And even more curiously, a massive experiment with DL led to the same discovery! One neuron (the grandmother neuron) learned to pick cats (yes, cats!) after seeing 10 million *unlabelled* videos.

In 2012, a research team led by Andrew Ng and Jeff Dean, fed 10 million random YouTube videos to the Google Brain Simulator built on 16,000 computers with one billion connections. Google Brain developed that peculiar neuron which fired only for videos featuring cats even though there were no labels (cats/no cats) on the training videos.

DeepMind Technologies was founded in the UK in 2010 and swiftly bought by Google in 2014. The company has created a neural network that learns how to play video games in a similar way that humans do, based only on the pixels shown on the computer screen.



How Google's AlphaGo Beat a Go World Champion

In 2016 the AlphaGo program developed by DeepMind beat a human professional Go player Lee Sedol, the world champion at the time. World is competing to develop faster and cleverer DL models capable of defeating star human players of any game. The more complex the game, the better!

OK, these applications were more for fun. According to almighty Wikipedia, DL have been used in many more areas:

9. Automatic speech recognition
10. Natural language processing
11. Drug discovery and toxicology
12. Customer relationship management
13. Recommendation systems
14. Generating video from still images
15. Bioinformatics
16. Medical image analysis
17. Image restoration
18. Financial fraud detection
19. Military

The list keeps growing. Where is data (and these days data is everywhere), DL can prove to be useful.

But don't get too excited yet! DL are NOT the answer to everything! They are good but not perfect. Moreover, they can be fooled. And ever since the rise of DL there has been a parallel stream of research looking into the flaws and brittleness of DL. [15, 19]



Do you know what this creature is? Yes, an ostrich (*struthio camelus*). You would imagine that a DL NN would easily learn to recognise an ostrich; its appearance is so unmistakable.



But look an laugh! Our super-clever DL thinks *all* these images are ostriches too!

How could this happen? The thing is, we can change a tiny amount of pixels in an image and flip the DL's decision into a completely different category. The resultant image looks remarkably like the original but the DL no longer gets it. This is called *brittleness* of the decision and is the opposite term to *robustness*. The subtle changes in the mistaken images above are done on purpose. They are skilfully crafted into the image to make the DL's decision flip.



The problems of DL do not stop with images. Last year, I was at a conference in Bilbao, Spain. I wanted to check the weather for the afternoon, and voiced the question to my phone: “What is the weather like in Bilbao today?” Check out what the DL NN understood! Hilarious! ‘Bill’s bowel’ – ‘Bilbao’? Really?

But imagine that fooling DLs is not done just for the fun of it. Take a DL that is used to detect financial fraud. A malicious attack on the data submitted to the DL may rob you of thousands

of pounds. There is a whole area of research called *Adversarial Machine Learning* which looks into this problem.

8.3 Structure and operation of DLs

There are many varieties of DL depending on the task being solved. For image classification, for example, the first layers of a DL are structured in a way to resemble the functioning of the visual cortex in response to a specific stimulus. For audio processing, image segmentation, retail analysis, DLs will have not only bespoke architectures but also tailor-made training algorithms.

One of the most advanced and publicly visible application of DL is for image classification. The challenges of image classification are beautifully summarised in a collage I found while surfing the Internet⁷. I pinched the idea and created one of my own (see Figure 8.5).

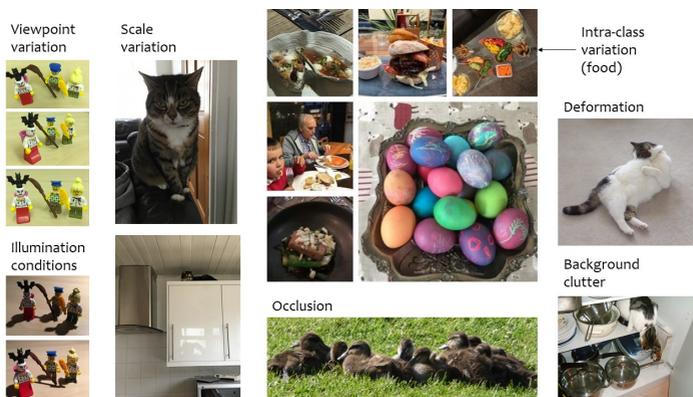


Figure 8.5: Challenges of image classification.

⁷<http://cs231n.github.io/classification/>

DL works a treat for this type of data with all its challenges. The specific type of DL for this task are *Convolutional Neural Networks* (CNNs / ConvNets). ConvNet architectures make the explicit assumption that the inputs are images, which allows for simplifying the network compared to a fully connected version.

CNNs make use of several different types of neurons which we have not seen during our travels thus far. The neurons at the first layer (called *convolutional layer*) take a 3-dimensional input. This input is a *block* from the input image as illustrated in Figure 8.6.

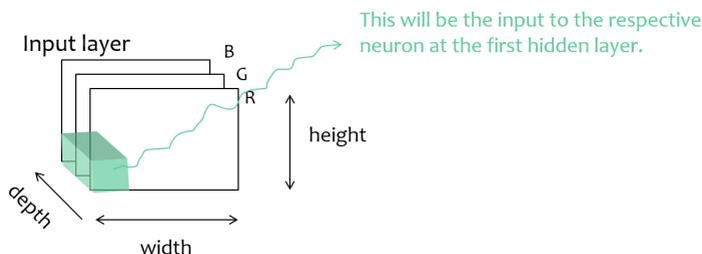
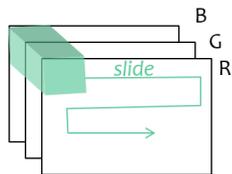


Figure 8.6: Illustration of the input to the neurons at the first CNN layer.

The size of the input block is $a \times b$ (pixels) $\times 3$ (colours). Here a and b are hyper-parameters of the algorithm. (Keep an eye on the number of parameters that we need to set up in order to train our CNN.) The number of weights for this neuron will be $3ab + 1$, where 1 is added for the bias weight. In image processing parlance, we can call this neuron a *filter*.



The question is now, which block of the image do we submit to which neuron at the first layer? CNN takes advantage of the fact that the input is an image, and not a random collection of blocks of the desired size.

Thus, we slide the block across the whole image with step s pixels, where s is called the “stride”, and is another hyper-parameter (three so far). One neuron at the first layer will take one block as its input. This action will produce an image of the same size as the input with new pixel values calculated through the filter; this operation is called *convolution*.

What happens to the corner and edge pixels? We pad the sides with zeros. The size of the padding is another hyper-parameter (making four now). Usually it is chosen so that the output of the convolution (after sliding the block across the whole image) is of the same size as the initial image. For example, if we have a $5 \times 5 \times 3$ block, it will stick out of the image by 2 pixels, and they will be assigned value 0 (black).

As we slide *the same* filter across the whole image, the number of tunable parameters for generating the new image will be $3ab + 1$.

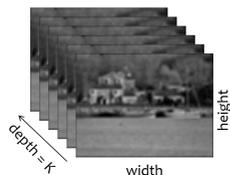
⊕ ⊕ ⊕ **Example 8.3.1**

Let’s see how a convolution layer will operate on the lovely image of Menai Straight from Figure 7.11. Figure 8.7 shows the original image of size $615 \times 820 \times 3$ and the results of a convolution pass.



Figure 8.7: An example of the result from a convolutional pass on an image with an averaging filter.

Consider a block of size $20 \times 20 \times 3$ with an averaging filter which has identity output and zero bias weight. In other words, all 1200 weights will be equal to $1/1200$, and the net sum will be the output of the neuron. Notice the slightly darker edges. This is because of the zero-padding (black). $\ominus \ominus \ominus$



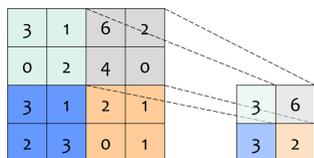
Now take K such filters with different weights. They form the first hidden layer of the CNN, called the “convolution layer” (CONV). The number K is called *depth*, and it is up to us to pick it (the fifth hyperparameter).

So, this convolution layer will consist of K stacked filter outputs of size $\text{width} \times \text{height} \times K$, and will serve as the input to the next layer. The interesting thing about the CONV layer is that, depending on the chosen filters, it may discover interesting (small) colour, texture and orientation patterns in the original image. The further layers are meant to use these patterns in forming higher-level concepts.

Recall that the neurons in each depth slice (filter) use the same weights and bias. This is called *parameter sharing*. Parameter sharing is a way to ensure that the CNN training is manageable.

And the story goes on. CONV is typically followed by a RELU layer. RELU returns the same structure as CONV but all negative outputs are set to zero.

Following RELU, we typically have a POOL layer which reduces the pixel dimensionality.



In the max-pool implementation, to reduce the width and the height of the image by a factor of 2, we replace every 2×2 square of each depth slice by the maximum number among the 4 numbers in the square as shown on the left.

Note that neither RELU nor POOL have trainable weights; they both perform fixed functions.

A final fully connected layer (FC) completes the CNN. This layer is the traditional NN layer where all neurons at the layer before are connected with all neurons at the subsequent layer. In classification, FC has c neurons, one for each class. The tag of the neuron with maximum activation for a given input image \mathbf{x} will determine the label for that image. A basic CNN configuration is shown in Figure 8.8.

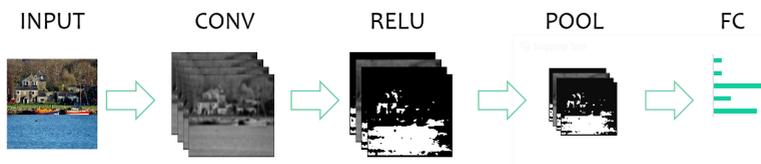


Figure 8.8: A basic CNN configuration.

⊕ ⊕ ⊕ Example 8.3.2

Consider the CNN configuration in Figure 8.8 for solving a 5-class problem. Assume that the input image is of size $32 \times 32 \times 3$, the block sizes are $a = 5$, $b = 5$, there are $K = 12$ depth slices in CONV, and POOL reduces the width and the height of the image by a factor of 2. Calculate the number of trainable parameters of this CNN.

Solution. Each of the 12 depth slices in CONV will need $5 \times 5 \times 3 + 1 = 76$ weights. Then CONV requires $76 \times 12 = 912$ weights. Neither RELU nor POOL will add to this count. After POOL, the output size will be $12 \times 16 \times 16 = 3072$. There will be 5 output neurons in FC because there are 5 classes. Each of the 3,072 neurons at POOL will be connected to the 5 output neurons (plus a bias weight, giving a total of $(5+1) \times 3,072 = 18,432$ weights from POOL to FC. Then the overall total number of tunable parameters is $912 + 18,432 = 19,374$.
 $\ominus \ominus \ominus$

In reality, CNN are a lot more complex, for example:

$INPUT - \underbrace{C - R - C - R - C - R - P}_{\text{repeat}} \dots C - R - C - R - C - R - P - FC$

Training of a CNN is no mean feat. The configuration will dictate the suitable training methods and procedures. Most such procedures are based on the error backpropagation algorithm which we studied already in Section 7.1.3. Getting the training right is an art that can be learned with a lot of practice. The problem is that, with data size in order of millions, and with thousands of trainable parameters, training a single instance of CNN takes weeks even with the best that modern technology can offer (computer clusters, GPUs, etc.). Gaining experience in training CNN, especially starting from scratch, may not be on the cards for all of us. Yet, if we are faced with a problem similar to one already approached by a CNN, we can use a pre-trained CNN! Bring it on! For example, if our classes of objects we want to recognise in images are in the list of classes of a pre-trained CNN for object recognition, we can run our images through the CNN and restrict the output to classes of interest only.

8.4 Which CNN is the best?

Of course, there isn't any one CNN that dominates the rest of them. Different problems call for very different CNN structures and training. But there have been frequent challenges on image classification. Table 8.1 contains some data about the most popular CNNs to date.

Table 8.1: Popular CNNs

Year	Name	Details
1990	LeNet	The first successful application of CNN due to Yann LeCun
2012	AlexNet	The first work that popularised CNN in computer vision. Won the 2012 ImageNet challenge ILSVRC.
2013	ZF Net	Due to Zeiler and Fergus. Won the 2013 ImageNet challenge ILSVRC. Improvement over AlexNet – expanded the size.
2014	GoogLeNet	Due to Szegedy et al. from Google. Inception-v4 Won the 2014 ImageNet challenge ILSVRC. Developed ‘Inception’ module. Reduced the total number of weights of the CNN.
2014	VGG Net	Due to Simonyan and Zisserman. Runner-up in the 2014 ImageNet challenge ILSVRC. Showed that the depth of the network is critical. Back to large number of parameters. Pre-trained models are available.
2015	ResNet	Residual Network. Due to He et al. Won the 2015 ImageNet challenge ILSVRC. The top choice as of May 2016.

Will DL go out of fashion soon? Who knows? Like every other area of human knowledge, DL will find their happy equilibrium between demand and supply. Some day.

Conclusion

Disclaimer: All images in this book are either my own, or sourced from Google under the licence 'Labelled for reuse'.

Disclaimer out of the way, remember the 'pattern recognition cycle'? It starts with Real World and ends with Real World as shown in Figure 1.6. The user comes to you with their data and their questions. Now that you have gone through this module, you should be a lot more knowledgeable about the possible spells and magic potions to help you bring a solution back to the user.

Imagine this is the last lecture of the semester. What do you think I do at the last lecture (pick one)?

- a.) Prepare a revision of the material.
- b.) Give the class a mock exam (better still, give them the real exam questions).
- c.) Other.

The correct answer is c.) Other. This last lecture is the last time I see the class before they graduate (however few troopers come to that lecture, just a few weeks before Christmas). I am adamant that all students who graduate from university must know the international student anthem. So, yes, I ask them to sing. Check

it! Ask my former students. I stand them up, show them a few YouTube renditions of *Gaudeamus igitur* (from New Zealand, Peru, Romania, Russia, USA, Spain, Indonesia Italy, Mexico, Norway and Poland - I just pick 3-4 of those to play) only to discover that UK students don't have a clue what this is! Then I display the lyrics on the screen and ask them to sing along with an endearing group of Finish graduates.⁸ Ah, yes, and I threaten the class that I will fail anybody who does not sing. (I am only human, I have to have fun too!) I love this! Celebration of life while it lasts. Actually, I can't claim novelty or originality for asking the class to sing *Gaudeamus igitur*. My friend Ina Slavova did this, maybe 30 odd years ago as a young lecturer in computer science at the New Bulgarian University in Sofia. I laughed my head off when she told me.

Now I will leave you with the lyrics (only two of the verses!) and will hope you can sing it too:

Gaudeamus igitur
Iuvenes dum sumus.
Post iucundam iuventutem
Post molestam senectutem
Nos habebit humus.

Vivat academia!
Vivant professores!
Vivat membrum quodlibet;
Vivant membra quaelibet;
Semper sint in flore.

⁸https://www.youtube.com/watch?v=Aa_5aRxqi8Q

Bibliography

- [1] K. Bache and M. Lichman. UCI machine learning repository. <http://archive.ics.uci.edu/ml>, 2013. University of California, Irvine, School of Information and Computer Sciences.
- [2] Anthony J. Bagnall, Aaron Bostrom, Gavin C. Cawley, Michael Flynn, James Large, and Jason Lines. Is Rotation Forest the best classifier for problems with continuous features? *CoRR*, abs/1809.06705, 2018.
- [3] J. C. Bezdek, J. M. Keller, R. Krishnapuram, L. I. Kuncheva, and N. R. Pal. Will the real iris data please stand up? *IEEE Transactions on Fuzzy Systems*, 7(3):368–369, 1999.
- [4] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.
- [5] Corinna Cortes and Vladimir N. Vapnik. Support-vector networks. *Machine Learning.*, 20(3):273–297, 1995.
- [6] Jia Deng, Wei Dong, Richard Socher, Li jia Li, Kai Li, and Li Fei-fei. Imagenet: A large-scale hierarchical image database. In *In Proceedings of the International Conference of Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

- [7] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley, 2001.
- [8] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, 15:3133–3181, 2014.
- [9] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.
- [10] David J. Hand. Measuring classifier performance: a coherent alternative to the area under the ROC curve. *Machine Learning*, 77(1):103–123, 2009.
- [11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.
- [12] L. I. Kuncheva. *Combining Pattern Classifiers. Methods and Algorithms*. Wiley, 2nd edition, 2014.
- [13] Geogre Lakoff. *Women, Fire and Dangerous Things*. The University of Chicago Press, 1987.
- [14] Gordon D. Murray. A cautionary note on selection of variables in discriminant analysis. *Journal of the Royal Statistical Society. Series C*, 26(3):246–250, 1977.
- [15] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. *CoRR*, abs/1412.1897, 2014.
- [16] J. J. Rodríguez, L. I. Kuncheva, and C. J. Alonso. Rotation forest: A new classifier ensemble method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1619–1630, Oct 2006.

- [17] Jürgen Schmidhuber. Deep learning in neural networks: An overview. 2014. Technical Report IDSIA-03-14.
- [18] E. H. Shortliffe and B. G. Buchanan. A model of inexact reasoning in medicine. *Mathematical Biosciences*, 23(3-4):351—379, 1975.
- [19] Christian Szegedy, Google Inc, Wojciech Zaremba, Ilya Sutskever, Google Inc, Joan Bruna, Dumitru Erhan, Google Inc, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. 2014. arXiv:1312.6199 [cs.CV].
- [20] Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, May 2004.
- [21] Andrew F. Weller, Anthony J. Harris, and J. Andrew Ware. Artificial neural networks as potential classification tools for dinoflagellate cyst images: A case using the self-organizing map clustering algorithm. *Review of Palaeobotany and Palynology*, 141:287—302, 2006.
- [22] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *ECCV*, 2016.

Appendix A

Maths you should know

What do you need to remember from your school maths?

1. Equation of a line in 2d

$$ax + by + c = 0.$$

2. Equation of a plane in 3d

$$ax + by + cz + d = 0.$$

3. Deriving the equation of a line in 2d from the coordinates of 2 points $A(x_1, y_1)$ and $B(x_2, y_2)$.

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1}.$$

Example: $A(3, -2)$ and $B(1, -1)$.

$$\frac{x - 3}{1 - 3} = \frac{y - (-2)}{-1 - (-2)}, \quad \frac{x - 3}{-2} = \frac{y + 2}{1}$$

$$x - 3 = -2y - 4,$$

and, finally

$$x + 2y + 1 = 0.$$

4. Plotting a line in 2d in your notes. Pick two points on the line, plot them on paper, and connect them using a straight edge. For example, plot the line given by the equation

$$6x - 2y + 8 = 0.$$

Pick $x = 0$. Calculate $y = -8/(-2) = 4$. First point is $P(0, 4)$. Pick $y = 0$. Calculate $x = -8/6 = -1.33$. Second point is $Q(-1.33, 0)$. (See Figure A.1.)

5. Equation of a circle

$$(x - c_x)^2 + (y - c_y)^2 = r^2,$$

where the centre is at $C(c_x, c_y)$, and the radius is r , and equation of a sphere

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2,$$

where the centre is at $C(c_x, c_y, c_z)$, and the radius is again r .

6. How the equation of line/plane is used to find out whether two points lie on the same side of the line/plane.

Substitute the coordinates of the two points in the left-hand-side of the equation of the line/plane. If the signs of the sums are the same, the two points lie on the same side of the line/plane.

7. Euclidean distance in 2d, 3d and n -d. Consider two points in \mathbb{R}^n : $X(x_1, x_2, \dots, x_n)$ and $Y(y_1, y_2, \dots, y_n)$. The Euclidean distance is

$$d(X, Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}.$$

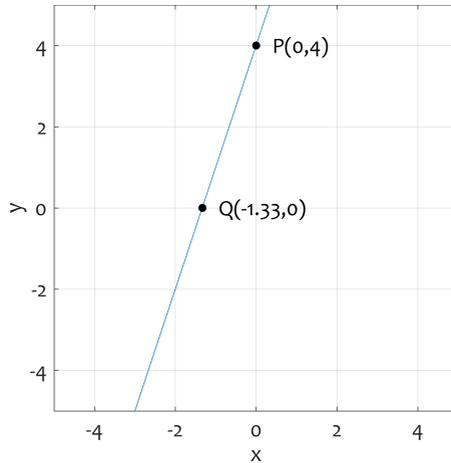


Figure A.1: Example of plotting a line given by its equation.

8. How we find the equation of a line which passes through a given point $A(a_1, a_2)$ and is orthogonal to vector $\mathbf{v} = [v_1, v_2]^T$?

Start by constructing an equation of a line using the components of \mathbf{v} as the coefficients

$$v_1x + v_2y + c = 0.$$

This way we guarantee that the line is orthogonal to \mathbf{v} . Second, point A must satisfy the equation because A lies on the line. Therefore

$$v_1a_1 + v_2a_2 + c = 0,$$

and

$$c = -v_1a_1 - v_2a_2.$$

The equation of the line is

$$v_1x + v_2y - v_1a_1 - v_2a_2 = 0.$$

Example: Find the equation of a line which passes through a given point $P(6, -4)$ and is orthogonal to vector $\mathbf{v} = [-2, 10]^T$?

$$-2x + 10y + 12 + 40 = 0$$

$$x - 5y - 26 = 0.$$

Check with P : $6 - 5(-4) - 26 = 6 + 20 - 26 = 0$.

Index

- 1nn, 90, 92
- adversarial machine learning, 209
- attribute, 18
- bagging, 105
- Bayes, 12, 60
- Bayes classifier, 60
- Bayes theorem, 12
- binary classification, 17, 32
- bootstrap sample, 105
- centroid linkage, 136
- circle, 224
- class, 16
 - positive, negative, 47
- class labels, 18
- classification region, 28
- classifier, 27, 63
 - decision tree, 97, 122
 - training, 98
 - ensemble, 104
 - if-then, 82
 - largest prior, 84
 - largest prior, 54
 - majority, 84
 - nearest mean (NMC), 64
 - nearest neighbour (1-nn), 90
 - OneR, 85
 - rule-based, 82
 - support vector machine (SVM), 101
 - ZeroR, 84
- classifier ensembles, 104
 - AdaBoost, 107
 - bagging, 105
 - base classifier, 105
 - boosting, 107
 - homogeneous and heterogeneous, 105
 - random forest, 110
 - random subspace, 108
 - unstable classifiers, 105
- cluster, 125
- cluster analysis, 125
- clustering, 125
 - agglomerative, 128

- hierarchical, 128
- k-means, 127, 143
- mean (centroid) linkage, 136
- non-hierarchical, 142
- single linkage, 127, 128
- single linkage, chain effect, 136
- confusion matrix, 41, 43
- convolutional neural networks, 210
 - CONV layer, 212, 213
 - parameter sharing, 212
 - POOL layer, 212
 - RELU layer, 212
- covariance matrix, 77
- cross-validation, 40
- cross-validation
 - fold, 40
- crowdsourcing, 23
- data
 - imbalanced, unbalanced, 54
- data set, 20
 - big, 23
 - iris, 22
 - labelled, 20
 - UCI repository, 22
 - wide, 23, 118
- decision tree, 97
 - decision stump, 96
 - intermediate nodes, 97
 - leaves, 97
 - random tree, 110
 - root, 97
- deep learning, 201
- dimensionality reduction, 115
- discriminant function, 27, 29
- distance
 - Euclidean, 91
 - Hamming, 91
 - Manhattan, city-block, 92
 - nearest neighbour, 129
- epoch, 177, 192
- error rate, 43
- Euclidean distance, 64, 91, 224
- F measure, 57
- false negative, 47
- false positive, 47
- feature, 18, 113
 - redundant, irrelevant, 114
- feature selection, 113
 - diagram, 115
 - exhaustive search, 119
 - filter approach, 122
 - sequential forward selection (SFS), 120
 - wrapper approach, 122
- feature space, 19
- feedforward neural networks, 174
- generalisation, 41
- GM measure, 57
- grand truth, 17

- if-then classifier, 82
- imbalanced data, 54
- k-means, 127, 143
- k-nearest neighbour classifier (k-nn), 90
 - algorithm, 92
- KAGGLE, 13
- knn, 90, 92
 - MATLAB, 93
- leave-one-out (LOO), 40
- line in 2D, 33, 223
- loss matrix, 56
- Majority classifier, 84
- MATLAB, 34, 45, 71, 93
 - ROC curve, 52
- matrix
 - loss, 56
- mean linkage, 136
- MNIST dataset, 17
- multi-label classification, 17
- multiclass classification, 17
- mutually exclusive, 17
- nearest mean classifier (NMC), 64
- nearest neighbour classifier (1-nn), 90
- NETFLIX, 12
- neural networks, 155
 - activation function, 161
 - backpropagation, 157
 - bias input, 159
 - bias weight, 159
 - biological neuron, 159
 - deep learning, 157, 201
 - convolution, 211
 - convolutional neural network (CNN), 210
 - epoch, 177
 - error backpropagation algorithm, 175
 - feedforward, 174
 - Hebbian learning, 190
 - learning rate, 178
 - learning vector quantisation (LVQ), 197
 - MLP, 171
 - momentum, 178
 - multi-layer-perceptron (MLP), 171
 - net sum, 160
 - neuron, 159
 - firing, 161
 - perceptron, 156, 165
 - radial basis functions (RBF), 179
 - RBF, 171, 179
 - sigmoid activation function, 162
 - SOM, 171
 - activation pattern, 191
 - threshold logic unit (TLU),

- 163
 - vector quantisation (VQ), 197
- neuron, 159
- object, 18, 19
- OneR classifier, 85
 - training, 85
- overfitting, 25, 36
- overtraining, 36
- pattern recognition cycle, 25, 217
- perceptron, 156, 165
 - convergence theorem, 168
 - training algorithm, 165
- plane, 223
- principal component analysis (PCA), 116
- probabilities
 - class-conditional, 60
 - posterior, 60
 - prior, 60
- probability
 - unconditional, 60
- RBF, 179
 - training, 183
- recall and precision, 57
- reinforcement learning, 156
- ROC curve, 49
 - area under (AUC), 53
 - operational point, 50
- ROC curves, 47
- rule-based classifiers, 82
- sensitivity, 48
- SFS (see feature selection), 120
- sigmoid activation function, 162
- single linkage, 127, 128
 - chain effect, 136
- specificity, 48
- sphere, 224
- supervised pattern recognition, 25
- SVM, 101
- threshold logic unit (TLU), 163
- training and testing, 39
 - cross-validation, 40
 - data shuffle, 39
 - hold-out (H-method), 39
 - leave-one-out, 40
 - resubstitution (R-method), 39
- training and testing protocols, 37
- true negative, 47
- true positive, 47
- UCI repository, 22
- unsupervised learning, 125
- unsupervised pattern recognition, 25, 125
- Voronoi cell, 71
- Voronoi diagrams, 71
- ZeroR classifier, 84