

Paul Fieguth

An Introduction to Pattern Recognition and Machine Learning



Springer

An Introduction to Pattern Recognition and Machine Learning

Paul Fieguth

An Introduction to Pattern Recognition and Machine Learning



Springer

Paul Fieguth
Department of Systems Design Engineering,
Faculty of Engineering
University of Waterloo
Waterloo, ON, Canada

ISBN 978-3-030-95993-7 ISBN 978-3-030-95995-1 (eBook)
<https://doi.org/10.1007/978-3-030-95995-1>

© Springer Nature Switzerland AG 2022

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

In 1996, I had the good fortune of being hired by the Department of Systems Design Engineering at the University of Waterloo. As a department which emphasizes systems theory, design methodology, and out-of-the-box thinking, it was a natural home for interdisciplinary topics such as pattern recognition, and I had the opportunity to teach pattern recognition at both the undergraduate and graduate levels.

At the time, pattern recognition and machine learning were something of a niche area, not offered in many programs of study, and if at all, then at the graduate/research level, making systems design engineering all the more unique in having offered a pattern recognition course at the undergraduate level since the 1970s. Today, of course, these fields have exploded to become one of the hottest areas of study and research, with students from nearly every field clamouring to enrol in such courses and with researchers in nearly every line of work seeking ways in which machine learning, more broadly, or deep learning, more specifically, might be applied to their domain.

This explosion in popularity is not without reason. There are two key changes which, among many others, have changed the field:

1. Pattern recognition has become much more capable: deep/convolutional neural networks have allowed high-level, highly abstract, and nonlinear problems to be solved, opening up vast domains of data processing which, only a few years earlier, were not even remotely under consideration.
2. Sensing and Measurement have become far more ubiquitous: the billions of cell-phones, cars, and Internet-of-Things devices have created a networked world of sensors, of data, and of opportunities to make inferences from such data.

Although there are a great many online tutorials and how-to guides for deep learning, to really understand the process of classification and inference data is quite nuanced. The goal of this book is to explore that nuance and associated insights, and hopefully to provide some balance and context to the breathless excitement and hype/exaggeration which is commonly encountered today in any concept associated with artificial intelligence.

A number of people need thanking in making it possible for me to undertake a project of this scope.¹ First and foremost, a great deal of love and thanks to my wife, Betty, who is an ardent supporter of all of my projects, both academic and non-academic, and whether they make any sense or not.

The text would never have come into being without Professor Ed Jernigan, the department chair who hired me into systems design, who had the foresight to create an undergraduate pattern recognition course *long* before that was a normal thing to do, and who warmly included me as a colleague in his research lab.

The text would probably also not have come into being without the enthusiastic and tirelessly energetic Professor Alex Wong, who was a very strong supporter of an undergraduate text. He had extensive suggestions and provided many ideas and innovations from his own years in teaching pattern recognition.

My appreciation for the patience of my chair (Professor Lisa Aultmann-Hall) and of my dean (Professor Mary Wells), in having a colleague who was somewhat distracted, or at least whose time was somewhat compromised, particularly throughout the 2020–2021 Covid-19 period. Thanks also to Anya Fieguth for allowing her artwork ([page 355](#), [page 378](#)) to appear.

It is, as an author, very easy to reach a point of cognitive dissonance, whereby one is convinced of the clarity or elegance of an exposition, making it critical to have *other* eyes provide honest feedback. My thanks to Alex Wong and Eddie Park for their feedback, but most particularly to Nicholas Pellegrino who provided an amazingly extensive, careful, and detailed reading.

¹ Producing all of the figures in this text required the development of 15,933 lines of code, with hindsight a perhaps ill-advised use of the author's spare time.

Finally, I would like to dedicate this text to two close colleagues who will sorely be missed:

In memory of Igor (Nov 27, 2020), who was a passionate, inspired teacher, and truly demonstrated a love and commitment to all of his students.

In memory of Pearl (Nov 28, 2020), who was a close colleague for 10 years, and had a generous heart and boundless energy.

*Paul Fieguth
Waterloo, ON, Canada*

Contents

Preface	v
Table of Contents	ix
List of Examples	xv
List of Algorithms	xvii
Notation	xix
1 Overview	1
2 Introduction to Pattern Recognition	5
2.1 What Is Pattern Recognition?	5
2.2 Measured Patterns	9
2.3 Classes	11
2.4 Classification	14
2.5 Types of Classification Problems	16
Case Study 2: Biometrics	19
Numerical Lab 2: The Iris Dataset	23
Further Reading	27
Sample Problems	27
References	28
3 Learning	29
Case Study 3: The Netflix Prize	44
Numerical Lab 3: Overfitting and Underfitting	46
Summary	50
Further Reading	51
Sample Problems	51
References	53

4 Representing Patterns	55
4.1 Similarity	55
4.2 Class Shape	57
4.3 Cluster Synthesis	73
Case Study 4: Defect Detection	74
Numerical Lab 4: Working with Random Numbers	76
Further Reading	79
Sample Problems	79
References	82
5 Feature Extraction and Selection	83
5.1 Fundamentals of Feature Extraction	83
5.2 Feature Extraction and Selection	93
Case Study 5: Image Searching	103
Numerical Lab 5: Extracting Features and Plotting Classes	104
Further Reading	108
Sample Problems	110
References	114
6 Distance-Based Classification	117
6.1 Definitions of Distance	118
6.2 Class Prototype	124
6.3 Distance-Based Classification	132
6.4 Classifier Variations	134
Case Study 6: Hand-writing Recognition	138
Numerical Lab 6: Distance-Based Classifiers	141
Further Reading	143
Sample Problems	144
References	150
7 Inferring Class Models	151
7.1 Parametric Estimation	152
7.2 Parametric Model Learning	154
7.3 Nonparametric Model Learning	164
7.3.1 Histogram Estimation	165
7.3.2 Kernel-Based Estimation	168
7.3.3 Neighbourhood-based Estimation	172
7.4 Distribution Assessment	174
Case Study 7: Object Recognition	179
Numerical Lab 7: Parametric and Nonparametric Estimation	180

Contents	xi
Further Reading	183
Sample Problems	184
References	191
8 Statistics-Based Classification	193
8.1 Non-Bayesian Classification: Maximum Likelihood	194
8.2 Bayesian Classification: Maximum a Posteriori	198
8.3 Statistical Classification for Normal Distributions	201
8.4 Classification Error	204
8.5 Other Statistical Classifiers	211
Case Study 8: Medical Assessments	213
Numerical Lab 8: Statistical and Distance-Based Classifiers	218
Further Reading	220
Sample Problems	221
References	230
9 Classifier Testing and Validation	231
9.1 Working with Data	231
9.2 Classifier Evaluation	239
9.3 Classifier Validation	249
Case Study 9: Autonomous Vehicles	255
Numerical Lab 9: Leave-One-Out Validation	257
Further Reading	260
Sample Problems	260
References	265
10 Discriminant-Based Classification	267
10.1 Linear Discriminants	269
10.2 Discriminant Model Learning	271
10.3 Nonlinear Discriminants	280
10.4 Multi-Class Problems	285
Case Study 10: Digital Communications	288
Numerical Lab 10: Discriminants	291
Further Reading	294
Sample Problems	294
References	298
11 Ensemble Classification	299
11.1 Combining Classifiers	301
11.2 Resampling Strategies	305
11.3 Sequential Strategies	312

11.4 Nonlinear Strategies	319
11.4.1 Neural Network Learning	320
11.4.2 Deep Neural Network Classifiers	325
Case Study 11: Interpretability and Ethics of Large Networks	332
Numerical Lab 11: Ensemble Classifiers	336
Further Reading	338
Sample Problems	339
References	344
12 Model-Free Classification	347
12.1 Unsupervised Learning	348
12.1.1 K-Means Clustering	351
12.1.2 Kernel K-Means Clustering	358
12.1.3 Mean-Shift Clustering	363
12.1.4 Hierarchical Clustering	365
12.2 Network-Based Clustering	370
12.3 Semi-Supervised Learning	373
Case Study 12: Ancient Text Analysis: Who Wrote What?	376
Numerical Lab 12: Clustering	378
Further Reading	381
Sample Problems	381
References	387
13 Conclusions and Directions	389
Appendices	395
A Algebra Review	397
Further Reading	404
Sample Problems	405
References	406
B Random Variables and Random Vectors	407
B.1 Random Variables	407
B.2 Expectations	409
B.3 Conditional Statistics	410
B.4 Random Vectors and Covariances	411
B.5 Outliers and Heavy-Tail Distributions	416
B.6 Sample Statistics	420
Further Reading	422
Sample Problems	422
References	424

Contents	xiii
C Introduction to Optimization	427
C.1 Basic Principles	427
C.2 One-Dimensional Optimization	428
C.3 Multi-Dimensional Optimization	431
C.4 Multi-Objective Optimization	434
Further Reading	435
Sample Problems	436
References	436
D Mathematical Derivations	437
Index	467

List of Examples

Case Study 2	Biometrics	19
Case Study 3	The Netflix Prize	44
Case Study 4	Defect Detection	74
Case Study 5	Image Searching	103
Case Study 6	Hand-writing Recognition	138
Case Study 7	Object Recognition	179
Case Study 8	Medical Assessments	213
Case Study 9	Autonomous Vehicles	255
Case Study 10	Digital Communications	288
Case Study 11	Interpretability and Ethics of Large Networks	332
Case Study 12	Ancient Text Analysis: Who Wrote What?	376
Example 2.1	Pattern Recognition of Text	7
Example 2.2	Pattern Recognition of the Mind	8
Example 2.3	What Is a Class?	12
Example 2.4	The Types of Pattern Recognition Problems	20
Example 3.1	Robustness in Learning	35
Example 3.2	Regression and Classification	36
Example 3.3	An Overview of the Use of Data in Learning	39
Example 3.4	Wrapper-Based Learning	43
Example 4.1	Face Recognition	58
Example 4.2	How to Form an Ellipsoid	63
Example 4.3	Quadratic Forms	65
Example 4.4	Ellipsoids and Real Data	67
Example 4.5	Class Data, Covariances, Eigendecompositions, and Ellipses	70
Example 6.1	Euclidean Distance and Measurement Units	121
Example 7.1	Learning of Mixture Models	161
Example 7.2	Nonlinear Maximum Likelihood	162

Example 8.1	Are we Classifying or Estimating?	195
Example 8.2	Classifier Probability of Error	207
Example 8.3	Classifier Probability of Error for Gaussian Statistics . .	210
Example 9.1	Data Augmentation	233
Example 9.2	Receiver Operating Characteristics Examples	246
Example 10.1	Mean-Squared Error Discriminant	276
Example 10.2	Radial Basis Functions	284
Example 11.1	Bootstrap and Estimation	306
Example 11.2	Biological and Artificial Neurons	321
Example 11.3	Illusions in Machine Learning	333
Example 12.1	Vector Quantization	355
Example 12.2	Bag-of-Words and Visual Representation	359
Example B.1	Power Laws and Infinite Variance	418

List of Algorithms

Numerical Lab 2	The Iris Dataset	23
Numerical Lab 3	Overfitting and Underfitting	46
Numerical Lab 4	Working with Random Numbers	76
Numerical Lab 5	Extracting Features and Plotting Classes	104
Numerical Lab 6	Distance-Based Classifiers	141
Numerical Lab 7	Parametric and Nonparametric Estimation	180
Numerical Lab 8	Statistical and Distance-Based Classifiers	218
Numerical Lab 9	Leave-One-Out Validation	257
Numerical Lab 10	Discriminants	291
Numerical Lab 11	Ensemble Classifiers	336
Numerical Lab 12	Clustering	378
Algorithm 6.1	Prototype-Distance Classification	130
Algorithm 7.1	Gaussian Mixture Models	163
Algorithm 9.1	Leave-One-Out Validation: Regression	252
Algorithm 9.2	Leave-One-Out Validation: Classification	253
Algorithm 11.1	AdaBoost – Adaptive Boosting	310
Algorithm 12.1	K-Means Clustering	354

Notation

To keep the text as accessible as possible, the text tries to maintain a relatively simple and consistent mathematical notation.

First and foremost, we will regularly need to distinguish between scalars, vectors, and matrices:

x	A <i>lower case</i> variable: a scalar, a single value
\underline{x}	An <i>underlined</i> variable: a column vector
X	An <i>upper case</i> variable: a matrix
\mathcal{X}	A <i>calligraphic</i> variable: an abstracted data type, normally a space or a set.

The above rules continue to apply when we are talking about parts of vectors or matrices:

x_i	A scalar, the i th element of vector \underline{x}
\underline{x}_i	A column vector, the i th vector in some sequence
x_{ij}	A scalar, an entry in matrix X
	May also rarely, for notational clarity, be shown as X_{ij}

and, although not occurring frequently, to constants as well:

0	The scalar value of zero
$\underline{0}$	A column vector of zeros (a zero row vector would be $\underline{0}^T$)
$\mathbf{0}$	A matrix of zeros (may be square, rectangular, or triangular)
	In any matrix shown in the text, blank areas are understood to be zero.

A few pieces of notation with regards to matrices:

A^{-1}	The inverse of matrix A
$\det(A)$	The determinant of matrix A
A^T	The transpose of matrix A
\underline{x}^T	A row vector, the transpose of column vector \underline{x}
$(\ast)(\dots)^T$	The same as $(\ast)(\ast)^T$ for any expression \ast ; a convenient shorthand, particularly when expression \ast is long or complicated

The varied roles of absolute-value delimiters:

$ a $, $ \underline{w} $	The magnitude of scalar a , the length of vector \underline{w}
$ A $	The determinant of matrix A , although will normally prefer $\det(A)$
$ \mathcal{H} $	The number of elements in set \mathcal{H}

More specific to the pattern recognition context, first we have classes

C	A class , a category of pattern
K	The number of classes
\mathcal{C}	A set of K classes , $\mathcal{C} = \{C_1, C_2, \dots, C_K\}$

We collect information of each pattern in the form of measurements and features:

y	A measurement vector in m -dimensional space, $y \in \mathbb{R}^m$
\underline{x}	A feature vector in n -dimensional space, $\underline{x} \in \mathbb{R}^n$
$\underline{\theta}$	A vector of q parameters , somehow describing the shape of a class

In describing notions of patterns and classifiers, we will frequently need notions of *similarity*, which will be based on some sense of distance or probability:

$d(\underline{x}_1, \underline{x}_2)$	Distance between features points \underline{x}_1 and \underline{x}_2
$d(\underline{x}, C)$	Distance from a feature point \underline{x} to class C
$p()$	A probability density
$\mathbf{P}()$	A probability
$p(\underline{x} C)$	The probability density in feature space \underline{x} of class C
$\mathbf{P}(C \underline{x})$	The probability of class C given feature \underline{x}
$\mathbb{E}[]$	Statistical expectation

We will use \sim to mean “is described by”, thus

$C \sim \theta$	Class C is described by parameter vector θ
$C \sim (\mu, \Sigma)$	Class C is described by an ellipsoid of shape Σ centered on \underline{x}
$C_{\kappa x} \sim \dots$	Class C_κ is described in feature domain x as ...
$C \sim p(\underline{x})$	Class C is described by probability distribution $p()$
$\underline{x} \sim \Sigma$	Feature vector \underline{x} has covariance Σ
$\underline{x} \sim (\mu, \Sigma)$	Feature vector \underline{x} has mean μ and covariance Σ
$\underline{x} \sim \mathcal{N}(\mu, \Sigma)$	Feature vector \underline{x} is Gaussian (normal) with mean μ and covariance Σ
$\mathcal{N}(\underline{x}; \mu, \Sigma)$	The value of the Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ evaluated at \underline{x}

Finally, we will need to describe **minimizing** over a set or a function,

$\min(\{a_i\})$	The minimum over a set of values
$\min_{\theta} f(\theta)$	The minimum value of function $f()$
$\arg_{\theta} \min f(\theta)$	The value of θ which minimizes function $f()$

A brief alphabetic list of the notation which is used persistently throughout the book. Where appropriate, links are provided to the corresponding location where concepts are discussed in the text; the **Index** can offer further concept locations.

b	Estimator bias	\mathcal{B}	Basin of attraction
c	Class data point or value	\mathcal{C}	Set of classes
$d()$	Distance function	\mathcal{D}	Dataset
e	Classification error	\mathbb{E}	Expectation
$f()$	Feature function	\mathcal{L}	Loss function
$g()$	Classification function	\mathbb{L}	Neural network layer
$h()$	Linear discriminant function	\mathcal{N}	Normal (Gaussian) distribution
i	Data point index	\mathbb{N}	Neuron
j	Eigenvalue or dimension index	$\mathcal{O}()$	Computational complexity
k	Number of samples (e.g., k NN)	\mathcal{R}	Classification region
m	Measurement dimensionality (\mathbb{R}^m)	\mathbb{R}	Real value
n	Feature dimensionality (\mathbb{R}^n)	\mathcal{X}	Feature space
p	Probability density	\mathcal{Y}	Measurement space
q	Parameter dimensionality, Degrees of freedom	α	χ^2 confidence
		$\delta()$	Delta function

s	Kernel scaling	ϵ	Small amount or perturbation
s	Classifier score	ζ	Distance
u	Neuron inputs and outputs	η	Moment power
v	Eigenvector	$\underline{\theta}$	Parameter vector
w	Normal vector	κ	Class index
w_o	Discriminant offset	λ	Eigenvalue
x	Feature vector	μ	Mean
y	Measurement vector	ν	Statistical threshold
z	Class prototype	ξ	Statistical threshold
C	Class	ρ	Minkowski distance power
F	Feature matrix	σ	Standard deviation
H	Hypothesis	σ^2	Variance
I	Identity matrix	τ	Classification Threshold
J	Criterion	ϕ	Probability density kernel
L	Standard L^1, L^2 penalty criteria	χ	Chi-Squared distribution
P	Probability	ψ	Neuron activation function
S	Ensemble count	φ	Nonlinear transformation
V	Matrix of eigenvectors	ϑ	Probability parameter
		Λ	Covariance matrix, typically diagonal
		Σ	Covariance matrix
		Ψ	Transformation kernel



Overview

Pattern Recognition and Machine Learning were once something of a niche area, which has now exploded to become one of the hottest areas of study and research. Students from nearly every field of study clamour to study pattern recognition courses, researchers in nearly every discipline seek ways in which machine learning/deep learning might be applied to their domain, and companies seem enthusiastic in their rush to incorporate “intelligence” (of some sort) into fridges, toasters, and washing machines.

What Is Pattern Recognition?

In many ways, pattern recognition is not at all a new discipline. It has *long* been important in statistics to perform statistical inference of some unknown quantity on the basis of related measurements, what is broadly referred to as an inverse problem. If the unknown quantity is *continuous* in nature — height, speed, time interval etc. — then the field of study regarding such inference is referred to estimation theory. In contrast, if the unknown quantity is *discrete* in nature — species of bird, type of protein, status of a machine (normal/broken) — then the inference is known as pattern recognition.

That is, pattern recognition is entirely focused on determining a particular identity (the pattern “class”) based on measured information, and the process which selects the class is known as a classifier:



The central task of pattern recognition consists of two parts:

FEATURE EXTRACTION: Which measurements are relevant to the classification process, and which measurements are irrelevant? Can we find *features*, some

function of the given measurements, that more compactly and reliably encode the salient information?

CLASSIFICATION: Given measurements or features, how do we determine their associated *class*, their underlying discrete identity?

A great many classification approaches have been developed, from the very simple (straight line or plane boundary between two classes), mathematically rigorous (statistically optimal classifiers), to much more advanced (large ensembles or aggregates).

Deep learning and associated deep neural networks dominate much of the related topics of pattern recognition, machine learning, artificial intelligence, and computer vision. Neural networks have been particularly successful at high-level language and vision problems, such as text translation, object recognition, or video captioning, but at a cost of very high computational complexity and, in most cases, absolutely no interpretability regarding what the network is doing or how it is accomplishing its task. This text will offer a systematic study of the mathematical, methodological, and conceptual developments that ultimately lead to deep networks, which are therefore discussed in their logical place as part of nonlinear ensembles of classifiers in [Sections 11.4](#) and [12.2](#), however there are a great many other books which discuss the practical aspects of programming network architectures and network learning.

How to Read This Book

The chapters are organized conceptually and pedagogically to be read/taught from beginning to end, so that approach would be recommended for anyone who is willing to invest the time.

Furthermore, as is perhaps true of most subjects, pattern recognition is best learned by *doing*, and so a substantial number of sample problems are offered at the end of each chapter. Pattern recognition can be thought of as two parallel disciplines, one analytical/mathematical, and the other experimental/computational. This text tries to provide a balance of both, with an emphasis on conceptual understanding (with supporting derivations in [Appendix D](#)), but also with a fully worked numerical/computational lab study at the end of every chapter and programming/computational questions at the end of each chapter.

For those readers who are inclined to jump around in a text to get an appreciation for and an overview of pattern recognition, you might begin by looking at the examples and case studies (listed on [page xv](#)) since these are written in an encapsulated style, intended to be readable on their own.

For those readers eager to jump in and quickly actually *do* some pattern recognition, you should probably look through [Chapter 2](#) to understand the overall concepts and terminology, and then progress to distance-based classification in [Chapter 6](#), since distance-based approaches are fairly intuitive and quick to implement. The computational lab study at the end of every chapter may also help as a quick introduction to basic techniques.

Finally, for those readers who want to jump straight into Deep Neural Networks, there are many other textbooks and online guides that can do this for you. [Section 11.4](#) does discuss nonlinear ensembles, which includes deep networks, and which the reader could take a look at (along with that chapter's [Further Reading](#)), however the preceding 319 pages aim to give much deeper insight into the pattern recognition problem, and so the reader will be missing a great deal of context and understanding if jumping straight to large nonlinear networks.

Most of this text, and nearly all of the examples and case studies, are intended to be accessible and appreciated without necessarily following the details of the mathematics. To avoid cluttering the main body of the text, many of the mathematical details and derivations were moved to [Appendix D](#), however a deeper mathematical understanding is one of the central goals of this text, and the reader is very much encouraged to follow the derivations in [Appendix D](#) in parallel with the text. For those readers whose mathematics background needs refreshing, the appendices provide most of the material needed on algebra ([Appendix A](#)), random vectors ([Appendix B](#)), and optimization ([Appendix C](#)).

This book is, to be sure, only an introduction, and there is a great deal more to explore. Directions for further reading are proposed at the end of every chapter.



Introduction to Pattern Recognition

2.1 What Is Pattern Recognition?

Pattern recognition is a process by which some input is measured, analyzed, and then classified as belonging to one of a set of classes.

Although this opening definition may sound somewhat abstract, in actual fact the process of pattern recognition and classification is a continual, never-ending aspect of every-day human existence:

Pattern recognition task	Possible classes
What is in front of you as you walk?	Door vs. Window Sidewalk vs. Road
What music are you listening to?	Familiar or Unfamiliar Genre (Rock, Classical, ...) Name of Composer or Group
Is the traffic intersection safe to cross?	Green vs. Red light Pedestrian Walk vs. Stop Car Present vs. Not Present
Reading a page in a textbook	Letters of the Alphabet Text vs. Graphics Languages
You smell something in your apartment ...	Cookies finished baking? (Yes/No) Is something burning? (Yes/No) Wet dog/Skunk/Dead mouse/...

As a human experience, pattern recognition refers to a perceptual process in which some form of sensory input is sensed, analyzed, and recognized

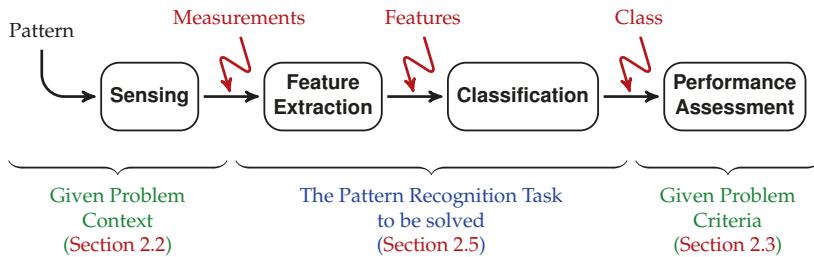


Fig. 2.1. PATTERN RECOGNITION: A pattern is sensed, giving rise to measurements, from which feature are extracted and given to a classifier, whose job it is to select the class associated with the sensed pattern. In most cases the measurements and the classes are characterized (green) by the given problem definition, and the pattern recognition task (blue) is to infer a strategy for feature extraction and classification.

(classified), either subconsciously (by instinct) or consciously (based on previous experience). Patterns may be presented in any sensory modality: vision, hearing, touch, taste, or smell.

As a technical discipline, pattern recognition refers to a process in which an input object is measured, analyzed, and classified by a machine as being more or less similar to some class in a set of classes.

There are nearly endless contexts to which pattern recognition may be applied, from broad problems such as object recognition, to human-mimicking tasks such as text recognition (as in [Example 2.1](#)) or face recognition, to applications such as medical diagnosis or fault detection in physical systems. Motivating contexts include cases in which the amount of data may be too large, the number of decisions too great, or the pattern distinctions too imperceptible for a human to effectively undertake the classification.

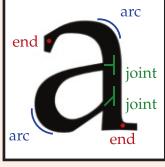
The conceptual framework for pattern recognition, whether human-based or machine-based, is illustrated in [Figure 2.1](#). In general,

- for a given definition of the information at hand (the measurements, [Section 2.2](#)),
- and some definition of what problem we wish to solve (the classes, [Section 2.3](#)),
- the goal of pattern recognition is to provide a machine with a kind of perceptual capability to automatically extract useful information from measured data (classification, [Section 2.4](#)).

The following three sections address these issues, in turn.

Example 2.1: Pattern Recognition of Text

When you read a printed character on this page, your eye images the character on your retina, where it is sensed and converted into a neural representation which is subsequently analyzed in your brain. If your memory indicates prior experience with the character, or with a symbol sufficiently similar to it, you perceive or recognize the symbol, associating a label and meaning with it. What appears to us to be a relatively simple process becomes ever more impressive as we attempt to design a machine which might accomplish the same task, known as optical character recognition (OCR). What features might a machine use in recognizing text?

Pattern	Attributes to Measure	Measurements	Strengths and Weaknesses
"a" →		Vector of pixel values	Fast, easy, explicit Sensitive to changes in font style, size, and rotation
"a" →		Vector of shape properties	Robust to changes in size and rotation Complicated features to extract
"a" →	Complex Nonlinear Algorithm	Vector of values, but with no intuition	Possibly very flexible May be very hard to learn Difficult to analyze

For a machine needing to deal with printed characters in only a single font, a simple classification scheme might sense an array of pixel intensity values (top example, above) and compare it to arrays stored in memory corresponding to known characters. Such a template matching approach is simple and fast, but will be subject to a variety of limitations: any distortion, translation, rotation, scale change, or even lighting variations will affect the degree to which the given measurement matches a template.

If the machine must accept a variety of fonts, as humans would in reading, a memorized template matching scheme becomes impractical, since an unreasonable number of templates would need to be stored and compared. An alternate approach is to seek distinctive character features (middle example, above), such as strokes, arcs, loops etc. It might also be possible to train a black-box strategy (bottom), whereby some complex nonlinear function is learned, producing features which no longer have any intuitive meaning regarding recognizing characters, but which turn out to be effective features in classifying typed characters.

Further Reading: Q. Ye, D. Doermann, "Text detection and recognition . . .," *IEEE PAMI* (37) #7, 2015.
H. Lin, P. Yang, F. Zhang, "Review of scene text detection . . .," *ACM in Eng.* (27), 2019.

Example 2.2: Pattern Recognition of the Mind

The retina in the eye is densely packed with light-sensitive cells, so that it may be tempting to think that our brain effectively sees and perceives the world as a great many pixels, much like the first feature in [Example 2.1](#). However there are many simple mind tricks or optical illusions that can make it clear that a systematic, pixellated view of the world is not really how we function.

With randomly fluctuating rotation, scale, and vertical offset ...

The typesetting may be weird, but for the human brain this is very easy to read,

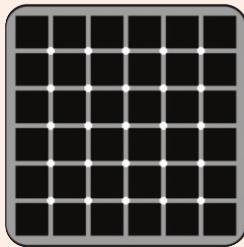
... but exceptionally difficult for a computer to do so.

Further evidence that our brain recognizes overall words or groups of words, rather than individual letters, stems from our ability to read text with permuted or missing letters:

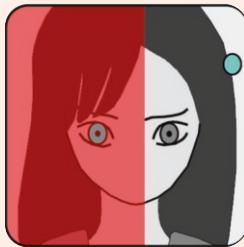
It deons't mettar in waht oerdr the lrtees in a wrod are,
as lnog as the frist and lsat ltteres rmeain in the rgiht pacle.

*t *ls* d**sn't m**tt*r wh**th*r y** h*v* *ll *f th* l**t**rs in *v*r* w**rd,
*s l**ng *s y**r br**n h*s *n**gh *nf*rm*t**n t* f**ll *n th* g*ps.

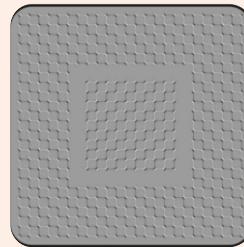
Indeed, the immense popularity of optical illusions in great part stems from us not being able to consciously understand why our brain's visual system is seeing a particular behaviour,



There are no black dots in the white circles.



The two eyes are the same colour. There is no blue pigment, at all, in the left eye.



This is a static image, yet try zooming in and scrolling.

... but which, among other reasons, stems from the fact that the eye is *not* just sending pixellated images to the brain. Rather, there is a great deal of *feature extraction* taking place, much already in the retina, which evolutionarily proved very helpful in running through a forest, but perhaps not so useful in staring at deliberately manipulated images on a page. To be fair, computer-based image recognition is susceptible to its own optical illusions ([Example 11.3](#)), in many cases far more primitive than those affecting the human visual system.

Further Reading:

D. Gershgorn, "Fooling the machine," *Popular Science*, 2016.

D. Purves, R. Lotto, S. Nundy, "Why We See What We Do," *American Scientist* (90) #3, 2002.

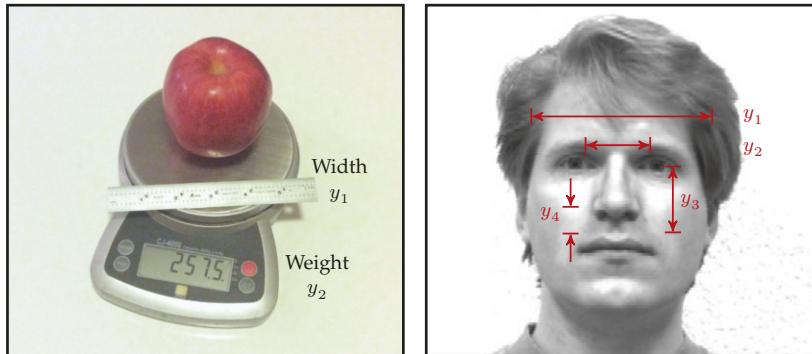


Fig. 2.2. MEASUREMENTS: There is a wide variety of measurements which could conceivably be extracted for any object of interest, whether the size/weight of a piece of fruit, left, or various dimensions extracted from a face, right. (Right image from the Yale Face Database [2, 5]).

2.2 Measured Patterns

The word “pattern” may bring to mind texture, fabric, or shape. However in the context of *pattern recognition*, the notion of pattern is far more broad, and can apply to any *thing* that can be distinguished from another *thing*. Really, “identity” might have been a better choice of word: we would like to infer the unknown *identity* of an object, whether the type of wildflower, type of songbird, or the name of the person facing a camera — each of these has a certain identity which we would like to determine from measurements.

A pattern is assumed to have certain *properties* or *attributes* which distinguishes it from other patterns. One or more *measurements* are taken of a pattern, as shown in Figure 2.2, which (should) reflect either directly or indirectly the attributes associated with the pattern. Indeed, one of the steps in pattern recognition is to assess to what extent a given measurement is relevant or helpful, as opposed to irrelevant or useless, to the classification task at hand. The selection of appropriate measurements is an essential part of the design stage of any pattern recognition solution development, since measurements may cost money and/or time, and poor measurements lead to poor performance of the resulting classifier.

As shown in Figure 2.1, *features* are functions of the measurements,

$$\text{Measurements } \underline{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \xrightarrow{\underline{x} = f(\underline{y})} \text{Features } \underline{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad (2.1)$$

The process of transforming measurements into features is intended to somehow facilitate classification, normally in one or both of the following ways:

1. By reducing the dimensionality of the problem: $n < m$

We need to develop a classifier in the n -dimensional feature space. Fewer dimensions are easier to visualize, easier to understand, are normally associated with learning fewer parameters, and will typically lead to more robust classifiers.

2. By creating features in which patterns are more clearly distinguished:

In any given problem, we may be constrained in terms of what sorts of measurements are available, perhaps on the basis of cost or the kinds of measurement instruments available. As a result, each measurement may only very weakly distinguish different pattern classes, and it may be that some function $f()$ of the measurements can yield a feature which is much more discriminating.

It is important to understand that the feature extraction function $f()$ can focus the information from \underline{x} , or it can remove irrelevant information from \underline{x} , but $f()$ never adds information. This is known as the *Data Processing Theorem*:

If $\underline{x} = f(\underline{y})$, then \underline{x} can never have more information than was present in \underline{y} .

An effective feature extraction function $f()$ can make the pattern recognition problem *easier*, however, in principle, the best possible classifier based on the measurements \underline{y} should perform at least as well as the best possible classifier based on the features \underline{x} .

Features may be intuitive or they may be quite abstract. Consider, for example, the measurements (with given units) which we might take of an electric motor:

$$\text{Measurements } \underline{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} \text{Motor voltage (Volts)} \\ \text{Motor current (Amperes)} \\ \text{Motor speed (RPM)} \\ \text{Motor winding temperature (\textdegree C)} \\ \text{Surrounding air temperature (\textdegree C)} \end{bmatrix} \quad (2.2)$$

then each of the individual measurements is clear, and a feature such as

$$x = f(\underline{y}) = y_1 \cdot y_2 \quad (2.3)$$

is easily intuitively understood as the motor power¹ in Watts, whereas another feature, such as

$$x = f(y) = \sqrt{y_1 - y_2} - \frac{y_3}{y_4} \quad (2.4)$$

is uninterpretable, in units which do not make any physical sense, but which could perhaps be effective as a feature for classification.

It should be pointed out that (2.1) is not the *only* way of encoding the information regarding a pattern. There is a field known as *Syntactic Pattern Recognition*, in which patterns and features are understood more as grammatical/linguistic constructs, but which will not be considered in this text (although discussed briefly in [Chapter 13](#)).

2.3 Classes

The whole purpose of pattern recognition or classification² consists of assigning an object to a *class*. A class is a particular pattern, or possibly a group of patterns which are *similar* or *equivalent* in some sense. In a given problem, the set of classes \mathcal{C} is defined as

$$\mathcal{C} = \{C_1, C_2, \dots, C_K\}, \quad (2.5)$$

such that we have K different classes³ from which to choose. Whether we know the classes, or what we know about the classes, or whether we even know the *number* of classes K , will depend on the kind of pattern recognition problem to be solved, as will be discussed in [Section 2.5](#).

The notion of a *class* presupposes some sense that members of a class share some common properties or attributes. However the definition of class can very much be a function of the problem being solved, as is illustrated in [Example 2.3](#).

¹ Technically, for an AC (alternating current) motor, one needs to distinguish between instantaneous power, at an instant of time, and average power, integrated *over* time. This distinction need not concern us here.

² This text will consider the terms “recognition” and “classification” as equivalent.

³ The careful reader will observe the inconsistency in identifying the number of classes as K , when the [textbook notation](#) indicates that scalars will be lower case, and upper case variables are used for matrices or abstract concepts, such as classes. However the use of K for the number of classes is exceptionally widely adopted in pattern recognition, so it seemed preferable to retain it here to avoid confusion.

Example 2.3: What Is a Class?

A class captures the essence of a pattern, such that there can be many examples of the class, all of which are *similar* or *equivalent* in some sense. The definition of class is not inherent or absolute, rather it will depend on which problem is being solved.

So, for example, in a face recognition problem each person is their own class, so the set of classes would be defined as

$$\mathcal{C} = \{ \text{"Paul Fieguth"}, \text{"Bob"}, \text{"Jane"}, \text{"Ali"}, \dots \}$$

so that I would be a member of the “Paul Fieguth” class:



$\in \text{"Paul Fieguth"}$

You might argue that there cannot be “many examples” of this class, as was suggested at the top of this discussion, since I am only one person. But in the face recognition problem there can in fact be *many* pictures of me, *all* of which should be associated with my class.

At a university we might define a different set of classes

$$\mathcal{C} = \{ \text{"Professors"}, \text{"Staff"}, \text{"Undergraduate Students"}, \text{"Graduate Students"} \}$$

such that

“Paul Fieguth” $\in \text{"Professor"}$

Such a set of class definitions might seem to make sense from an organization/categorization perspective, however in most cases there are not likely measurable or observable attributes that allow for categorization as undergraduate vs. graduate, or professor vs. staff, so it is probably actually not helpful to formulate a pattern recognition problem this way.

One *could* use pattern recognition to estimate age, in which case one might have classes like

$$\mathcal{C} = \{ \text{"0–10 years"}, \text{"10–20 years"}, \text{"20–30 years"}, \dots \}$$

such that now I appear in a class as

“Paul Fieguth” $\in \text{"50–60 years"}$

Although this is a legitimate pattern recognition problem, it is a bit artificial, in that really what we have is a continuous underlying value, *age*, which we have chosen to (somewhat arbitrarily) discretize into classes (as is discussed in further detail in [Example 8.1](#)). Really we do not actually have inherently distinct classes here, rather we should instead formulate this as a parameter estimation problem, such that continuous parameter *age* is estimated from measurements, as will be discussed in [Chapter 7](#).

Example continues ...

Example 2.3: What Is a Class? (continued)

One case where classes *would* be inherent could, for example, appear in a taxonomy of life,

$$\mathcal{C} = \{ \text{"Mammals"}, \text{"Reptiles"}, \text{"Amphibians"}, \text{"Fish"}, \text{"Sharks"}, \dots \}$$

such that

$$\text{"Paul Fieguth"} \in \text{"Mammals"}.$$

We will need some way of describing classes. Such descriptions largely fall into one of four types:

VIA PROTOTYPE, an idealized representation or notion of the “essence” of the class. The advantage is that each class is unambiguously defined, however with no scope for variability.

VIA PARAMETERIZED SHAPE, a generalization of the prototype, in which the class has a known shape (rectangular or elliptical, say), where the shape is described in some number of parameters (ellipse centre, rotation, and axis lengths, for example). The description is more flexible than that of a single prototype, but still requires the type of shape to be assumed or known.

VIA STATISTICAL DISTRIBUTION, such that we have some description of the likelihood or probability of a class member having a particular set of measurements or features. This approach is very comprehensive, however there will be circumstances when the statistics are not known and may be difficult to infer.

VIA SAMPLES, such that a set of given samples (many apples, or tigers, or bicycles) directly characterizes the class. When such samples are given the representation is highly convenient, since nothing further needs to be done to describe the class, however there may be storage and computational challenges, since all of the data need to be saved and then searched every time a classification needs to be undertaken.

There is some fluidity between these forms, in that we may try to infer a prototype or a distribution from given samples, or to generate typical samples from a given distribution. Furthermore, in principle the prototypes, distributions, and samples can all be defined both in the measurement space $y \in \mathbb{R}^m$ and in the feature space $x \in \mathbb{R}^n$.

It should be clear that patterns do not need to be identical to belong to the same class: not all pictures of me ([Example 2.3](#)) are the same, or of tigers, or of apples.

There are at least two sources of variability present in the measurements associated with a single class:

1. The inherent variability within a class: Every class will consist of members which differ in some way. The degree and nature of the inherent variability will depend greatly on the class definition. So the “Fruit” class contains all manners of variability in colour, size, and shape; the “Apple” class is much more specific, but apples do come in different colours and patterning; the “Granny Smith Apple” class is even more specific, but still will have apples of different sizes or with more or fewer blemishes.
2. Noise or random variations⁴ in measurement: Every measurement involves some sort of physical process which will be subject to error, such as thermal noise in electronics, or quantization noise in converting an analogue signal to a digital representation.

In many pattern recognition problems the class set \mathcal{C} is predefined and has been specified as part of the problem to be solved. Ideally, however, there would be significant value in putting thought into class definitions in the design stage of a problem, to avoid class definitions which are articulated poorly or vaguely.

2.4 Classification

A *classifier* is some function $g()$, possibly analytical (i.e., an equation) or a computer algorithm, which assigns a class label to a given feature:

$$g(\underline{x}) \in \mathcal{C} = \{C_1, C_2, \dots, C_K\} \quad (2.6)$$

There is a strong relationship between feature extraction and classification, such that

- Good features allow for simpler classifiers, whereas
- Complex classifiers can compensate for weaker features, those features which are unable to fully separate the pattern classes.

A simple example of a classifier is illustrated in [Figure 2.3](#). In this case $K = 2$, meaning that the classifier is discriminating between only two classes. The classifier is a function of two-dimensional $\underline{x} \in \mathbb{R}^2$, and in [Figure 2.3](#) the classification boundary of $g()$ is chosen to be a straight line, although *many* other classifiers are possible.

⁴ Noise and Randomness are inherently statistical concepts, which are reviewed in [Appendix B](#).

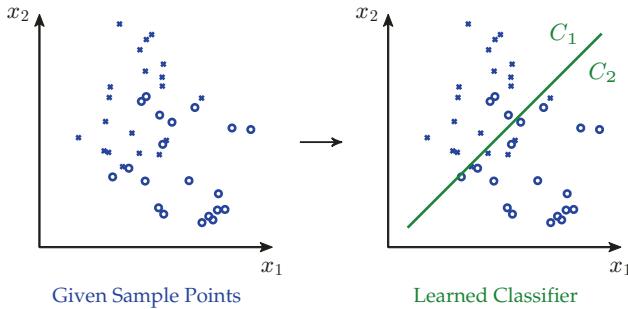


Fig. 2.3. PATTERN RECOGNITION I — CLASSIFIER LEARNING: A classifier, here a straight line (right) dividing a feature space into classifications C_1 and C_2 , can be learned from a given set of sample points (left).

There are two fundamental steps associated with classification:

CLASSIFIER LEARNING is illustrated in [Figure 2.3](#). The method by which classifier $g()$ is learned is normally a function of how classes are described. That is, we will see methods for learning $g()$ on the basis of class prototypes, statistical distributions, or directly from sample data.

The whole topic of *Learning* is discussed in further detail in [Chapter 3](#), and specific types of classifiers will be discussed starting in [Chapter 6](#).

CLASSIFIER TESTING OR VALIDATION is illustrated in [Figure 2.4](#). Another important aspect of pattern recognition is the notion of testing, where we evaluate the performance of a classifier to assess how well it provides the correct class prediction. There are many ways to assess classifier performance; two of the most basic are

- Training accuracy: determine how well a classifier can assign correct classes to samples it was trained on, and
- Testing accuracy: determine how effectively a classifier can assign correct classes to test samples it has *not* been exposed to.

Classifier error will be assessed in detail in the statistical context, in [Chapter 8](#), and broader strategies for classifier *Validation and Testing* will be discussed further in [Chapter 9](#).

An important application of pattern recognition is for visual perception tasks, where the goal is to take an image (or video) as an input, and make predictions on the content of the image. Four illustrative examples are shown in [Figure 2.5](#), where we go from whole scene categorization and optical character recognition, to more complex visual perception problems such as object detection within an image and answering complex high-level questions about a scene.

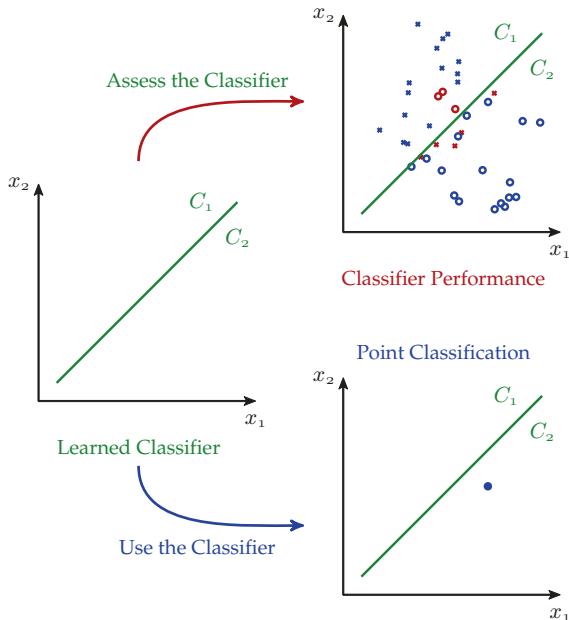


Fig. 2.4. PATTERN RECOGNITION II — CLASSIFIER TESTING: What can we do with the learned classifier, left, from [Figure 2.3](#)? We could assess its performance (top), for example by counting how many sample points are classified correctly (blue) and incorrectly (red). Or we could apply the classifier (bottom) to a new, unknown point and then classify it.

2.5 Types of Classification Problems

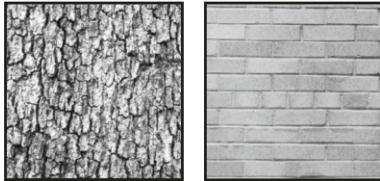
With measurements, classes, and classification defined, the final step in introducing pattern recognition is to better understand what sorts of classification problems we might encounter. The range of problems is very large, from simple to complex, low dimensional to high dimensional, and given detailed models or training examples to being given almost no information at all.

From this vast range of pattern recognition problems, we can most fundamentally group these problems into four scenarios on the basis of what is known and what is not known, as is illustrated in [Figure 2.6](#):

SCENARIO 1: Model is known or given

This is the most information we can hope to have regarding a pattern recognition problem, in that we are told the behaviour of the measurements for each of the pattern classes. Normally this behaviour is characterized in

Texture Classification (Brodatz [3]):
Classify each texture



Digit Recognition (MNIST [6]):
Which digit is which?

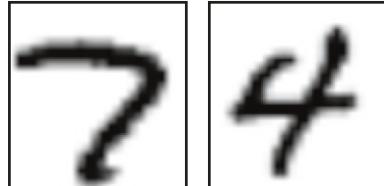


Image Segmentation (MS-COCO [4]):
Find the airplane, the car, the bus ...



Image Segmentation (VisualQA [1]):
Did the batter hit the ball?



Fig. 2.5. PATTERN RECOGNITION ON IMAGES: Since the human visual system is so dominant in human perception, a great deal of pattern recognition focuses on image-related problems. Here four examples are shown, from comparatively straightforward (top), the classification or recognition of whole images, to rather advanced (bottom), such as recognizing the objects within an image or being able to answer complex high-level questions.

a statistical⁵ fashion, such as

$$p(y|C) = \text{The distribution of measurement vector } y \text{ given class } C \quad (2.7)$$

Such detailed information will be available only in those contexts where the physical process is known by which a given pattern class gives rise to measurements. It is very convenient to have such detailed information, since this allows statistical decision theory (Chapter 8) to explicitly define the optimal classifier, in the sense of minimizing the probability of classification error.

SCENARIO 2: Model is not known, labelled data are⁶ available

Although we do not have an exact description of the problem, as in (2.7), we are given labelled data, meaning data pairs of the form

$$\{y_i, C_\kappa\} \longrightarrow \text{The } i\text{th measurement vector } y_i \text{ is known to have come from class } C_\kappa \quad (2.8)$$

⁵ Statistical pattern recognition is discussed in Chapter 8, and the background mathematics on statistics and distributions can be found in Appendix B.

⁶ I know it may look a bit odd, but in fact the word “data” is *plural*, so data *are* available. The singular is “datum”.

The four fundamental types of Pattern Recognition problems,

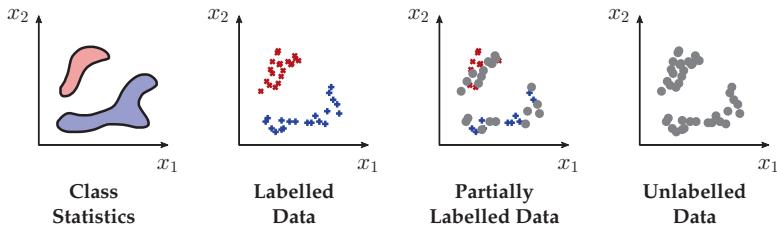
Model-Based

Supervised

Semi-Supervised

Unsupervised

... based on what information you are given ...



One example of each ...

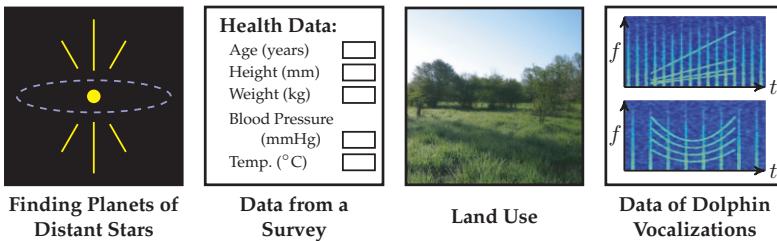


Fig. 2.6. THE FUNDAMENTAL PROBLEMS: There are four fundamental pattern recognition problems, ordered from the most detailed problem description (left) to the most ambiguous (right).

Labelled data do not just *magically* appear, of course; normally they have been labelled or tagged by a human observer, so we refer to this scenario as *supervised*, which can become exceptionally expensive or labour-intensive with larger datasets. Given labelled data, we have two broad approaches:

1. [Chapter 6](#): We can derive a classifier directly from the given labelled measurements.
2. [Chapter 7](#): We can learn an empirical probability model, as in [\(2.7\)](#), based on the labelled data, and then use the classifier which follows from the model.

SCENARIO 3: Model is not known, some data are labelled, some are not

This scenario would seem to be a trivial extension of Scenario 2, in that one could choose to learn a classifier (whether via [Chapter 6](#) or [Chapter 7](#)) on the basis of only those data points which are labelled. However such an approach *ignores* all of the *unlabelled* data; since labelled data can be expensive, requiring manual labelling, it is possible that *most* of the data will be unlabelled. This type of problem is referred to as semi-supervised, since some degree of human input is required.

Semi-supervised problems appear commonly in cases where we wish to use a small set of samples that have been manually labelled for classification (e.g., face images tagged by a human observer), but then also to leverage a very large set of unlabelled data. In actual fact, *most* pattern recognition problems are of this form, it is then more a question of whether the number of labelled data are sufficient for learning (back to Scenario 2), or not. Although semi-supervised learning is treated only very modestly, at the end of [Chapter 12](#), it builds very directly on all of the methods of inference, classification, and clustering, which are discussed in the rest of this text.

SCENARIO 4: Model is not known, no labelled data are available

Pattern measurements are available, however the points have no associated class information; this is known as an *unsupervised* problem. The range of problems here is still very broad, depending on whether we are told the *number* of classes, or their typical size or separation, or perhaps nothing at all. We refer to these as *clustering* problems, to be discussed further in [Chapter 12](#).

Case Study 2: Biometrics

Biometrics are measurements that we can take of human characteristics. Most commonly we associate biometrics with the ability to validate who a person is, to uniquely recognize their identity. Common biometrics would include measurements of any of the following:

- Face
- Fingerprint
- Iris (the coloured region in your eye around the pupil)
- Retina (the pattern of arteries in the back of your eye)
- Veins (the vein structure in some part of your skin, normally the hand or arm)
- DNA

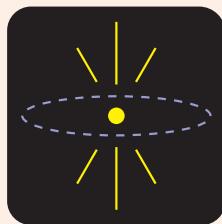
With the exception of DNA, where taking a measurement is much more invasive, all of the other biometrics on this list are based on extracting features from an image which can be acquired remotely (without touching the body).

To be sure, biometrics are not *only* about security and personal identification, and there is much that can be measured about a physical body that might not be helpful in recognizing an individual, but might be *quite* helpful in sports analytics, such as designing a superior golf club. However for the purpose of this case study our focus will be on the use of pattern recognition for uniquely identifying a person.

Example 2.4: The Types of Pattern Recognition Problems

Scenario 1 — Model-Based: Planets around Distant Stars

Scientists are making claims about planets around distant stars, light-years away, and whether these planets are earth-like or not, having oxygen atmospheres. How is this even remotely possible?!



Scientists cannot actually *image* planetary atmospheres, let alone even see the planet itself at all. Instead, scientists have to *infer* the presence of a planet, its size, and its atmospheric constituents by having a *model* for the intensity of light, over time and as a function of wavelength, from that star. When a planet passes in front of the star, the star appears to dim slightly, and the planet's atmosphere contains molecules which influence the starlight at certain wavelengths. So we might have a vector of measurements, something like

$$[I_{\lambda_1}^b \ I_{\lambda_2}^b \ \dots \ I_{\lambda_q}^b \ I_{\lambda_1}^d \ I_{\lambda_2}^d \ \dots \ I_{\lambda_q}^d]$$

for intensity observations when the star is bright I^b and dimmed I^d , and at wavelengths $\lambda_1, \dots, \lambda_q$ chosen to be at spectral lines of those atmospheric components of interest, such as water, oxygen, methane etc.

K. Heng, "A new window on alien atmospheres," *American Scientist* (105) #2, 2017

A second much simpler example could be a fetal heart monitor. Medical models suggest a normal fetal heart rate of 110–160 beats per minute. As a result, it would be easy to develop a monitor which sounds an alarm if the heart rate goes below 110 or above 160.

But where did such a model come from? Well, it came from a long history of observing the heart rates of many fetuses, and which of these were under some sort of distress or not. In other words, we gathered labelled data on heart rate (the measurement) and distress or not (the label), and then aggregated these data into a model.

So the model in a model-based approach could be derived from science or from labelled data — both approaches are legitimate.

Scenario 2 — Supervised: Surveys

We are exposed to nearly endless surveys (government census, political surveys, movie preference surveys, etc.) — every such survey represents *labelled data*, from which we can undertake supervised learning.

Suppose we wish to identify whether someone is a child or adult based on their height. What we can do is survey the heights (the measurement) of a large number of individuals, each person identified as "adult" or "child" (the label). Having each data point labelled is very convenient, because

Health Data:	
Age (years)	<input type="text"/>
Height (mm)	<input type="text"/>
Weight (kg)	<input type="text"/>
Blood Pressure (mmHg)	<input type="text"/>
Temp. (°C)	<input type="text"/>

Example continues ...

Example 2.4: The Types of Pattern Recognition Problems (continued)

it allows us to know “truth” for each measurement, and to validate whether a proposed classifier is actually classifying correctly or not for each given measurement.

Scenario 3 — Semi-supervised: Land Use

With rare exceptions, nearly every pattern recognition problem begins as unlabelled, and the question then becomes how *much* labelling is needed for adequate learning, or how much learning can be afforded given a limited budget.

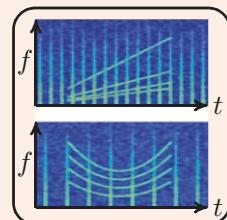


The illustrated example is that of land use: there are endless satellite maps and airplane photos of the earth’s surface (those are our *unlabelled* data), but sending out a team of people to actual do field surveys, on site (or perhaps trained operators sitting at a computer terminal, identifying houses and roads) costs money. That is the cost, however, of labelling the data: to actually *know* what land use (desert, city, agriculture, forest, water, swamp, ...) is actually present at a given location. To be sure, the actual land-use categories (classes) could vary significantly from one context to another, depending on what you would like to study.

Although labelling may be expensive, even labelling only a modest number of points provides significantly more context than having no labelled data at all. At the same time, the *unlabelled* data points are also useful, since they may give us a much richer sense of class shape than the few labelled data points may be able to do.

Scenario 4 — Un-supervised: Whale and Dolphin Vocalizations

Suppose we wish to identify the vocalizations of dolphins or whales, where the figure illustrates vocalization strength (colour) and frequency f as a function of time t .



We have no way of labelling the data, since there is no information available to us regarding the meaning or intent behind the vocalizations. Nevertheless there *are* patterns present in the data — there are particular sounds which are articulated, repeatedly, which could be represented as a cluster of points in some feature space. And there are *different* sounds that are made, each of which would appear as a *separate* cluster, assuming the feature space to have been appropriately chosen.

The pattern recognition challenge would then be a clustering problem ([Chapter 12](#)). The goal would be to look for naturally-occurring groups (each of which might be a word or a sentence), or possibly groups-of-groups, allowing us to begin understanding some aspect of the richness, complexity, or variations in dolphin and whale communications.

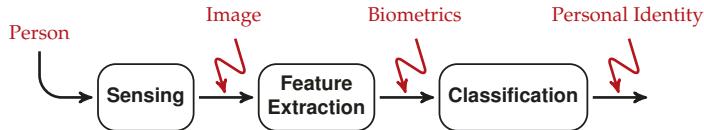


Fig. 2.7. BIOMETRIC RECOGNITION: In order to recognize a specific person on the basis of remotely-sensed biometrics, we need to acquire (sense) an image of some part of the body which has a unique signature (fingerprint, retina etc), extract biometric features from this image, and then develop a classifier that reliably recognizes the individual.

As we know from [Figure 2.1](#), which has been re-cast into the context of this case study in [Figure 2.7](#), a biometric-based human recognition system would have three basic components:

1. **Image Acquisition:** Outside the scope of this text, there are many design decisions related to imaging hardware, such as cost, ease of use, lighting, choices of sensor (infrared versus visible, grey-scale versus colour). Poor hardware, which yields images which vary significantly for the same person, places a greater burden on the subsequent steps of feature extraction and classification, so one goal is for the hardware to produce images as consistently repeatable as possible.
2. **Feature Extraction:** Also outside the scope of this text, given an image we now need to extract some vector of values which encode the behaviour observed in the image. Examples could include face geometry (as in [Figure 2.2](#)), a texture model of the iris, the locations and angles of arteries in the retina or lines in a fingerprint. This step is essentially in the domain of image processing: how to extract information from a given image.
3. **Classification:** Given a vector of features, the core component of the biometric system is a classifier, determining which person has just been measured.

The classifier returns one member of the class set

$$\mathcal{C} = \{\text{NoMatch}, \text{Person}_1, \text{Person}_2, \dots, \text{Person}_K\}. \quad (2.9)$$

It is important to have the *NoMatch* class present, otherwise the classifier is forced into classifying a given feature vector, even when there is little evidence that the feature vector is a good match. The reason why this is important is because of a significant asymmetry in the costs of two kinds of error:

$$\begin{aligned}
 &\text{Classifying } \underline{x} \text{ as } \text{NoMatch} \text{ where } \text{Person}_1 \text{ is correct} \leftarrow \text{Frustration} \\
 &\text{Classifying } \underline{x} \text{ as } \text{Person}_2 \text{ where } \text{Person}_1 \text{ is correct} \leftarrow \text{Security Breach}
 \end{aligned} \quad (2.10)$$

Avoiding frustration and security breaches are not the only criteria at hand. In general, a successful biometric strategy must satisfy

- Universality: Every person should be measurable, regardless of age and health
- Uniqueness: The feature vectors extracted for a given person should be robustly unique
- Consistency: For a given person the feature vector should be highly repeatable from one try to the next, and should be slowly (or not at all) varying over time.

Clearly there are other criteria, such as non-invasiveness, social acceptability, or how easily the system would be to defeat via nefarious means.

How we would assess and validate the resulting classifier is a complicated topic which will be discussed at length later in this text ([Chapters 3, 4, 8, and 9](#)). In a nutshell, consistency is evaluated on the basis of class *spread* in feature space, uniqueness is assessed based on class *overlap*, and the two types of error in [\(2.10\)](#) would be evaluated based on test data (as discussed in [Chapter 9](#)).

Lab 2: The Iris Dataset

The end of each chapter contains a worked numerical lab. The intent of the labs is to bring out some of the details in implementing pattern recognition concepts, so that the reader can see all of the steps laid out. The code is written in MATLAB, which was chosen because it is widely available at educational institutions, and because the code is fairly intuitive to anyone who programs in C or Python. Many tutorials can be found online, however no particularly detailed understanding of MATLAB is required to follow the lab discussions.

We have not yet learned any classifiers, but we can start by trying to understand what a pattern recognition dataset might look like. The *Iris Dataset* contains flower data for three types of Iris:

Iris Setosa



(Creative Commons, Tiaa Monto)

Iris Versicolor



(Creative Commons, Danielle Langlois)

Iris Virginica



(Creative Commons, Public Domain)

Now certainly there is nothing profound about classifying these three particular plants, since there are many other types of Iris, many other types of flowering plants, and indeed many other pattern recognition problems of much greater significance. However the dataset is classic, having originally been published in 1936, it is relatively simple, and it is widely used in the pattern recognition field.

With reference to [Figure 2.1](#), we need to begin by identifying the classes and the measurements. There are three classes,

$$\mathcal{C} = \{C_1, C_2, C_3\} = \{\text{"Iris Setosa"}, \text{"Iris Versicolor"}, \text{"Iris Virginica"}\}$$

and for each plant four measurements were taken:

$$\underline{y} = \begin{bmatrix} \text{Sepal Length} \\ \text{Sepal Width} \\ \text{Petal Length} \\ \text{Petal Width} \end{bmatrix}$$

This dataset and following code are available for you from the [the textbook data site](#):

```
load Iris
IrisData
```

which leads to the following output being generated:

```
IrisData =
```

5.1000	3.5000	1.4000	0.2000	1.0000
4.9000	3.0000	1.4000	0.2000	1.0000
4.7000	3.2000	1.3000	0.2000	1.0000

... plus 147 more lines of data. Converting this into more human-readable form, what the data are saying is the following:

Sepal length	Sepal width	Petal length	Petal width	Class
5.1 cm	3.5 cm	1.4 cm	0.2 cm	Iris Setosa
4.9 cm	3.0 cm	1.4 cm	0.2 cm	Iris Setosa
4.7 cm	3.2 cm	1.3 cm	0.2 cm	Iris Setosa
⋮	⋮	⋮	⋮	⋮
5.5 cm	2.4 cm	3.7 cm	1.0 cm	Iris Versicolor
⋮	⋮	⋮	⋮	⋮
5.9 cm	3.0 cm	5.1 cm	1.8 cm	Iris Virginica

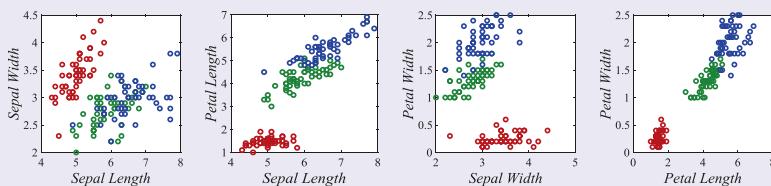
One of our first goals in a pattern recognition problem is to try to understand the nature of the problem:

- What are the class shapes? ... Round?, Elongated?, Split into several pieces?
- How localized are the patterns? ... Compact and small?, Spread out?
- How much do the classes overlap? ... Separated?, Touching?, Overlapping?
- How might we separate the classes? ... Straight line?, Curve?, Complicated?

Our first challenge is that we have four measurements, therefore each data point is a vector in four dimensions, which is quite difficult to visualize. What we *can* do is plot very nicely in *two* dimensions, so let us start by looking at a few 2D examples. In each case we are plotting sample points (circles) from each of the three classes (coloured) based on two of the measurements:

```
% loop over three classes and plot
clf
cols = 'rgb';
p=1;
for f = [ 1 2; 1 3; 2 4; 3 4],      % define the pairs of features to plot
    subplot(1,4,p)
    p = p + 1;
    for c=1:3,
        % plot the points associated with class in variable "c"
        q = find(IrisData(:,5)==c);
        plot(IrisData(q,f(1)), IrisData(q,f(2)),[ 'o' cols(c)]);
        hold on
    end
    xlabel(IrisMeas(f(1)))
    ylabel(IrisMeas(f(2)))
end
```

The resulting plots then look like



Class points: Iris Setosa Iris Versicolor Iris Virginica

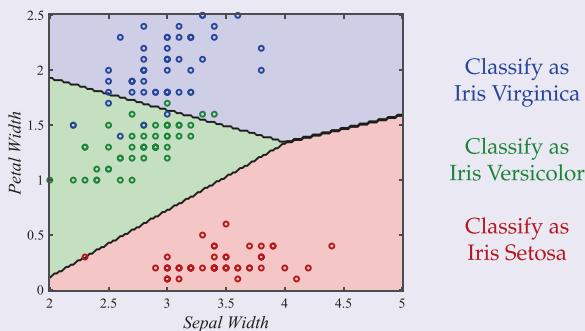
What do we observe here?

1. Iris Setosa (red) is, in general, quite easily separated from Iris Versicolor and Iris Virginica, which overlap in each of the pairs of measurements which we plotted.
2. Class compactness and degree of overlap vary from one measurement to another. The sepal data (leftmost plot) are least compact and most overlapping, and the petal data (rightmost plot) is more compact, and only slightly overlapping.
3. Generally the classes are compact. It is primarily the sepal width measurement (first and third plots) in which the classes exhibit outliers (data points somewhat apart from the rest of the class).
4. The visualization is necessarily incomplete. If the classes overlap in 2D, you *cannot* tell from the 2D plots whether the classes overlap or not in 4D. Ideally we would be motivated to extract two or three features from the four measurements, to allow for better visualization.

A classifier is a function $g()$,

$$g(y) \in \mathcal{C} = \{\text{"Iris Setosa", "Iris Versicolor", "Iris Virginica"}\} \quad (2.11)$$

which maps every possible 4D measurement to one of the three classes. The classifier is not “required” to actually use all four of the measurements so, for example, we might express the classifier as a function of two measurements, y_2 and y_4 :



We see that the classifier does not classify all points correctly — there are green data points in the blue domain, and blue points in the green

domain. The classifier is also not optimal — it would be easy to adjust the classification boundaries slightly to improve the performance. How to deduce such a classifier, and the options we have available to us in selecting a classifier, will be one of the central themes in this book.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links: [Pattern](#), [Pattern recognition](#) [Biometrics](#)

The most fundamental background on the algebra, statistics, and optimization theory needed to understand this book are discussed in [Appendix A](#) through [Appendix C](#).

The concepts in this chapter span much of pattern recognition, but were deliberately discussed only very briefly. Really the entire rest of this textbook represents further reading to the ideas introduced in this chapter.

Sample Problems

Problem 2.1: Short Answer

In your own words, give a short definition of each of the following:

- Data Processing Theorem
- Feature extraction
- Classification

Problem 2.2: Short Answer

Given an example, different from the ones given in this chapter, for each of the four fundamental types of Pattern Recognition problems:

- Model-Based
- Supervised
- Semi-Supervised
- Unsupervised

Problem 2.3: Lab/Computational

[Lab 2](#) code and associated Iris data are available to you from the [the textbook data site](#).

- (a) Using the code from the website, or by writing your own, reproduce the plot on [page 25](#).
- (b) Generalize the figure by plotting *three* of the features in a 3D plot.
- (c) Rotate the 3D plot of part (b) to allow the three classes to be maximally distinguished. What are you actually doing, in rotating the plot this way?

Problem 2.4: Reading — Semi-Supervision

The problem of semi-supervision seems to be located half way between the supervised labelled-data problem and the unsupervised unlabelled-data problem. In principle we might imagine solving it as

1. A labelled-data problem, progressively classifying the unlabelled points to their closest labelled counterparts, or
2. An unlabelled-data problem, performing unsupervised clustering on the data points.

Look up semi-supervision and offer a summary of the prevailing strategies.

References

1. S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. Zitnick, D. Parikh, Vqa: Visual question answering, in *International Conference on Computer Vision (ICCV)* (2015)
2. P. Bellhumer, J. Hespanha, D. Kriegman, Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Trans. Pattern Anal. Mach. Intell.* **17**(7), 711 (1997)
3. P. Brodatz, *Textures: A Photographic Album for Artists and Designers* (Dover, 1966)
4. T. Lin et al., Microsoft coco: Common objects in context. arXiv:1405.0312v3 (2015)
5. A. Georghiades, P. Belhumeur, D. Kriegman, From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intell.* **23**(6), 643 (2001)
6. Y. LeCun, C. Cortes, C. Burges, *The MNIST Database of Handwritten Digits*



Learning

What Does It Mean to Learn?

Really, this is a terribly complicated question, overlapping with metaphysical questions of human consciousness on the one hand, or perhaps theories of education and pedagogy on the other. Even if we restrict the concept of “learning” to computers, we would still need to understand something about the many strategies by which deep neural networks, with millions to billions of parameters, do their learning.

So perhaps we need a slightly simpler question: *What does it mean to have learned?*

In other words, how do we recognize that an algorithm or a method, regardless of how simple or complex, has succeeded in learning? The object being learned is a classifier $g(\underline{x})$, some sort of black-box/function/algorithm/method/concept/system which takes a given feature \underline{x} and returns the associated class C :

$$g(\underline{x}) \in \mathcal{C} = \{C_1, C_2, \dots, C_K\} \quad (3.1)$$

We suppose that we are given a dataset \mathcal{D} of N feature-class¹ pairs

$$\mathcal{D} = \{(\underline{x}_i, c_i)\} \quad \text{where } \underline{x}_i \in \mathbb{R}^n, c_i \in \mathcal{C}, 1 \leq i \leq N \quad (3.2)$$

The most primitive type of learning is just memorization, in which case we would consider $g()$ to have learned from the given dataset based on the number of feature-class pairs it successfully reproduces:

¹ A notational subtlety: c_i , $i = 1, \dots, N$ is the class associated with the i th data point, whereas C_κ , $\kappa = 1, \dots, K$ denotes the κ th class. So C_κ is a label, the name of a class, whereas c_i is a variable, which takes on one of C_1, \dots, C_K .

$$\text{Correct Count} = \sum_i \delta(c_i, g(\underline{x}_i)) \quad \delta(a, b) = \begin{cases} 1 & a = b \\ 0 & a \neq b \end{cases} \quad (3.3)$$

In principle, memorization is actually a credible approach² to developing a classifier, however in general there are two significant limitations:

1. Memorizing is *hard*, and
2. Memorizing is *not enough*.

Memorization is hard because we wish to limit the amount of memory required to represent or store the classifier, and also to limit the computational complexity of operating the classifier. This is a very real concern: there are image databases with 300,000 images occupying 20 gigabytes of storage. As a result, the classifier $g()$ in (3.1) should actually be written as a function of a parameter vector $\underline{\theta}$,

$$g(\underline{x}, \underline{\theta}) \quad \underline{\theta} \in \mathbb{R}^q \quad (3.4)$$

such that *everything* that the classifier needs to remember about $\{(x_i, c_i)\}$ has to be compressed or encoded into vector $\underline{\theta}$, a process which obviously gets more and more difficult as parameter q , the dimensionality of $\underline{\theta}$ in (3.4), is made smaller. Parameter q is referred to as the number of *degrees of freedom*, the number of parameters in the model for $g()$. The learning challenge then becomes to somehow optimize³ $\underline{\theta}$ to find a classifier to maximize the number of examples which are memorized correctly:

$$\hat{\underline{\theta}} = \arg_{\underline{\theta}} \max \sum_i \delta(c_i, g(\underline{x}_i, \underline{\theta})) \quad (3.5)$$

Next, memorization alone is not enough, because the point of learning is not just to remember, it is to *generalize*, to be able to reach correct conclusions about instances which you have *not* seen. For example, one of the first things young children learn are the names of animals, and children very reliably learn to correctly recognize animals, generalizing from having seen only relatively few examples in books or in real life. In other words, to demonstrate that they have *learned*, we would expect children to recognize a picture of a dog or cat which they had never seen before.

The first challenge, in generalization, is what assumptions can you correctly make about the things which you haven't seen? How do you *know* that the patterns which you have observed in the presented data correctly generalize

² For example, the *nearest neighbour* classifier of Section 6.3 essentially memorizes all of its given training data.

³ See Appendix C for an overview of optimization, including a clarification of the "arg max" notation of (3.5).

True Class			Classification		
c_1	c_2	c_3	$g(\underline{x}_1) = \text{Dog}$	$g(\underline{x}_2) = \text{Dog}$	$g(\underline{x}_3) = \text{Cat}$
Dog	Dog	Dog	✓	✓	✗
Dog	Dog	Cat	✓	✓	✓
Dog	Cat	Dog	✓	✗	✗
Dog	Cat	Cat	✓	✗	✓
Cat	Dog	Dog	✗	✓	✗
Cat	Dog	Cat	✗	✓	✓
Cat	Cat	Dog	✗	✗	✗
Cat	Cat	Cat	✗	✗	✓

50% 50% 50% ← Average Performance

Fig. 3.1. NO FREE LUNCH: There is no objectively best classifier, averaged over all possible outcomes all classifiers perform the same as random guessing. A given classifier $g()$ is asked to classify three previously-unseen features $\underline{x}_1, \underline{x}_2, \underline{x}_3$ having true associated classes c_1, c_2, c_3 , where the features are classified into two possible classes of *Cat* and *Dog*. Averaged over all possible truths (left), the classifier $g()$ does no better than random guessing. Indeed, averaged over all truths, *every* classifier will perform the same.

to other data? The short answer is that you *don't*, the topic of the *No Free Lunch Theorem* of Figure 3.1:

Averaged over all possibilities for the unseen data, no classifier generalizes better than any other!

In other words, *all* classifiers are just guessing how to generalize, and in the absence of any pattern or structure or prior knowledge, one classifier is as good as another. The reason classifiers *do* work, and why the discipline of pattern recognition actually makes any sense, is because nearly all data we encounter and seek to generalize *do* possess some sort of pattern or structure. However it is important to keep in mind that any claim about some classifier being "better" than another always needs to be taken in context: for a particular problem or dataset a given classifier may very well be better than another, however the No Free Lunch Theorem asserts that a given classifier can never be objectively be better, universally, over all possible patterns.

Since we have not yet developed any classifiers, it will be simpler to demonstrate the principles of learning in a familiar context, such as fitting a polynomial to a set of data points, as illustrated in Figure 3.2. A p th order polynomial is described by $q = p + 1$ parameters⁴ or degrees of freedom; in the notation of (3.4), we would write the polynomial as

$$g(x, \underline{\theta}) = \theta_p x^p + \theta_{p-1} x^{p-1} + \dots + \theta_1 x^1 + \theta_0 x^0. \quad (3.6)$$

⁴ Recall that a parabola (*second* order) has *three* coefficients.

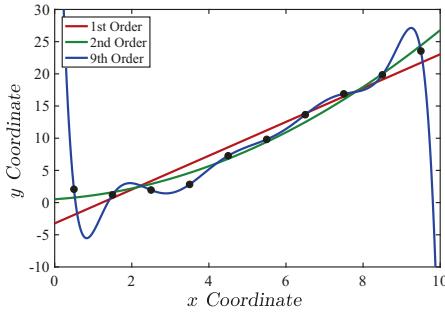


Fig. 3.2. What does it mean to *fit* a model to data? Here we have ten data points (black dots), which come from some unknown model with added random noise. We fit polynomials to the points, for which the first-order (linear regression), second-order (parabolic regression), and ninth-order fits are shown. A p th-order polynomial can always be found that passes through $p + 1$ given points, so here the ninth-order polynomial fits the points exactly, however it seems like a very unlikely generalization of the data. In general, per the principle of Occam's Razor, we prefer the simplest or lowest-order model which is consistent with the data.

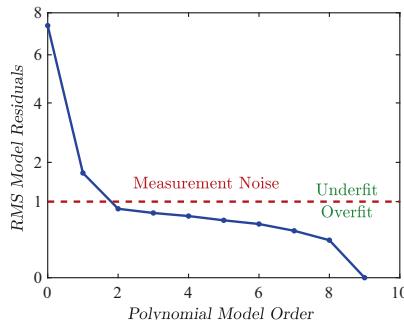


Fig. 3.3. Following up on Figure 3.2, we can plot the root-mean-square (RMS) difference between the polynomial fit and the data points, as a function of the polynomial order. In this case the measurement noise level was assumed to have been provided, it was not learned.

In general we wish to fit the data with the lowest-order polynomial which is consistent with the data, sometimes referred to as the *Principle of Parsimony* or *Occam's Razor*.

Given a data point (x, y) , we measure the goodness of fit of the polynomial by examining the residual

$$\text{Residual} = y - g(x, \underline{\theta}), \quad (3.7)$$

measuring the inconsistency between the model and a given point. Given N data points (x_i, y_i) , the average squared residual, the root-mean-square (RMS) of the model residuals, is computed as

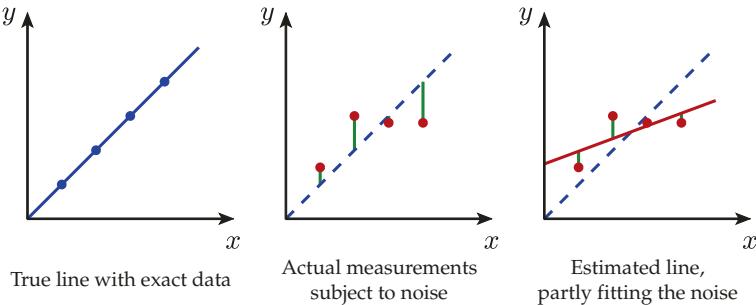


Fig. 3.4. OVERFITTING: Any learning, whether of linear regression (here) or a pattern recognition classifier (Figure 3.5), is said to be overfitting if it begins to tune its parameters to the behaviour of the noise, rather than of the underlying phenomenon we wish to learn. The estimated line (red) is quite plausible, given the four data points (red dots), however it is clear how the line has accommodated (fit) the noise, to make the residuals (green) smaller.

$$\text{RMS}(\underline{\theta}) = \left(\frac{1}{N} \sum_{i=1}^N (y_i - g(x_i, \underline{\theta}))^2 \right)^{1/2}, \quad (3.8)$$

as plotted in Figure 3.3. Since all p th order polynomials are included in the possible $(p+1)$ th order polynomials, as the order is increased the polynomial fit *cannot* get worse, meaning that the RMS residual cannot increase, as we see in Figure 3.3. In this example the added noise has a standard deviation of $\sigma = 1$, meaning that for the correct model, $\text{RMS}(\underline{\theta}_{\text{exact}}) = 1$. As a result,

- Any RMS difference below σ must be overfitting, meaning that $g(x, \underline{\theta})$ is partly fitting to noise, by taking some of the noise into account when learning $\underline{\theta}$;
- Any RMS difference above σ suggests that the learned model has not adequately generalized, or has not been given adequate flexibility in $\underline{\theta}$ (enough degrees of freedom q) to capture the variations that need to be captured.

Although Figure 3.3 makes the distinction appear simple, one of the challenges in applying the Principle of Parsimony or Occam's Razor is that we normally *don't know* the noise statistics, and therefore don't actually know whether we are over- or under-fitting.

The model $g(x, \underline{\theta})$ and the data (x_i, y_i) (for polynomial fitting) or (x_i, c_i) (for classification) each possess some number of degrees of freedom. The degrees of freedom in the model we explicitly control via q , the number of values in parameter vector $\underline{\theta}$. In contrast, the degrees of freedom in N noisy data points consists of some number (typically *much* less than N) of degrees associated

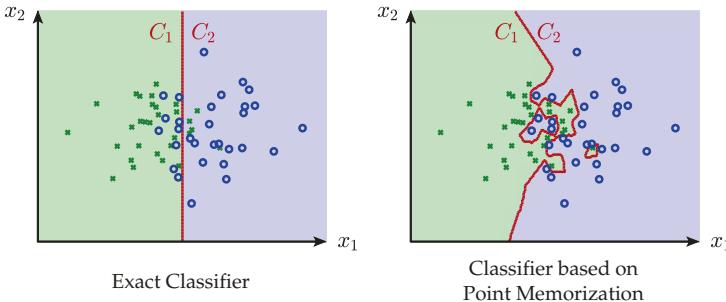


Fig. 3.5. OVERFITTING: As in Figure 3.4, but here for a pattern recognition classifier. We will have to wait for Chapter 6 for the details of the classifier to be discussed, however the principle is the same as in regression: Any learning is overfitting if it tunes its parameters to the behaviour of the noise. That tuning is obvious here, in that the memorized classifier (right) is tuning its decision (coloured background) on the basis of individual training points, significantly distracted from the correct or ideal classification (left).

with the underlying (unknown) behaviour, plus N degrees,⁵ one for each added noise term.

Each increase in q , each degree of freedom added to the model, gives us one more parameter of flexibility. The key idea is that such an increase should be associated with a meaningful improvement in the modelling of the underlying behaviour, leading to a meaningful drop in the RMS difference of (3.8), a drop *greater* than what we would associate with fitting one more degree of freedom of noise. That is, we require

$$\text{RMS}(\underline{\theta} \in \mathbb{R}^{q+1}) < \text{RMS}(\underline{\theta} \in \mathbb{R}^q) - (\text{RMS Drop from Noise Fit}) \quad (3.9)$$

Precisely this idea is the basis of the *Akaike Information Criterion* (AIC), which asserts a penalty with every increase in q . As a result, rather than minimizing $\text{RMS}(\underline{\theta})$, which is easily accomplished by overfitting and allowing $q \rightarrow \infty$, we now choose q to minimize⁶

$$\text{AIC}(q) = \text{RMS}(\underline{\theta} \in \mathbb{R}^q) + \text{Penalty}(q) \quad (3.10)$$

as illustrated in Figure 3.6. The Akaike Information Criterion is an elegant idea, but can be difficult to apply. Really, the easiest way to objectively test the generalization ability of a learned classifier is to *not* train with all of the available data, as is illustrated by the sequence of learning frameworks presented in Example 3.3. That is, to keep some of the data hidden from the

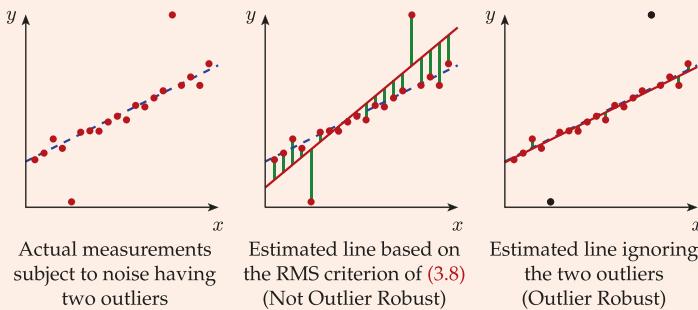
⁵ Assuming the noise to be independent. If the noise terms are correlated then the effective degrees of freedom is less than N .

⁶ The actual formulation of the AIC is somewhat more complicated than as shown in (3.10); the interested reader is referred to the Further Reading.

Example 3.1: Robustness in Learning

Figure 3.4 illustrated the phenomenon of overfitting, whereby the learned parameters (such as the slope and intercept in regression) fit to the noise. Presumably as the number of data points, N , increases, the problem of overfitting slowly disappears, since there are more and more data points constraining the values of the parameters? If all of the data points are subject to Gaussian noise with the same variance then yes, the problem of overfitting decreases as N increases. However there are many contexts in which there is the occasional measurement which is an *outlier*, wildly inconsistent from other measurements, perhaps due to data corruption or instrument failure.

A human, asked to draw a straight line through data points, having a small number of outliers, would most likely ignore the outliers, treating them as irrelevant or failed measurements, however the RMS criterion of (3.8) has no such ability, computing the squared-distance to *all* measurements, making it indeed quite sensitive to outliers:

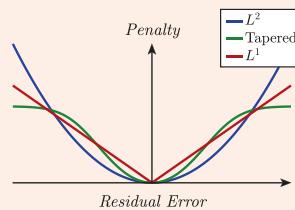


Why is the middle line so badly skewed, when there are only two outliers and eighteen good data points? The answer is that the RMS criterion of (3.8) seeks to minimize the average *squared* distance, and if an outlier is far away, its *squared* error can be *very* large, overwhelming or at least seriously influencing the effect of the many non-outlier data points.

So how do we make learning *robust* to outliers? Really this is a vast topic, which we can only begin to touch here. A variety of approaches is possible:

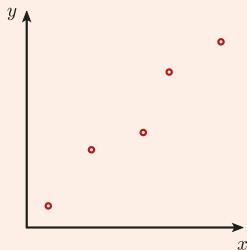
1. Detect and remove the outliers (as was done in the right-most panel, above),
2. Choose parameters in $\underline{\theta}$ insensitive to outliers, or
3. Choose an optimization metric insensitive to outliers.

As an illustration of the third point, the squared (L^2) criterion of (3.8) could be replaced with other functions, such that the residuals ($y_i - g(x_i, \underline{\theta})$) could be penalized in different ways, as sketched here. For example, an absolute (L^1) or a tapered function penalize large residuals much more modestly than the L^2 criterion, but at greater computational complexity.

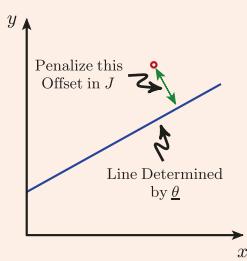
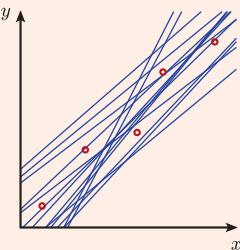


Example 3.2: Regression and Classification

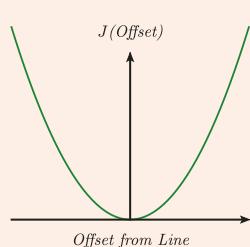
Much of this chapter has looked at learning in the context of a simple regression problem, so let's start there:



Linear regression takes a set of points (x_i, y_i) (red, left) and seeks a line of best fit. Each line is parameterized by vector $\underline{\theta}$, for example containing the slope and intercept. Of the infinitely many possible lines (blue, right), we need some strategy by which to find the optimal line.



The key is that we need an objective function $J(\underline{\theta} | (x_i, y_i))$, which measures the goodness of fit of a line, represented by $\underline{\theta}$, with the given points (left). A common objective (right) is to penalize the square of the offset (the shortest distance) from each point to the line.

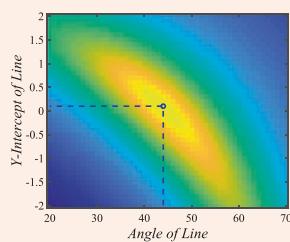


For each possible line, described by

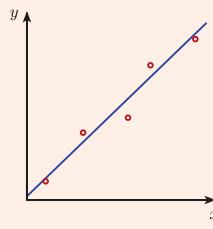
$$\underline{\theta} = \begin{bmatrix} \text{Angle of Line} \\ \text{Y-Intercept of Line} \end{bmatrix}$$

we can assess the penalty associated with the line as

$$\text{Penalty}(\underline{\theta}) = \sum_i J(\text{Offset from line } \underline{\theta} \text{ to } (x_i, y_i))$$



The penalty is plotted, right, as a function of angle and intercept, with dark blue indicating a poor fit, and bright yellow the best. This is an optimization problem (Appendix C), in this case performed by exhaustively searching the domain of $\underline{\theta}$. The optimal (minimized penalty) location is marked, implying an optimal y-intercept just slightly above zero, and a line at an angle of just under 45° , leading to the optimal line as plotted at right.



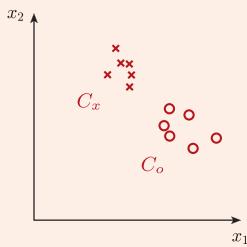
It is important to understand that this line is optimal *with respect to the penalty function J*: changing the definition of J will lead to a *different* optimal line.

Having discussed regression in some detail, we now follow the *same* sequence of steps for classification. The reader is encouraged to repeatedly compare

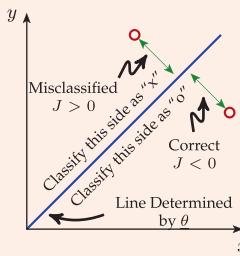
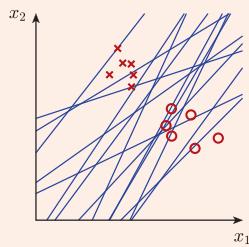
Example continues ...

Example 3.2: Regression and Classification (continued)

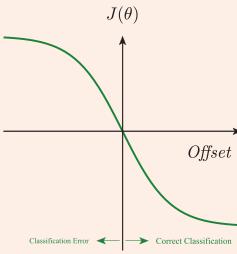
the figures and descriptions on this and the previous page. Regression and classification are, in fact, remarkably similar:



Classification takes two sets of points $\underline{x}_i \in C_x$ and $\underline{x}_j \in C_o$, for classes C_x and C_o , left, and seeks an optimal line of best separation. Of the infinitely many possible separating lines (blue, right), we need some strategy by which to find the optimal line, for which we still need an objective function $J(\underline{\theta} | \{C_x, C_o\})$.

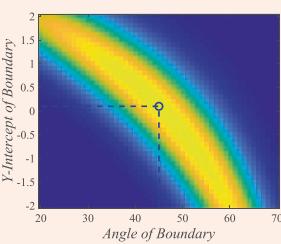


However rather than wanting points near the line, we now need to have points on the correct side of the line, preferably far from the line. Since we are minimizing J , we wish to reward ($J < 0$) points on the correct side of the line, and penalize ($J > 0$) points on the incorrect side, as sketched (left).

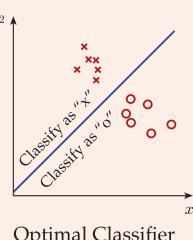


We are most concerned with having points correctly classified, and less concerned with having them far from the boundary, therefore the slope of J is maximized at zero, with the degree of benefit ($J < 0$) or penalty ($J > 0$) tapering as we move far from the line (above right). Each possible line is parameterized by $\underline{\theta}$ as before, with the penalty computed to both classes,

$$\text{Penalty}(\underline{\theta}) = \sum_i J(\text{Offset from line } \underline{\theta} \text{ to } \underline{x}_i \in C_x) + \sum_j J(\text{Offset from line } \underline{\theta} \text{ to } \underline{x}_j \in C_o)$$



The penalty is plotted, above right, as before exhaustively searching the domain of $\underline{\theta}$. The optimal (minimized penalty) location is marked and the corresponding optimal classifier is plotted, right.



As before it is important to understand that this classifier is optimal *with respect to the penalty function J* : changing J will lead to a *different* optimal classifier.

Further Reading: Finding a dividing line between two classes is the subject of [Chapter 10](#), and the algebraic commonality between regression and classification is developed further in [Examples 10.1](#) and [8.1](#) and in [Appendix D](#).

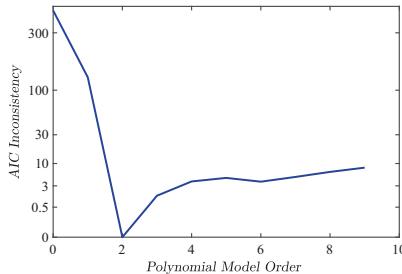


Fig. 3.6. AIC: Following up on [Figure 3.3](#), the Akaike Information Criterion (AIC) of [\(3.10\)](#) gives us an objective way of choosing the appropriate model size or complexity. To simplify the plot, here we show $\text{AIC}(q) - \text{AIC}(2)$, the deviation of the AIC from its optimum at $q = 2$.

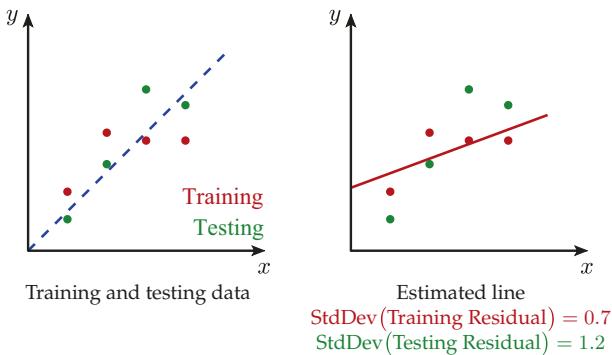


Fig. 3.7. We have separate training and testing data (left), such that the learned model (red line) is deduced from the training data, but assessed against the testing data. Observe the degree to which the estimated line fits to the noise, based on the difference between the fit to training data (overfit, under-reporting model inconsistency) and the fit to testing data (which is an accurate, objective assessment).

learning process, making it impossible to overfit to the hidden data, and then to see how well the classifier performs on that unseen data. The principle of separating *training* and *testing* data is illustrated in model III in [Example 3.3](#) and simulated in [Figure 3.7](#): the model will naturally tend to fit the noise of the *training* data, but cannot do so with the *testing* data. Since a model may, in principle, memorize all of the training data, the performance of the model on the training data is nearly irrelevant, and it is the model performance on the testing data which would be reported in any objective assessment.

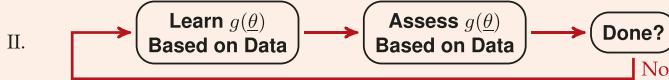
The danger of separating into training and testing data is that there is a temptation to observe the outcome of the testing assessment, and then to return to re-do the training. The problem is that this iterative process begins to fit the model to the testing data, in principle once again allowing us to overfit to the testing data.

Example 3.3: An Overview of the Use of Data in Learning

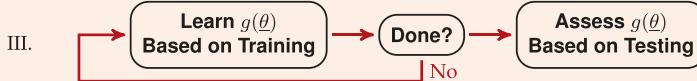
We begin with the simplest possible approach to learning, from a single dataset:



It is difficult to know whether $g(\theta)$ is overfit or not, and the assessment will not be able to reveal underfit/overfit to us. Frequently the learning process is iterative, such that the learning continues until some assessment criterion is met:

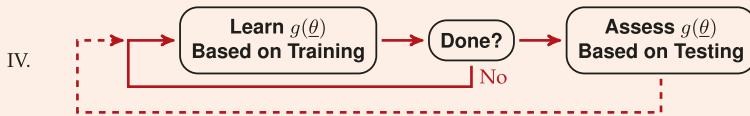


This approach is, of course, completely ill-advised!, since repeatedly learning and assessing on the same dataset will almost certainly lead to overfitting. Instead, we really need to separate the data into separate training and testing sets:



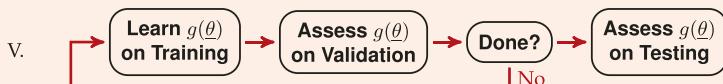
The test data *does* now give us a meaningful assessment of the quality of the learning, and the performance of the learned model on the training and testing sets will allow us to have some sense of the degree of overfitting to the training data, versus successful generalization to the testing data.

A poor practice is to have undertaken separate training/testing, as above, but then to have gained insight from the test results, which influences how $g(\theta)$ is learned in a *next* iteration, with the undesirable consequence that learning is beginning to fit the testing data:



A related approach is to, say, learn a number of models $g_1(\theta_1), g_2(\theta_2), \dots$ from the training data, then test *all* of the models on the testing data to see which one is the best. This sounds innocent enough, however consider learning *millions* of models ... the “best” one will then be the model which happened to best fit, likely *overfit*, the testing data.

It is, of course, perfectly legitimate to wish to train many models, or to iteratively train a model and then to gain insight from testing. However the only way to do so without tainting the learning process, without any possibility of implicitly fitting the testing data, is to divide the data into *three* parts, of training, validation, and testing:



A solution (as outlined in model V in [Example 3.3](#)) is to further separate the available data into training, validation, and testing data, such that a model is learned from (and likely overfitting) the training data, repeatedly assessed for generalization (and possibly slightly overfitting) to the validation data, allowing re-learning, but then assessed *once* on the testing data, at the end of learning.

The final difficulty is that there is rarely enough data, which presents a dilemma in dividing the data into training, validation, and testing pieces. In general, the error bars⁷ (the standard deviation) in estimating the error of a classifier is

$$\text{Estimated Classifier Error} = \text{True Classifier Error} \pm \frac{\sigma}{\sqrt{N}} \quad (3.11)$$

given N data points of variability σ . In other words, to cut the uncertainty in *half*, we need *four* times as much data. And *each* of training, validation, and testing will be needing as much data as possible, in order to effectively perform their respective tasks (of learning, validation, and final assessment). In other words, given N data points in total, we need to allocate these data such that

$$N_{\text{Training}} + N_{\text{Validation}} + N_{\text{Testing}} = N \quad (3.12)$$

with the dilemma that increasing N_{Training} leads to a better classifier but with a limited ability to assess it, whereas increasing N_{Testing} leads to a better assessment, but of a poorer classifier.

We need to better understand how we will undertake classifier assessment. For a regression-type model, the model prediction is a continuous-valued $g(\underline{x})$, such that we would like to minimize the squared-residual $(y - g(\underline{x}))^2$, as was discussed around [\(3.8\)](#). In contrast, for a classification-type model, the model prediction is a *discrete*-valued

$$g(\underline{x}) \in \mathcal{C} = \{C_1, C_2, \dots, C_K\}. \quad (3.13)$$

In classification it does not help to be *close* to the right class; the classification is either correct, or it is incorrect. As a result, the most basic assessment of a classifier is its overall probability of classification error

$$\mathbf{P}_g(e) \simeq \frac{1}{N} \sum_{i=1}^N \left(1 - \delta(c_i, g(\underline{x}_i)) \right) \quad \text{given test data} \quad (\underline{x}_1, c_1), \dots, (\underline{x}_N, c_N) \quad (3.14)$$

⁷ The error bars in [\(3.11\)](#) are derived in [Section 7.2](#).

There are at least three nuances present in (3.14):

1. $\mathbf{P}_g(e)$ in (3.14) represents the *exact* (actual) classification error of classifier g , whereas the evaluation over the testing data (\underline{x}_i, c_i) is a sample *estimate*⁸ of $\mathbf{P}_g(e)$, hence the “ \simeq ” in (3.14). The error in the estimate of $\mathbf{P}_g(e)$ will be proportional to $1/\sqrt{N}$.
2. We may not have the same amount of sample data from each class, the so-called *class imbalance* problem. Such a problem is particularly common in medical diagnoses, where most tests return negative (no problem found), and only rarely positive (some medical issue identified). The estimated $\mathbf{P}_g(e)$ in (3.14) will be based on how the testing data are distributed over the K classes, which may or may not be what we want.
3. We may have biases in our sample data, separate from the imbalance problem, such that the testing data may be biased in terms of its distribution *within* a class. For example for face detection in an image patch, addressing the classification problem

$$C_0 = \text{No face is present} \quad C_1 = \text{Face is present in image patch} \quad (3.15)$$

if the training and testing data both contain primarily examples of white male faces, the returned $\mathbf{P}_g(e)$ may be an unreliable indicator of the actual performance of detector $g()$ when given images from a broader cross-section of society.

The overall classification error $\mathbf{P}(e)$ is convenient, being a single number which summarizes the performance of the classifier, however such a summary greatly obscures *where* the errors are taking place, which classes are most frequently confused with other classes. Our most valuable tool in this regard is what is known as a *confusion matrix*, which lists all pairwise probabilities of the form

$$\begin{aligned} \mathbf{P}_g(C_j | C_k) &= \text{Probability that classifier } g() \text{ says } C_j \text{ given a data point from } C_k \\ &\simeq \sum_{i=1}^N \underbrace{\delta(C_j, g(\underline{x}_i))}_{\text{Did } g() \text{ say } C_j?} \quad \underbrace{\delta(C_k, c_i)}_{\text{Is } c_i \text{ from class } C_k?} \quad / \underbrace{\sum_{i=1}^N \delta(C_k, c_i)}_{\# \text{ data points in } C_k} \end{aligned} \quad (3.16)$$

where, as in (3.14), each of these probabilities in (3.16) is *estimated* from testing data, and therefore subject to the nuances just discussed. The confusion matrix of probabilities then becomes

⁸ See Appendix B, particularly Section B.6, for a brief overview of sample statistics.

$$\begin{array}{c}
 \text{Is classified as ...} \\
 \begin{array}{cccc}
 C_1 & C_2 & \cdots & C_K
 \end{array}
 \end{array}
 \quad
 \begin{array}{ccccc}
 \text{True Class} & C_1 & \boxed{\begin{array}{cccc}
 P_g(C_1|C_1) & P_g(C_2|C_1) & \cdots & P_g(C_K|C_1)
 \end{array}} & & (3.17) \\
 \vdots & \vdots & \vdots & \vdots \\
 C_K & \boxed{\begin{array}{cccc}
 P_g(C_1|C_K) & P_g(C_2|C_K) & \cdots & P_g(C_K|C_K)
 \end{array}}
 \end{array}$$

Depending on what sort of presentation is more intuitive or more informative, it is also common to have confusion matrices listing pairwise classification counts

$$\begin{aligned}
 S_g(C_j|C_k) &= \text{Number of times that classifier } g() \text{ says } C_j \text{ given data from } C_k \\
 &= \sum_{i=1}^N \delta(C_j, g(\underline{x}_i)) \cdot \delta(C_k, c_i)
 \end{aligned}
 \quad (3.18)$$

for which the confusion matrix then becomes

$$\begin{array}{c}
 \text{Is classified as ...} \\
 \begin{array}{cccc}
 C_1 & C_2 & \cdots & C_K
 \end{array}
 \end{array}
 \quad
 \begin{array}{ccccc}
 \text{True Class} & C_1 & \boxed{\begin{array}{cccc}
 S_g(C_1|C_1) & S_g(C_2|C_1) & \cdots & S_g(C_K|C_1)
 \end{array}} & & (3.19) \\
 \vdots & \vdots & \vdots & \vdots \\
 C_K & \boxed{\begin{array}{cccc}
 S_g(C_1|C_K) & S_g(C_2|C_K) & \cdots & S_g(C_K|C_K)
 \end{array}}
 \end{array}$$

The main difference in interpreting (3.19) versus (3.17) is that the counts $S_g()$ are integers, and therefore more easily read by eye than probabilities, and that $S_g()$ is an actual count from a data set (no “ \simeq ” in (3.18)), and not an estimate. In any given problem, whether we are using probabilities (3.17) or counts (3.19) should be clear from the context; however if not, we will refer to a confusion matrix of probabilities or of counts, respectively.

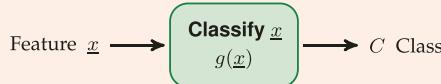
We will return to confusion matrices and their interpretation in future chapters. Briefly, however, the ideal is to have all of the off-diagonal values (which correspond to classification errors) to be zero, or close to zero. Non-zero off-diagonal values correspond to errors, and any large off-diagonal values identify those pairs of classes which significantly overlap, or at least those classes which classifier $g()$ finds the most difficult to distinguish.

Confusion matrices illustrate the *wrapper* principle of Example 3.4, widely used in this book. The idea behind wrapper-based methods is to separate the conceptual complexity of one layer (say, that of classification) with that of another (say, that of classifier assessment). That is, in assessing a classifier (3.14), (3.16), (3.18) we would prefer not to need to have to know the detailed behaviour of the classifier itself, that is whether the classifier is distance-based (Chapter 6), statistical (Chapter 8), discriminant-based (Chapter 10), or aggregated as an ensemble (Chapter 11). We will see wrapper methods next used for feature selection in Section 5.2.

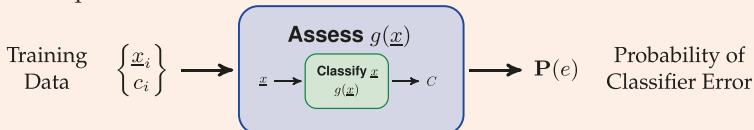
Example 3.4: Wrapper-Based Learning

Wrapper-based approaches involve wrapping some objective, such as measuring or minimizing the probability of classification error, around some other operation or function, such as a classifier.

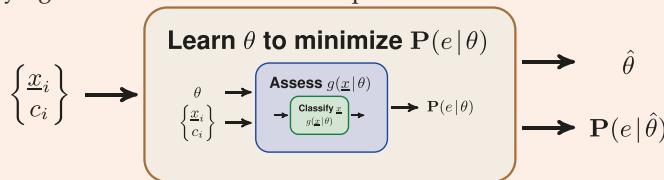
For example, suppose we begin with some given classifier $g()$:



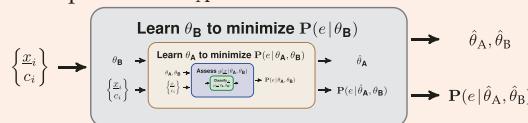
In some cases we may study the mathematical properties of a given classifier in detail, however in general we will prefer to *wrap* an assessment around the classifier, such that the assessment does not need to know anything about the internal workings of the classifier, it just calls the classifier many times to assess its performance:



Nearly all classifiers need to learn, having a parameter θ to be tuned, for example on the basis of given training data. So now we wish to have an optimization process (further described in [Appendix C](#)), a minimization, *wrapped* around the assessment, in order to minimize the probability of classification error, but where the minimization does not need to know any of the details of the underlying classification or assessment processes:



Although it may seem conceptually cumbersome, in many cases we may wish to wrap one optimization process around another, such that the error minimization as a function of parameter θ_B is *wrapped* around a separate minimization over parameter θ_A :



For example, suppose that θ_A determines *where* the classification boundary is placed, whereas θ_B determines *how many* boundaries there are. Then θ_B affects the degrees of freedom in the classifier, leading to issues of overfitting and model order selection, as discussed throughout this chapter, whereas θ_A has no influence on degrees of freedom. Furthermore θ_B is a single integer, whereas θ_A is a vector of real values, so learning/optimizing θ_A, θ_B might be based on very different approaches.

Case Study 3: The Netflix Prize

How does a website determine “Other books you might like to read” or “If you liked this movie, here is another you might enjoy.” Such questions are, of course, central to much of online shopping, since an online site selling thousands to millions of different products has to have some targeted way of presenting a small but plausible list to a given person. That Facebook or Google now use pattern recognition⁹ for targeted advertising, friend considerations, or news articles is taken for granted,¹⁰ however in 2006 such user-profiling was far less developed.

In 2006 Netflix was a rapidly growing company and faced the challenge of how to recommend movies and television programs. In a move that brought a remarkable degree of interest and publicity, the 2006 Netflix Prize was one of the first large-scale pattern recognition prizes, an open competition with a US\$1,000,000 prize as an incentive to researchers around the world.

So what is the problem that Netflix wanted to solve? Suppose each movie can receive a grade G from one to five stars. Ideally Netflix would like to know which movies people would *like* to see, which would mean those movies to which a person would assign a high grade. That is, we would like a predictor of the form

$$\text{Movie } M \quad \text{User } U \quad \xrightarrow{\text{Predict}} \quad \text{Grade } G_{MU} \quad (3.20)$$

The grade G can be treated as continuous, $1 \leq G \leq 5$, in which case the predictor is more of a regression, or G can be treated as discrete, $G \in \{1, 2, 3, 4, 5\}$, in which case the predictor is more of a classifier.

There was also a desire to take into account that people’s tastes might change over time, so the actual predictor included a date attribute:

$$\text{Movie } M \quad \text{User } U \quad \text{Date } D \quad \xrightarrow{\text{Predict}} \quad \text{Grade } G_{MUD} \quad (3.21)$$

Since many people provide a grade after watching a movie, Netflix already had an extensive database of entries

$$\{M_i \quad U_i \quad D_i \quad G_i\} \quad (3.22)$$

⁹ Almost universally, but exceptionally vaguely, referred to as “algorithms” in articles discussing targeted advertising. Really almost *anything* running on a computer is an algorithm.

¹⁰ “Taken for granted” should not be understood to mean “acceptable” or “ethical”. There are many activities — identity theft, political meddling, or algorithms presenting gender or racial biases — which raise serious ethical concerns, as further discussed in [Case Study 9](#) and [Case Study 11](#).

and the idea was to use this database in order to learn a movie grade predictor. Netflix was, of course, greatly concerned with the overfitting problem discussed in this chapter: they wanted a predictor that *generalized* movie ratings, not just memorized them from the database, and so the data were separated into training and testing portions, as in [Example 3.3](#).

However Netflix was *also* concerned about overfitting to the testing data, as shown in case IV in [Example 3.3](#), so it was important to prevent a given competitor from using their performance on the testing data to feed back into the learning system, thereby (after many iterations on the testing data) slowly possibly overfitting the data.

There are several strategies that can be used to prevent overfitting:

- Many modern contests require that you submit your computer code to the contest organizers, and that they provide the test data (unknown to the competitor) to the code and return only the overall result. This does, however, require a degree of standardization of programming language and program interfacing, and is not always convenient.
- An alternative is to provide the testing dataset $\{M_i \ U_i \ D_i\}$ to the competitor, who then submits the predicted grades $\{G_i\}$ to the contest organizers, but that there is a limit as to how often grades can be submitted, which limits the number of times that a competitor could attempt to close the loop, so to speak, from the testing data to the predictor. In the Netflix Prize, only one submission could be made per day.
- Finally, one can make overfitting much more difficult by reporting the performance for only a *fraction* of the testing dataset. For the Netflix Prize, performance was reported only for half of the testing data, but where the competitor did not know *which* of the entries were being included in the reported score, and which not.

The various datasets used were organized as follows:

Data Set	Number of entries	Comment
Training set	99.1 Million	
Probe set	1.4 Million	Statistically similar to test and quiz datasets
Test set	1.4 Million	Score is reported back to contestant
Quiz set	1.4 Million	Score is not reported

Particularly challenging in this prediction problem was the significant variance.¹¹ Although users rated about 200 movies, on average, there was one user who rated over 17,000. Similarly whereas movies were rated by approximately 5000 users, on average, there were some movies that had as few as only three ratings.

The contest ended in 2009 when two teams successfully topped the performance benchmark that Netflix had indicated would be required to be eligible for the \$1,000,000 grand prize.

Since that time, and particularly with the widespread adoption and standardization of methods for deep learning, such pattern-recognition contests have become quite common, appearing as part of many academic conferences and on platforms offering industry-sponsored challenges.

Lab 3: Overfitting and Underfitting

We would like to explore the appearance of underfitting/overfitting in classifier learning. Since we have not yet studied any classifiers in detail, we will assume the existence of a two-class kNN (*k* Nearest Neighbour) function:

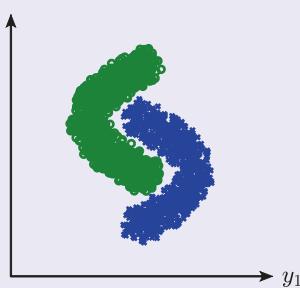
```
% learn a two-class knn classifier and return the classification of the
    test data
% c = knn(training_set1 , training_set2 ,k, test_data)
```

In a nutshell, the kNN classifier (discussed in [Chapter 6](#), particularly [Section 6.3](#)) looks at the *k* closest points and makes a classification decision based on those points:

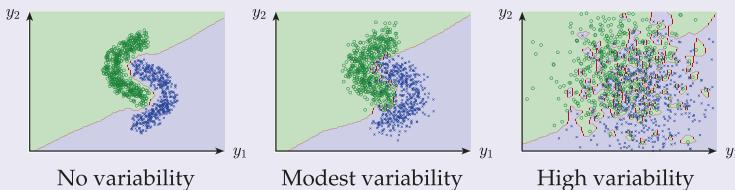
- At $k = 1$, the classifier is highly responsive to local patterns in the data, but is greatly susceptible to overfitting by memorizing each data point.
- At $k \gg 1$, the classifier is robust to noise, by ignoring individual outlying points, but may be susceptible to underfitting by being unresponsive to local behaviours in the data.

¹¹ Although outside the scope of this text, high-variance features are inherent to many social phenomena, like movie popularity, because of a phenomenon known as *preferential attachment*, which leads to heavy-tail or power-law distributions (briefly discussed in [Section B.5](#)), commonplace in economics and in the fascinating field of Complex Systems [5].

```
load TwoArc
p1train = p1(:,1:500); p1test = p1(:,501:end);
p2train = p2(:,1:500); p2test = p2(:,501:end);
clf; plot(p1(1,:), p1(2,:), 'og');
hold on; plot(p2(1,:), p2(2,:), 'xb');
```



The “TwoArc” data (available from [the textbook data site](#)) are plotted at right: we have two classes (green, blue) which are not *linearly separable*, meaning that the classes cannot be separated by a straight line. To obtain a degree of intuition, the following panels show the behaviour of the $k = 1$ classifier for three different amounts of variability added to the data points. A clear responsiveness to class shape can be seen in the leftmost panel, whereas overfitting can clearly be seen in the rightmost panel since the noisy data lead to many small islands of classification decision:



We have two parameters in our problem: the number of points k that the classifier examines, and the amount of added variability. So we begin by defining the ranges of each to test:

```
% Define noise and classifier parameter ranges
srange = linrange(0,sqrt(2),20).^2;
krange = [1:5 round(exprange(6,499,15))];
```

We have created a linear range for variability s and an exponential range for parameter k , based on the following functions:

```
% give an exponentially distributed range with a requested # samples
% r = exprange( lo, hi, numsamples )
function r = exprange( lo, hi, num )
    r = exp(log(lo):(log(hi)-log(lo))/(num-1):log(hi));
end
% give a uniformly distributed range with a requested # samples
% r = linrange( lo, hi, numsamples )
function r = linrange( lo, hi, num )
    r = lo:(hi-lo)/(num-1):hi;
end
```

We will be counting the number of times the classifier misclassifies, so we need to initialize those arrays:

```
% Initialize error results arrays
err = zeros(numel(srang),numel(krange)); errn = err; errt = err;
```

The following code runs the classifier many times, for different amounts of variability and parameter k . Since the variability is random, there will be fluctuations in the results, so the whole processes is run 20 times (along the lines of (3.11)):

```
% Loop over passes to average out statistical variations
passes = 20;
for p=1:passes,
    % Loop over noise levels
    for si = 1:length(srang),
        % Loop over classifier parameter k
        for ki = 1:length(krange),
            p1n = p1train+srang(si)*randn(size(p1train));
            p2n = p2train+srang(si)*randn(size(p2train));
            p1tn = p1test+srang(si)*randn(size(p1train));
            p2tn = p2test+srang(si)*randn(size(p2train));

            % All kNNs trained on the same noisy training data

            % kNN tested on noisy training data
            errt(si,ki) = errt(si,ki) + sum([(knn(p1n,p2n,krange(ki),p1n)
                ~= 1) ... (knn(p1n,p2n,krange(ki),p2n) ~= 2)]);

            % kNN tested on noisy testing data
            errn(si,ki) = errn(si,ki) + sum([(knn(p1n,p2n,krange(ki),p1tn)
                ~= 1) ... (knn(p1n,p2n,krange(ki),p2tn) ~= 2)]);
        end
    end
end
```

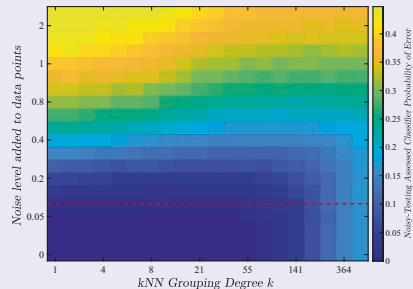
We counted the number of classification errors, so to compute probabilities we divide by the total number of times the classifier was tested, which is the number of passes multiplied by the number of test points:

```
% Convert from counts to probabilities
errn = errn / (passes * (size(p1test,2)+size(p2test,2)));
errt = errt / (passes * (size(p1test,2)+size(p2test,2)));

% Visualize kNN error assessment
clf
```

```
imagesc(errn)
colorbar;
axis xy
```

Further code for axis labels etc. is not shown, for brevity. Note from the axis values that both axes are compressed at the high end, to bring out details at small k and small noise level. There are quite a number of interesting behaviours to observe here.

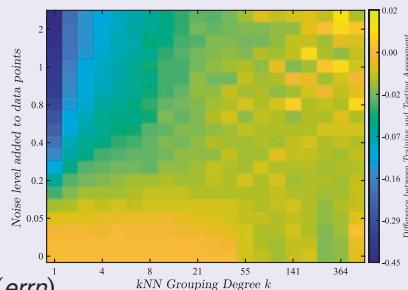


The overall behaviour shows an increase in classifier error as the data point variability increases (up), which makes sense: the more variable (noisy) the class, the more the classes overlap.

For a sufficiently small amount of class variability (bottom), the behaviour seems independent of noise level. This effect stems from the classes being slightly separated, such that it takes a certain amount of variability (red dashed line) before the data points in the classes actually begin to overlap; below the red line the classifier performance is not particularly affected by the variability.

For low variability the performance is best for small k , whereas for high variability the performance is best for large k . This behaviour is consistent with the description of k NN at the beginning of this lab (page 46): for small k the classifier is sensitive to local details, and as k increases the number of data points considered by the classifiers similarly increases. For low variability the classes do not overlap, and a good classifier needs to carefully locate its classification boundary between the bent cluster shapes, requiring small k . For high variability the clusters badly overlap, and focusing on only the local points (small k) could lead to overfitting, since with so much overlap we need to be looking at relatively many points (large k) to reliably know how to classify.

So how can we quantify the degree of overfitting? The classifier saw only the training data during the learning phase, so we can compare how the classifier performed when tested on training data (*errt*) as opposed to when tested on the unseen testing data (*errn*).



The plot, right, shows the difference $\text{err}_t - \text{err}_n$, so that a negative value (blue) implies overfitting. Overfitting appears most striking for small k and high variability, where the classes significantly overlap and a small value of k leads to fitting details in the training data, as we saw on page 47.

The irregularity in the upper-right is due to statistical variability. The classifier learning and testing was run repeatedly (twenty times), which produced reliable results most of the time, however for large variability and large k the statistical fluctuations are apparently so significant that *many* more learning-testing runs would have been required.

Summary

A model $g(\underline{x}, \underline{\theta})$ seeks to learn a vector of parameters $\underline{\theta}$ to match given training data (\underline{x}_i, c_i) . Adding a degree of freedom to $\underline{\theta} \in \mathbb{R}^q$ “eats up” one degree of freedom from the training data, causing the model to begin to fit the noise in the training data. The choice of the number of model parameters q is a delicate balance between

UNDERFITTING, such that q is too low, and the model is unable to match the behaviour underlying the data; and

OVERFITTING, in which q is too large, such that the excess degrees of freedom in the model are fitting to noise in the training data, rather than improving the model’s ability to meaningfully generalize.

We need some objective way of assessing whether our model is overfitting, which requires data which have not been used in the training process. As a result, data must be divided into training and testing, or ideally all three of training/validation/testing.

The assessment of a model will be described in further detail in Chapter 9, however at its most basic level a model is evaluated based on probability of classification error $P(e)$, or by examining the confusion matrix.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links: [Overfitting](#), [Degrees of Freedom](#), [Confusion Matrix](#), [Machine Learning](#), [Occam's Razor](#), [Akaike Information Criterion](#), [No Free Lunch Theorem](#), [Wrapper function](#)

The reader is first referred to [Chapter 9](#), which discusses the question of classifier validation, which is a natural partner to the topic of learning.

There are a great many books on pattern recognition and machine learning, all of which address aspects of learning, underfitting, and overfitting [[1–3](#), [6, 7, 9](#)].

Regarding Occam's Razor, the Principal of Parsimony, and the overall question of what it means to learn without overfitting, the book by Sober [[8](#)] offers a nice overview.

Learning a classifier or, more generally, a model, is a special case of what is known as an inverse problem. That is, if $p(x|C_i)$ represents the distribution, essentially a generative model, describing the transformation from class C_i to feature space \underline{x} , then a classifier $g(\underline{x}, \underline{\theta})$ is precisely the *inverse*, seeking to transform from feature space \underline{x} to class space \mathcal{C} . For a deeper understanding you might consider Chapter 11 of [[5](#)] or, a bit more advanced, Chapter 2 of [[4](#)], and the references therein.

Finally, in terms of the No Free Lunch Theorem, two of the original papers on the topic:

- D. Wolpert, D. Macready, "No Free Lunch Theorems . . .," *IEEE Trans. Evolutionary Computation* (1) #67, 1997
D. Wolpert, "The Lack of A Priori Distinctions . . .," *Neural Computation* (8) #7, 1996

Sample Problems

Problem 3.1: Short Answer

Give a short definition of each of the following:

- Degrees of freedom
- Overfitting
- Underfitting
- Occam's Razor
- Training vs. Testing data
- Confusion matrix

Problem 3.2: Analytical — Confusion Matrices

Suppose we are given the following confusion matrix (of counts) based on testing data for three classes C_1, C_2, C_3 :

12	4	2
3	8	0
0	0	9

- (a) How many test data points are in class C_1 ?
- (b) What is the estimated overall probability of classification error $\mathbf{P}(e)$?
- (c) Briefly, what can you say regarding the spatial arrangement of the three classes in their feature space \underline{x} ?

Problem 3.3: Computational — Robustness and Outliers

[Example 3.1](#) looked at the question of robustness, and the extent to which extreme outliers could influence an estimate.

Let us create two data sets $\{a_i\}, \{b_i\}$:

$$a_i \sim \mathcal{N}(0, 1) \quad b_i = \frac{x_i}{y_i} \quad \text{where} \quad x_i, y_i \sim \mathcal{N}(0, 1) \quad 1 \leq i \leq 20 \quad (3.23)$$

We would like to compute the “middle” of each set of numbers. There are at least three ways of finding the middle:

- Find the mean
- Find the median
- Find half way between the minimum and the maximum values.

We would like to assess the robustness of these three notions of “middle”. So, set up a loop to do the following 50 times:

- Synthesize 20 random data points $\{a_i\}$ per [\(3.23\)](#)
- Synthesize 20 random data points $\{b_i\}$ per [\(3.23\)](#)
- For each of $\{a_i\}$ and $\{b_i\}$ compute and plot the mean, median, and min-max-halfway.

What do you observe?

Problem 3.4: Reading — Overfitting

Overfitting is one of the most fundamental concerns in pattern recognition, and it is easy for poorly-designed algorithms or testing regimes to be susceptible to overfitting.

Prepare a short overview of the problem of overfitting, how we detect or infer it, and strategies to avoid it.

References

1. C. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2011)
2. M. Deisenroth, A. Faisal, C. Ong, *Mathematics for Machine Learning* (Cambridge University Press, 2020)
3. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley Interscience, 2009)
4. P. Fieguth, *Statistical Image Processing and Multidimensional Modeling* (Springer, 2010)
5. P. Fieguth, *An Introduction to Complex Systems* (Springer, 2021)
6. K. Murphy, *Machine Learning: A Probabilistic Perspective* (MIT Press, 2012)
7. S. Rogers, M. Girolami, *A First Course in Machine Learning* (Chapman and Hall, 2016)
8. E. Sober, *Ockham's Razors: A User's Manual* (Cambridge University Press, 2015)
9. A. Webb, K. Copsey, *Statistical Pattern Recognition* (Wiley, 2011)



Representing Patterns

What does it mean to “represent” a pattern? When we think of a “pattern”, we normally have some unifying concept in mind, some behaviour that all elements of that pattern have in common. For example, our mental model of *lemons* might be of yellow objects, somewhat elongated, round in cross section, fit easily in the hand, and relatively smooth skin, meaning that the “lemon” class should occupy a small, focused part of the colour — shape — size — texture feature space. So our notion of representing a pattern is intimately connected to the shape of its associated pattern class.

4.1 Similarity

Intuitively, two patterns are similar if they share common properties. That is not to say that two similar patterns *necessarily* have similar measurements, since it may be possible to select measurements which are only weakly related, or completely unrelated, to the patterns at hand.

For example, apples all have a relatively common shape, but can vary widely in mass, therefore we would expect two members of the “apple” class to have similar geometric properties, but not necessarily to be similar in mass, making a measurement of mass a poor choice of feature. Similarly in terms of face recognition, measurements which are sensitive to whether I am wearing sunglasses or a hat would *not* lead to the desired degree of pattern similarity.

In terms of the face recognition example, it is not enough to find a definition of a pattern which keeps all samples of the “Paul Fieguth” face pattern similar, regardless of whether I am wearing sunglasses or a hat, since at the same time

all *other* face patterns need to be strikingly *dissimilar*, somewhere far away from the “Paul Fieguth” location in \mathcal{X} , the chosen feature space.

As a result, we require a feature $\underline{x} \in \mathcal{X} = \mathbb{R}^n$ such that

1. Features from the same pattern class are highly similar, and
2. Features from differing pattern classes are highly dissimilar.

In principle, this then leads directly to the most basic definition of a classifier. Given an observed feature \underline{x} to classify, and given a set of classes C_1, C_2, \dots, C_K , we wish a classifier $g(\underline{x})$ to return the most similar class:

$$g(\underline{x}) = C \quad \text{such that} \quad \text{Similarity}(\underline{x}, C) \geq \text{Similarity}(\underline{x}, C_\kappa) \quad \text{for all } \kappa. \quad (4.1)$$

Of course what (4.1) does *not* specify is how $\text{Similarity}(\underline{x}, C)$ is defined, where it comes from, or how we might select it. Indeed, functions measuring pattern similarity are at the heart of pattern recognition, and will be discussed throughout this chapter.

Although talking about pattern “similarity” is fairly intuitive, mathematically it is more convenient¹ to talk about pattern dissimilarity by means of a *distance* function, such that (4.1) becomes

$$g(\underline{x}) = C \quad \text{such that} \quad \text{Distance}(\underline{x}, C) \leq \text{Distance}(\underline{x}, C_\kappa) \quad \text{for all } \kappa. \quad (4.2)$$

To be more specific,

$$\begin{aligned} \text{Dissimilarity between feature points } \underline{x}_1 \text{ and } \underline{x}_2 &= \text{Distance}(\underline{x}_1, \underline{x}_2) \\ &= d(\underline{x}_1, \underline{x}_2) \end{aligned} \quad (4.3)$$

where there are only two things that we require of $d()$:

1. $d(\underline{x}_1, \underline{x}_2) \geq 0$ — Distances are never negative
2. $d(\underline{x}_1, \underline{x}_1) = 0$ — The distance between two identical points is zero

There are many possible distance functions which can be defined, three of which are illustrated in [Figure 4.1](#), however we will have a great deal more to say about distance functions in [Chapter 6](#).

Although the examples in [Figure 4.1](#) seem fairly intuitive and straightforward, in actual fact there are many challenges in determining suitable measures of distance:

- Measurements may have fundamentally different units that are not natural to compare, such as *seconds*, *kilometres*, and *kilowatts* in taking measurements of an electric vehicle.

¹ For example, if two feature points $\underline{x}_1 = \underline{x}_2$ are exactly the same, how do we measure their perfect similarity? It is easier to talk about how far apart two points are, in which case two identical points have a distance or separation or *dissimilarity* of zero.

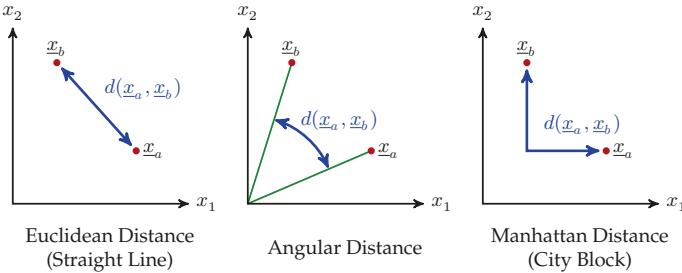


Fig. 4.1. DISTANCES: How far apart are two feature points \underline{x}_a and \underline{x}_b ? There are many possible notions of distance, of which three are shown here. All three of these definitions satisfy our requirements that distances are never negative, and that the distance from a point to itself must be zero.

- Features may not be in any units whatsoever, as was discussed at (2.4) in [Chapter 2](#), giving us no intuitive basis for understanding how differences in one feature might compare with another.
- We may have a large number n of features, meaning that we cannot visualize the n -dimensional feature space, whereas in two or three dimensions we may be able to visualize the data to get some intuition or insight about plausible distance measures.
- Measurements may be discrete (not continuous). Such measurements may be ordered, such as *number of people* or *letter of alphabet*, or unordered (cardinal), such as *marital status* or *religious denomination*, etc. Unordered measurements are particularly challenging, since there may be no objective way of assessing how different one value is from another.

For now, these unusual cases do not actually need to concern us, and instead we will focus mostly on understanding class shapes in low-dimensional continuous feature spaces $\mathcal{X} = \mathbb{R}^n$.

4.2 Class Shape

A *class* describes a type of pattern. Since nearly every pattern involves some degree of variability², each class has some extent or shape in the feature space \mathcal{X} . As was described on [page 13](#), there are at least four ways of thinking about describing the shape of a class, however mathematically these boil down to two:

² Otherwise pattern recognition would actually be pretty easy!

Example 4.1: Face Recognition

Face recognition has become a mainstream topic, sometimes in controversial subjects such as video surveillance or police use, and sometimes in much more every-day contexts, such as automatically logging into your phone or tagging photos on social media.

These different tasks do lead to slightly different definitions of the pattern recognition problem to solve. For example, logging into your phone is a two-class problem

$$\text{Set of Classes } \mathcal{C}_{\text{Phone}} = \{\text{"You", "Everyone else"}\} \quad (4.4)$$

since all that needs to be determined is whether you are facing the phone, no-one else needs to be recognized. If we are tracking people in a crowd, then

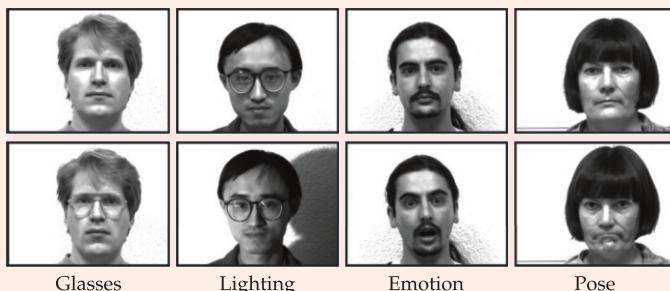
$$\text{Set of Classes } \mathcal{C}_{\text{Tracking}} = \{\text{Unknown Person \#1, Unknown Person \#2, ...}\} \quad (4.5)$$

We don't actually classify *who* the person is, we just want to know that we are consistently tracking the *same* person. Finally for photo tagging purposes, we do actually need to recognize the specific person,

$$\text{Set of Classes } \mathcal{C}_{\text{Tagging}} = \{\text{"Bob", "Jane", "Dave", ...}\} \quad (4.6)$$

Face recognition is quite a challenging problem. There is the broader question of how to design a suitable classifier, to be discussed in future chapters, however even with a classification framework in place there remains the question of how to determine what types of quantifiable features would be useful? Possible features might include eye spacing, iris colour, eye-to-nose distance, mouth width, raw pixels from a face image etc.

As was discussed in [Section 4.1](#), we need features which distinguish between different people, but which are also insensitive to the variations that a given face can exhibit. Images of four faces, taken from the Yale Face Database [1, 4], illustrate some of the challenging variations associated with face recognition, including wearing glasses, image lighting, facial expressions, or head positioning (pose):



Of the features we had listed, mouth width or eye-nose separation could be affected by emotion, eye spacing could be skewed by wearing glasses, and the raw image pixels (including iris colour) could be changed by lighting intensity and colour (sunlight vs. fluorescent light, for example).

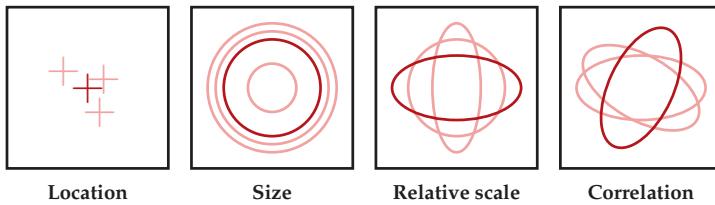


Fig. 4.2. PARAMETRIC CLASS SHAPE: The most fundamental parameters controlling the shape of a class involve its location, its size, the relative scale of its features, and the correlation of its features. Clearly this list is not exhaustive, since more unusually-shaped classes will not be fully characterized by these four attributes.

1. A representation is *parametric*, as shown in Figure 4.2, if it is described by an explicit model which is a function of a predetermined number q of parameters $\theta_1, \dots, \theta_q$. We will commonly encounter three types of models for describing a pattern class:

- A prototype, an idealized essence of a class, parameterized as a point \underline{x} in feature space \mathcal{X} . In an n -dimensional feature space, such a point is described by $q = n$ parameters.
- A shape model, almost always including parameters for the class centre and size, but may have further parameters, depending on the range of shapes permitted. A sphere in n dimensions requires $q = n + 1$ parameters: n parameters for the centre of the sphere, and 1 parameter for its radius.
- A statistical distribution $p(\underline{x} | C)$ gives the relative likelihood for any point \underline{x} to be a member of class C . The number of parameters will vary significantly with the chosen type of probability distribution.

Understanding notation³ “ \sim ” to mean “is described by”, then the above models might be written as follows:

$$\begin{array}{ccc} C \sim \{\underline{z}\} & C \sim \{\underline{\theta}\} & C \sim p(\underline{x} | \underline{\theta}) \\ \uparrow & \uparrow & \uparrow \\ \text{Prototype} & \text{Shape Parameters} & \text{Distribution} \end{array} \quad (4.7)$$

2. In contrast, a representation is *non-parametric*, as shown in Figure 4.3 if there is *no* model, rather the class shape is described *implicitly* by a large number of data points

$$C \sim \{\underline{x}_i, 1 \leq i \leq N\}. \quad (4.8)$$

As shown in Figure 4.4, it is also possible for a distribution to be non-parametric. That is, whereas the distribution in (4.7) came from some

³ A summary of textbook notation can be found starting on page xix.

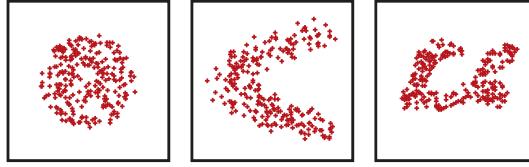


Fig. 4.3. NON-PARAMETRIC CLASS SHAPE: In contrast to Figure 4.2, here three classes are characterized implicitly, non-parametrically, on the basis of their sample points. We could imagine proposing parametric representations, perhaps circular (left) or arc-shaped (middle), however the strength of the non-parametric form is its ability to represent highly unusual shapes (right) that could be quite difficult to describe mathematically. On the other hand, clearly the 250 points here representing each class represent far more degrees of freedom than the few parameters needed in Figure 4.2.

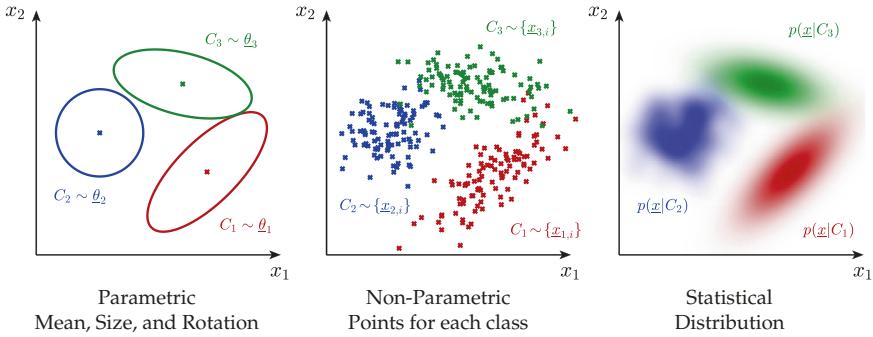


Fig. 4.4. CLASS REPRESENTATIONS: A given class can be described parametrically (left, as in Figure 4.2) or non-parametrically (middle, as in Figure 4.3). In this particular example the distributions (right) for C_1 and C_3 are Gaussian, and therefore parametric (in the form of (4.7)). In contrast, the distribution for C_2 is non-parametric (as in (4.9)), meaning that the distribution is based on underlying data, and therefore looks slightly irregular; the method to infer such a non-parametric distribution will be described in Chapter 7. Each of these forms is a legitimate way of describing a class; whether the description is also *effective* would have to be determined by assessing the resulting classifier.

specified family (Gaussian, uniform, exponential etc.) it is possible to specify a distribution, implicitly, from a large set of data points, in which case there is no assumed form to the distribution and no parameter vector θ to learn, rather a non-parametric distribution

$$C \sim p(\underline{x} | \{\underline{x}_i, 1 \leq i \leq N\}). \quad (4.9)$$

It is a fundamental choice in representing a class as parametric or non-parametric, illustrated in Figure 4.4, and this choice has significant implications for computational complexity, memory requirements, and overall flexibility, as summarized in Table 4.1. It is not possible to give universal

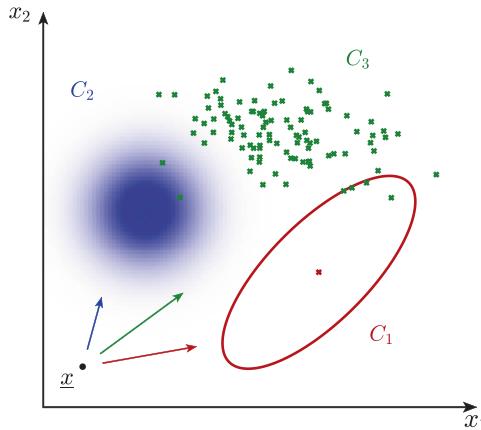


Fig. 4.5. MIXED REPRESENTATIONS: Although unusual, and perhaps inconvenient, there is nothing invalid or illegal about each class being described in a different way. What is essential is that each class exists in the *same* feature space \mathcal{X} , and that we have some way of describing the similarity between a given point \underline{x} (lower left) and each of the classes.

Parametric	Non-Parametric
The class is <i>explicitly</i> described in terms of a model, which itself is a function of relatively few parameters	The class is <i>implicitly</i> described in terms of a collection of data points. No model is assumed
Parameters can effectively be learned from relatively <i>few</i> data points	<i>Many</i> data points are required to meaningfully characterize class shape
Typically computationally efficient and requiring little storage	Typically computationally demanding and requiring significant storage
A model needs to be assumed or asserted	No model assumption of any kind
Model parameters need to be inferred from data	Nothing needs to be learned, the given data points fully characterize the class

Table 4.1. PARAMETRIC AND NON-PARAMETRIC REPRESENTATIONS: A pattern class can be described in terms of some sort of model (parametric, as in Figure 4.2), or without any model, just on the basis of a large number of data points (non-parametric, as in Figure 4.3).

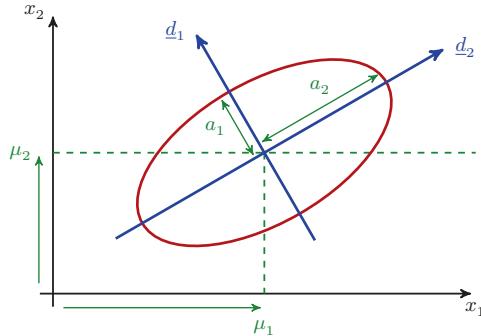


Fig. 4.6. ELLIPSOID: Our most fundamental class shape is that of a multi-dimensional ellipsoid (red), described in terms of the ellipse centre $\underline{\mu}$, the directions of the principle axes \underline{d}_i , and the half-axis lengths a_i .

advice on model selection, however the Principle of Parsimony of [Chapter 3](#) would well be heeded: try to use as simple a representation as possible for a given context. Thus the q degrees of freedom should be kept modest relative to the complexity of the problem, both to avoid overfitting and to avoid unneeded computational burdens. It is important to emphasize that representation is specific to each class, and not necessarily to the feature space. As a result it is possible, even if generally not advisable, to represent each class differently, as shown in [Figure 4.5](#).

The two extremes of class representation are via a single prototype, as the most simple model, or via *all* of the data points, as the most general. Neither of these extremes is particularly helpful in obtaining a deeper mathematical understanding of features and classes. Instead, the model that is of particular significance is to represent class shape as a multi-dimensional hyper-ellipse⁴

$$C \sim \{\underline{\mu}, \underline{d}_1, \underline{d}_2, \dots, a_1, a_2, \dots\} \quad (4.10)$$

for centre μ , axis directions \underline{d}_i , and axis lengths a_i , as sketched in [Figure 4.6](#), and as developed/constructed in [Example 4.2](#). The number of dimensions is left unspecified in (4.10), since we could equally well be describing a class in the m -dimensional measurement space or the n -dimensional feature space.

The familiar equation for an ellipse in two dimensions,

$$\left(\frac{x_1 - \mu_1}{a_1} \right)^2 + \left(\frac{x_2 - \mu_2}{a_2} \right)^2 = 1, \quad (4.15)$$

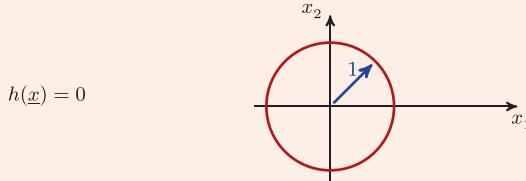
⁴ We will use the terminology of an ellipse in two dimensions, an ellipsoid (egg-shape) in three dimensions, and a hyper-ellipse in four or more dimensions, just as circle (2D)/sphere (3D)/hyper-sphere (>3 D).

Example 4.2: How to Form an Ellipsoid

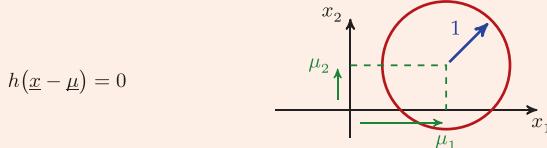
The equation for a unit circle/sphere/hyper-sphere, centered on the origin, is just

$$\sum_j x_j^2 = 1 \longrightarrow \sum_j x_j^2 - 1 = 0 \longrightarrow \underline{x}^T \underline{x} - 1 = 0 \quad (4.11)$$

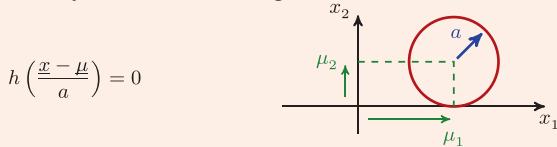
so we will define $h(\underline{x}) = \underline{x}^T \underline{x} - 1$, such that $h(\underline{x}) = 0$ defines a circle or sphere:



We then apply a coordinate shift by μ to move the centre of the circle:

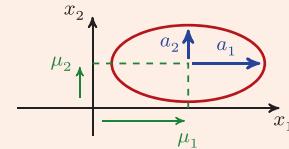


Next we re-scale by a factor a , which grows or shrinks the circle:



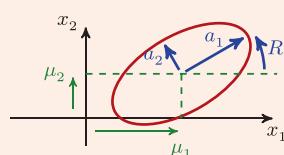
The re-scaling can be applied to each coordinate separately, producing an ellipse:

$$h \left(\begin{bmatrix} 1/a_1 & & \\ & \ddots & \\ & & 1/a_n \end{bmatrix} (\underline{x} - \underline{\mu}) \right) = 0$$



A rotation R rotates the ellipse:

$$h \left(\begin{bmatrix} 1/a_1 & & \\ & \ddots & \\ & & 1/a_n \end{bmatrix} R^T (\underline{x} - \underline{\mu}) \right) = 0$$



Example continues ...

Example 4.2: How to Form an Ellipsoid (continued)

The preceding sequence of operations is able to produce *any* 2D ellipse/multi-dimensional hyper-ellipse, by specifying a centre vector $\underline{\mu}$, a diagonal matrix of scaling factors A , and a rotation matrix R .

We can expand the last equation to see the hyper-ellipse written out. Since $h(\underline{x}) = \underline{x}^T \underline{x} - 1$, then

$$h(M(\underline{x} - \underline{\mu})) = (M(\underline{x} - \underline{\mu}))^T (M(\underline{x} - \underline{\mu})) - 1 = (\underline{x} - \underline{\mu})^T M^T M(\underline{x} - \underline{\mu}) - 1 \quad (4.12)$$

for any matrix M . Therefore

$$h\left(\begin{bmatrix} 1/a_1 & & \\ & \ddots & \\ & & 1/a_n \end{bmatrix} R^T (\underline{x} - \underline{\mu})\right) = 0 \quad (4.13)$$

becomes

$$(\underline{x} - \underline{\mu})^T \cdot R \underbrace{\begin{bmatrix} 1/a_1^2 & & \\ & \ddots & \\ & & 1/a_n^2 \end{bmatrix}}_Q R^T \cdot (\underline{x} - \underline{\mu}) = 1 \quad (4.14)$$

from which we can very clearly identify the parameter matrix $Q = R(A^2)^{-1}R^T$ as being built up from squared-scaling A^2 and rotation R elements.

is fairly simple and intuitive, however this way of writing the equation becomes nearly impossible when rotated arbitrarily in many dimensions. Instead, although it will take a little getting used to, it will be much simpler to write a multi-dimensional hyper-ellipse as

$$(\underline{x} - \underline{\mu})^T Q (\underline{x} - \underline{\mu}) = 1, \quad (4.16)$$

where this form is derived in [Example 4.2](#). It must be pointed out that while [\(4.16\)](#) is capable of generating any ellipse, it is *also* capable of representing hyperbolas, as developed in [Example 4.3](#), so we will need to be somewhat discriminating in our choice of the parameters in matrix Q .

The motivation for [\(4.16\)](#) goes somewhat deeper yet. By far the most common probability density function is the well-known Gaussian or Normal distribution.⁵ In one dimension, it is the common “bell curve”

⁵ For an overview of basic probability theory see [Appendix B](#).

Example 4.3: Quadratic Forms

The equation

$$(\underline{x} - \underline{\mu})^T Q (\underline{x} - \underline{\mu}) = 1 \quad (4.17)$$

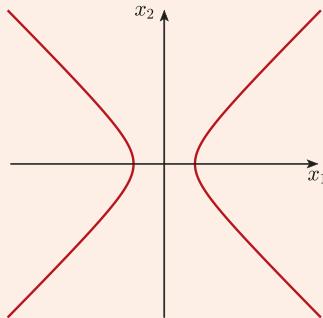
can represent any ellipse, by appropriate choices of Q , however (4.17) is not necessarily an ellipse. Suppose, for example, that we choose the two-dimensional example

$$\underline{\mu} = \underline{0} \quad Q = \begin{bmatrix} 1 & \\ & -1 \end{bmatrix} \quad (4.18)$$

Then (4.17) becomes

$$\underline{x}^T Q \underline{x} = 1 \longrightarrow \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 1 & \\ & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 1 \longrightarrow x_1^2 - x_2^2 = 1 \quad (4.19)$$

which is, in fact, a *hyperbola*, as plotted below:



For (4.17) to produce ellipses, we need to place some constraints on the permitted choices of Q . In particular (from Example 4.2) we wish Q to perform only scaling and rotation, which will mean that Q is *symmetric and positive-definite*, meaning that

- Q is symmetric;
- All of the eigenvalues of Q are positive.

Since the eigenvalues of a diagonal matrix are simply the elements along the diagonal, from (4.18) we can see that the choice of Q in (4.18) had a negative eigenvalue and therefore did not lead to the equation for an ellipse.

Further Reading: Algebra is reviewed in Appendix A and covariances in Appendix B.4.

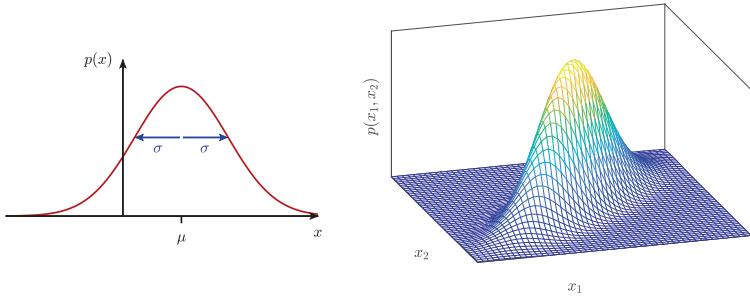


Fig. 4.7. One-dimensional (left) and two-dimensional (right) examples of the Gaussian/Normal probability distribution, further described in Appendix B. The distribution is characterized in terms of its centre (the mean) and its spread (the variance σ^2 in one dimension, or a covariance matrix Σ in higher dimensions).

$$x \sim \mathcal{N}(\mu, \sigma^2) \longrightarrow p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (4.20)$$

for mean μ and standard-deviation σ or variance σ^2 . In n dimensions (4.20) generalizes to

$$p(\underline{x}) = \frac{1}{\det(\Sigma)(2\pi)^{n/2}} \exp\left(-\frac{1}{2}(\underline{x}-\mu)^T \Sigma^{-1} (\underline{x}-\mu)\right) \quad (4.21)$$

for mean μ and covariance⁶ Σ , such that

$$\Sigma_{i,j} = \text{Correlation}(x_i, x_j) = \mathbb{E}[(x_i - \mu_i) \cdot (x_j - \mu_j)] \quad (4.22)$$

describes the correlation between x_i and x_j , where $\mathbb{E}[\cdot]$ is the expectation⁷ operator. The shape of this distribution is sketched in one and two dimensions in Figure 4.7.

We know that the most likely value of \underline{x} , the peak of $p(\underline{x})$, will be at the mean μ , with the probability density smoothly decreasing as one moves away from the mean, since the Normal distribution does not have any sudden boundary or edge. To obtain some insight into the shape of the multi-dimensional Normal, we start by evaluating a contour/surface of constant probability-density, thus

$$p(\underline{x}) = \frac{1}{\det(\Sigma)(2\pi)^{m/2}} \exp\left(-\frac{1}{2}(\underline{x}-\mu)^T \Sigma^{-1} (\underline{x}-\mu)\right) = \text{Constant} \quad (4.23)$$

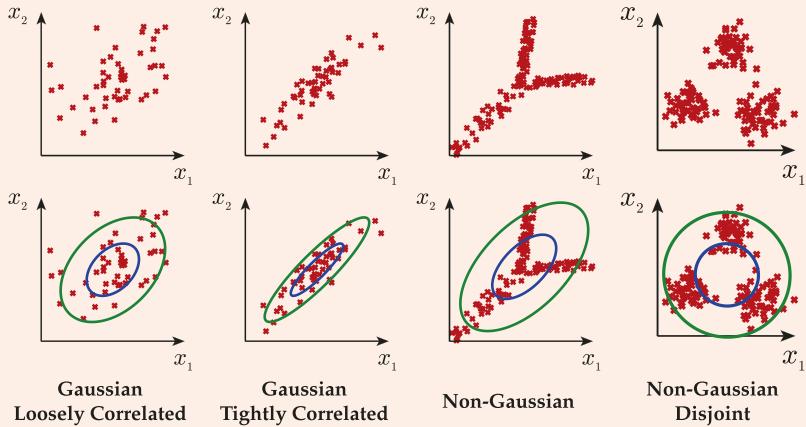
Since the first part of $p(\underline{x})$, outside of the exponential, is not a function of \underline{x} , we can ignore it as far as the constant probability-density shape is concerned. Thus

⁶ Covariance matrices are described in detail in Appendix B.4.

⁷ Expectations are described in Appendix B.2.

Example 4.4: Ellipsoids and Real Data

Given *any* data set $\{\underline{x}_i\}$ we can calculate its corresponding covariance, and from there its unit standard-deviation contour, however that does not necessarily mean that the ellipse is a good representation:



In the lower row, each of the panels shows a one (blue) and two (green) standard-deviation contour. For the two Gaussian examples (left), the one standard-deviation contour correctly encloses the approximately 39% of points that we expect to have within that contour (as discussed in [Example 4.5](#)). In contrast, for the disjoint class (right) the fraction of points within the blue contour is near zero, and it is questionable to what extent the contours have any significant meaning with respect to the class shape in the two non-Gaussian examples.

The point here is not to suggest that the Gaussian ellipsoid is a poor representation; in contrast, it is widely used and is an excellent starting point in data analysis. However there can be many class shapes for which an ellipsoid is a poor match.

$$\exp\left(-\frac{1}{2}(\underline{x} - \mu)^T \Sigma^{-1} (\underline{x} - \mu)\right) = \text{Constant} \quad (4.24)$$

Since the exponential is an invertible function, if there is some contour or surface such that the *exponential* is constant, it follows that over the contour or surface the *exponent* must also be constant:

$$(\underline{x} - \mu)^T \Sigma^{-1} (\underline{x} - \mu) = \text{Constant} \quad (4.25)$$

Note that we have not specified *which* constant (i.e., which value of probability density back in [\(4.23\)](#)) we were interested in. Without any loss of generality, presumably it is simplest to set that constant equal to one, thus the contour/surface represented by

$$(\underline{x} - \mu)^T \Sigma^{-1} (\underline{x} - \mu) = 1, \quad (4.26)$$

which is, in fact, *exactly* the same form as the hyper-ellipse from (4.16), as before. Therefore

every hyper-surface of constant probability density for any multi-dimensional Normal distribution has the shape of a hyper-ellipse.

If we look at the exponent for the one-dimensional case, which is easier to interpret, we understand that

$$\frac{(x - \mu)^2}{\sigma^2} = 1 \rightarrow x = \mu \pm \sigma \quad (4.27)$$

meaning that we are identifying those points lying at a distance of one σ (one standard deviation) from the mean. The same interpretation applies in higher dimensions, such that we refer to (4.26) as the *unit standard-deviation contour*, which then also serves as a measure of distance (as in Section 4.1), such that the distance from any point \underline{x} to mean μ , in the units implied by the class statistics of Σ , is,

$$d(\underline{x}, \mu) = (\underline{x} - \mu)^T \Sigma^{-1} (\underline{x} - \mu) \quad (4.28)$$

As a last step, we need to understand the connection between the ellipse parameters Q , from (4.16), and the statistical covariance Σ , from (4.26):

$$\underbrace{(\underline{x} - \mu)^T Q (\underline{x} - \mu)}_{\text{Ellipse equation from (4.16)}} = 1 \quad \underbrace{(\underline{x} - \mu)^T \Sigma^{-1} (\underline{x} - \mu)}_{\text{Unit standard-deviation from (4.26)}} = 1 \quad (4.29)$$

From Example 4.2, the ellipse form of Q is built up from scaling and rotations, as

$$Q = R (A^2)^{-1} R^T \quad (4.30)$$

Similarly, for any covariance we can find the eigendecomposition⁸

$$\Sigma \underline{v}_j = \lambda_j \underline{v}_j \rightarrow \Sigma V = V \Lambda \quad (4.31)$$

where, from (A.32), V is a matrix of eigenvectors, arranged next to each other column-by-column, and Λ is a diagonal matrix of eigenvalues:

$$A \underline{v}_j = \lambda_i \underline{v}_j, \quad 1 \leq j \leq n \rightarrow V = \begin{bmatrix} | & | \\ v_1 & \cdots v_n \\ | & | \end{bmatrix} \quad \Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \quad (4.32)$$

⁸ Eigendecompositions are discussed in Appendix A.

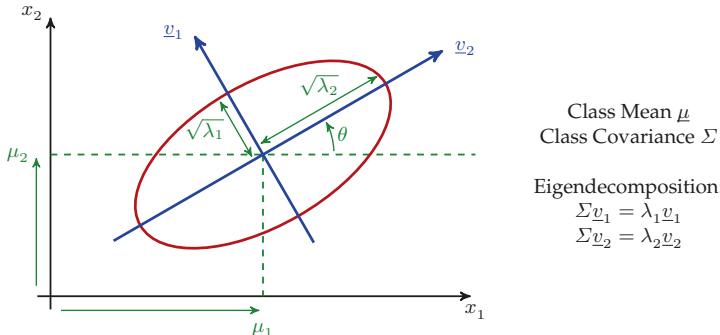


Fig. 4.8. ELLIPSOID: Building on [Figure 4.6](#), the ellipsoid associated with a given class covariance Σ is described in terms of the directions of its principle axes (the eigenvectors) and the extent of the ellipsoid along each axis (the square-roots of the eigenvalues), determined by the eigendecomposition of Σ , as shown.

The eigenvector matrix V is an *orthogonal* matrix⁹, also known as a rotation matrix, which has the remarkable property that $V^T = V^{-1}$, therefore

$$\Sigma V = V \Lambda \rightarrow \Sigma = V \Lambda V^T \rightarrow \Sigma^{-1} = (V \Lambda V^T)^{-1} = V \Lambda^{-1} V^T \quad (4.33)$$

Returning to [\(4.29\)](#), we now have the remarkably similar

$$Q = R(A^2)^{-1}R^T \quad \Sigma^{-1} = V \Lambda^{-1} V^T \quad (4.34)$$

clearly associating rotation R with rotation matrix V , which transforms the original coordinate system to point in the directions of the eigenvectors, and associating A^2 with Λ , such that the eigenvalues control the scaling, as summarized in [Figure 4.8](#):

- The centre of the ellipse is equal to the mean μ
- The axis directions of the ellipse are the eigenvectors of the covariance Σ
- The half axis lengths of the ellipse are the square roots of the eigenvalues of Σ

Since we now understand, from [Figure 4.8](#), that the eigendecomposition of a covariance Σ describes the unit standard-deviation contour, presumably there is some way of understanding the role of the covariance elements themselves, *without* any eigendecomposition. For a two-dimensional covariance¹⁰ matrix

⁹ There are some assumptions implicit here, discussed in [Appendix A](#). Matrix V will be orthogonal only if the eigenvectors are normalized to unit length and if matrix A is symmetric. Since we will mostly be taking eigendecompositions of covariances or other symmetric matrices, we will assume the orthogonality of V throughout this text, except where noted otherwise.

¹⁰ A generic 2×2 matrix is normally written as $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$. However since covariances *must* be symmetric matrices, rather than writing $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and reminding ourselves that $b = c$, we will always just directly write the symmetric form $\begin{bmatrix} a & b \\ b & c \end{bmatrix}$, as in [\(4.35\)](#).

Example 4.5: Class Data, Covariances, Eigendecompositions, and Ellipses

Suppose we look again at the Iris data from page 25; one of those plots is reproduced here, at right. The collections of data points represent non-parametric descriptions of the classes, but ellipses would be much simpler.

We begin by computing the sample mean

$$\hat{\mu} = \frac{1}{N} \sum_i y_i$$

and sample covariance

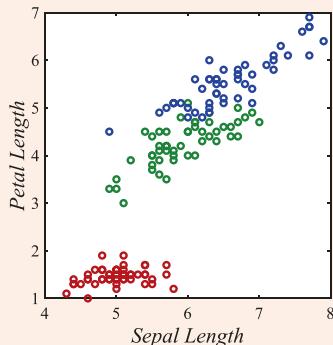
$$\hat{\Sigma} = \frac{1}{N-1} \sum_i (y_i - \hat{\mu})(y_i - \hat{\mu})^T$$

for each class. The covariances are shown, at right, with numbers rounded for simplicity. Keep in mind that the covariances are in squared units, so it is the square-root of the diagonal elements which gives us the standard deviations.

We observe that the off-diagonal elements are positive for green and blue, since the measurements in those classes are positively correlated, whereas the correlation in the red class is close to zero. We also clearly see how the covariance entries increase from the smallest (red) to the largest (blue) class.

We can compute the eigendecompositions from the covariances. In principle, the eigendecomposition of a 2×2 matrix covariance can be computed by hand, however doing so is not of particular interest in this text.

Do note that eigenvectors scale, in that if \underline{v} is an eigenvector of Σ , then so are $-\underline{v}$ and $2\underline{v}$. It is common practice to normalize the eigenvector lengths, however that is not done here, in order to keep the numbers



Sample Means and Covariances

$$\begin{aligned}\hat{\mu}_1 &= \begin{bmatrix} 5.0 \\ 1.5 \end{bmatrix} & \hat{\Sigma}_1 &= \begin{bmatrix} 0.10.0 \\ 0.00.0 \end{bmatrix} \\ \hat{\mu}_2 &= \begin{bmatrix} 5.9 \\ 4.3 \end{bmatrix} & \hat{\Sigma}_2 &= \begin{bmatrix} 0.30.2 \\ 0.20.2 \end{bmatrix} \\ \hat{\mu}_3 &= \begin{bmatrix} 6.6 \\ 5.6 \end{bmatrix} & \hat{\Sigma}_3 &= \begin{bmatrix} 0.40.3 \\ 0.30.3 \end{bmatrix}\end{aligned}$$

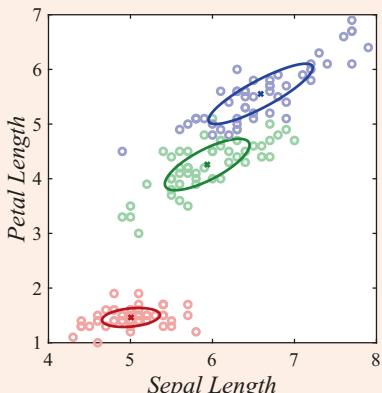
Eigendecompositions

$$\underline{v}_1 = \begin{bmatrix} 1 \\ 0.2 \end{bmatrix} \lambda_1 = 0.13 \quad \underline{v}_2 = \begin{bmatrix} 0.2 \\ -1 \end{bmatrix} \lambda_2 = 0.03$$

$$\underline{v}_1 = \begin{bmatrix} 1 \\ 0.9 \end{bmatrix} \lambda_1 = 0.43 \quad \underline{v}_2 = \begin{bmatrix} 0.9 \\ -1 \end{bmatrix} \lambda_2 = 0.06$$

$$\underline{v}_1 = \begin{bmatrix} 1 \\ 0.9 \end{bmatrix} \lambda_1 = 0.66 \quad \underline{v}_2 = \begin{bmatrix} 0.9 \\ -1 \end{bmatrix} \lambda_2 = 0.05$$

Resulting Unit-Standard Deviation Contours



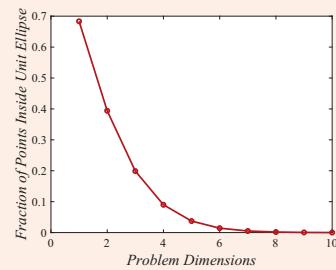
Example continues ...

**Example 4.5: Class Data, Covariances, Eigendecompositions, and Ellipses
(continued)**

simple. The normalization *is* essential, however, in order for matrix V in (4.32) to be an orthogonal/rotation matrix.

For each class the sample mean (shown as an \times) is plotted, with the surrounding unit standard-deviation ellipse (which [Lab 5](#) will show how to plot) implied by the sample covariance, but where the ellipse geometry is determined based on the eigenvalues and eigenvectors as in [Figure 4.8](#). You will observe that many of the data points lie outside of the ellipse, a behaviour which is, indeed, expected: The purpose of the ellipse is to serve as a description of the characteristic class shape, it is not intended to circumscribe or enclose all of the cluster data points.

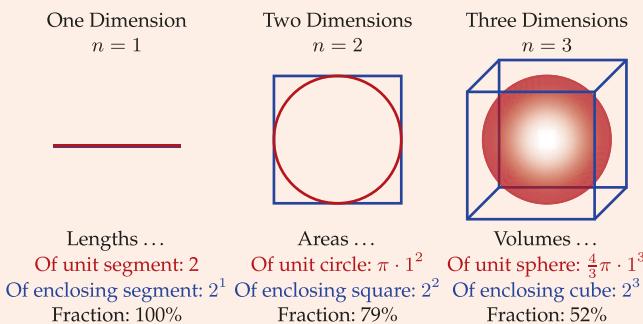
As plotted, right, for the one-dimensional Normal distribution, 68% of points lie within one standard deviation of the mean, meaning that 32% would be expected to lie *outside*. In two dimensions it is only 39% of the points which are within the unit standard-deviation contour, thus 61% lying outside, roughly consistent with what we saw on the previous page.



So why does the probability of lying within one standard deviation *decrease* with higher dimensions? Shouldn't it be a fixed fraction?

There are essentially two influencing effects:

1. The point has to be close to the mean in *each* of the n dimensions. So if there is a probability of 0.683 of lying within one standard-deviation in one dimension, then there is a probability of 0.683^n of lying that close in all of n dimensions, meaning the probability of lying within an n -dimensional hyper-cube.
2. However a hyperellipse or hypersphere occupies an ever smaller fraction of its surrounding hypercube as the number of dimensions increases:



The actual probability, plotted above-right, is slightly larger than the product of these two effects, since the probability distribution within the hypercube is not uniform, rather it is higher at the mean.

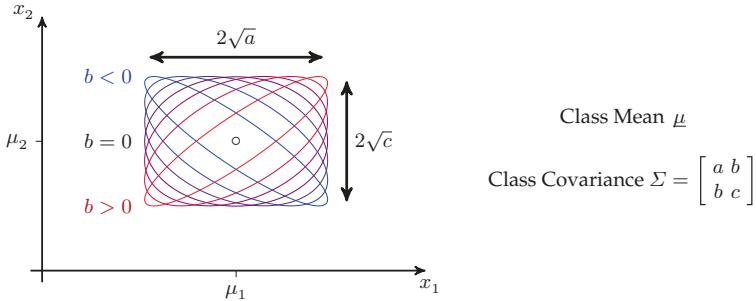


Fig. 4.9. As an alternative to the representation in Figure 4.8, an ellipsoidal class can also be directly specified by its mean μ and covariance Σ , without any reference to the eigendecomposition. Shown here for the two-dimensional case, the variances a and c control the bounding box around the class, whereas the correlation b controls the shape of the class within that bounding box. Note that whereas b influences the angle of the ellipse, it is really not a rotation parameter.

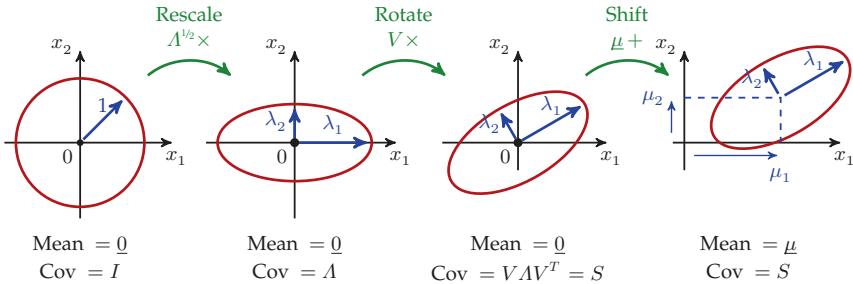


Fig. 4.10. CLUSTER SYNTHESIS: Analogous to the construction of ellipses in Example 4.2, from (4.44) we can generate random points obeying any given mean μ and covariance S by starting with independent random values (left) and then rescaling, rotating, and shifting to produce the desired class shape (right).

$$\Sigma = \begin{bmatrix} a & b \\ b & c \end{bmatrix} \quad \rightarrow \quad \begin{array}{c} \text{Standard deviations } \sqrt{a}, \sqrt{c} \\ \text{Correlation } b \end{array} \quad (4.35)$$

we expect the unit standard-deviation ellipse to be constrained to be within one standard deviation, \sqrt{a} and \sqrt{c} , of the mean, with the correlation b controlling the shape of the ellipse within that constrained box, precisely what we see in Figure 4.9.

So we now have a fairly complete understanding of multi-dimensional Normal distributions, the effects of their covariances and respective eigendecompositions, and the ellipse/hyper-ellipse associated with the unit standard-deviation contour. An illustration with real data is shown in Example 4.5.

4.3 Cluster Synthesis

The eigendecomposition applied to a covariance allows us to understand the geometry, the axis lengths and directions, of the associated unit standard-deviation contour, as we saw in Figure 4.8. However, we should be able to similarly apply this process in reverse, starting with the desired geometry, and deducing the associated covariance. Indeed, this is not at all just a theoretical exercise — in pattern recognition we very often need to create simple data sets to test a proposed classifier or other technique.

We begin with random numbers: any numerical mathematics package will have a function defined to return a random zero-mean unit-variance normal (Gaussian) value,

$$v \sim \mathcal{N}(0, 1), \quad (4.36)$$

with successive random numbers independent¹¹ of one another. So by stacking n successive values into a vector \underline{v} we obtain

$$\underline{v} \in \mathbb{R}^n \quad \underline{v} \sim \mathcal{N}(\underline{0}, I) \quad (4.37)$$

This is therefore our starting point: we wish to use an independent/uncorrelated noise vector $\underline{v} \sim I$ to synthesize a class having given class statistics:

$$\underline{v} \in \mathbb{R}^n \quad \underline{v} \sim \mathcal{N}(\underline{0}, I) \quad \longrightarrow \quad \underline{x} \in \mathbb{R}^n \quad \underline{x} \sim \mathcal{N}(\underline{\mu}, S) \quad (4.38)$$

by finding a transformation $T\underline{v} + \underline{\alpha}$ such that

$$(T\underline{v} + \underline{\alpha}) \sim \mathcal{N}(\underline{\mu}, S). \quad (4.39)$$

From (B.37) and Appendix D we know that

$$\begin{aligned} \mathbb{E}[T\underline{v} + \underline{\alpha}] &= T\mathbb{E}[\underline{v}] + \underline{\alpha} = \underline{0} + \underline{\alpha} = \underline{\alpha} && \leftarrow \text{Needs to equal } \underline{\mu} \\ \text{Cov}(T\underline{v} + \underline{\alpha}) &= T \cdot \text{Cov}(\underline{v}) \cdot T^T = T \cdot I \cdot T^T = TT^T && \leftarrow \text{Needs to equal } S \end{aligned} \quad (4.40)$$

From (4.33) we know the relationship between a covariance and its eigendecomposition:

$$S = V\Lambda V^T \quad (4.41)$$

which we can rewrite as

$$S = V\Lambda V^T = (V\Lambda^{1/2})(V\Lambda^{1/2})^T = TT^T \quad (4.42)$$

¹¹ The numbers are generated by a deterministic algorithm and so are not, in fact, random or independent. However such algorithms have been very carefully developed over many years to yield values which would be assessed as random/independent by any statistical test.

Therefore what we require in transformation T is just

$$T = VA^{1/2}. \quad (4.43)$$

Since A is a diagonal matrix, the square root $A^{1/2}$ is simple, easily found by taking the square root of the diagonal elements. Therefore, we can synthesize data for any hyper-ellipsoidal class in n dimensions as

$$\underline{v} \in \mathbb{R}^n \quad \underline{v} \sim \mathcal{N}(\underline{0}, I) \quad \longrightarrow \quad (\underline{\mu} + VA^{1/2}\underline{v}) \sim \mathcal{N}(\underline{\mu}, S) \quad (4.44)$$

The intuition behind (4.44) is illustrated in [Figure 4.10](#) and explored numerically in [Lab 4](#).

Case Study 4: Defect Detection

The detection of defects is one of the most fundamental aspects of quality control in nearly any product, whether bugs in a computer program, a failure in a mechanized assembly line, or some component which has not met the physical specifications that were required.

For whatever quality control step we are facing, the basic pattern recognition formulation would appear to be relatively straightforward:

$$\text{Measured Features } \underline{x} \xrightarrow{\text{Classify}} C_{\text{Good}} \text{ or } C_{\text{Defective}} \quad (4.45)$$

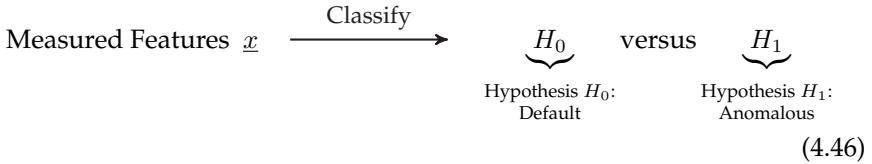
Given the ideas just discussed in this chapter, it seems plausible to develop a class model for each of these two classes. However at this point we suddenly encounter a problem: we know what it means for a sample to lie in C_{Good} , but we do not have a description of what $C_{\text{Defective}}$ means.

For example consider a very simple mechanical part, such as a bolt. It is easy to describe what a “good” bolt is — correct dimensions, correct mass etc. But there are *millions* of ways for a bolt to be “defective” — it might be dented or scratched, it might contain a flaw that makes the metal weak, the thread could have the wrong pitch, the head or length dimensions could be incorrect etc. Here then, is the problem: $C_{\text{Defective}}$ has a very complex class shape (since there are so many ways in which a bolt could be defective), and so we would need *many* samples to learn the class (probably non-parametrically, as discussed in [Section 4.2](#)), at the same time we would expect defective bolts to be rare,¹² so in most cases we will have very few samples to work with.

In summary, we can describe C_{Good} , but will typically not have enough data to comprehensively describe $C_{\text{Defective}}$.

¹² If defective parts are *common*, then you seriously need to switch suppliers!

This defect detection problem is a special case of what is known as *anomaly detection* or *hypothesis testing*:



(4.46)

We assume that $p(\underline{x} | H_0)$ is known (whether parametric or non-parametric), but nothing is known about $p(\underline{x} | H_1)$, since we do not know what sorts of \underline{x} will be measured from a defective bolt.

For a given classifier, what we *can* say is how likely it is that we will declare good part to be defective:

$$\alpha = \mathbf{P}(\text{Classify as } H_1 | \text{Sample actually from } H_0) \quad (4.47)$$

Here α is a confidence parameter, however α is not something which we *deduce* from data, rather it is a value which we *assert* and then find a corresponding classifier.

Let us consider a concrete example as part of this case study. Suppose you wish to assess whether a given coin is fair (H_0) or unfair (H_1), by observing how the coin lands when you flip it N times. We know the statistics for a fair coin, which is that there is a 50–50 probability of the coin landing on heads or tails. As a result h , the number of heads, must be binomially distributed:

$$h | H_0 \sim \text{Binomial} \quad (4.48)$$

As N increases the Binomial distribution is increasingly well approximated as Gaussian, therefore (4.48) simplifies as

$$h | H_0 \sim \mathcal{N}(Np, Np(1 - p)) \quad (4.49)$$

for $p = \frac{1}{2}$, the probability of the coin landing as heads. We can normalize (4.49) so that

$$q = \frac{h - N/2}{\sqrt{N/4}} \rightarrow q | H_0 \sim \mathcal{N}(0, 1) \quad (4.50)$$

So we do not know what statistics an unfair coin would possess, we just know that a fair coin should satisfy the distribution of (4.50), which is to have q close to zero, and so an unfair coin should lead to q deviating from zero (i.e., h deviating from $N/2$). We can then evaluate¹³ what range of h a coin would

¹³ The tails of the Gaussian distribution cannot be integrated by hand, so the evaluation needs to be done numerically, as is discussed later in the text in Section 8.4.

have to fall into, as a function of the confidence parameter α , supposing we flip a coin $N = 100$ times:

Confidence α	20%	5%	1%	0.1%	0.01%
Fair coin range for h	44–56	40–60	37–63	36–64	32–68

That is, the classifier

$$|h - 50| \begin{cases} \text{Unfair} \\ \geqslant \\ \text{Fair} \end{cases} 6 \quad (4.51)$$

has a 20% probability of declaring a fair coin as unfair. To be much more certain about your conclusion, the classifier

$$|h - 50| \begin{cases} \text{Unfair} \\ \geqslant \\ \text{Fair} \end{cases} 18 \quad (4.52)$$

will have only a one in 10,000 chance of declaring a fair coin as unfair, but with an increased likelihood that some *unfair* coins are declared as fair.

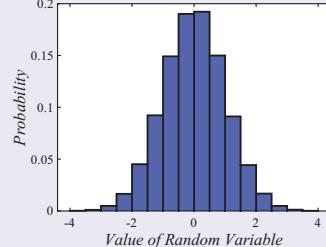
Lab 4: Working with Random Numbers

[Lab 2](#) and [Lab 3](#) both assumed an available dataset. This makes sense — in nearly all pattern recognition problems we need to analyze data which are given to us.

However it is important to understand how to work with and manipulate random numbers. We start most simply just by generating a large number of Gaussian (Normal) random variables, and seeing how they are distributed:

```
% start by understanding basics of random
values
pts = randn(1,1e5);
histogram(pts,[-4:0.5:4], 'normalization',
'probability')
```

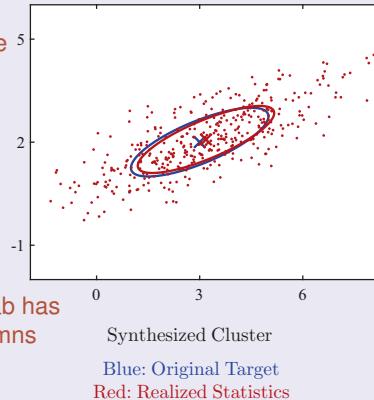
The plotted histogram (to be discussed in more detail in [Chapter 7](#)) is essentially the probability distribution of the random variables, and we can very clearly observe that the distribution is zero mean and symmetric about the mean. Although the tails of the Gaussian distribution theoretically go to infinity, in practice with 100,000 points we see that not a single point was more than 4 units away from the mean of zero.



We won't get very far in pattern recognition just having random variables in one dimension. From [Section 4.3](#) we saw in [\(4.44\)](#) how to

create random clusters of points subject to a given mean and covariance, based on the eigendecomposition of the covariance. We can now try this:

```
% Choose a cluster to create:  
% Number of points, mean, and covariance  
N = 400;  
mu = [3;2];  
S = [4 1.5; 1.5 1];  
  
% Take eigendecomposition of covariance  
[v,d] = eig(S);  
  
% Build up new random points  
% Everything is transposed because Matlab has  
% data points in rows rather than in columns  
pts = mu' + (v*sqrt(d)*randn(2,N))';  
  
% Find mean and covariance of new points  
mean(pts)  
cov(pts)  
plot(pts (:,1), pts (:,2), ' .');
```



Target: $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$	Mean $\begin{bmatrix} 4.00 & 1.50 \end{bmatrix}$ Cov $\begin{bmatrix} 1.00 & 1.50 \end{bmatrix}$
Synthesized: $\begin{bmatrix} 3.18 \\ 2.08 \end{bmatrix}$	Mean $\begin{bmatrix} 3.94 & 1.52 \end{bmatrix}$ Cov $\begin{bmatrix} 1.52 & 0.95 \end{bmatrix}$

It works! Our random samples have been transformed to very nearly have the statistics of S . The slight inconsistency between the original target and the synthesized behaviour is due to the finite number, N , of points.

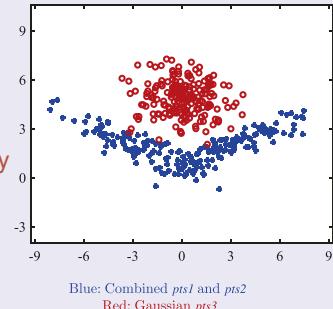
The above code gave us a single Gaussian cluster. In practice, we will need multiple classes, and also more interesting ones. We will now synthesize three sets of points, where the first two will be combined to give us a non-Gaussian cluster:

```
% Define three means and covariances  
mu1 = [3;2]; S1 = [5 2; 2 1];  
mu2 = [-3;2]; S2 = [5 -2; -2 1];  
mu3 = [0;5]; S3 = [2 0; 0 1];  
  
% Synthesize three sets of points  
[v,d] = eig(S1); pts1 = mu1' + (v*sqrt(d)*randn(2,100))';  
[v,d] = eig(S2); pts2 = mu2' + (v*sqrt(d)*randn(2,100))';  
[v,d] = eig(S3); pts3 = mu3' + (v*sqrt(d)*randn(2,200))';
```

```
% Plots the synthesized points
% pts1 and pts2 belong to the same class, pts3 is a second class
clf
plot(pts1 (:,1), pts1 (:,2), '*b');
hold on
plot(pts2 (:,1), pts2 (:,2), '*b');
plot(pts3 (:,1), pts3 (:,2), 'or');

% Scale axes equally for better interpretability
axis equal
```

So the code created two clusters and combined them (blue), creating a decidedly non-Gaussian/non-ellipsoidal class. Such combinations could easily be the basis for creating other unique class shapes.



As was pointed out in the [discussion](#) near the beginning of this lab, Gaussian distributions have very thin tails, so it is very rare to have significant outliers. In *practice*, however, we can most definitely encounter serious outliers, so we may need to develop classifiers to be robust to such outliers. We can easily synthesize outliers by generating a small fraction of points with a much larger variance. So, for example, for a given random vector $\underline{x} \sim \mathcal{N}(\underline{\mu}, S)$ we could synthesize from the distribution

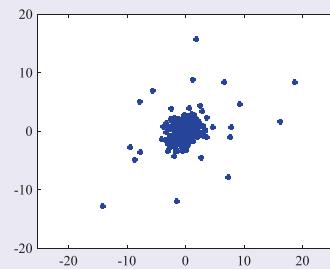
$$(1 - p) \cdot \mathcal{N}(\underline{\mu}, S) + p \cdot \mathcal{N}(\underline{\mu}, s \cdot S)$$

for outlier probability p and outlier amplification factor s .

```
% Create a class with outliers by superimposing points
mu = [0;0]; S = [3 1; 1 3];
[v,d] = eig(S);
pts = [mu+(v'*sqrt(d)*randn(2,200))'; % regular points
       mu+5*(v'*sqrt(d)*randn(2,20))']; % outlier points

clf
plot(pts (:,1), pts (:,2), '*b');
axis equal
```

In the code we have an amplification factor of $s = 5$ and we simulate 20 outliers on top of 200 regular points, thus $p = 1/11$. The resulting cluster is plotted, right, where we can very clearly see the dense main cluster, surrounded by a scattering of outlier points.



Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Most of the background material for this chapter can be found under the discussion of [matrix covariances](#) in [Appendix B](#) and of [the eigendecomposition](#) in [Appendix A](#). For a different perspective, you may wish to consult

Wikipedia Links: [Distance metric](#), [Covariance matrix](#), [Eigendecomposition](#)

To be sure there are a great many books discussion pattern recognition and fundamentals of pattern theory, although nearly all are aimed at a fairly advanced level, which is precisely the motivation for this present text.

One of the most significant books in the pattern recognition domain is that of Duda et al. [3], however there are many others from which to choose [2, 7, 8].

Much too difficult as introductory texts, the works by Grenander [5, 6] are exceptional in their depth and scope.

Sample Problems

Problem 4.1: Short Answer

Give a short definition of each of the following:

- Distance metric
- Prototype
- Set of classes
- Eigendecomposition
- Hypothesis testing

Problem 4.2: Short Answer

Offer brief answers to each of the following:

- (a) Give a few examples of distance metrics and offer a few tradeoffs/asumptions
- (b) Summarize what is meant by a Parametric versus a Non-parametric method
- (c) The text will generally be limited to illustrating one- and two-dimensional problems. What are some of the challenges associated with applying Pattern Recognition to high-dimensional problems?

- (d) This text shows many ellipses — shifted, rotated, rescaled etc.
- What do the ellipses represent? That is, why are we working with ellipses?
 - Why are axis-aligned ellipses convenient? How do they make a pattern recognition problem easier or simpler?

Problem 4.3: Conceptual — Class Ellipses

Draw a sketch showing the unit standard-deviation shape for the following two clusters, without computing any eigendecomposition:

$$\underline{\mu}_1 = \begin{bmatrix} 0 \\ -2 \end{bmatrix} \quad \Sigma_1 = \begin{bmatrix} 3 & -2 \\ -2 & 3 \end{bmatrix} \quad \underline{\mu}_2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \quad \Sigma_2 = \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix}$$

Problem 4.4: Conceptual — Class Ellipses

Suppose we have a cluster in *three* dimensions:

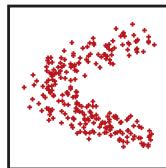
$$\underline{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \quad \underline{\mu}_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \quad \Sigma_x = \begin{bmatrix} 3 & 1 & 1 \\ 1 & 5 & -3 \\ 1 & -3 & 4 \end{bmatrix}$$

It is very hard to sketch a three-dimensional shape, however we *do* know how to sketch in two dimensions. Sketch the shape of the cluster in each of the following two-dimensional feature spaces:

- Feature space $\underline{x} = \begin{bmatrix} y_1 \\ y_3 \end{bmatrix}$.
- Slightly harder, feature space $\underline{x} = \begin{bmatrix} y_1 \\ y_2 + 2y_3 \end{bmatrix}$.

Problem 4.5: Conceptual — Class Representation

Suppose we have the arc-shaped class



from [Figure 4.3](#). Briefly describe a few options for representing this class. That is, what are the options for mathematically describing the shape of this class?

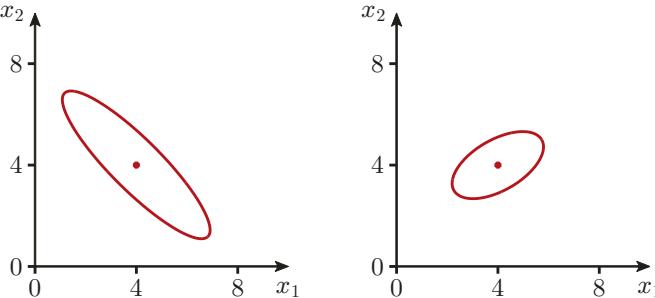
Problem 4.6: Analytical — Covariance Eigendecomposition

- (a) A 2×2 covariance has eigenvalues $\lambda_1 = 1, \lambda_2 = 4$ and eigenvectors

$$\underline{v}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \underline{v}_2 = \begin{bmatrix} 4 \\ -2 \end{bmatrix}.$$

What is the covariance? Sketch the unit-deviation contour assuming that the mean is at the origin.

- (b) For each of the two following¹⁴ unit standard-deviation contours, find the approximate corresponding covariances, eigenvalues, and eigenvectors:



Problem 4.7: Analytical — Covariance Eigendecomposition

It is convenient that the eigenvectors of covariance matrices always appear to be orthogonal. We can prove this:

Given a real, symmetric matrix A , prove that eigenvectors corresponding to different eigenvalues *must* be orthogonal.

Problem 4.8: Numeric/Computational — Cluster Synthesis

We saw cluster synthesis described in [Section 4.3](#) and [Lab 4](#). Develop short programs to synthesize each of the following:

- (a) We would like to create a negatively-correlated non-zero mean Gaussian cluster. So, for example, we would like the cluster statistics to be

$$\mu = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 4 & -2 \\ -2 & 3 \end{bmatrix} \quad (4.53)$$

- (b) We would like to create a ring-like cluster (like the one shown in [Figure 12.5](#)). The cluster is most easily created in polar coordinates:

¹⁴ Please note that figure templates for most problem figures are linked to the text home page at [textbook web site](#).

- The radius is Gaussian distributed, so $r \sim \mathcal{N}(1, 0.05)$
- The angle (in radians) is Uniformly distributed, so $\theta \sim \mathcal{U}(0, 2\pi)$

Problem 4.9: Real-World, Open-Ended — Distance Metrics

Any pattern recognition problem critically relies on some understanding of what a class looks like, and some sense of what a corresponding effective distance metric would be. In most problems we would need real data in order to try to visualize the problem, however a familiar problem should give us some understanding, so for each of the following contexts offer some thoughts about what an effective distance metric might be:

- (a) Suppose we want to develop a classification algorithm to match fingerprints.
 - (i) For a given person, what aspects of their fingerprints can vary from day to day? These would normally *not* represent very good measurements, and any distance metric should not be too sensitive to such aspects of a fingerprint.
 - (ii) What might be more robust, time-invariant aspects of fingerprints, which would play a more significant role in a distance metric?
- (b) Suppose we wish to build a pattern-recognition system to recognize human faces. How would you start? What might you measure? What sort of “distance” might be appropriate here? (See also [Example 4.1](#))
- (c) Suppose you want to build a pattern-recognition system to recognize handwritten letters. How might you go about solving this problem? How might we define a “distance” between one handwritten letter and another? (See also [Example 2.1](#) and [Case Study 6](#))

References

1. P. Bellhumer, J. Hespanha, D. Kriegman, Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Trans. Pattern Anal. Mach. Intell.* **17**(7) (1997)
2. C. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2011)
3. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley Interscience, 2009)
4. A. Georghiades, P. Belhumeur, D. Kriegman, From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intell.* **23**(6), 643 (2001)
5. U. Grenander, *General Pattern Theory* (Oxford Science Publications, 1994)
6. U. Grenander, *Elements of Pattern Theory* (Johns Hopkins University Press, 1996)
7. S. Rogers, M. Girolami, *A First Course in Machine Learning* (Chapman and Hall, 2016)
8. A. Webb, K. Copsey, *Statistical Pattern Recognition* (Wiley, 2011)



Feature Extraction and Selection

Chapter 4 explored questions of pattern shape, and how the shape of a pattern class might be understood and represented. However what actually *determines* the size or shape of a given class? Ideally a given pattern class, such as the “lemon” class from the beginning of Chapter 4, should occupy a small, focused part of the colour — shape — size — texture feature space. So a key part of representing a pattern is determining the set of features which most effectively discriminate one pattern from another, and which then determine the shape of each pattern class within the feature space.

5.1 Fundamentals of Feature Extraction

The class shapes of Chapter 4 exists in some measurement space \mathcal{Y} or feature space \mathcal{X} . In other words, repeating (2.1), it is essential that we understand how feature vector $\underline{x} \in \mathcal{X}$ may be derived from measurement vector $\underline{y} \in \mathcal{Y}$:

$$\text{Measurements } \underline{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \xrightarrow{\underline{x} = f(\underline{y})} \text{Features } \underline{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad (5.1)$$

As was already introduced in Section 2.2, the action of $f()$, normally taking us from some number of measurements to fewer features (Figure 5.1), is a special case of what is known as *dimensionality reduction*. Function $f()$ is subject to the Data Processing Theorem, meaning that $f()$ may simplify the problem, but it is never *adding* information in any way; really, by reducing the dimensionality of the problem, $f()$ can only be *losing* information.

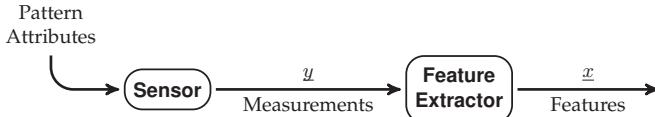


Fig. 5.1. FEATURES: A feature is some value, typically a real number, which is related to the given classification problem. The purpose of feature extraction is to identify those functions of the measurements which are related to the problem, eliminating redundancy and irrelevance.

Therefore, to avoid losing any information, the most trivial case is to set the features equal to the measurements:

$$\underline{x} = \underline{y} \quad \text{or, equivalently,} \quad \underline{x} = I \cdot \underline{y}. \quad (5.2)$$

That is, the transformation operator from measurements to features is just I , the identity matrix. In cases where the measurements are judiciously chosen — relevant and non-redundant — keeping all of the measurements as features may very well be a good choice. However, in most pattern recognition problems we will be given many measurements (say, the millions of pixels in an image), and we will be better off to reduce this to a far smaller set of features.

Although keeping all of the measurements technically preserves more information, there are practical reasons why having fewer features might be preferable, aspects of which are explored in [Figure 5.2](#):

- A finite sample size (finite number N of samples) limits the dimensionality n for which statistical estimates are reliable.
- Having features which are correlated with one another may, in fact, degrade performance. As a result, for some classifiers, performance can be improved by removing such redundancy.
- In most cases, classifier complexity increases with the number n of features.

On an intuitive level, the extraction of features might exploit prior knowledge about the measurements or certain behaviours and sensitivities of the chosen classification scheme. However such an intuitive approach is strongly application dependent, and here we are interested in more general approaches.

Following on [\(5.2\)](#), the next simplest case, conceptually, is to preserve only a *single*, scalar linear function of the measurements,

$$\underline{x} = f(\underline{y}) = f^T \underline{y} \equiv f \bullet \underline{y} \quad (5.3)$$

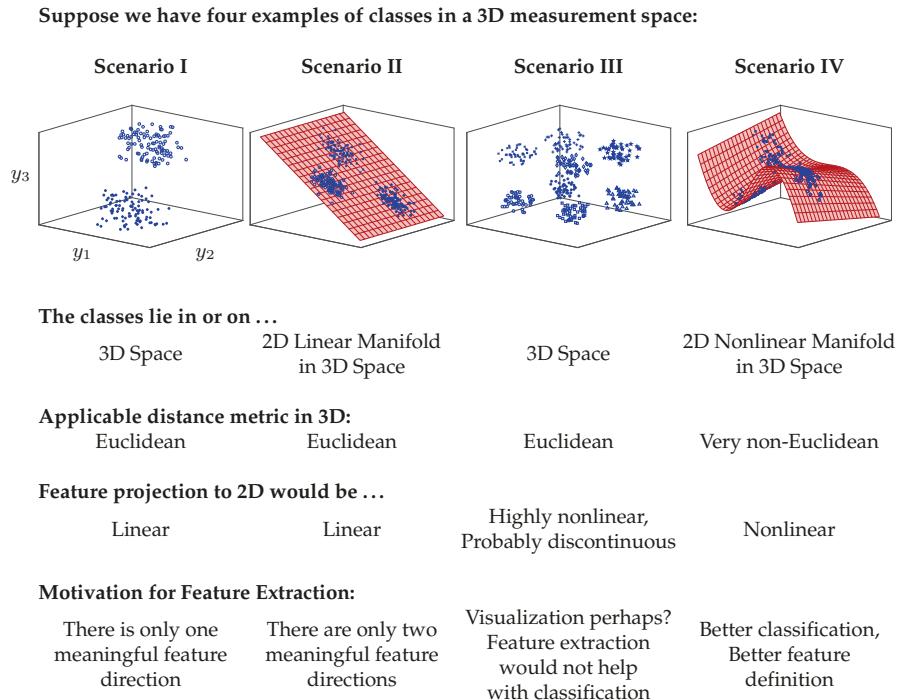


Fig. 5.2. FEATURE EXTRACTION: There are many possible ways that classes could be arranged in a higher-dimensional space, of which four are illustrated here. For the first three scenarios the classification problem could readily be solved in the original space, so the motivation for feature extraction is to make the problem smaller or for visualization purposes. Observe that in Scenario III the eight clusters are well-separated in the given measurement space, but will almost certainly overlap in any pair of extracted features, unless the features were very unusually defined (Problem 5.4) to significantly move the classes around. Scenario IV would require an unusual distance metric in the given space, and might be better solved if we could find nonlinear features which followed the manifold. A manifold is most easily thought of as a two-dimensional surface (a curved sheet of paper) in a three-dimensional space, but more generally is a lower-dimensional hyper-surface existing in a higher-dimensional space.

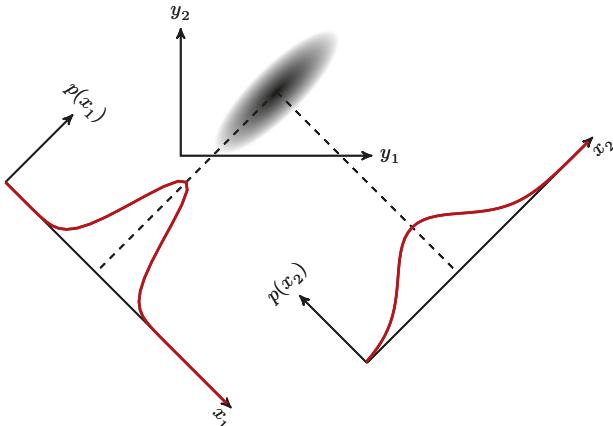


Fig. 5.3. LINEAR PROJECTION: Given a two-dimensional distribution characterizing a class, we could choose to extract one or more features by projecting this distribution onto one-dimensional axes. The figure illustrates two projections, onto features x_1 and x_2 . Clearly, the shape of the projected class can be a function of the feature, here significantly narrower in feature x_1 than in feature x_2 .

where the feature extraction function $f()$ is expressed as a dot product.¹ Two illustrative one-dimensional linear functions of a two-dimensional distribution are shown in Figure 5.3. These linear functions are effectively a projection of a class shape (Section 4.2), whether in the form of a geometric shape, a distribution, or a set of data points, from a higher dimension onto a single feature axis, by preserving the information only in some direction f . From (B.37) and as we already saw in Section 4.3, we do know the algebra associated with Figure 5.3, how a linear transformation influences class statistics:

$$\text{Mean}(f^T y) = f^T \cdot \text{Mean}(y) \quad \text{Var}(f^T y) = f^T \cdot \text{Var}(y) \cdot f \quad (5.4)$$

With the mathematics of the transformation known,² the question, then, is *how* an effective feature direction f is to be chosen. The two plausible, basic strategies are sketched in Figure 5.4:

1. **Find a direction f in which individual classes are smallest or most compact.** Presumably such a feature is effective, since it is the direction in which a pattern class has little variability, and therefore hopefully is most distinct.

¹ There is the possibility for confusion as to whether \cdot indicates multiplication or a dot product. We will need dot products only rarely, but will use the larger ($a \bullet b$) in those cases to indicate the dot product, and the smaller ($a \cdot b$) to indicate multiplication.

² It is important to understand that (5.4) and the discussion in Appendix B.4 apply only to *linear* transformations. For nonlinear transformations the mathematics are *much* more complicated.

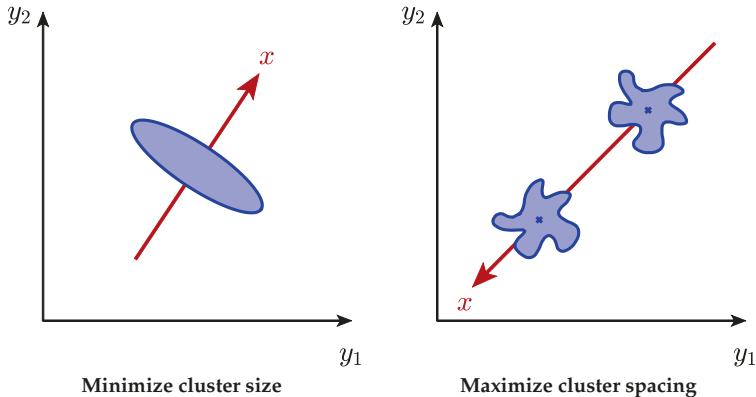


Fig. 5.4. FEATURE EXTRACTION: The two most basic strategies in deciding upon a promising direction (red) for a feature x : Select a feature direction in which individual classes are most compact (left), or select a feature in which the classes are maximally separated (right).

For a given class $C \sim \Sigma$, from [Figure 4.8](#) we know the class to be most compact in the direction of the eigenvector corresponding to the smallest eigenvalue of Σ :

$$\begin{aligned} \text{Given the eigendecomposition } \Sigma \underline{v}_j = \lambda_j \underline{v}_j \\ \rightarrow f_{\text{Compact}} = \underline{v}_\alpha \text{ where } \lambda_\alpha = \min\{\lambda_1, \dots, \lambda_m\} \end{aligned} \quad (5.5)$$

Clearly this definition of feature generalizes to multiple features, $n > 1$, whereby we select the n eigenvectors associated with the n smallest eigenvalues.

2. **Find a direction f in which classes are maximally separated.** Presumably such a feature is effective, since it is the direction in which classes are most different, and therefore hopefully most easily separated.

This direction is sometimes taken to be the difference between the cluster means,

$$f_{\text{Separate}} = \mu_1 - \mu_2 \quad (5.6)$$

when given only two clusters, or the direction associated with a *largest* eigenvalue:

$$\begin{aligned} \text{Given the eigendecomposition } \Sigma_T \underline{v}_j = \lambda_j \underline{v}_j \\ \rightarrow f_{\text{Separate}} = \underline{v}_\beta \text{ where } \lambda_\beta = \max\{\lambda_1, \dots, \lambda_m\} \end{aligned} \quad (5.7)$$

We then require some definition of this overall or total covariance³ Σ_T :

$$\Sigma_T = \underbrace{\text{cov}(\mu_1, \dots, \mu_K)}_{\text{Covariance over means}} + \underbrace{\frac{1}{K} \sum_{\kappa=1}^K \Sigma_\kappa}_{\text{Covariance over classes}}$$

$$\Sigma_T = \text{cov}(y_1, \dots, y_N) \quad (5.8)$$

$\underbrace{}$
Covariance over data

As before, this definition of feature can generalize to multiple features $n > 1$ whereby the eigenvectors associated with the n largest eigenvalues are kept.

Both of the above are variations of *principal components analysis*, a very common strategy in dimensionality reduction. Given a random vector y , such that

$$y \in \mathbb{R}^m \quad y \sim \Sigma \quad (5.9)$$

then the principal components problem asks to identify the n best linear features $f_1, f_2, \dots, f_n \in \mathbb{R}^m$, in order to minimize the mean-squared error (MSE) in reconstructing y from the features:

$$\text{Find } f_1, f_2, \dots, f_n \text{ to minimize } \text{MSE} \left(y - \sum_{j=1}^n \alpha_j f_j \right) \quad (5.10)$$

$$\text{where } \text{MSE}(y - \bar{y}) = \frac{1}{m} \sum_{j=1}^m (y_j - \bar{y}_j)^2$$

for the best possible $\alpha_1, \dots, \alpha_n$. The result, derived in [Appendix D](#), is to keep the n orthogonal linear functions of y having the greatest variance. The optimal orthogonal functions are the n eigenvectors corresponding to the largest eigenvalues of Σ :

$$\begin{aligned} \text{Given } \Sigma v_j = \lambda_j v_j \quad \text{where } \lambda_1 \geq \lambda_2 \geq \dots \lambda_m \geq 0 \\ \text{then let } f_1 = v_1 \dots f_n = v_n \end{aligned} \quad (5.11)$$

In other words, the principal components, the linear functions of y having the greatest significance, are those feature directions (the eigenvectors) of greatest variance, absorbing or preserving the greatest amount of statistical information from y . One limitation with principal components is that it tacitly treats all of the measurements as being in comparable units since the MSE, the distance metric in [\(5.10\)](#), treats each measurement equivalently.

³ The derivation for the covariance over classes in [\(5.8\)](#) can be found in [Appendix D](#).

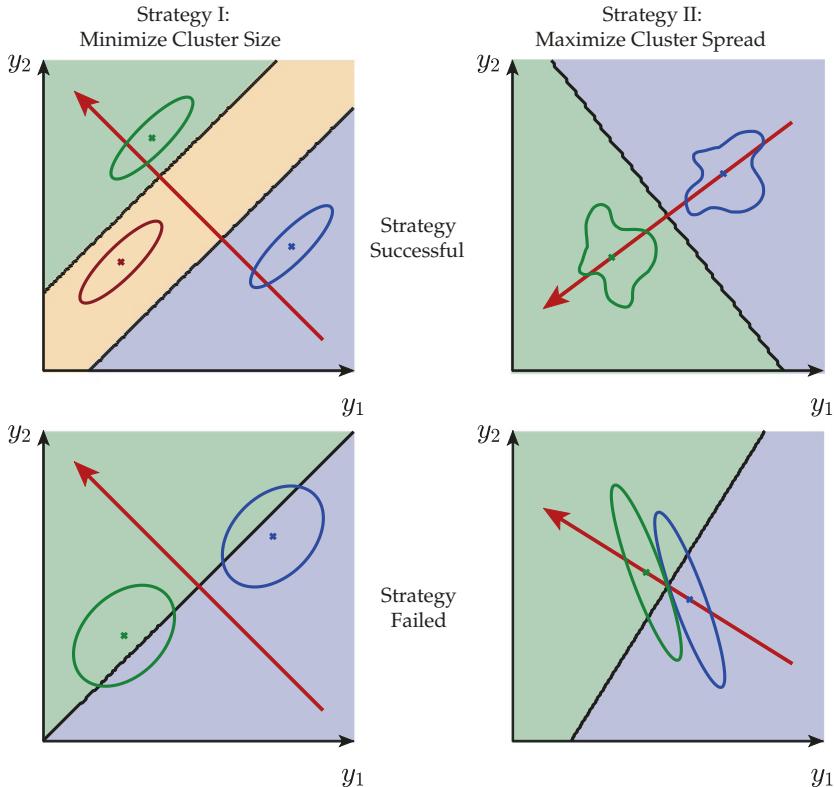


Fig. 5.5. FEATURE EXTRACTION: There are simple illustrations where the two strategies of Figure 5.4 either succeed (top row) or fail (bottom row). In all panels the selected feature direction is given by the red arrow, and the background shading shows the resulting classification. The successful strategies (top) have each class entirely within its corresponding classification region, as opposed to crossing over classification regions (bottom).

Having the eigendecomposition understood as *optimal* in some statistical sense by no means guarantees that it is optimal for a given pattern recognition problem. Indeed, we need only a few sketches in Figure 5.5 to quickly identify counterexamples to the two strategies just discussed, and which were somewhat optimistically idealized in Figure 5.4. In particular, minimizing class size is helpful only if that direction is *also* effective at separating classes, and maximizing class spacing can be a poor choice if that direction is inconsistent with the class shapes. So really what we seek are features which maximize the class separation *relative* to the class size, as illustrated in Figure 5.6.

Suppose we are given classes C_1, \dots, C_K in the m -dimensional measurement space of \mathcal{Y} , with class means and covariances

$$C_{\kappa|y} \sim (\mu_\kappa, \Sigma_\kappa) \quad (5.12)$$

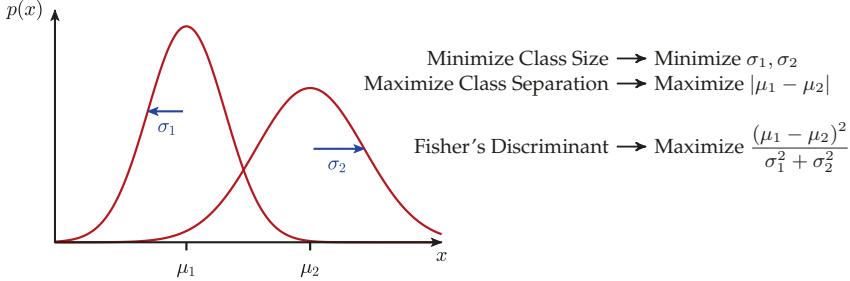


Fig. 5.6. FEATURE EXTRACTION: Building on Figures 5.4 and 5.5, it is not just class size or separation on their own, which matter, rather class separation *relative* to class size, which is the criterion asserted by Fisher's discriminant.

For a feature direction f , such that $x = f^T y$, the classes projected onto x have means and variances

$$C_{\kappa|x} \sim (\mu_{\kappa|x}, \sigma_{\kappa|x}^2) \quad \text{where} \quad \mu_{\kappa|x} = f^T \mu_{\kappa} \quad \sigma_{\kappa|x}^2 = f^T \Sigma_{\kappa|y} f \quad (5.13)$$

An illustration for the $K = 2$ class case in two dimensions is shown in Figure 5.7, where it is clear that the statistics of the projected classes vary with the feature

$$f = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \quad \text{for projection angle } \theta. \quad (5.14)$$

If we let Σ_W represent the average *within* class statistics

$$\Sigma_W = \sum_k \Sigma_k \quad (5.15)$$

and Σ_T the *total* statistics, as in (5.8), then Fisher's criterion⁴ says to

$$\text{Find } f \text{ to maximize } J(f) = \frac{f^T \Sigma_T f}{f^T \Sigma_W f} \quad (5.16)$$

that is, to maximize the projected class spread *relative* to the projected class size. To maximize $J(f)$, we could perform an optimization, for example searching over θ in (5.14) to find the best feature in Figure 5.7. However, we will obtain further insight by finding the maximum analytically, by differentiating⁵ with respect to f :

$$\frac{\partial J(f)}{\partial f} = 0 \quad \rightarrow \quad \frac{\partial J(f)}{\partial f} = \frac{2 \Sigma_T f}{f^T \Sigma_W f} - 2 \Sigma_W f \frac{f^T \Sigma_T f}{(f^T \Sigma_W f)^2} = 0 \quad (5.17)$$

⁴ The linear transformation of covariances is discussed in Appendix B.4 and derived in Appendix D.

⁵ Vector derivatives are briefly described in Appendix A, particularly Table A.1.

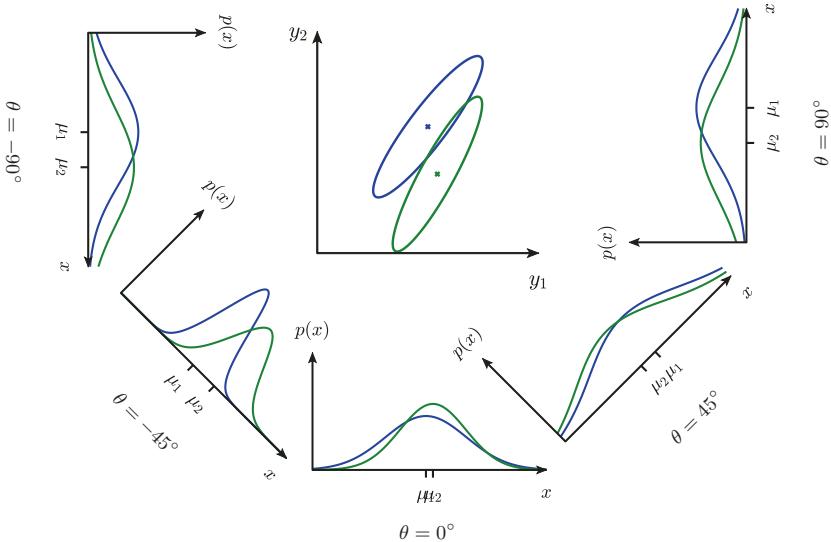


Fig. 5.7. The principle of Fisher’s Discriminant was articulated in [Figure 5.6](#), such that a goal of feature extraction could be to maximize class separation *relative* to class size. So given measurements in two dimensions, building on the projection illustrated in [Figure 5.3](#), we can now ask *which* projection might lead to the best feature since, as is clear here, the means and variances are a function of projection direction.

Multiplying through by the scalar,⁶ factor $(f^T \Sigma_W f)$ and simplifying⁷

$$\Sigma_T f = \frac{f^T \Sigma_T f}{f^T \Sigma_W f} \Sigma_W f \quad \rightarrow \quad (\Sigma_W^{-1} \Sigma_T) f = J(f) f, \quad (5.18)$$

which we recognize as an eigendecomposition. That is, $J()$, the thing we wish to maximize, must be an eigenvalue of matrix $(\Sigma_W^{-1} \Sigma_T)$, an attractive result:

The Fisher discriminant which we seek, f_{Fisher} , is the eigenvector corresponding to the largest eigenvalue of $(\Sigma_W^{-1} \Sigma_T)$.

In the event that we have a $K = 2$ class problem, and where Σ_T from [\(5.8\)](#) is chosen as,⁸

⁶ Even though [\(5.17\)](#) looks fairly complicated, it is always important to keep in mind what sort of object we are manipulating. So both $(f^T \Sigma_T f)$ and $(f^T \Sigma_W f)$ are a row vector · matrix · column vector product, which is just a scalar. So anything you can do with a scalar can also be done with $(f^T \Sigma_W f)$.

⁷ Clearly [\(5.18\)](#) is assuming the class covariance Σ_W to be invertible, which should essentially always be the case; see the discussion in [Appendix B](#).

⁸ See [derivation](#) in [Appendix D](#).

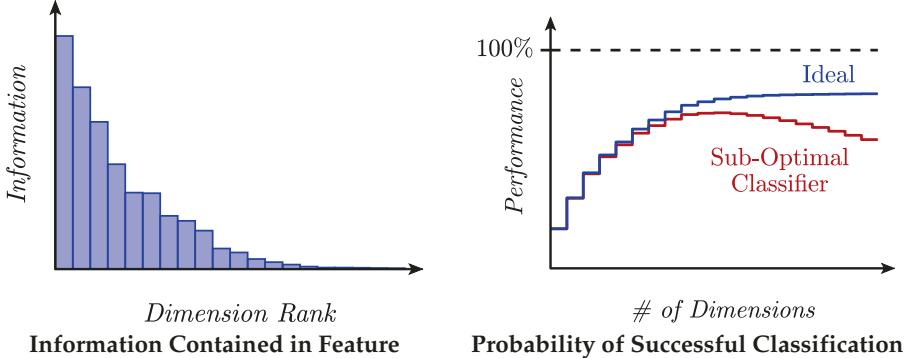


Fig. 5.8. DIMENSIONALITY REDUCTION: One of the primary goals of feature extraction is to limit the number of features; that is, to limit the dimensionality of the feature space. Given a set of features, we could sort them by their relevance or information content, left. Although keeping more features ideally leads to better performance, the *degree* of improvement will level off, right, and for certain classifiers (red) performance can decrease as the number of features increases. Observe that keeping *all* features most certainly does not guarantee perfect performance.

$$\Sigma_T = \text{cov}(\mu_1, \mu_2) = \frac{1}{2}(\mu_1 - \mu_2)(\mu_1 - \mu_2)^T \quad (5.19)$$

then (5.18) simplifies more easily without an eigendecomposition, as

$$f = \frac{1}{J(f)} (\Sigma_W^{-1} \Sigma_T) f = \frac{1}{J(f)} \Sigma_W^{-1} \frac{1}{2} (\mu_1 - \mu_2) (\mu_1 - \mu_2)^T f \propto \Sigma_W^{-1} (\mu_1 - \mu_2) \quad (5.20)$$

Since it is only the *direction* of f that matters, and not its magnitude (length), the constant of proportionality is irrelevant, and we select Fisher's discriminant as

$$f_{\text{Fisher}} = \Sigma_W^{-1} (\mu_1 - \mu_2), \quad (5.21)$$

meaning that the feature is selected as the direction between the two means, rotated by an amount Σ_W^{-1} controlled by the class shape.

Finally, we need to allow for $K > 2$ classes. Fisher's criterion in (5.16) was written only for a single feature f , whereas for many classes we may very well need *multiple* features ($n > 1$), so that the feature extraction is expressed as

$$\underline{x} = F^T y \quad F \in \mathbb{R}^{m \times n}, \quad (5.22)$$

such that each column of F describes one feature direction. The criterion in (5.16) takes the ratio of two scalars, the spread between classes relative to the size of individual classes, a concept which does not obviously generalize

to multiple features. We take advantage that a matrix *determinant* measures the *size of volume* of a given covariance, therefore maximizing the multi-dimensional “size” between classes relative to the multi-dimensional “size” of individual classes can be expressed as

$$\text{Find } F \text{ to maximize } J(F) = \frac{\det(F^T \Sigma_T F)}{\det(F^T \Sigma_W F)} \quad (5.23)$$

As before, in (5.17), to maximize $J(F)$ we find its extremum by differentiating⁹ with respect to matrix F :

$$\frac{\partial J(F)}{\partial F} = 0 \quad \longrightarrow \quad (\Sigma_W^{-1} \Sigma_T) F = F \Lambda, \quad (5.24)$$

with the details derived in [Appendix D](#). That is, we solve the same eigen-decomposition as before, in (5.18), but now keeping in F the n eigenvectors corresponding to the n largest eigenvalues of $(\Sigma_W^{-1} \Sigma_T)$.

We are left with deciding how large n should be, such that the features contain meaningful information regarding the classes, and not beginning to fit the noise or variability associated with the measurements. For Fisher’s method of (5.24), we are limited to $n < K$, for K classes, since we require at most $K - 1$ directions to separate K compact classes.¹⁰ More generally, the question of selecting n very much parallels the question of selecting model order in [Chapter 3](#), as was illustrated in [Figure 3.3](#), and the selection of n lies at the crux of dimensionality reduction, as illustrated in [Figure 5.8](#).

One difference is that for the model learning in [Figure 3.3](#) there was a “true”, correct model order that we wanted to infer, so in principle we could formulate strategies to estimate the model order and validate the performance of each strategy. In contrast, here there may not be any “true” feature dimensionality n , so there is no way to validate whether our chosen n is “correct” or not, and instead we typically need to choose n based on the [feature-dimensionality tradeoffs](#) as discussed at the beginning of this section.

5.2 Feature Extraction and Selection

The Fisher discriminant $J(f)$ from (5.16) is compelling and well motivated, and in certain specific contexts (to be discussed in [Chapter 8](#)) maximizing $J()$

⁹ A matrix derivative seems confusing, but is really just a matrix of derivatives of scalar function $J()$ with respect to individual elements of F . The needed derivatives are briefly described in [Appendix A](#), particularly [Table A.1](#).

¹⁰ ... and, depending on the definition of Σ_T back in (5.8), $\text{rank}(\Sigma_T) = K - 1$, which limits the number of non-zero eigenvalues of $\Sigma_W^{-1} \Sigma_T$ to $K - 1$.

does in fact lead to the lowest probability of classification error. However in general the Fisher discriminant is more or less a *guess* of the statistical attributes or shapes or behaviours of a set of classes and, at the same time, the strengths and weaknesses of a given classifier. After all, the data processing theorem of [Section 2.2](#), the principle of parsimony in [Chapter 3](#), and the tradeoff implied by the sub-optimal classifier in [Figure 5.8](#) (the possibility that a classifier may perform *worse* if n is sufficiently increased) all pull us in different directions:

- The data processing theorem argues towards keeping *all* measurements, performing no feature extraction;
- The principle of parsimony argues towards keeping as *few* features as possible;
- The tradeoff implied by sub-optimal classifiers suggests that we *don't know* how many features to keep, and that we may need to infer n empirically from the problem.

As a result, we cannot expect a single universal or standardized approach to feature extraction to work well in every scenario. The following six topics are intended to illustrate the breadth of the process of inferring features from measurements:

1. **Different Criteria:** The Fisher criterion of [\(5.16\)](#), [\(5.23\)](#) is only one possible criterion by which to extract a feature; indeed, we had already seen two other criteria, that of minimizing the class size [\(5.5\)](#) or of maximizing the class spread [\(5.7\)](#).

Many other criteria could be formulated. For example, Fisher's criterion of [\(5.23\)](#),

$$\text{Find } F \text{ to maximize } J(F) = \frac{\det(F^T \Sigma_T F)}{\det(F^T \Sigma_W F)} \quad (5.25)$$

lumped all classes together, using a single covariance Σ_W to represent an overall average class shape, and another single covariance Σ_T to capture the distribution of classes. Neither Σ_W nor Σ_T necessarily captures the details of individual class shapes or pairwise separations. Perhaps we would like to ensure that *all* pairs of classes are well separated, therefore proposing a modified criterion something like

$$\text{Find } F \text{ to maximize } \min_{ij} \frac{\det(F^T \Sigma_T^{ij} F)}{\det(F^T (\Sigma_W^i + \Sigma_W^j) F)} \quad (5.26)$$

that is, to maximize the worst-case separation, where Σ_W^i is the covariance of class i , and Σ_T^{ij} is the total covariance of the pair of classes i, j . The “min” operator makes this a nonlinear optimization problem, and so much harder to solve, but possibly with far more effective features.

Other criteria could be proposed, such as based on correlation (between feature and class), or information-theoretic notions of entropy or mutual information (see [Further Reading](#)).

- 2. Features in Particular Contexts:** Pattern Recognition is widely applied in a variety of fields, some of which are fairly specialized in ways that motivate a focused/specialized approach to feature extraction. Two domains are particularly significant:

- **Image Processing:** An image can contain millions of pixels, so

$$y \in \mathbb{R}^m \quad m \approx 10^7 \quad (5.27)$$

It almost certainly does not make sense to try to find eigendecompositions of $10^7 \times 10^7$ covariance matrices. Furthermore, the objects we would like to classify (a car, a face, an animal) could appear anywhere in the image, perhaps at different rotation angles and illuminations, and so would require a vast number of samples to adequately cover all variations. Furthermore, in any given image, normally the majority of the pixels are so-called background (not part of the object of interest), and so are not relevant to the classification.

Instead, a great many image-related features have been developed: edge detectors, corner detectors, line/shape detectors (Hough transform), scale-invariant feature detectors (SIFT, SURF). The details of these methods are outside the scope of this text, however pattern recognition and image processing/computer vision are highly complementary fields, and a greater understanding of image processing and computer vision is highly recommended ([Further Reading](#)).

- **Speech Processing:** Thirty seconds of speech recorded at 10 kHz results in 300,000 samples. As with image processing, we would not want to be estimating $300,000 \times 300,000$ sample covariances. However, even if we could, the amount of speech contained in 30 seconds could be highly variable, from a few words to a great many. Instead, the audio signal is broken into small pieces, each assumed to contain a single sound, from which the so-called *Cepstrum*¹¹ coefficients

$$\underline{x}_{\text{Cepstrum}} = |\mathcal{F}^{-1} \{ \log (|\mathcal{F}(y)|) \}| \quad (5.28)$$

for Fourier transform \mathcal{F} and inverse Fourier transform \mathcal{F}^{-1} , yielding a feature vector $\underline{x}_{\text{Cepstrum}}$ which can be used as the basis for the recognition of sounds and then speech.

¹¹ “Cepstrum” is an anagram of “Spectrum”, intending to convey a sense of re-arranging the spectrum.

3. Feature selection: So far we have considered feature extraction,

$$\underbrace{\underline{x} = \underline{f}^T \underline{y}}_{\text{Single linear feature}} \quad \text{or} \quad \underbrace{\underline{x} = \underline{F}^T \underline{y}}_{\text{Multiple linear features}} \quad \text{or} \quad \underbrace{\underline{x} = f(\underline{y})}_{\text{Non-linear features}} \quad (5.29)$$

such that the feature \underline{x} is some function of all of the measurements \underline{y} . In cases where there are a *great* many measurements (such as in DNA sequencing), where many or most of the measurements are irrelevant to the classification problem at hand, we would prefer feature *selection*, which is to select a *subset* of the measurements as the features. That is, each element of \underline{x} is a single element of \underline{y} . We can express this selection mathematically using indices,

$$\underline{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_{j_1} \\ \vdots \\ y_{j_n} \end{bmatrix} \quad \text{for } 1 \leq j_1 < \dots < j_n \leq m \quad (5.30)$$

or via a so-called *selection matrix* S , $\underline{x} = S^T \underline{y}$, where

$$\begin{aligned} S_{\alpha\beta} \in \{0, 1\} &\leftarrow \text{All elements of } S \text{ are binary} \\ \sum_{\alpha} S_{\alpha\beta} = 1 &\leftarrow \text{Exactly one element in each column equals one} \end{aligned} \quad (5.31)$$

such that the 1's in S pull out the selected values of \underline{y} .

Feature selection has significant advantages over feature extraction: each feature preserves the interpretability and units of the original measurement, and the required *number* of measurements is greatly reduced. That is, if a large number m of measurements is proposed in a trial and feature selection is able to reduce this to $n \ll m$ measurements, then the $m - n$ unused measurements no longer need measuring, which can be a significant simplification and cost saving. In contrast, with feature extraction, even if $n \ll m$, typically *all* of the measurements are still required.

On the other hand, feature selection has significant disadvantages, specifically with regards to optimization. That is, feature selection is *discrete* (a given measurement is either used or not-used), as opposed to feature extraction, which is continuous. The feature selection problem is therefore nonlinear. We still need to specify a criterion $J(S)$, where J could be any of the criteria which we have already seen, such as (5.5), (5.7), (5.25), (5.26); we then need to optimize

$$\text{Find } 1 \leq j_1 < \dots < j_n \leq m \text{ to minimize } J(\underline{x}) = J(y_{j_1}, \dots, y_{j_n}) \quad (5.32)$$

The problem is that this is, in principle, a search over all $\binom{m}{n}$ possibilities for the values of indices $j_1 < \dots < j_n$. Probably one of the simplest alternative strategies is a greedy growing approach:

$$\text{First find } 1 \leq j_1 \leq m \text{ to minimize } J(y_{j_1}) \quad (5.33)$$

$$\text{Then find } 1 \leq j_2 \leq m \text{ to minimize } J(y_{j_1}, y_{j_2}) \quad (5.34)$$

$$\vdots$$

This approach seems optimal, but it is not: y_{j_1} is the best *single* feature, however that does not necessarily make y_{j_1} part of the best *pair* of features. There are many alternatives, such as genetic algorithms or simulated annealing, which allow for indices to be both added and removed from the feature set.

- 4. Wrapper approaches:** The wrapper principle was introduced in [Example 3.4](#). The wrapper idea is to solve a given problem, such as feature extraction or feature selection, *without* needing to know the details of the classifier or the statistics of the classes, rather to have the nature/behaviour of the classifier vis-a-vis the underlying problem guide and implicitly specify what a good feature might be.

For example, returning to the context of [Figure 5.7](#), we have a linear feature from [\(5.14\)](#),

$$f(\theta) = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \quad \text{for projection angle } \theta. \quad (5.35)$$

We had asserted that maximizing $J()$ would lead to a good classifier, however we can solve for the optimal feature more explicitly by wrapping it *around* an assessment of the probability of classification error $\mathbf{P}(e)$ of the classifier $g(f)$ which results from feature $f(\theta)$:

$$\text{Optimal Feature Angle } \hat{\theta} = \arg_{\vartheta} \min \mathbf{P}(e | g(f(\vartheta))) \quad (5.36)$$

giving us an optimization problem ([Appendix C](#)) to be solved.

Essentially the same approach as in [\(5.36\)](#) can be undertaken for multiple features

$$\underline{x} = F(\underline{\theta})^T \cdot \underline{y} \quad \text{or} \quad \underline{x} = f(\underline{y}, \underline{\theta}) \quad (5.37)$$

in linear and non-linear cases, respectively, where $\underline{\theta}$ is a vector encoding one or more parameters associated with the feature extraction.

Linear feature extraction can be solved analytically, as we illustrated for three criteria in [Section 5.1](#), so the computational complexity associated

with wrapper methods can be unattractive. In contrast, feature selection is necessarily nonlinear (because it is discrete), therefore there are no closed-form solutions and wrapper methods are commonly used. That is, $J()$ in (5.32)–(5.34) is a wrapper function, as in (5.36), so that the greedy method of (5.33)–(5.34) becomes

$$\text{First find } 1 \leq j_1 \leq m \text{ to minimize } \mathbf{P}(e|g(y_{j_1})) \quad (5.38)$$

$$\text{Then find } 1 \leq j_2 \leq m \text{ to minimize } \mathbf{P}(e|g(y_{j_1}, y_{j_2})) \quad (5.39)$$

That is, the minimization is *wrapped* around the probability of error assessment $\mathbf{P}()$, which itself is wrapped around the classifier $g()$.

5. Random Features: In contrast to the systematic/intuitive preceding approaches to feature extraction, an unusual alternative is to choose *random* features. That is, given a measurement y , a feature may be found as

$$\underline{x} = F^T y \quad F \in \mathbb{R}^{m \times n} \quad F_{\alpha, \beta} \in \mathcal{N}(0, 1). \quad (5.40)$$

That is, that the transformation matrix F is just a set of random values, typically Gaussian, as shown in (5.40), but could be other distributions as well.

A limitation is that the resulting features x_j are difficult to interpret,¹² since each feature is essentially a nonsensical function of the measurements. However in a great many contexts a set of random functions is actually a very effective strategy for dimensionality reduction, since the random functions tend to explore the measurement space \mathcal{Y} very thoroughly, whereas deliberately chosen features may make inappropriate assumptions which match \mathcal{Y} poorly. In other words, a set of random vectors are likely to point in a whole variety of directions, whereas pre-defined features will fit our expectations of the behaviour of the data, which may (accidentally or deliberately) ignore certain patterns in the measurements which turn out to be quite important.

Methods of random features are quite widely used, particularly in the area of *Compressed Sensing*, in which a random transformation is used to reduce the number of measurements, typically from an image of $m \times m$ pixels to a much smaller set of n features. Obviously many details of the image will be lost in undertaking such a transformation; however if you are not interested in the image itself, rather its classification into one of K classes, then in many cases the classification performance from the n random features is just as good as from the original image.

¹² Although interpretability limitations are not unique to random features, and would be the case of nonlinear features as well, particularly in large nonlinear networks, as in (5.44), and discussed in more detail in Section 11.4. The broader question of interpretability is discussed further in Case Study 11.

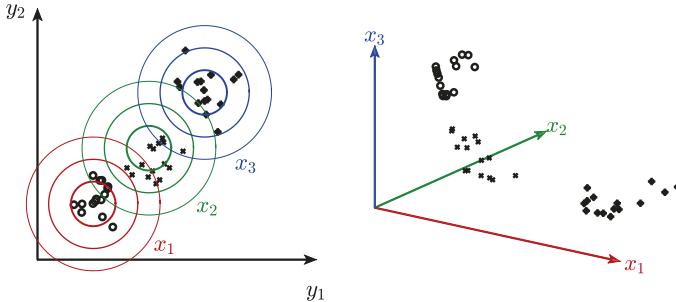


Fig. 5.9. NONLINEAR FEATURES: In many cases a *non-linear* feature may be preferable to linear. This figure illustrates radial features (left), such that feature r_j measures the distance to the centre of class C_j . The classes are well separated in the resulting feature space (right).

6. Nonlinear Features: All of the discussion in [Section 5.1](#) focused on linear features, primarily because the analysis of linear transformations is far more straightforward than of nonlinear transformations. Specifically, from [\(B.37\)](#) we know the effect of a linear transformation on class statistics. Building on [\(5.1\)](#), given a linear feature $\underline{x} = F^T \underline{y}$ from measurements $\underline{y} \sim \Sigma_y$, then the class statistics in the feature space \underline{x} are known:

$$\underline{y} \sim \Sigma_y \quad \xrightarrow{\underline{x} = F^T \underline{y}} \quad \underline{x} \sim F^T \Sigma_y F \quad (5.41)$$

The obvious disadvantage of limiting our attention to linear features is that in some contexts a bending or reshaping of the measurement space (see [Figure 5.2](#)) leads to a far superior feature than would be possible with any linear feature. A simple example of a nonlinear feature is illustrated in [Figure 5.9](#) of three radial features:

$$x_j = d(\underline{\mu}_j, \underline{y}) \quad (5.42)$$

where the j th radial feature is defined as the distance $d()$ to some location (such as a class mean) $\underline{\mu}_j$.

The example of [Figure 5.9](#) seeks to illustrate how a nonlinear feature could be defined, however the distribution of classes in the resulting feature space \mathcal{X} is not particularly more convenient than in the measured space \mathcal{Y} . A more compelling illustration of the benefit of the nonlinear reshaping of a measured space is shown in [Figure 5.10](#): two classes which cannot be separated by a straight line may, in fact, be separable by a straight line in a nonlinear feature space.

Given a nonlinear feature $\underline{x} = f(\underline{y})$, using the feature is not unusually difficult: we can transform the given data points $\underline{x}_i = f(\underline{y}_i)$, as we see in [Figures 5.9](#) and [5.10](#), and in principle the class statistics in the feature

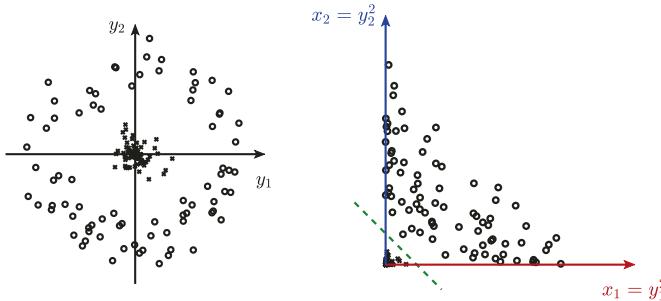


Fig. 5.10. NONLINEAR FEATURES: In many contexts a sufficiently rich set of nonlinear features allow for simpler class separability in the resulting feature space. Here two classes are *not* linearly separable (left), but are easily separated via a straight line (green) in a nonlinear feature space (right).

domain \mathcal{X} can be found empirically, to be discussed in [Chapter 7](#). It is not *using* the feature which is difficult, rather in *identifying* the nonlinear feature in the first place: how do we find $f()$? Broadly, there are two approaches:

1. Kernel methods, to be discussed in [Chapter 10](#);
2. Neural network methods, to be discussed in [Section 11.4](#).

In the Image Processing and Computer Vision domains, it is now very common to extract nonlinear features via deep neural networks, which will be discussed further in [Chapter 11](#). Although frequently treated as a somewhat magical or mysterious topic, in many ways deep neural networks are nothing more than a very large wrapper, optimizing an objective (typically called a Loss Function $\mathcal{L}()$) with respect to both feature extraction and classification.

That is, the information flow is very similar to (5.1), which we have seen repeatedly:

$$\underbrace{\underline{y}}_{\text{Measurement}} \rightarrow \underbrace{\underline{x} = f(\underline{y}, \underline{\theta}_{\text{Feature}})}_{\text{Feature Extraction}} \rightarrow \underbrace{c = g(\underline{x}, \underline{\theta}_{\text{Class}})}_{\text{Classification}} \rightarrow \underbrace{\mathcal{L}(C_{\text{True}}, c)}_{\text{Assessment}} \quad (5.43)$$

The resulting deep network is solved in very much the similar way as the wrapper of (5.36), such that given training samples (\underline{y}_i, C_i) the network weights in $\underline{\theta}$ are optimized to minimize the loss:

$$\text{Optimal Network} = \arg_{\theta_{\text{Feature}}, \theta_{\text{Class}}} \min \mathcal{L}\left(C_i, g\left(f(\underline{y}_i, \theta_{\text{Feature}}), \theta_{\text{Class}}\right)\right) \quad (5.44)$$

What makes deep networks stand out from other methods, such as those developed in this chapter, is that $f()$ and $g()$ are both nonlinear, and

also the vast size of the optimization, with $\theta_{\text{Feature}}, \theta_{\text{Class}}$ easily containing hundreds of *millions* of parameters.

7. Vector Embedding: Most of the classification methods in this text will have a fairly restrictive notion of their input feature. That is, the feature must be a vector of n real numbers, where n is fixed and does not vary from one sample to the next.

However, a great many phenomena which we routinely encounter may not fit this rigid framework:

- In **image processing**, as was already discussed under (5.27), a given image may contain a few objects or many objects, and the features which we could extract from an image (edges, corners etc.) may similarly vary in number.
- In the analysis of **spoken speech**, discussed above (5.28), the number of sounds/letters/words/sentences in a section of audio may be highly variable.
- In **text analysis**, such as some sort of automated processing and recognition of text on a patient's medical record or the analysis of the works of Shakespeare, the number of words will clearly vary greatly from one problem to the next.
- Any problem based on cardinal¹³ (non-ordered) data, which do not have a natural numeric representation.

A vector embedding is some mechanism whereby any of the preceding variable-length or unordered data are cast into a vector:

$$\begin{array}{c} \text{Irregular, Unordered} \\ \text{Variable-Length, Variable-Context} \\ \text{Data} \end{array} \xrightarrow{\text{Vector Embedding}} \underline{x} \in \mathbb{R}^n \quad (5.45)$$

In most problems the details of such embedding will be highly context-specific: more to do with the nuances of the problem being studied, and less to do with pattern recognition methods. Therefore the details of embedding are necessarily largely outside the scope of this text.

However one relatively broadly-applicable strategy to address vector embedding is known as *Bag of Words*, which converts a variable number of pieces of input information into a feature of fixed length describing some *distribution* of the input. In general, we can pre-define a set of n codes, such

¹³ An example of cardinal data would be the names of trees: Maple, Beech, Oak, Pine. They are all well-defined, distinct labels, but have no inherent ordering.

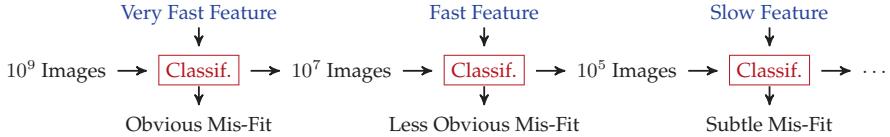


Fig. 5.11. IMAGE SEARCHING: The feature extraction process of Figure 5.1 can be generalized to apply features sequentially, as illustrated here, rather than all at once in a single feature vector \underline{x} . The key advantage is that the sequential approach allows us to begin with a vast number of images (left), applying a weak but exceptionally fast feature. This first step reduces the size of the dataset such that more discriminating (but also computationally more demanding features, right) are applied only to a small fraction of the original problem.

that the feature vector consists of counts of how often each code appeared in the input:

$$\underline{x}^T = \begin{Bmatrix} \# \text{ times} & \# \text{ times} & \dots & \# \text{ times} \\ \text{Code 1} & \text{Code 2} & \dots & \text{Code } n \end{Bmatrix} \quad (5.46)$$

which therefore gives us a fixed feature vector \underline{x} of length n , regardless of the length or complexity of the input text. So, for example, given the code words

"All" "Animals" "Are" "But" "Equal" "Farm" "Horse" "More" "Others"
"Some" "Than"

then the text

"All animals are equal, but some animals are more equal than others"¹⁴

is encoded into a feature vector as

$$\underline{x}^T = \begin{Bmatrix} \text{All} & \text{Animals} & \text{Are} & \text{But} & \text{Equal} & \text{Farm} & \text{Horse} & \text{More} & \text{Others} & \text{Some} & \text{Than} \\ 1 & 2 & 2 & 1 & 2 & 0 & 0 & 1 & 1 & 1 & 1 \end{Bmatrix} \quad (5.47)$$

In practice, of course, for real text analysis the number n of codes (unique words) would need to be far larger than eleven. However, there is significant latitude in how the codes are specified; for example, we could define codes in (5.46) as

$$\underline{x}^T = \{ \# \text{ nouns} \quad \# \text{ verbs} \quad \# \text{ adjectives} \quad \# \text{ adverbs} \quad \dots \}, \quad (5.48)$$

which would result in a *much* shorter feature vector, but one which might fail to capture meaningful patterns in a text. Obviously there are many

¹⁴ From *Animal Farm* by George Orwell, 1945.

variations, such that a given code could represent a word stem, capturing plurals, capitalizations, and different tenses, such that

Code = “Run” includes “Run”, “run”, “running”, “runs”, “ran” etc.

On the other hand, since (5.46) keeps only a count of code appearances, we might wish to preserve contextual information about which words are next to other words. For this purpose there are so-called bi-gram models, which take q words to form $n = q^2$ codes consisting of all word *pairs*, or tri-gram models to form $n = q^3$ word triples. Clearly the bi-gram and tri-gram models offer much more word-position information, but at a vastly increased feature vector length, which may introduce concerns regarding adequate training data and overfitting (Chapter 3).

The application of Bag-of-Words to image processing/computer vision contexts will require that we understand something about clustering, a topic which will be discussed in Chapter 12, so we will return to Bag-of-Words in Example 12.2 and Case Study 12.

Case Study 5: Image Searching

Most image searching is undertaken on the basis of text keywords — you type in a few words, and Google Images and other similar tools will search for web sites or specific images associated with the keywords. Although there are major logistical challenges associated with acquiring and storing many billions of images, the underlying pattern recognition problem is not necessarily particularly difficult, since most images are associated with a caption, a title, metadata, or text in the surrounding web page, all of which can be matched with the provided search keywords.

In contrast, *reverse* image search is a very different problem, whereby you provide an image, and the task is to find matching images, either from a database, or from across the Internet. That is, we are no longer matching on the basis of words (relatively easy), rather on the basis of image similarity (much more difficult). In order of increasing difficulty, we have

1. Given an image, find exact matches.
2. Given an image, find exact matches, but which may have been cropped, blurred, rotated, or printed/re-scanned. Such a search might be undertaken, for example, to look for copyright infringement.
3. Given an image, find other images which are visually or thematically similar.

4. Given a very rough (human-drawn) sketch of an image, find images which match the sketched elements.

There are tools which have been developed to tackle these challenges, such as TinEye, Google Images, and Pixsy, among others.

Assuming that the computer server infrastructure is in place to store and search trillions of images, one of the remaining key challenges is feature extraction: the identification of quantitative image features. What attributes do we require in a good feature:

- Fast to compute, since the number of images on the internet is vast;
- Adequately discriminating, so that irrelevant images are rejected;
- Adequately flexible, so that similar images are *not* rejected.

The choice of feature will almost certainly be problem-dependent, such that the features for identifying exact matches (category #2, above) will almost certainly be quite different from those features matching a rough sketch (category #4, above).

Commonly used features could include global features, such as size, aspect ratio, or colour distribution, or local features, such as edges, corners, or gradients. Since the *number* of edges, corners, gradients etc. can vary greatly from one image to another, almost certainly some sort of feature aggregation or embedding, such as [Bag of Words](#), needs to be used.

A second strategy, illustrated in [Figure 5.11](#), is to apply features sequentially, such that only weakly discriminating, but very fast, features (such as overall average image colour) are applied first, rejecting only those images which are obviously a poor fit, but which subsequently leaves only a modest fraction of the original dataset to be analyzed by more sophisticated features (such as face or corner detectors).

Lab 5: Extracting Features and Plotting Classes

Since we have already seen the Iris dataset in [Lab 2](#), let us apply the methods of this chapter to that dataset. In particular, you may recall that the Iris dataset had four measurements, meaning that each measured data point lies in a four-dimensional space, which is very difficult to visualize. We would therefore be very motivated to identify a lower-dimensional feature space, specifically the Fisher features of [Section 5.1](#).

We begin by setting up the problem, computing covariances, and finding the eigendecomposition of $(\Sigma_W^{-1} \Sigma_T)$ from (5.24):

```
% define class colours and lighter alternate colours
cols = [[1 0 0]' [0 1 0]' [0 0 1]'];
lcols = 0.8 + 0.2*cols;

% load Iris dataset
load Iris

% Find statistics for each class and overall dataset
% Matlab 'cov' function expects one data point per row
St = cov(IrisData (:,1:4));
Sw = zeros(4);
for c=1:3,
    % The fifth row of IrisData contains the class labels
    q = find(IrisData(:,5)==c);
    Sw = Sw + cov(IrisData(q,1:4));
end
Sw = Sw/3;

% Find best Fisher features
[V,D] = eig(inv(Sw)*St);
```

From the eigendecomposition we obtain the following result:

$V =$

-0.2087	0.0065	0.7819	0.3515
-0.3862	0.5866	-0.0995	-0.4246
0.5540	-0.2526	-0.0464	-0.4801
0.7074	0.7695	-0.6136	0.6824

$D =$

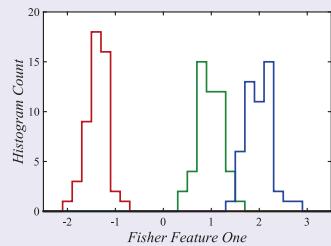
32.7464	0	0	0
0	1.2681	0	0
0	0	0.9866	0
0	0	0	0.9866

Each column of V is one eigenvector, and D is a diagonal matrix with the eigenvalues along the diagonal. Recall that each eigenvalue represents the value of the Fisher discriminant $J(f)$ from (5.16), so it is clear that the first feature discriminates far more significantly, about ten times, than the next three features combined. We can transform the original measurements into these features by multiplying $V^T y$:

```
% Transform Iris data onto features
% IrisData needs to be transposed to move data points into columns
x = [(V*(IrisData (:,1:4)))' IrisData (:,5)];
```

Having computed the points in the feature space, we can then develop a histogram of the points in that first feature:

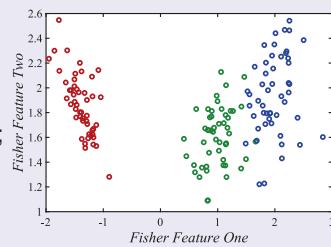
```
clf
for c=1:3,
    q = find(x(:,5) ==c);
    histogram(x(q,1),[-2.5:0.2:3.5], ,
        'DisplayStyle', 'stairs');
    hold on
end
xlabel('Fisher Feature One')
ylabel('Histogram Count');
```



From the plots in [Lab 2](#) it was our expectation that the three classes would not fully separate, so it is not a surprise that two of the histograms still overlap. What we have lost, of course, is any intuition around the units of the feature, since it is a weighted combination of the four measurements with which we began.

We can easily visualize the feature points in the most significant two dimensions:

```
% loop over three classes and plot
clf
for c=1:3,
    q = find(x(:,5)==c);
    plot(x(q,1),x(q,2), 'o', 'color', cols(c,:));
    hold on
end
xlabel('Fisher Feature One')
ylabel('Fisher Feature Two');
```



The three classes are now very nearly separated, and having reduced the problem from four measurements to two features, this is easy to visualize. These features would lead to a comparatively simple classification problem.

Finally, since [Section 4.2](#) spoke at some length regarding the ellipse representation of classes, we would like to understand quite concretely how to find such an ellipse from data, and to then plot it:

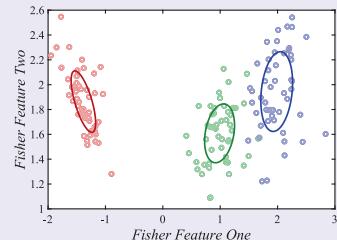
```
% plot an ellipse for each class
for c=1:3,
    % plot data points in light colour
    q = find(x(:,5) ==c);
    plot(x(q,1),x(q,2), 'o', 'color', lcols(c,:));

    % extract mean and covariance
    m = mean(x(q,1:2));
    S = cov(x(q,1:2));

    ellipse_plot( m, S, '—', cols(c,:));
end
xlabel('Fisher Feature One')
ylabel('Fisher Feature Two')

% create a stand-alone function since plotting class ellipses will be
useful
function ellipse_plot( class_mean, class_cov, linestyle , color )
    % find eigendecomposition of covariance
    [v,d] = eig(class_cov);

    % plot ellipse based on eigendecomposition
    th = 0:0.01:(2*pi);
    pts = class_mean(:)+sqrt(d(1,1))*v(:,1)*cos(th)+sqrt(d(2,2))*v(:,2)
    *sin(th);
    plot(pts (1,:) ,pts (2,:) , linestyle , 'color' , color);
end
```



So we find the mean $\underline{\mu}$ and covariance Σ of each class, find the eigen-decomposition V, Λ of the covariance, and then plot the ellipse as

$$\underline{\mu} + \sqrt{\lambda_1} \underline{v}_1 \cos(\theta) + \sqrt{\lambda_2} \underline{v}_2 \sin(\theta) \quad \text{over } 0 \leq \theta \leq 2\pi$$

Keep in mind that it does not matter which eigenvector is chosen as \underline{v}_1 and which for \underline{v}_2 ; there is no particular significance that \underline{v}_1 was multiplied by $\cos(\theta)$ and \underline{v}_2 by $\sin(\theta)$. The only essential step is that eigenvalues always need to be associated with their respective eigenvectors.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links — Fundamentals: [Feature Extraction](#), [Feature Selection](#), [Feature Selection Wrapper Method](#), [Dimensionality Reduction](#) [Linear Discriminant Analysis](#), [Fisher's Linear Discriminant](#)

Wikipedia Links — Specialized Methods: [Bag of Words](#), [Bag-of-words model in Computer Vision](#), [Information Theory](#), [Entropy](#)

As in other chapters, there are a number of excellent texts on pattern recognition in order to pursue the material of this chapter more deeply [2, 4, 10, 12].

Deep neural networks, mentioned on [page 100](#), are at this point nearly universally used for feature extraction, particularly for problems having many inputs (such as analyzing images or video). Such networks are further developed in [Section 11.4](#), and there are a great many books on the subject for the reader to follow up upon [3, 7, 8].

In terms of *random* features, interested readers may wish to look at [5] or one of [Random Projection](#) or [Compressed Sensing](#).

One of the goals of an *effective* feature is to maximally capture the relevant information present in the measurements. Although slightly beyond the scope of this text, although a topic which we will return to again in [Section 11.3](#), it is possible to define the concept of *information* much more formally, using a branch of mathematics known as *information theory*, which is central to the related engineering domain of *communications theory*. Essentially every measurement $y \in \mathcal{Y}$, whether a signal on a fibre-optic cable or taken from a sensor, contains information, where the *amount* of information I is a function of the degree to which we could have predicted the measured value ahead of time:

$$I(y) = -\log_2(\mathbf{P}(y)) \quad \mathbf{P}(y) = \text{the probability that we had expected value } y \quad (5.49)$$

Thus, the more surprising (unexpected) the measurement, the more information it conveys. A given measurement could take on a wide variety of values, of course, so its *average* information, $H(y)$, also known as the *entropy*, is more relevant:

$$H(y) = \mathbb{E}[I(y)] \rightarrow H(y) = \underbrace{\sum_{y \in \mathcal{Y}} -\mathbf{P}(y) \log_2(\mathbf{P}(y))}_{\text{If } y \text{ is discrete valued}}$$

$$H(y) = \underbrace{\int_{y \in \mathcal{Y}} -p(y) \log_2(p(y)) dy}_{\text{If } y \text{ is continuous valued}} \quad (5.50)$$

Strictly speaking, it is not the information conveyed by y that is important to us, rather the *relevant* information, the information about the unknown class c . This is known as the *mutual* information, the average information over the normalized joint distribution:

$$I(c, y) = \sum_{c \in \mathcal{C}} \sum_{y \in \mathcal{Y}} \mathbf{P}(c, y) \log_2 \frac{\mathbf{P}(c, y)}{\mathbf{P}(c)\mathbf{P}(y)} \quad (5.51)$$

for measurement y assumed to be discrete-valued. The situation becomes significantly more complex for continuous-valued and/or multiple measurements. Interested readers may wish to look ahead to [Section 11.3](#), or to any of a great many texts, including [1, 9], or one of [Information Theory](#), [Entropy](#), or [Mutual Information](#).

Although pattern recognition can be applied to *any* context in which we wish to make inferences from measured data, a large number of significant problems come from the visual domains of image processing/video processing/computer vision. This preponderance and importance of visual problems most likely reflects the way in which the human visual system similarly dominates the human senses, for most people, and also the ubiquity of inexpensive cameras. However what makes the study of vision-based pattern recognition problems fascinating is, in part, the following juxtaposition:

- Visual classification is almost trivially easy for the human brain, such that young children can reliably classify a great number of animals, people, vehicles etc.
- Visual classification is frustratingly difficult for computational algorithms, such that even after years of study and thousands to millions of learning attempts, the resulting algorithms are frequently unreliable, or at least uncertainly reliable.

The field is far too large to even summarize here, however there are endless texts on the subject, of which [6, 11] would be a good start, or one of [Image Processing](#) or [Computer Vision](#).

Finally, as we have seen throughout this chapter, whether we are learning a deep neural network, as in (5.44), or a classic feature extractor optimizing an information theoretic criterion, as in (5.51), much of the computational work in pattern recognition becomes a matter of *optimization*. An overview of optimization, with further links and references, is provided in [Appendix C](#).

Sample Problems

Problem 5.1: Short Answer

Give a short definition of each of the following:

- Feature extraction
- Feature selection
- Manifold
- Fisher's discriminant
- Hypothesis testing

Problem 5.2: Short Answer

Offer brief answers to each of the following:

- (a) Why is there a need for "feature extraction" at all? What are the possible benefits which follow from doing feature extraction?
- (b) What makes Feature Selection a harder problem to solve than Feature Extraction?
- (c) In what sorts of circumstances might you prefer to use feature extraction over feature selection or vice-versa?
- (d) Both feature selection and extraction reduce the number of features.
 - (i) When is having fewer features a good thing?
 - (ii) Under what circumstances might you want more (i.e., additional) features?
- (e) Fisher's method is a relatively successful approach for feature extraction. Explain the intuition/rationale of Fisher's criterion. What is the criterion trying to do?

Problem 5.3: Conceptual — Fisher's Criterion

Sketch a counterexample to Fisher's criterion: Find two classes which are linearly separable, but which Fisher's linear discriminant fails to separate.

Problem 5.4: Conceptual — Manifolds

Scenario III from [Figure 5.2](#) does not really have a linear projection onto a 2D space that keeps all classes separated, however there are definitely *non-linear* ways of mapping the problem into 2D.

Supposing that the eight classes shown in Scenario III are centered on the vertices of a cube, where the cube is centered on the origin and where each side has a length of two. Propose a nonlinear 2D feature extraction that *does* keep the classes separated.

Problem 5.5: Conceptual — Feature Extraction

There are several simple strategies for keeping a good single feature for the two-class case:

1. Keep the direction connecting the two class means.
2. Keep the minimum intra-class direction.
3. Keep the maximum inter-class direction.

For each of the above three strategies, undertake the following:

- (a) State the underlying principle behind each method: what is the approach trying to do? Write down the mathematical definition for the feature direction.
- (b) Sketch an example of where each method works well; that is, sketch two class shapes and the resulting feature direction.
- (c) For each method, sketch an example where the method fails, but where the two classes are, in fact, linearly separable.

Problem 5.6: Conceptual/Analytical — Feature Extraction

Suppose we are given two Gaussian classes in 2D space \mathcal{Y} as shown in Figure 5.12.

Sketch the following feature directions:

- (a) \underline{w}_1 = the direction minimizing the size of C_1
- (b) \underline{w}_2 = the direction minimizing the size of C_2
- (c) \underline{w}_3 = the direction between the class means
- (d) \underline{w}_4 = the best single feature from feature selection

Each of these four feature directions \underline{w}_j defines a new feature

$$x_j = \underline{w}_j^T \underline{y}$$

Sketch four plots, showing the one-dimensional class PDFs

$$p(x_j|C_1), p(x_j|C_2) \quad \text{for each of } j = 1, 2, 3, 4. \quad (5.52)$$

Problem 5.7: Analytical — Feature Extraction

Suppose we have two Gaussian classes in two dimensions y_1, y_2 :

$$\mu_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \quad \Sigma_1 = \begin{bmatrix} 1 & b \\ b & \frac{1}{4} \end{bmatrix} \quad \mu_2 = \begin{bmatrix} +1 \\ 0 \end{bmatrix} \quad \Sigma_2 = \begin{bmatrix} 1 & b \\ b & \frac{1}{4} \end{bmatrix} \quad (5.53)$$

Furthermore, in order for the covariances to be positive definite it must be true that

$$-0.5 \leq b \leq 0.5. \quad (5.54)$$

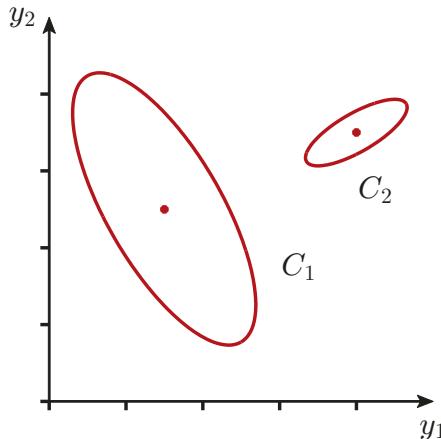


Fig. 5.12. The two classes for Problem 5.6.

- (a) Sketch the unit standard-deviation contours for the two classes for the case that $b = -0.2$.
- (b) Find Fisher's discriminant \underline{w} as a function of b .
- (c) Given Fisher's discriminant \underline{w} from part (b), let a new feature be

$$x = \underline{w}^T \underline{y}$$

So now we have two classes in feature x . Find the means and variances for the two classes on x as a function of b .

- (d) Sketch the transformed problem on x for $b = +0.2$.

Problem 5.8: Analytical — Feature Selection and Extraction

We want to look at feature selection and extraction. Suppose we are given two equally-likely Gaussian classes A, B in 3D measurement space \mathcal{Y} :

$$\mu_A = \begin{bmatrix} 4 \\ 2 \\ 0 \end{bmatrix} \quad \mu_B = \begin{bmatrix} 0 \\ 3 \\ 0 \end{bmatrix} \quad \Sigma_A = \Sigma_B = \begin{bmatrix} 20 & 8 & 2 \\ 8 & 14 & 8 \\ 2 & 8 & 20 \end{bmatrix} \quad (5.55)$$

Inverting 3×3 matrices by hand is somewhat tedious, so here are the numerical results for the matrix eigendecomposition and inverse of Σ :

$$\lambda_1 = 6, \quad \underline{v}_1 = \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix} \quad \lambda_2 = 18, \quad \underline{v}_2 = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \quad \lambda_3 = 30, \quad \underline{v}_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\Sigma^{-1} = \begin{bmatrix} 0.05 & -0.05 & -0.01 \\ -0.05 & 0.30 & -0.04 \\ -0.01 & -0.04 & 0.05 \end{bmatrix}$$

- (a) We want to do feature *selection*, meaning that we either keep a measurement as a feature, or we don't keep it at all. Find the *two* features optimizing Fisher's criterion.

Compute the class means and covariances in the resulting feature space.

- (b) We want to do feature *selection*. This time, find the *one* feature optimizing Fisher's criterion.

Compute the class means and variances in the resulting feature space.

- (c) We want to do feature *extraction*. Let's keep the *two* feature directions in which the classes are most compact.

Identify the feature, compute the class means and covariances in the resulting feature space, and sketch the unit-standard deviation class contours for both classes in the feature space.

- (d) We want to do feature *extraction*, but this time we want to keep Fisher's direction.

Identify Fisher's feature and compute the class means in the resulting feature space.

Problem 5.9: Numeric/Computational

We would like to quantitatively assess how information is compressed in dimensionality reduction, along the lines of the left plot in [Figure 5.8](#). Specifically, suppose we have a vector y of 20 measurements, thus

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{20} \end{bmatrix} \quad \Sigma = \text{cov}(y) \quad \Sigma_{ij} = e^{-\beta|i-j|} \quad \beta > 0 \quad (5.56)$$

We call Σ a Toeplitz matrix — all of the elements are constant along each diagonal in Σ .

Parameter β controls how correlated the measurements are. For $\beta \rightarrow \infty$ the measurements become uncorrelated, and for $\beta \rightarrow 0$ the measurements become perfectly correlated. Note, however, that the variance of each measurement is one, regardless of β .

We will be finding the principal components of y by taking the eigendecomposition of Σ . Undertake the following steps:

- Set $\beta = 0.1$
- Write a simple piece of code to create 20×20 matrix Σ

- Take the eigendecomposition and sort the eigenvalues $\{\lambda_j\}$ from largest to smallest
- The variance captured by the first q features (principal components) is $v_j = \sum_{j=1}^q \lambda_j$
- Observe the value of v_{20}
- Plot v_j as a function of j

Repeat the above steps for $\beta = 0.01$ (more correlated) and $\beta = 1$ (less correlated).

What do you observe in terms of the value of v_{20} and the shape of the plot as a function of β ?

Problem 5.10: Real-World, Open-Ended — Features

The identification of which features to extract is a critical step in pattern recognition, and can have a large impact on classifier complexity and accuracy. For each of the following contexts, discuss the sorts of features that might allow for effective classification:

- (a) We would like to design a system to classify a video as “television show” or “commercial.”
- (b) We want to develop a classifier to distinguish between different bird species (robin, sparrow, hawk etc.). Suggest a few measurements that might permit effective classification.
- (c) A substantial, current socio-environmental problem is urban sprawl. One of the biggest problems is even knowing *how much* sprawl has taken place. There has recently been some interest in quantifying sprawl based on satellite measurements. What sorts of features might a satellite be able to measure that would allow you to set up a pattern recognition problem to discriminate between the classes C_{City} and C_{Country} ?

References

1. R. Ash, *Information Theory* (Dover Publications, 1990)
2. C. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2011)
3. C. Bishop, G. Hinton, *Neural Networks For Pattern Recognition* (Oxford University Press, 2005)
4. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley Interscience, 2009)
5. Y. Eldar, G. Kutyniok, *Compressed Sensing: Theory and Applications* (Cambridge University Press, 2012)

6. R. Gonzalez, R. Woods, *Digital Image Processing* (Pearson, 2017)
7. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016)
8. J. Kelleher, *Deep Learning* (MIT Press, 2019)
9. S. Kullback, *Information Theory and Statistics* (Dover Publications, 1997)
10. S. Rogers, M. Girolami, *A First Course in Machine Learning* (Chapman and Hall, 2016)
11. R. Szeliski, *Computer Vision: Algorithms and Applications* (Springer, 2010)
12. A. Webb, K. Copsey, *Statistical Pattern Recognition* (Wiley, 2011)



Distance-Based Classification

In principle, distance-based classification is a very straightforward idea. With the feature space \mathcal{X} having been defined in [Chapter 5](#), given a feature point \underline{x} , presumably the class “closest” to \underline{x} is the most similar, and therefore it is plausible to classify the point \underline{x} as belonging to that closest class, as shown in [Figure 6.1](#). That is, given a definition of distance $d(\underline{x}, C_\kappa)$ measuring the distance (dissimilarity) between feature point \underline{x} and class C_κ , then distance-based classification says

$$\text{Classify } \underline{x} \text{ as } \bar{C} \text{ if } d(\underline{x}, \bar{C}) \leq d(\underline{x}, C_\kappa) \quad \text{for all } 1 \leq \kappa \leq K, \quad (6.1)$$

thus choosing the closest among the K classes.

The use of a distance function to perform classification therefore seems trivially straightforward — there is little more to say than what is shown in [\(6.1\)](#),

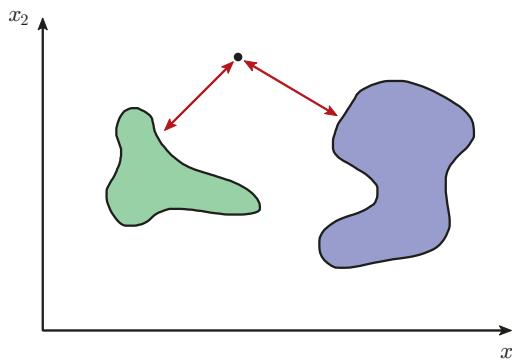


Fig. 6.1. CLASS DISTANCE: If we wish to classify a point (black dot), what does it mean for the point to be *closer* to one class or another? Distance-based classification requires us to offer an explicit definition of such distances.

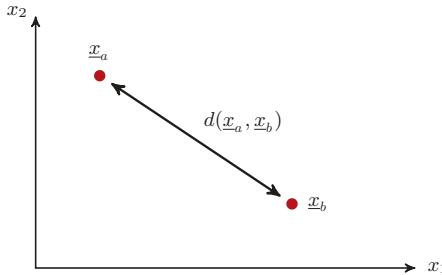


Fig. 6.2. DISTANCE: The most fundamental question is how to define the distance between two points $\underline{x}_a, \underline{x}_b$ in some feature space \mathcal{X} . The feature space is shown here as two-dimensional, but could equally well be a much higher n -dimensional space.

which is just to choose the closest class. The key challenge, however, is the definition of the distance function $d()$ itself. To define a distance function we essentially need two pieces:

1. How do we define the point-to-point distance $d(\underline{x}, \bar{\underline{x}})$ from \underline{x} to $\bar{\underline{x}}$?
2. In measuring the distance to a *class*, where do we measure *to*?

The following two sections focus on precisely these questions.

6.1 Definitions of Distance

We begin with the question of what it means to define a *distance*, either between two measurements y_1, y_2 in measurement space \mathcal{Y} , or between two features x_1, x_2 in feature space \mathcal{X} , as shown in Figure 6.2. The reader will recall that aspects of distances were already explored in the context of *similarity* in Section 4.1. In particular, the three illustrations of distance definitions from Figure 4.1 are repeated and expanded upon in Figure 6.3.

Our starting point is the Euclidean distance, shown in the left column of Figure 6.3, the most basic and intuitive of all distance functions. The Euclidean distance,

$$d_E(\underline{x}, \underline{x}') = \left(\sum_j (x_j - x'_j)^2 \right)^{1/2}, \quad (6.2)$$

is the straight-line distance between two points in a space, the distance you would measure with a ruler or a string in two or three dimensions.

An alternative way of defining or understanding a distance function is in terms of a constant-distance contour: the set of all points some fixed distance from a given point \underline{x}' :

$$\text{Contour of all points a distance } \zeta \text{ from point } \underline{x}' = \{ \underline{x} \mid d(\underline{x}, \underline{x}') = \zeta \} \quad (6.3)$$

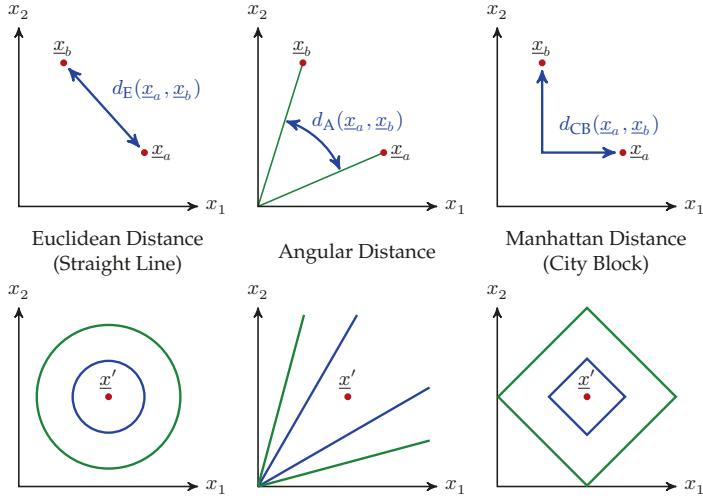


Fig. 6.3. DISTANCES: There are many possible notions of distance, of which three are shown here, repeated from Figure 4.1. The class shape implied by a distance function is typically much more easily visualized (bottom row) by plotting the contours of constant distance, contour $\zeta = 1$ (blue) and $\zeta = 2$ (green), to a given point (red).

In the case of the Euclidean distance d_E , the contour of constant distance is a circle/sphere/hyper-sphere (depending on the number of dimensions), however other definitions of distance lead to other definitions of constant-distance contours, as shown in Figure 6.3.

From the discussion of patterns and class shape in Chapter 4, the reader should understand that although the Euclidean distance may be intuitive, in the relatively abstract world of feature spaces it is not at all obvious that classes will have circular/spherical shapes, and therefore it is similarly not obvious that the Euclidean distance is somehow correct or natural.¹ Indeed, the Euclidean distance is but a special case of the broader family of Minkowski distances,

$$d_\rho(\underline{x}, \underline{x}') = \left(\sum_j |x_j - x'_j|^p \right)^{1/p} \quad 0 \leq \rho \leq \infty, \quad (6.4)$$

controlled by continuous parameter ρ , for which several constant-distance contours are plotted in Figure 6.4. It is useful for the reader to understand, mathematically, how the value of ρ in (6.4) leads to the contours observed in Figure 6.4, however in practice *all* of the Minkowski contours (at least for $\rho \geq 1$) are relatively compact square/circle-like, and so do not offer a meaningful alternative to the Euclidean distance.

¹ Indeed, see the discussion in Example 6.1.

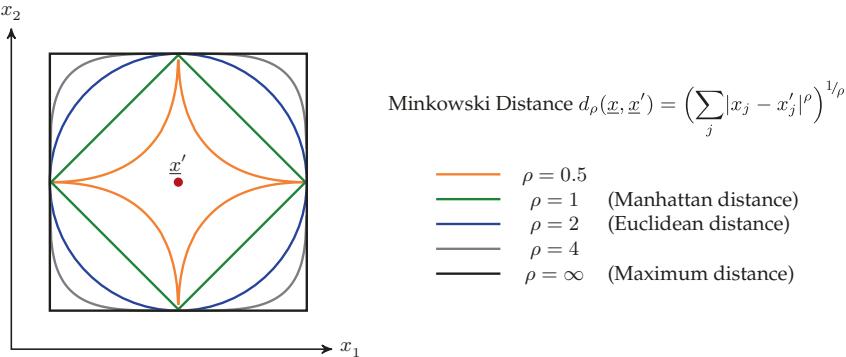


Fig. 6.4. MINKOWSKI DISTANCES: The Manhattan and Euclidean distances of Figure 6.3 are actually special cases of the family of Minkowski distances, parameterized by a single parameter p . Five contours are plotted here, all at a constant distance of 1.0 from point \underline{x}' . Values of $p < 1$ are mathematically valid, however the resulting star-shaped distance contours will be correct for only fairly unusually-shaped classes.

Far more important, and as discussed in Example 6.1, is that measurement dimensions are rarely in directly comparable units (such as a person's height and age — these are very different things), the specific units (millimeters versus centimeters versus meters) introduce scaling, and (as discussed in Section 5.2) the process of extracting features can lead to an even more confusing mix of units.²

At the very least we would want to be able to undo any arbitrary scaling introduced by a choice of units (the factor of ten, for example, in choosing to measure in millimeters rather than in centimeters), leading to the scaled Euclidean distance (Figure 6.5)

$$d_S(\underline{x}, \underline{x}') = \left(\sum_j (\beta_j |x_j - x'_j|)^2 \right)^{1/2} \quad (6.6)$$

which could easily generalize to the scaled Minkowski distance, if desired, by replacing the digit "2" with ρ . Based on the discussion in Chapter 5, we can imagine two plausible approaches to selecting the rescaling parameters $\{\beta_j\}$:

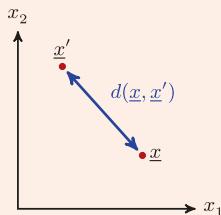
1. Based on the average class shape Σ_W , from (5.15), such that

$$\beta_j = \frac{1}{\sqrt{(\Sigma_W)_{j,j}}}, \quad (6.7)$$

² One exception, as discussed in Section 5.2, is that in the case of feature *selection* (and not extraction), the features are individual measurements, and therefore *do* preserve the units of the measurements. However it is still the case that measurements taken in very different units will almost certainly not be meaningfully represented by a Euclidean or Minkowski distance.

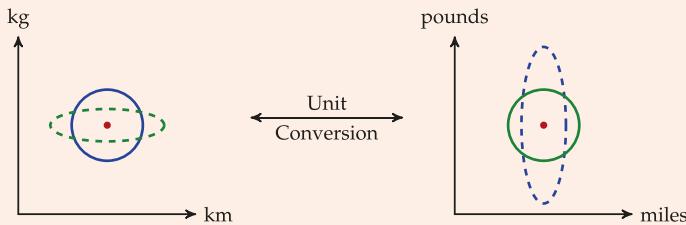
Example 6.1: Euclidean Distance and Measurement Units

When looking at a piece of paper, the Euclidean distance may very well appear to be the most natural definition of distance. After all, a ruler measuring the distance between two points on a page is measuring the straight-line distance which will (by Pythagoras) be equal to the Euclidean distance.



However in a measurement space \mathcal{Y} or feature space \mathcal{X} , there is really no particular reason for the Euclidean distance to be "natural" or "correct".

For example, suppose that in some application we have measurements of length and weight. Are our lengths measured in miles or kilometers, and the weight in pounds or kilograms? You may argue that it doesn't matter, that the units are easily converted from one form to another, which is true; however the Euclidean distance in kilometer-kilogram space implies quite a different distance compared to the Euclidean distance in mile-pound space:



So which distance is "correct"? ... The Euclidean distance in km-kg space (blue), or the Euclidean distance in mile-pound space (green)?

In general, neither will be correct, and we should not be attempting to force the Euclidean distance onto any measurement or feature space. Rather, we should understand that each measurement or feature is associated with a degree of scaling (based on its units), and so any distance function computed over different measurements or features should at least correct for such scaling:

$$\underbrace{d_E(\underline{x}, \underline{x}') = \left(\sum_j (x_j - x'_j)^2 \right)^{1/2}}_{\text{Euclidean Distance}} \quad \underbrace{d_S(\underline{x}, \underline{x}') = \left(\sum_j (\beta_j (x_j - x'_j))^2 \right)^{1/2}}_{\text{Scaled Euclidean Distance}} \quad (6.5)$$

Parameter β_j rescales feature x_j to make its spread similar to those of other rescaled features, such that the Euclidean distance can be applied in the rescaled space.

We would still need to learn suitable or optimal rescaling values $\{\beta_j\}$ from the data; in practice, we normally address this issue by using the fairly powerful Mahalanobis distance of (6.13), which includes rescaling within its degrees of freedom.

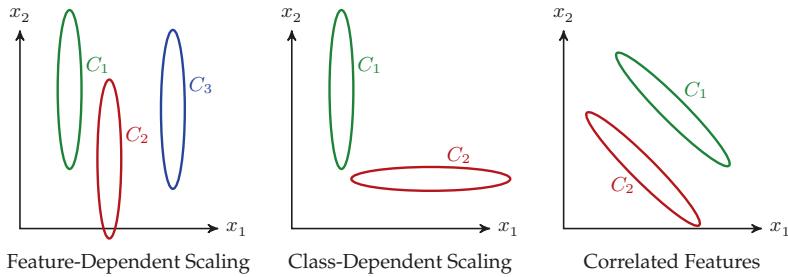


Fig. 6.5. The SCALED EUCLIDEAN DISTANCE of (6.6) allows each feature axis to be rescaled for a more meaningful notion of distance, which would be effective in cases (left) where different features have very different degrees of class elongation. However the scaled distance is *not* effective if the stretching is class-dependent (centre), or if the features are correlated (right).

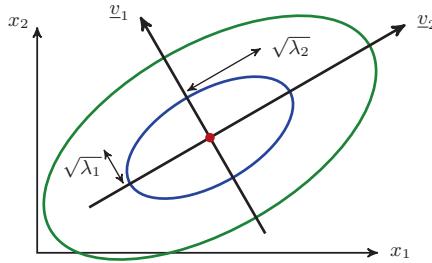


Fig. 6.6. The MAHALANOBIS DISTANCE is based on fitting a hyper-ellipse to a class, such that the hyper-ellipse represents a distance of one standard deviation from the class mean (red dot). Two contours of constant distance from the mean are shown, with contours \$\zeta = 1\$ (blue) and \$\zeta = 2\$ (green). The Mahalanobis distance builds on the class parameterization of Figure 4.6.

undoing the class extent (the standard deviation, found as the square-root of the j th diagonal element) along the j th dimension.

2. Based on the total variation Σ_T , from (5.8), such that

$$\beta_j = \frac{1}{\sqrt{(\Sigma_T)_{j,j}}}, \quad (6.8)$$

undoing the overall extent of the dataset along the j th dimension.

In practice, the scaled Euclidean distance of (6.6) and associated estimation of the scaling parameters $\{\beta_j\}$ is of limited interest to us, because this scaling is handled automatically by a far more general definition of distance: the Mahalanobis distance.

From the relatively detailed discussion in [Section 4.2](#), we began with a Gaussian distribution for a class, based on its mean $\underline{\mu}$ and covariance Σ :

$$p(\underline{x}) = \frac{1}{\det(\Sigma)(2\pi)^{n/2}} \exp\left(-\frac{1}{2}(\underline{x} - \underline{\mu})^T \Sigma^{-1} (\underline{x} - \underline{\mu})\right) \quad (6.9)$$

The contour or surface of constant probability density,

$$p(\underline{x}) = \text{Constant} \quad (6.10)$$

can be found as

$$(\underline{x} - \underline{\mu})^T \Sigma^{-1} (\underline{x} - \underline{\mu}) = \text{Constant} \quad (6.11)$$

Since the choice of constant is somewhat arbitrary, we chose the *unit* standard-deviation contour in [\(6.11\)](#) as

$$(\underline{x} - \underline{\mu})^T \Sigma^{-1} (\underline{x} - \underline{\mu}) = 1 \quad (6.12)$$

The Mahalanobis distance of [Figure 6.6](#) is then defined to measure distances in multiples of this unit standard-deviation contour; that is, how many standard-deviations is it from a feature point \underline{x} to the mean $\underline{\mu}$ of a given class:

$$d_M(\underline{x}, \underline{\mu}) = \left\{ (\underline{x} - \underline{\mu})^T \Sigma^{-1} (\underline{x} - \underline{\mu}) \right\}^{1/2} \quad (6.13)$$

A few comments with regards to the Mahalanobis distance:

- It is explicitly *parametric*,³ since the distance is formulated as a function of the class mean $\underline{\mu}$ and covariance Σ .
- The Mahalanobis distance is able to handle all of the circumstances of [Figure 6.5](#), since the class covariances Σ_κ will reveal feature scaling, class-dependent scaling, and correlated features.
- The Mahalanobis distance is class-specific, meaning that the definition of distance from point \underline{x} to class C_1 will, in general, be *different* from the definition of distance from \underline{x} to class C_2 .
- The Mahalanobis distance has explicitly built into its definition that it measures to the mean $\underline{\mu}$ of each class. That is, it has already answered the question which we intend to cover in [Section 6.2](#), which is *where* in a given class to measure *to*.

With several ideas for point-to-point distance functions $d(\underline{x}, \underline{x}')$ now defined, we turn our attention to the second issue identified at the beginning of this chapter: In measuring the distance to a *class*, where do we measure *to*?

³ Refer back to the discussion at the beginning of [Section 4.2](#) or ahead to [Chapter 7](#) in terms of parametric versus non-parametric methods.

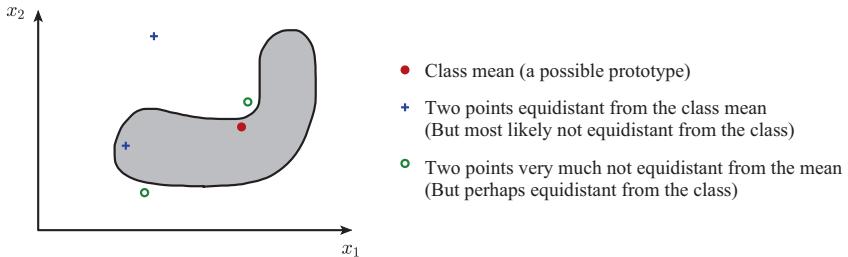


Fig. 6.7. PROTOTYPE: Suppose we are given an irregularly shaped class (grey). The mean (red) of a class is equidistant from two points (blue), however surely we would not consider those points similarly close to the class? The two green points appear to have approximately the same distance to the class, but have a very different relationship to the mean. For irregularly-shaped classes, a single fixed prototype is unlikely to meaningfully represent the class.

6.2 Class Prototype

In order to undertake distance-based classification, as in (6.1), it is really the point-to-class distance $d(\underline{x}, C_\kappa)$ which is needed. A point-to-point distance function $d(\underline{x}, \underline{x}')$ alone does not tell us how close a given feature point \underline{x} is to a class C , since we need to decide *where* in C to measure to. We define the class *prototype* \underline{z}_κ to be a point representative of class C_κ , such that the distance to a class is defined as

$$d(\underline{x}, C_\kappa) \equiv d(\underline{x}, \underline{z}_\kappa) \quad (6.14)$$

There are a few options in terms of how to consider choosing a prototype:

- The prototype \underline{z}_κ is a fixed location;
- The prototype $\underline{z}_\kappa(\underline{x})$ can be a function of where we are measuring *from*.

The latter case, where the prototype might need to vary, may seem confusing at first glance, however the situation is easily explained graphically, as in Figure 6.7: for an irregularly-shaped class, a single prototype will not meaningfully represent the overall class shape and what it means to be close to or far from the class. Examples of fixed and variable prototypes are shown in Figure 6.8.

Further prototype options build on the question of representation from Section 4.2:

- The class shape and its associated prototype is represented parametrically;
- The representation is *non-parametric*.

That is, do we have a parametric model for the class shape, or are the class shape and prototype represented implicitly via the data points $\{\underline{x}_{\kappa,i}\}$ associated with class C_κ ?

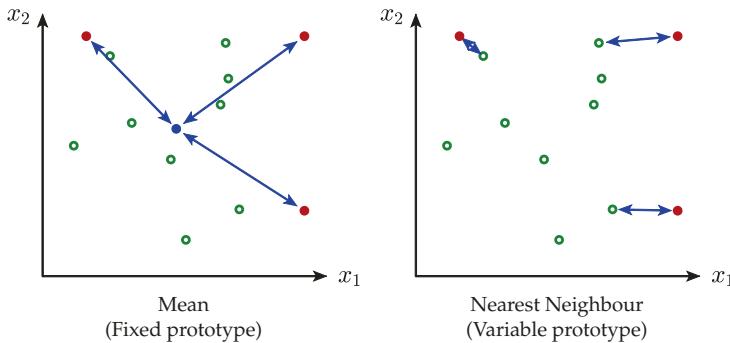


Fig. 6.8. PROTOTYPE DEPENDENCE: The point to measure *to* (the prototype) may (right) or may not (left) be a function of where we are measuring *from* (red dots).

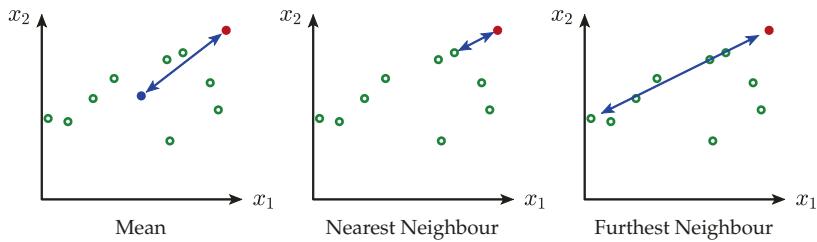


Fig. 6.9. PROTOTYPE: Given a cluster of points, where do we measure *to*? That is, given a point of interest (red) and a cluster of points (green) defining a class, how do we define the point–cluster distance? Three common possibilities, sketched here, are the mean of the points, the nearest point, and the furthest point.

Although the preceding discussion might give the impression of a great many possibilities in defining prototypes, in actual fact almost universally only the four approaches sketched in Figures 6.9 and 6.10 are used:

1. A fixed, parametric prototype located at the class mean:

$$\underline{z}_\kappa = \mu_\kappa = \frac{1}{N_\kappa} \sum_{i=1}^{N_\kappa} x_{\kappa,i} \quad (6.15)$$

The rationale for choosing the mean is that it is, as proved in Appendix D, the most representative point for a class in a squared-error sense. That is, the mean uniquely minimizes the average squared-Euclidean or squared-Mahalanobis distance to all points in a class. For classes which are not ellipsoidal in shape, the Mahalanobis contour may be a poor parametric representation of the class, in which case the mean may, in fact, be a rather poor prototype, as illustrated in Figure 6.7.

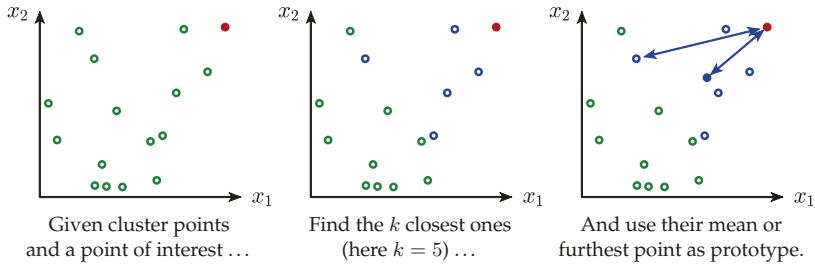


Fig. 6.10. kNN MEAN PROTOTYPE: The notion of prototype can be quite flexible, and a great many variations are possible. Even here, we could take the five closest points, in total, or the five closest points per class. And then, given five points, we could take the distance to the mean or to the furthest point.

2. A moving non-parametric prototype, the *nearest neighbour* (NN), located as that point \underline{x}' in class C_κ which is closest to the feature point \underline{x} from which we are measuring:

$$\underline{z}_\kappa(\underline{x}) = \underline{x}' \text{ such that } \underline{x}' \in \{\underline{x}_{\kappa,i}\}, d(\underline{x}, \underline{x}') \leq d(\underline{x}, \underline{x}_{\kappa,i}) \text{ for all } i \quad (6.16)$$

It is important to observe that the nearest neighbour \underline{x}' may very well depend on the choice of distance function from [Section 6.1](#). That is, the nearest neighbour under one definition of distance may not be the nearest neighbour for some other definition, as explored in [Problem 6.5](#).

3. Generalizing from the previous case, the non-parametric *k-nearest neighbour* is based on the k closest data points to feature point \underline{x} , as illustrated in [Figure 6.10](#). The actual prototype may be the mean of these k points (k NN-mean) or just the k th closest point (k NN).
4. Finally one other non-parametric prototype, the *furthest neighbour*, defined as that point \underline{x}' in class C_κ which is *furthest* from the feature point \underline{x} from which we are measuring:

$$\underline{z}_\kappa(\underline{x}) = \underline{x}' \text{ such that } \underline{x}' \in \{\underline{x}_{\kappa,i}\}, d(\underline{x}, \underline{x}') \geq d(\underline{x}, \underline{x}_{\kappa,i}) \text{ for all } i \quad (6.17)$$

This definition of prototype can seem counter-intuitive at first — are we not wanting to *minimize* distances? Absolutely: the distance-based classifier of [\(6.1\)](#) *always* minimizes distance. What we are considering here is *where* to measure to (that is, how $d(\underline{x}, C)$ in [\(6.1\)](#) is defined), and the furthest-neighbour prototype is essentially a fairly conservative notion of class closeness: a point \underline{x} is close to class C based on how close it is to the furthest point of C . So among a set of classes, point \underline{x} will be classified as that class whose *furthest* point is *closest*.

The choice of prototype significantly influences the associated computational complexity. In general, parametric methods are fast, since the parameterization (such as the mean) needs to be computed only once. On the other hand,

nonparametric methods are computationally demanding, with the computational complexity growing with increasing k .

Given an n -dimensional feature space with N points characterizing class C , suppose there are M points in feature space whose distance to class C we would like to evaluate:

MEAN PROTOTYPE : The mean needs to be computed over N data points, and then each of M points needs to be compared to the mean:

$$\text{Total complexity } \mathcal{O}(\underbrace{nN}_{\text{Compute mean}} + \underbrace{nM}_{\text{Distance from } M \text{ points to mean}}) \quad (6.18)$$

NN PROTOTYPE : No work needs to be done in advance. For *each* of the M points, the distance to *each*⁴ of the N points needs to be found in order to identify the closest:

$$\text{Total complexity } \mathcal{O}(\underbrace{0}_{\text{No work in advance}} + \underbrace{nNM}_{\text{Distances between } M \text{ and } N \text{ points}}) \quad (6.19)$$

k NN PROTOTYPE : No work needs to be done in advance. For *each* of the M points, the distance to *each* of the N points needs to be found, and the k smallest distances then identified:

$$\text{Total complexity } \mathcal{O}(\underbrace{nNM}_{\text{Distances between } M \text{ and } N \text{ points}} + \underbrace{knM}_{\text{Find } k \text{ smallest distances for each of } M}) \quad (6.20)$$

Combining a definition of prototype with a definition of distance from [Section 6.1](#) leads to a definition of the point-to-class distance $d(\underline{x}, C)$, which therefore allows us to talk about the constant-distance contour

$$d(\underline{x}, C) = \text{Constant}. \quad (6.21)$$

That contour or surface or hyper-surface in feature space \mathcal{X} is a constant distance from class C , for which three examples are illustrated in [Figure 6.11](#).

The choice of prototype and distance function can significantly influence the assumptions regarding the class shape. That is, the constant-distance contours of (6.21) should meaningfully reflect the shape of each underlying class.

⁴ It is possible to find the closest point without testing the distance to all N points by pruning unneeded points, as will be described in [Section 6.4](#), however the pruning process itself then introduces additional computational work.

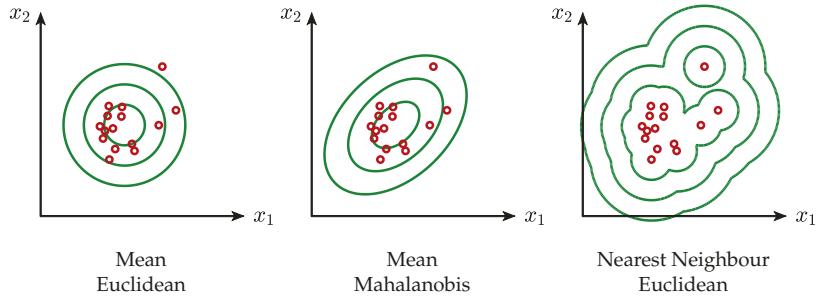


Fig. 6.11. CLUSTER DISTANCE CONTOURS: Combining the notion of *distance* from Figure 6.3 and that of *prototype* from Figure 6.9, we can now talk about the distance to a *cluster* of points. Three common examples of distance contours are given, each of which assumes a distance function (Euclidean, Mahalanobis) and a prototype (mean, nearest neighbour). In each case, three constant-distance contours are shown, corresponding to cluster distances of 1.0, 2.0, and 3.0.

Building on the discussion of learning in Chapter 3, there are two fundamental ways in which a prototype might be badly chosen:

1. The definition of prototype is based on a poor choice of class model;
2. The definition of prototype overfits noise (outlying data) describing the class.

In terms of modelling, parametric methods are subject to mis-fit if the assumption underlying the parameterization (such as a spherical, compact class) fails to match the actual class shape, whereas nonparametric methods based on local class structure ($k \ll N$) will be very flexible in terms of class shape, in that essentially nothing is assumed about the class, other than that a data point from a class is expected to be close to one or more other point from the same class.

The latter question, our sensitivity to outlying data, is illustrated and summarized in Figure 6.12 and Table 6.1:

- Prototypes computed on the basis of many data points (such as the mean) will be influenced by outliers, but only slightly, whereas prototypes computed on the basis of a single data point (such as NN) are maximally sensitive. The k NN outlier sensitivity will decrease as k increases.
- Parametric prototypes, such as a mean or covariance, will fit exceptionally well when the underlying class shape matches the parametric model assumptions, but will match very poorly if the class is irregular or highly inconsistent with the model.

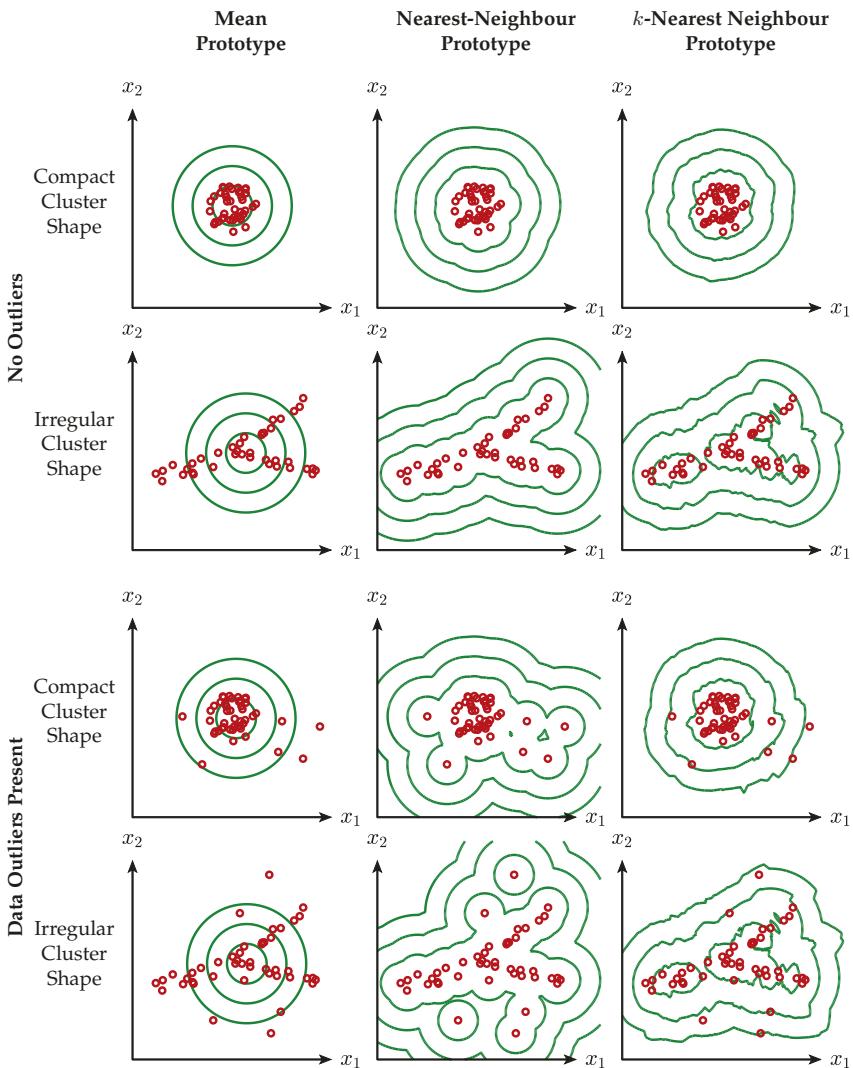


Fig. 6.12. ROBUSTNESS: The choice of prototype implies some sense of what it means to be close or far from a cluster. For a compact cluster without outliers (top) all prototypes produce similar distance contours. However irregularly shaped clusters or outlying points (remaining three rows) lead to significant variations between prototypes. In general, the mean prototype is inflexible to cluster shape and NN is highly sensitive to outliers, whereas kNN represents a flexible and robust compromise. The kNN results are shown for $k = 5$.

Prototype	Assumed class shape	Outlier robustness	Computational speed
Mean	Only hyper-spheres	Very good	Very fast
Mahalanobis	All hyper-ellipses	Good	Fast
NN	Flexible (any shape)	Poor	Slow
k NN or k NN-Mean	Flexible (any shape; details lost as k increases)	Improves as k increases	Increasingly slow as k increases

Table 6.1. The three basic criteria in choosing a definition of prototype would be the assumptions it makes regarding the class shape, the robustness to outlying data points (Figure 6.12), and the computational complexity.

Algorithm: Prototype-Distance Classification

```

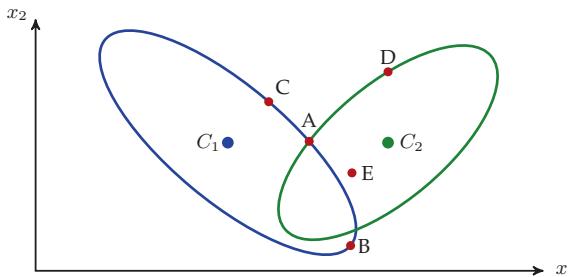
Goals: Classify a feature point  $\underline{x}$  based on  $K$  prototypes  $\{\underline{z}_\kappa\}$  and distance  $d()$ 
Function Class Label  $c = \text{Dist.Class}(\underline{x}, K, \{\underline{z}_\kappa\}, d())$ 
    MinDist =  $\infty$ 
    for  $\kappa = 1 : K$  do
         $\zeta = d(\underline{x}, \underline{z}_\kappa)$  Measure distance to class  $\kappa$ 
        if  $\zeta < \text{MinDist}$  then
            MinDist =  $\zeta$ 
             $c = \kappa$  Remember the class minimizing the distance so far
        end if
    end for

```

Algorithm 6.1. We can classify a feature point by measuring the distance to each prototype and returning the class associated with the smallest distance. This algorithm is essentially just implementing the classification of (6.22) based on (6.14)

- Prototypes based on a *linear* function, such as the mean or k NN-mean (not shown in Figure 6.12) will have some dependence on all (mean) or k (k NN-mean) data points, implying a sensitivity to outliers. To fully remove any outlier sensitivity requires a non-linear prototype, such as a median or k NN.

Of the three prototypes shown in Figure 6.12, only the k NN possesses both desirable aspects of flexibility (to class shape) and robustness (to outliers). Clearly the choice of k is critical, in that too small a value of k leads to outlier sensitivity, whereas too large a value of k leads to an insensitivity to details of the class shape.



Point	Distance to C_1	Distance to C_2	Classify as ...
A	1	1	On Boundary
B	1	> 1	C_1
C	1	> 1	C_1
D	> 1	1	C_2
E	> 1	< 1	C_2

Fig. 6.13. DISTANCE BASED CLASSIFICATION: Given two ellipsoidal classes, we propose to use the Mean prototype/Mahalanobis distance for classification. The unit-distance contours are shown, which allow us to know whether each of the five points A ... E has a distance greater than or less than one, to each of C_1 and C_2 . A computer simulation of this classifier is shown in Figure 6.14.

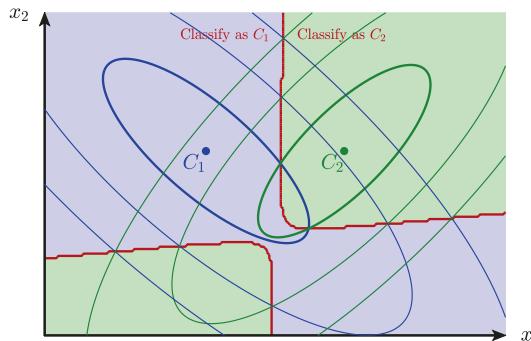


Fig. 6.14. A computer simulation of Figure 6.13: at every point in feature space (x_1, x_2) we can determine which class is the closest, here based on the Mean-Mahalanobis distance, with the red line showing the resulting classification boundary. The reader may find the classification regions unusual, however the *shape* of classes C_1 and C_2 means that the lower-left region is closer (as measured by Mahalanobis) to C_2 , and the lower-right to C_1 .

The significant disadvantage of the k NN approach is, of course, its computational complexity, already quantified in (6.20) and discussed further in Section 6.4.

6.3 Distance-Based Classification

Having established a definition of distance (Section 6.1) and a definition of prototype, where in a class to measure to (Section 6.2), we consequently have the point-to-class distance $d(\underline{x}, C)$ which is needed for distance-based classification, repeated from (6.1):

$$\text{Classify } \underline{x} \text{ as } \bar{C} \text{ if } d(\underline{x}, \bar{C}) \leq d(\underline{x}, C_\kappa) \quad \text{for all } 1 \leq \kappa \leq K. \quad (6.22)$$

Since many definitions of distance, such as Euclidean in (6.2) or Mahalanobis in (6.13), involve a square root, it is often analytically and computationally simpler to compare squared-distances, such that (6.22) can be written as

$$\text{Classify } \underline{x} \text{ as } \bar{C} \text{ if } d^2(\underline{x}, \bar{C}) \leq d^2(\underline{x}, C_\kappa) \quad \text{for all } 1 \leq \kappa \leq K. \quad (6.23)$$

The equivalence between (6.22) and (6.23) holds because squaring is monotonic for non-negative distances.⁵

For the two-class ($K = 2$) case, the notation of (6.23) simplifies as

$$\text{Classify } \underline{x} \text{ as } d^2(\underline{x}, C_1) \begin{array}{c} \xrightarrow{\text{Say } C_2} \\ \gtrless \\ \xleftarrow{\text{Say } C_1} \end{array} d^2(\underline{x}, C_2) \quad (6.24)$$

Thus, for example, the two-class Mahalanobis classifier can be written as

$$(\underline{x} - \mu_1)^T \Sigma_1^{-1} (\underline{x} - \mu_1) \begin{array}{c} \xrightarrow{\text{Say } C_2} \\ \gtrless \\ \xleftarrow{\text{Say } C_1} \end{array} (\underline{x} - \mu_2)^T \Sigma_2^{-1} (\underline{x} - \mu_2) \quad (6.25)$$

The two class case, showing constant-distance contours and five examples of classification, is illustrated in Figure 6.13; the parallel computer-generated classification, for every point in feature space, is shown in Figure 6.14. The shape of the classification regions can be a little unusual, as can be seen in Figure 6.14 and three further examples in Figure 6.15. The key to observe is that the classification boundary *must* occur at points of equal distance to the two classes. That is, the red boundary in Figure 6.14 can be seen to pass through the intersections of same distance contours to C_1 and C_2 , that is, to the intersection of

$$d(\underline{x}, C_1) = \zeta \quad \text{and} \quad d(\underline{x}, C_2) = \zeta \quad (6.26)$$

for some distance $\zeta > 0$.

⁵ See Problem 6.7 to explore how a non-monotonic operator does, in fact, lead to a changed classifier.

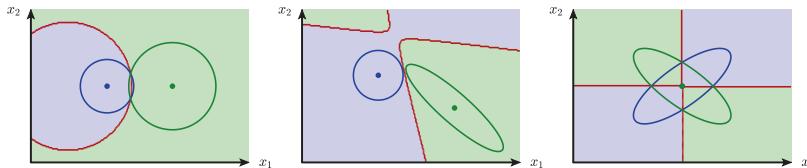


Fig. 6.15. Three additional simulations of Figure 6.14, all based on the Mahalanobis distance and mean prototype. Such boundaries can be sketched relatively easily by hand, as explored in Problem 6.9.

For the two-class case, what we can conclude, for now, is the following:

1. For the *Mean-Euclidean Distance (MED)* classifier, the classification boundary will be the right-bisecting line/plane/hyperplane between the two class means.
2. For the *Mean-Mahalanobis Distance (MMD)* classifier, the classification boundary will be a hyperbola (as in Example 4.3) or n -dimensional hyper-hyperboloid. In general, except in degenerate⁶ cases, the feature space will be divided into three regions, as in Figure 6.14.
3. For the *kNN-Euclidean* classifier, the boundary will normally be fairly irregular and with no particularly predictable shape or form.

If there are more than two classes then the classification boundary will have a significantly more complex behaviour, as can be seen in Figure 6.16. The boundary can be understood as being composed, piece-wise, of two-class boundaries, since at any point \underline{x} in feature space \mathcal{X} it is the two closest classes which will determine the behaviour of the boundary at that point.

Finally, as emphasized in Figure 6.17, nearly *any* classifier will perform well on compact, widely-separated classes. It is classes which are extended, irregularly shaped, closely-spaced or partially overlapping which lead to significant classification challenges, and where differences in classifiers become apparent and significant.

⁶ A circumstance is said to be *degenerate* if an infinitesimal perturbation causes the circumstance to disappear. For example, the case that two classes C_1 and C_2 have *exactly* the same covariance, $\Sigma_1 = \Sigma_2$, is degenerate, or similarly the case that a given class shape is perfectly circular, as in the left panel of Figure 6.15.

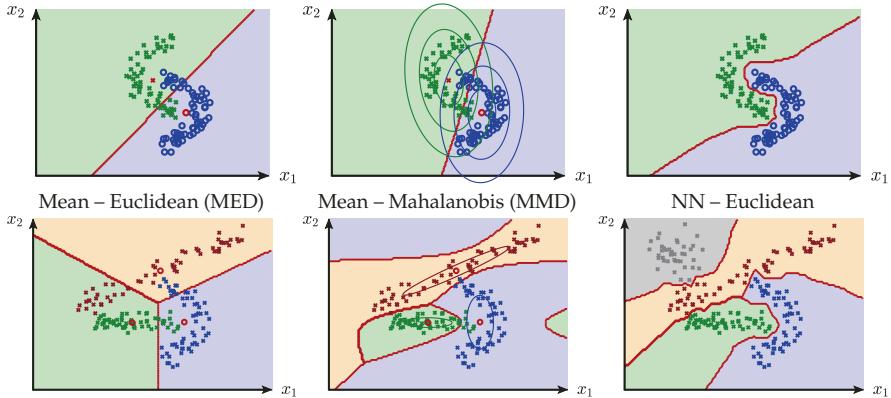


Fig. 6.16. CLUSTER CLASSIFICATION: Building upon the distance contours of Figures 6.14 and 6.15, by computing the distance to multiple classes, at every point in feature space (x_1, x_2) we can determine which class is the closest. This idea is very flexible, applying to any notion of distance and prototype, and to any number of classes.

6.4 Classifier Variations

A great many classifier variations are possible, since any combination of prototype and distance metric leads, in principle, to a legitimate classifier. However, as was discussed in Figure 6.16, much of distance-based classification falls into one of Mean–Euclidean, Mean–Mahalanobis, or NN/ k NN, and other metrics or combinations will be of limited interest to us.

But within the context of Mean–Euclidean, Mean–Mahalanobis, or NN/ k NN, there are two important concepts that we wish to explore in this section:

1. Computational and storage efficiency;
2. Behaviour and variations of k NN.

Computational and storage efficiency:

A distance-based classifier could, in principle, be given a very large number of data points. Normally this sounds like a good idea: we *like* to have a lot of training data. If the prototype is parametric (e.g., the mean), then the resulting classifier is very fast, since the parameter estimation process is undertaken only once, during classifier learning, and when it is time to actually classify then the training data points are no longer used, as was shown in (6.18). In contrast, as we saw in (6.19) and (6.20), for a non-parametric prototype, *every single* classification involves searching through all of the given data points, which can become a very significant computational burden for large data sets.

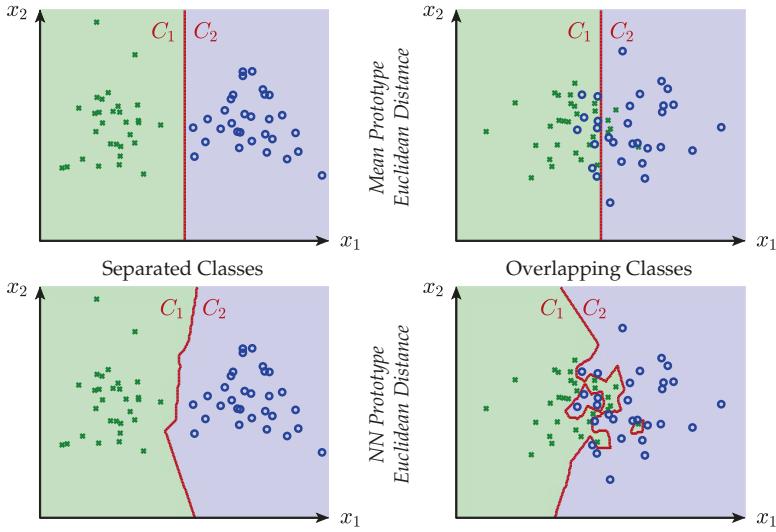


Fig. 6.17. The Mean–Euclidean classifier (top) always has a straight-line boundary, regardless of class size or shape. Furthermore, well-separated classes (left) will typically have approximately similar boundaries, regardless of classifier. However as classes overlap (right) the differences in prototype or distance assumptions lead to differences in the classification boundary which become more striking.

However although the NN *prototype* of (6.16) involves a search over all data points $\underline{x}_{\kappa,i}$ from all classes,

$$\underline{z}_\kappa(\underline{x}) = \underline{x}' \quad \text{such that} \quad \underline{x}' \in \{\underline{x}_{\kappa,i}\}, \quad d(\underline{x}, \underline{x}') \leq d(\underline{x}, \underline{x}_{\kappa,i}) \quad \text{for all } i \quad (6.27)$$

the resulting NN *classifier* can be identical to a NN classifier based on only a subset of the data points. That is, not every data point actually influences the resulting classification boundary. This behaviour may seem mystifying at first, but is easily understood from Figure 6.18: the only points which actually influence the location of the classification boundary are those points which are closest to the classification boundary, at least somewhere along the boundary. That is, if there is a data point in class C_κ which is far from the classification boundary, then that point will *never* be the nearest data point as we move along the boundary, meaning that it can be removed or pruned.

A second illustration with a much larger number of data points is shown in Figure 6.19. The idea of pruning data points is elegant and compelling, particularly given relatively few data points in two dimensions, as in Figures 6.18 and 6.19; the difficulty is that the process of pruning itself becomes computationally very challenging, particularly for large datasets in high-dimensional feature spaces.

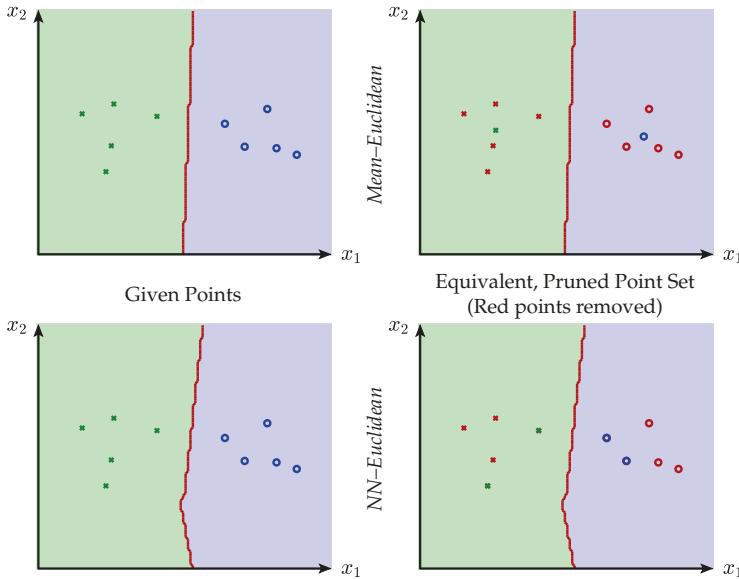


Fig. 6.18. POINT PRUNING: Given classes (left) defined by sample points, we do not necessarily need to store/remember all of the points in order to implement the resulting classifier, such that a pruned set (right) can omit the red-coloured points without any change in classification. In the case of the Mean-Euclidean classifier (top) this is obvious, since each class is completely characterized by a single prototype. However even the nearest-neighbour classifier (bottom) will not need all data points; the only points needed are those which are, at some point, closest to the classification boundary, and the red points can be removed (pruned) without influencing the classifier.

The concept of pruning will reappear in the context of Support Vector Machines in [Section 10.3](#).

Behaviour and variations of k NN:

As was discussed in [Section 6.2](#), we have multiple possible interpretations for how to form a prototype from the k closest points to a feature value \underline{x} :

$$d_{k\text{NN}}(\underline{x}, C_\kappa) = \text{Distance to } k\text{-th-closest point from } \underline{x}, \text{ given data points in } C_\kappa$$

$$d_{k\text{NN-Mean}}(\underline{x}, C_\kappa) = \text{Distance to mean of } k\text{-closest points from } \underline{x}, \text{ given data points in } C_\kappa$$

on the basis of which we choose the closest class, as usual. So, for example, given the former k NN prototype,⁷ we formulate its associated k NN classifier by computing all of the point-to-class distances

⁷ And also a distance metric, of course, from [Section 6.1](#). We are concerned here specifically with the prototype behaviour of k NN; in principle any distance metric could be used.

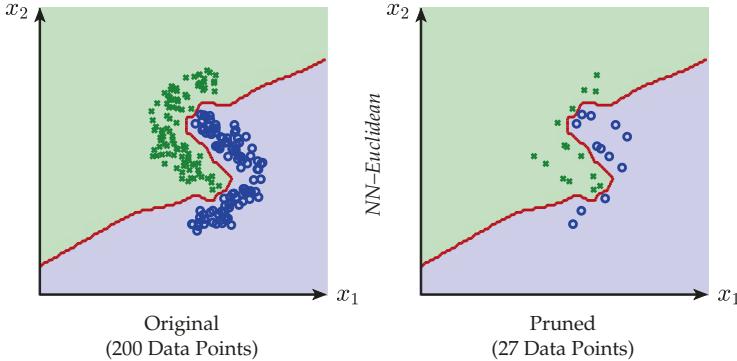


Fig. 6.19. POINT PRUNING: An application of the point-pruning approach to a larger dataset. As we saw in [Figure 6.18](#), the only points which actually influence the classification boundary are those which are, at some point, closest to the boundary, which can lead to an order of magnitude reduction in the points needing to be stored and searched.

$$d_{k\text{NN}}(\underline{x}, C_1) \quad \dots \quad d_{k\text{NN}}(\underline{x}, C_K) \quad (6.28)$$

and then classifying point \underline{x} as that class having the smallest distance to its k th-closest point. This per-class notion of k NN is also consistent with the per-class estimation of distribution, to be discussed in [Section 7.3.3](#).

However there are other possible definitions for k NN classification, and indeed the most widely-used does *not* use a per-class notion, rather is based on the k closest points to \underline{x} , globally, from *all* of the classes:

kNN-Global: Given a feature point \underline{x} , find the k nearest data points, and classify \underline{x} as that class which appears most frequently among the k data points.

In this case it is not possible to talk about a prototype for each class, since the method directly defines the classifier, and does not actually compute a distance to each class. The per-class and global approaches are compared in [Figure 6.20](#). For $k = 1$ (i.e., nearest neighbour), the k NN, k NN-Mean, and k NN-Global all give the same result; for increasing k all three exhibit reduced classification error ([Section 8.4](#) or [Chapter 9](#)), reduced sensitivity to outliers, and increased computational complexity.

If the number of classes $K = 2$ then choosing an odd value for k will guarantee that k NN-Global has a winning class which is unambiguous, however for any value $K > 2$ there is no value of $k > 1$ which ensures a unique solution, and choosing $k = 1$ just gives us the nearest-neighbour classifier, with associated concerns of overfitting and outlier sensitivity. The problem can be seen in

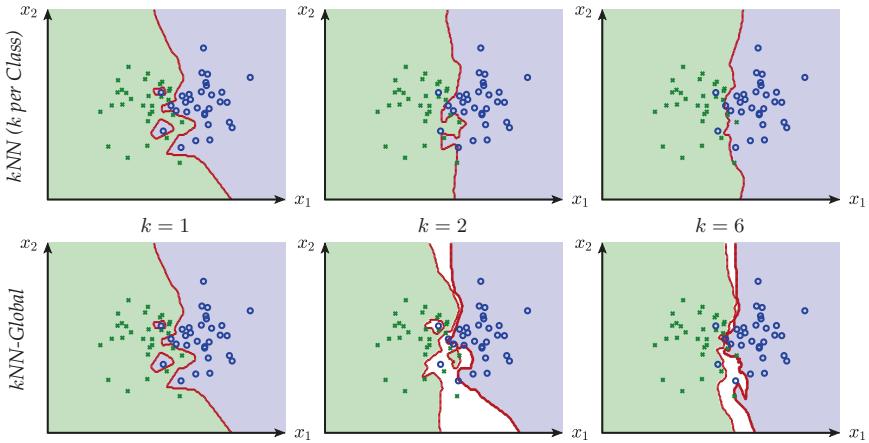


Fig. 6.20. *k*NN: Illustrations of the *k*NN classifier, based on prototypes for each class (top), or based on a majority vote of the *k* closest data points (bottom). Observe how the 1-NN classifier essentially memorizes every sample point, giving a highly irregular classifier, whereas as *k* increases the boundary becomes smoother, since individual outlying points have less of an influence. The white regions in the *k*NN-Global classifier are those points in which the classifier is undecided (has a tie among the classes).

the white regions in the bottom row of Figure 6.20. Strategies to avoid this impasse generally fall into one of two categories:

- **Varying *k*:** For any value of \underline{x} in which there is a tie among the closest *k* points, see whether one class has a plurality among the closest $k - 1$ points; if there remains a tie then see whether one class has a plurality among the closest $k - 2$ points etc. Since we know that every point is unambiguously classified for $k = 1$, the above strategy is guaranteed to classify every point.
- **Scoring/Weighting:** Rather than just seeking a class plurality among the closest *k* points, we could have a weighted score, such that data points close to \underline{x} have slightly more weight than those further away. The change in weight is intended to be slight, only to break the tie, not to be sufficiently large so that one point very close to \underline{x} would take precedence over several points (still among the *k* closest) further away.

Case Study 6: Hand-writing Recognition

The recognition of handwriting, whether free-form (a hand-written postcard) or highly constrained (a sequence of digits in known locations for the ZIP

code on an envelope), is easily learned by children in elementary school but has been a problem of great significance and challenge for a long time in pattern recognition.

In many ways we now take computer-based handwriting recognition for granted, since effective methods are implemented on most smartphones and tablets, and the problem has lost its prominence because of a number of significant shifts:

- People are writing much less than they once did
- Most envelopes and packages in the postal system have computer-generated bar codes
- Most handwriting that needs to be computer-recognized is mediated via a stylus
- Massive non-parametric pattern recognition networks have largely solved the problem

Nevertheless, the recognition of hand-writing is far from straightforward.

We have already seen examples of written digits in [Figure 2.5](#), samples from the MNIST Dataset [4] which has thousands of images of handwritten digits, a few more examples of which are shown here:



It is clear that there will be significant variations present, whether digits are straight or slanted, whether "7" is crossed, whether "4" is written open or closed etc.

However before we even begin to consider any feature extractor or distance metric, certainly we have to know what input data we are processing:

OFFLINE: The writing has already taken place, so the recognition needs to be on the basis of a scanned image. Example contexts would be reading addresses on envelopes by the postal system, reading assistance for people with limited vision, or translation/recognition of handwritten signs in a store window.

ONLINE: The recognition takes place simultaneously with the act of writing, so the input comes electronically from a pen, possibly including measurements of pressure, velocity or accelerometer data, and pen-tip location. Example contexts would be nearly any process of writing on an electronic surface.

Certainly the latter process, of online writing, is far more reliably solved, since the recognition system receives information regarding the *process* of writing, and not just a set of pixels of the end result. On the other hand, the online context will not have a constant-length feature vector for every letter or digit, since drawing “o” presumably takes less time and less motion than “ξ”.

It is rather proprietary and highly variable from system to system exactly what information might be collected by a stylus and passed on to a recognition system. Therefore let us assume that we are developing an offline system since, although probably more difficult to solve, is much easier to visualize and describe.

The key question would appear to be how to define an appropriate distance metric $d()$, however first we need to know in which feature space \mathcal{X} we are attempting to define a distance metric. The process can be chicken-and-egg

...

- Do we first choose a distance metric (say, Euclidean) and then seek a feature extractor which will keep classes Euclidean-separated?
- Or do we first develop what we believe to be a plausible feature extractor, and then attempt to visualize the feature space and to propose a distance metric on the basis of shapes and patterns of classes?

In some ways it may not even matter. Given a non-trivial task, such as hand-written digit recognition, the complexity of the problem has to be *somewhere*, so choosing a simple distance metric (Euclidean) will necessitate a complex feature extractor, or choosing simple features (pixel differences, local binary patterns) will lead to more complicated classification:

$$d_{\text{Simple}}(f_{\text{Complex}}(\mathbf{y}), \{f_{\text{Complex}}(\mathbf{y}_i)\}) \quad \text{vs.} \quad d_{\text{Complex}}(f_{\text{Simple}}(\mathbf{y}), \{f_{\text{Simple}}(\mathbf{y}_i)\}) \quad (6.29)$$

One of the key steps in visual recognition is to reduce variability. A “7” can be written large or small, shifted to the left or to the right, but in all cases is still the digit “7”. Ideally, if the given image can be reliably standardized by rescaling, rotating, shifting, and thresholding, then it becomes invariant to size, rotation, translation, and illumination.

Another key consideration is the amount of training data available. Non-parametric methods make far fewer assumptions, and therefore offer far more degrees of freedom, but which therefore require many more training samples to reliably learn.

In practice, a successful approach would either need an effective set of features based on image processing, a nonlinear feature transformation (as in [Section 10.3](#)), or a network of discriminants (as in [Section 11.4](#)).

Lab 6: Distance-Based Classifiers

We would like to explore simple distance-based classifiers. Since [Lab 3](#) already explored k NN, the reader may wish to look through the results of that lab again, in light of the much deeper understanding which we now have regarding k NN from this chapter.

So, since k NN was already explored in [Lab 3](#), let us implement and test the Mean-Mahalanobis classifier, building on [Lab 5](#) in which we showed how to plot an elliptical class shape in response to a data set. The key difference is that [Lab 5](#) focused on visualization, which would normally constrain us to two, maximally three dimensions, whereas the Mean-Mahalanobis classifier of this chapter really has no difficulty being developed in *any* number of dimensions. To emphasize that flexibility, the following code will generate the Mahalanobis classifiers for each of one, two, three, and four dimensional features:

```
load Iris

% Use variable for number of classes, since that makes code clearer
K = 3;

% Loop over number of dimensions to use
for dims = 1:4,

    % Find statistics separately for each of the three classes
    mu = cell(K,1); sigma = cell(K,1);
    for c=1:K,
        % Find points from class c and compute mean and covariance
        q = find(IrisData(:,5)==c);
        mu{c} = mean(IrisData(q,1:dims));
        sigma{c} = cov(IrisData(q,1:dims));
    end

    % compute the Mahalanobis distance for each point to each class
    dist = zeros(size(IrisData,1), K);
    for c=1:K,
        % Find points from class c and compute mean and covariance
        dist (:, c) = sum(((IrisData (:,1:dims)-mu{c}) * inv(sigma{c})) .* ...
                           (IrisData (:,1:dims)-mu{c}),2);
    end

    % classify each of the points based on the Mahalanobis classifier
    [~, cmin] = min(dist ,[],2);

    % generate the confusion matrix
    for i=1:K,
```

```

for j=1:K,
    cm(j, i) = sum( (IrisData(:,5)==j) & (cmin==i) );
end
end

% Extract overall probability of error from the confusion matrix
perr = 1-sum(diag(cm))/sum(cm(:));

```

The code is relatively straightforward and commented, however there is one line that merits a closer look. As we well know, the Mean-Mahalanobis (MMD) classifier computes the distance from data point \underline{x}_i to class C_κ as

$$(\underline{x}_i - \mu_\kappa)^T \Sigma_\kappa^{-1} (\underline{x}_i - \mu_\kappa) \quad (6.30)$$

We could have a for-loop to loop over all of the data points to separately compute (6.30) for each point, however it is *much* faster to compute to *all* of the points simultaneously, and is a good illustration of how to vectorize code.

All of our data points $\{\underline{x}_i\}$ are in

IrisData (:,1: *dims*)

one data point per row. Subtracting a row vector from a matrix

IrisData (:,1: *dims*) - *mu*{c}

subtracts that row from *each* row in the matrix, which is what we want, since the same mean is subtracted from every data point. Since the measurements are in rows they are already transposed, so multiplying by the inverse-covariance is straightforward:

((*IrisData* (:,1: *dims*) - *mu*{c}) * **inv**(*sigma*{c}))

At this point we have a $(N \times \text{dims})$ matrix, each row of which is a product vector which needs to be multiplied by a mean-removed data point. We cannot just multiply this by *IrisData* (:,1: *dims*) - *mu*{c}, since the dimensions do not work. MATLAB would allow us to multiply by the transpose, since the dimensions do work, however that is not at all the correct operation. Returning to (6.30), recognize that $(\underline{x}_i - \mu_\kappa)^T \Sigma_\kappa^{-1}$ is a row-vector, to be multiplied by column vector $(\underline{x}_i - \mu_\kappa)$, however a row-column product $\underline{z}^T \underline{z}$ is just a dot product, which can be implemented as a sum over the element-by-element product:

```
sum(((IrisData(:,1: dims) - mu{c}) * inv(sigma{c})) .* (IrisData(:,1: dims)
- mu{c}),2)
```

where the “2” at the end tells the summation to proceed over rows, and not over columns.

The resulting confusion matrices (as defined in (3.19)), for the $n = 1$ to $n = 4$ dimensional problems, are

$n = 1$	$n = 2$	$n = 3$	$n = 4$																																				
Classified	Classified	Classified	Classified																																				
Truth	Truth	Truth	Truth																																				
<table border="1"> <tr><td>40</td><td>10</td><td>0</td></tr> <tr><td>5</td><td>31</td><td>14</td></tr> <tr><td>1</td><td>12</td><td>37</td></tr> </table>	40	10	0	5	31	14	1	12	37	<table border="1"> <tr><td>49</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>34</td><td>16</td></tr> <tr><td>0</td><td>12</td><td>38</td></tr> </table>	49	1	0	0	34	16	0	12	38	<table border="1"> <tr><td>50</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>46</td><td>4</td></tr> <tr><td>0</td><td>2</td><td>48</td></tr> </table>	50	0	0	0	46	4	0	2	48	<table border="1"> <tr><td>50</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>47</td><td>3</td></tr> <tr><td>0</td><td>0</td><td>50</td></tr> </table>	50	0	0	0	47	3	0	0	50
40	10	0																																					
5	31	14																																					
1	12	37																																					
49	1	0																																					
0	34	16																																					
0	12	38																																					
50	0	0																																					
0	46	4																																					
0	2	48																																					
50	0	0																																					
0	47	3																																					
0	0	50																																					
$\mathbf{P}(e) = 28\%$	$\mathbf{P}(e) = 19\%$	$\mathbf{P}(e) = 4\%$	$\mathbf{P}(e) = 2\%$																																				

Since the same data were used to train and test the classifier, the results, above, are definitely at risk of being overfit. Nevertheless the results are most likely reasonable, because the Mean-Mahalanobis classifier has limited degrees of freedom (unlike, say, NN, as we will see in [Lab 9](#)) and so is much less likely to introduce overfitting.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links: [Distance metric](#), [Outlier](#), [Similarity learning](#), [Euclidean distance](#), [Mahalanobis distance](#), [K-nearest neighbours algorithm](#)

There has been extensive work on strategies regarding the k NN classifier, and a wide variety of illustrative videos and web pages can be found online. As discussed in [Section 6.4](#), much of the work related to k NN focuses on three questions:

- The selection of hyper-parameters (the choice of k , in particular)
- Strategies for avoiding or dealing with a tie among the k closest points
- Options for computationally efficient implementations.

The reader is referred to more advanced books on pattern recognition, for k NN in particular to [2, 3]. Finally, for the reader seeking much greater depth, [1] offers a compendium regarding nearly everything that is known about NN and k NN methods.

Sample Problems

Problem 6.1: Short Answer

Give a short definition of each of the following:

- (a) Distance metric
- (b) Minkowski distance
- (c) Mahalanobis distance
- (d) Unit-distance contour
- (e) Class prototype
- (f) Outlier

Problem 6.2: Short Answer

Offer brief answers to each of the following:

- (a) In general, for the Minkowski distance,
 - (i) In what context might we prefer $\rho = 2$ over $\rho = 1$ or $\rho = \infty$?
 - (ii) In what context might we prefer $\rho = 1$ or $\rho = \infty$ over $\rho = 2$?
- (b) For what sort of pattern recognition problem might we prefer the Mahalanobis distance over the Minkowski distance?
- (c) Suppose our classes have very unusual shapes (non compact, non elliptical). How might we approach distance-based classification in such a case?
- (d) For a set of K Gaussian clusters in n dimensions, under what conditions will the Euclidean and Mahalanobis distances yield the same classifier?
- (e) Suppose we wish to use the k -NN classifier. What are the tradeoffs in the choice of k ? In particular,
 - (i) What is the problem with choosing a value of k which is too *small*?
Draw a simple sketch to illustrate the problem.
 - (ii) What is the problem with choosing a value of k which is too *large*?
Draw a simple sketch to illustrate the problem.
- (f) Suppose we have chosen to use a NN classifier for some problem. Having selected NN, we still need to define the point-to-point distance:
 - (i) What common alternatives are there to the Euclidean distance?
 - (ii) Under what circumstances might we definitely want a non-Euclidean distance metric?

- (g) What does point pruning mean?, how do we do it?, and what is the benefit?

Problem 6.3: Conceptual — Nearest Neighbour

Each of the following statements is either true or false. If true, argue why; if false, develop a convincing counterexample:

- (a) A classifier based on NN prototypes is always more accurate than a classifier based on the mean prototype.
- (b) As k increases, the k NN prototype converges to the mean prototype.
- (c) As k increases, the k FN prototype converges to the mean prototype.

Problem 6.4: Conceptual — Nearest Neighbour

Draw a sketch illustrating very clearly how the Nearest Neighbour prototype is a function of where you are measuring from.

As a suggestion, supposing that you are given a dataset \mathcal{D} consisting of three points

$$\mathcal{D} = \{(-1, -1), (0, 1), (1, -1)\}$$

and you are asked to measure the NN-Euclidean distance to these points. Identify three 2D points such that the NN-Euclidean prototype is different for each one.

Problem 6.5: Conceptual — Nearest Neighbour

Specifying “Nearest Neighbour” alone does not uniquely specify the prototype, since we *also* need to know the distance function.

Suppose we are measuring the nearest neighbour distance from a point $(1, 1)$ to a class centered on the origin.

Sketch a class shape and offer three definitions of distance, such that the nearest neighbour to $(1, 1)$ is different for each of the three distance functions.

Problem 6.6: Conceptual — Classification

Suppose we have a 2D classification problem involving two classes, A and B . What can you say, in general, about the *shape* of the classification boundary in each of the following cases:

- (a) The Mean-Euclidean classifier
- (b) The Mean-Mahalanobis classifier
- (c) The NN-Euclidean classifier

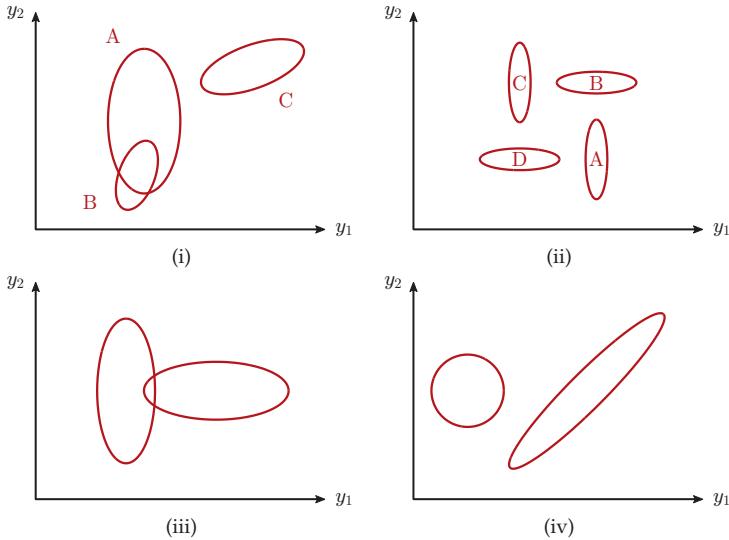


Fig. 6.21. Class shapes and sizes for Problems 6.8 and 6.9.

Problem 6.7: Analytic — Distances

In (6.22) and (6.23) we argued that comparing distances or squared-distances leads to the equivalent classifier,

$$d(\underline{x}, \underline{z}_1) \stackrel{C_2}{\underset{C_1}{\gtrless}} d(\underline{x}, \underline{z}_2) \quad \text{is the same as} \quad (d(\underline{x}, \underline{z}_1))^2 \stackrel{C_2}{\underset{C_1}{\gtrless}} (d(\underline{x}, \underline{z}_2))^2 \quad (6.31)$$

because the squaring operation $(d)^2$ is a monotonic function of $d > 0$, meaning that

$$a > b \iff a^2 > b^2 \quad \text{as long as } a > 0, b > 0 \quad (6.32)$$

Demonstrate that the classifier equivalence does *not* hold if the function $\varphi()$ is *not* monotonic; that is,

$$d(\underline{x}, \underline{z}_1) \stackrel{C_2}{\underset{C_1}{\gtrless}} d(\underline{x}, \underline{z}_2) \quad \text{is not the same as} \quad \varphi(d(\underline{x}, \underline{z}_1)) \stackrel{C_2}{\underset{C_1}{\gtrless}} \varphi(d(\underline{x}, \underline{z}_2)) \quad (6.33)$$

Two positive non-monotonic functions to try:

$$\varphi(d) = (d - 1)^2 \quad \text{or} \quad \varphi(d) = 2 + \sin(d) \quad (6.34)$$

Problem 6.8: Conceptual — Distance Classification

Suppose we have three Gaussian classes, A , B and C , for which the unit standard deviation contours are as shown in Figure 6.21(i).

Suppose that we had 50 labelled samples from A , and 100 labelled samples from each of B and C , and that we were to classify these points. Write down an approximate confusion matrix for each of the MED and MMD classifiers.

Problem 6.9: Conceptual — Distance Classification

For each of the four sets of classes defined in Figure 6.21,

- (a) Sketch the MED classifier,
- (b) Sketch the MMD classifier.

Problem 6.10: Conceptual/Analytic — Distance Classification

Given two classes with means and covariances:

$$\underline{\mu}_A = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad S_A = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \quad \underline{\mu}_B = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \quad S_B = \begin{bmatrix} 4 & 2 \\ 2 & 4 \end{bmatrix} \quad (6.35)$$

- (a) Sketch the unit standard-deviation ellipse for class A.
- (b) Sketch the unit standard-deviation ellipse for class B.
- (c) Draw a sketch showing the two classes and the MED classification boundary.
- (d) Use Fisher's method to identify a feature direction \underline{w} .
- (e) Draw a sketch showing the two classes and the straight-line boundary having \underline{w} as its normal vector.

Problem 6.11: Conceptual/Analytic — Distance Classification

Given three classes in 2D space y with means and covariances:

$$\underline{\mu}_A = \begin{bmatrix} 0 \\ 5 \end{bmatrix} \quad S_A = \begin{bmatrix} 16 & 2 \\ 2 & 1 \end{bmatrix} \quad \underline{\mu}_B = \begin{bmatrix} 8 \\ 3 \end{bmatrix} \quad S_B = \begin{bmatrix} 3 & 0 \\ 0 & 10 \end{bmatrix}$$

$$\underline{\mu}_C = \begin{bmatrix} 8 \\ 0 \end{bmatrix} \quad S_C = \begin{bmatrix} 1 & -1 \\ -1 & 4 \end{bmatrix}$$

- (a) Sketch the unit standard-deviation ellipses for all three classes.
- (b) Draw a sketch showing the three classes and the MED classification boundary.

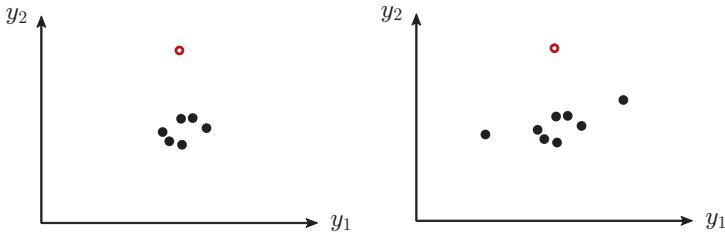


Fig. 6.22. Two sets of point clusters (solid black) and a defined distance point (open red) for Problem 6.12. The task is to plot the constant-distance contour, that curve of constant distance to the black points which passes through the red circle.

- (c) Similarly sketch (but don't try to compute) the MMD classification boundary.
- (d) We want to reduce this to a 1D problem. Identify an effective feature direction \underline{w} to keep.

Let a new feature be

$$x = \underline{w}^T y$$

Note that a classifier on x implies a classifier on y . That is, for each y we find $x = \underline{w}^T y$ and then we determine the class based on x .

- (e) Sketch (but don't compute) the classification boundary in the 2D domain of y , based on the MED classifier applied to x .
- (f) Sketch (but don't compute) the classification boundary in the 2D domain of y , based on the MMD classifier applied to x .

Problem 6.12: Conceptual — Distance Contours

Let our definition of distance be the normal Euclidean distance. Then, given a set of points and a definition of prototype, we can draw a constant-distance contour for the cluster.

For both of the point clusters in Figure 6.22, draw that constant-distance contour which passes through the red open circle (which denotes a distance, it is not a data point) for each of the following five prototype definitions:

- (a) Mean
- (b) Furthest-Neighbour
- (c) Nearest-Neighbour
- (d) 2-Nearest-Neighbour
- (e) 4-Nearest-Neighbour

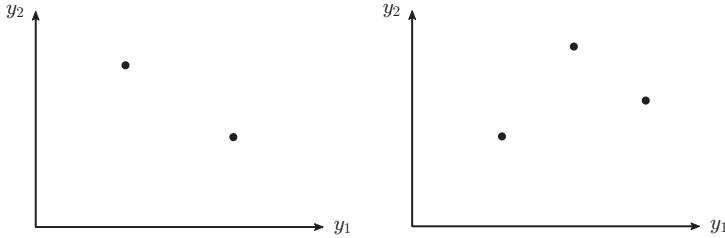


Fig. 6.23. Two sets of prototypes are given, for two classes (left) and three classes (right), for which classification boundaries are to be sketched in [Problem 6.13](#).

Problem 6.13: Conceptual — Minkowski Distance

We have looked at various definitions of point-to-point distance $d(\underline{x}, \bar{x})$, most commonly the Euclidean case. In this question we will consider the Minkowski distance family

$$d_\rho(\underline{x}, \bar{x}) = \left(\sum_{j=1}^n (x_j - \bar{x}_j)^\rho \right)^{1/\rho}$$

- (a) We begin in 2D: For each of $\rho = 1, 2, 4, \infty$, draw a sketch of a contour of constant distance from the origin.
- (b) Sketching is much harder, of course, in 3D. Instead, describe (in a few words) the 3D constant-distance shapes corresponding to each of $p = 1, 2, 4, \infty$.
- (c) A generic minimum-distance classifier is very easy to define. Given two fixed prototypes $\underline{z}_1, \underline{z}_2$, a point \underline{x} is classified as

$$d_\rho(\underline{x}, \underline{z}_1) \underset{C_1}{\underset{\geq}{\overset{C_2}{\gtrless}}} d_\rho(\underline{x}, \underline{z}_2)$$

We have studied this in the Euclidean case, but the concept generalizes just as well to any non-Euclidean distance, such as the Minkowski distance $d_\rho()$.

Sketch the classification boundaries for both sets of prototypes shown in [Figure 6.23](#) for each of $p = 1, 2, \infty$ (i.e., six sketches in all).

Problem 6.14: Numeric/Computational — Distance Contours

Implement a numerical solution to each part of [Problem 6.13](#). That is,

- (a) Produce the 2D constant-distance contours (consider using a contour plot)

- (b) Produce the 3D constant-distance contours (consider using an isosurface plot)
- (c) Produce the Minkowski classification boundaries. For the prototypes from [Figure 6.23](#), use

$$\begin{aligned} \text{Prototype Set I: } \underline{z}_1 &= \begin{bmatrix} 2 \\ 4 \end{bmatrix} & \underline{z}_2 &= \begin{bmatrix} 5 \\ 2 \end{bmatrix} \\ \text{Prototype Set II: } z_1 &= \begin{bmatrix} 2 \\ 2 \end{bmatrix} & z_2 &= \begin{bmatrix} 4 \\ 4 \end{bmatrix} & z_3 &= \begin{bmatrix} 5 \\ 3 \end{bmatrix} \end{aligned} \quad (6.36)$$

Problem 6.15: Reading — Distributed/Federated Learning

With this chapter we have begun to see classifiers being learned, so we can begin to think of broader generalizations to classifier learning. Read up on [Federated](#) or [Distributed Learning](#), and prepare an overview of the motivations and challenges in performing classifier learning this way.

References

1. G. Biau, L. Devroye, *Lectures on the Nearest Neighbor Method* (Springer, 2015)
2. L. Devroye, L. Gyorfi, G. Lugosi, *A Probabilistic Theory of Pattern Recognition* (Springer, 1996)
3. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley Interscience, 2009)
4. Y. LeCun, C. Cortes, C. Burges, *The MNIST Database of Handwritten Digits*



Inferring Class Models

Chapters 4 and 5 explored pattern shape and feature definitions. Throughout those discussions were a great many unknowns — parameters, shapes, distributions, probabilities etc.

In most real-life problems, learning the values of unknown parameters, shapes, distributions, and probabilities represents an essential part of inferring models from data. Given a pattern recognition problem involving K classes C_1, \dots, C_K , ideally we would like to understand the shape or distribution of each class in feature space \mathcal{X} :

$$p(\underline{x} | C_1) \quad p(\underline{x} | C_2) \quad \dots \quad p(\underline{x} | C_K) \quad (7.1)$$

where these distributions are to be learned, somehow, from given measured data $\underline{x}_{\kappa i}$ — the i th data point associated with class C_κ .

Very broadly, as described in Section 4.2 (particularly Table 4.1), how we choose to describe class distributions may fall into one of two forms:

PARAMETRIC: Each class C_κ is *explicitly* described in terms of an analytical model,

$$p(\underline{x} | C_\kappa) \equiv p(\underline{x} | \underline{\theta}_\kappa) \quad \leftarrow p() \text{ known, } \underline{\theta}_\kappa \text{ unknown} \quad (7.2)$$

where the form of model $p()$ is known (or assumed) and is a function of some parameter vector $\underline{\theta}_\kappa$. It is $\underline{\theta}_\kappa$ which needs to be learned from the data $\underline{x}_{\kappa i}$ associated with class C_κ , as described in Sections 7.1 and 7.2.

NON-PARAMETRIC: Each class is *implicitly* described in terms of a collection of data points,

$$p(\underline{x} | C_\kappa) \equiv p(\underline{x} | \{\underline{x}_{\kappa i}\}) \quad \leftarrow p() \text{ is an algorithm} \quad (7.3)$$

such that no particular analytical form is assumed for the shape of the class, as described in [Section 7.3](#), although clearly we will need to define or state what $p()$ is doing.

7.1 Parametric Estimation

We have already seen a number of examples of learning class shapes, particularly the ellipsoids of [Example 4.5](#) and [Lab 5](#). In those cases the equations for the ellipses (based on sample means and sample covariances) were just asserted. In this section we would like to understand where those equations came from, and how we might generalize to the inference of parameters associated with other class shapes, since not every class is necessarily well represented by an ellipse.

In order to estimate a vector $\underline{\theta}$ of unknown parameters, we need to describe the relationship, the connection, between $\underline{\theta}$ and our N given data points $\underline{x}_1, \underline{x}_2, \dots, \underline{x}_N$. Ideally this relationship takes the form of a [conditional probability distribution](#), representing the forward model relating the (unknown) parameters to the (known) data points:

Distribution of data points as a function of parameter $\underline{\theta}$: $p(\{\underline{x}_i\} | \underline{\theta})$ (7.4)

Knowing this relationship, the next question is how we might *invert* it, so to speak, to allow us to infer or estimate¹ $\underline{\theta}$ from the data points $\{\underline{x}_i\}$. The most fundamental approach is *Maximum Likelihood* (ML), to choose $\underline{\theta}$ as that vector of values which makes the observed points most likely:

$$\hat{\underline{\theta}}_{\text{ML}} = \arg_{\underline{\theta}} \max \underbrace{p(\{\underline{x}_i\} | \underline{\theta})}_{\text{Forward model}} \quad (7.5)$$

In most cases, if $p()$ is a continuous function,² the maximum likelihood estimate will be at a local maximum, found by differentiating³

$$\frac{\partial p(\{\underline{x}_i\} | \underline{\theta})}{\partial \underline{\theta}} \Bigg|_{\underline{\theta} = \hat{\underline{\theta}}_{\text{ML}}} = 0 \quad \text{or} \quad \frac{\partial \log p(\{\underline{x}_i\} | \underline{\theta})}{\partial \underline{\theta}} \Bigg|_{\underline{\theta} = \hat{\underline{\theta}}_{\text{ML}}} = 0 \quad (7.6)$$

¹ Here the word “estimate” is a formal mathematical approach, a part of *Estimation Theory*, meaning how to learn one thing from another, quite different from the English vernacular “estimate”, meaning an approximated assessment.

² Although maximum likelihood can also be solved for discontinuous $p()$, as illustrated in [Example 7.2](#).

³ To be more precise, the condition of (7.6) will clearly find both local maxima and minima. For continuous $p()$, unbounded $\underline{\theta}$, and only a single extremum, the maximum, then (7.6) is all we need. In any other circumstance finding the global maximum may be more difficult.

where the equivalence of these two forms stems from the fact that $\log()$ is a monotonically increasing function. That is, any maximum of $p()$ will also be a maximum of $\log p()$.

In the preceding maximum-likelihood discussion $\underline{\theta}$ was treated as a vector of unknowns, where we had no knowledge of any kind regarding the values in $\underline{\theta}$. There are many circumstances, however, in which we have information about $\underline{\theta}$ separate from $\{\underline{x}_i\}$, so-called *prior* information described by a distribution $p(\underline{\theta})$. In particular, we might have an existing (prior) estimate of $\underline{\theta}$, but which we would now like to update on the basis of new measurements without discarding or losing past information. This is referred to as a *Bayesian* approach, meaning a context in which the unknowns to be estimated are random variables, and subject to a distribution encoding the prior knowledge. The Bayesian approach which parallels maximum likelihood is known as Maximum a Posteriori (MAP).⁴

$$\hat{\underline{\theta}}_{\text{MAP}} = \arg_{\underline{\theta}} \max p(\underline{\theta} | \{\underline{x}_i\}) \quad (7.7)$$

That is, we now choose $\underline{\theta}$ to be the most likely vector of values based on the observed points, deceptively similar to the maximum likelihood statement preceding (7.5). It is important to recognize the difference:

- In (7.5) we have a distribution for \underline{x} , where $\underline{\theta}$ is just a vector of unknown parameters.
- In (7.7) we have a distribution for $\underline{\theta}$, so $\underline{\theta}$ is a random vector, and we are maximizing over the distribution of $\underline{\theta}$, not of \underline{x} .

The first challenge in solving the MAP problem of (7.7) is that we typically do not know $p(\underline{\theta} | \{\underline{x}_i\})$, however Bayes' Rule allows us to express it in terms of distributions which we *do* know:

$$\arg_{\underline{\theta}} \max p(\underline{\theta} | \{\underline{x}_i\}) = \arg_{\underline{\theta}} \max \frac{p(\{\underline{x}_i\} | \underline{\theta}) \cdot p(\underline{\theta})}{p(\{\underline{x}_i\})} = \arg_{\underline{\theta}} \max \underbrace{p(\{\underline{x}_i\} | \underline{\theta})}_{\substack{\text{Bayes' Rule} \\ \text{Forward model}}} \cdot \underbrace{p(\underline{\theta})}_{\substack{\text{Remove } p(\{\underline{x}_i\}) \\ \text{Prior model}}} \quad (7.8)$$

where the $p()$ term in the denominator can be removed, since it is not a function of $\underline{\theta}$, and therefore does not affect the value of $\underline{\theta}$ which maximizes the overall expression. As in (7.6) we may choose to take a logarithm,

$$\arg_{\underline{\theta}} \max p(\{\underline{x}_i\} | \underline{\theta}) \cdot p(\underline{\theta}) = \arg_{\underline{\theta}} \max [\log p(\{\underline{x}_i\} | \underline{\theta}) + \log p(\underline{\theta})] \quad (7.9)$$

and in any given problem we can choose whichever form is more simplified.

Although MAP (or related Bayesian methods) is widely used in inverse problems, remote sensing, and computer vision (see [Further Reading](#)), analytically

⁴ Meaning the maximum *afterwards*, presumably written in Latin since "Maximum Posterior" sounds a bit weird.

MAP is significantly more difficult to solve than ML, and ML-related methods are significantly more widely used in pattern recognition, so most of this text will focus on ML.

7.2 Parametric Model Learning

We have already seen repeated examples of parametric model learning:

- A mean (the middle of a class)
- A variance (a measure of the size of a class)
- A probability (the likelihood of a class)

Each of these can be derived more formally, using Maximum Likelihood, from first principles. Suppose we are given a set of Gaussian-distributed independent one-dimensional values $x_i \sim \mathcal{N}(\mu, \sigma^2)$. Even though we know that μ represents the mean of the distribution, let us formally derive its estimator. We begin with the formal definition of the maximum likelihood estimate of unknown value μ :

$$\hat{\mu}_{\text{ML}} = \arg_{\mu} \max p(\{x_i\} | \mu) = \arg_{\mu} \max \underbrace{\prod_{i=1}^N p(x_i | \mu)}_{\text{Independence of } \{x_i\}} \quad (7.10)$$

From (7.6) we know that we can take a logarithm; since the Gaussian distribution contains an exponential, a logarithm provides convenient simplification:

$$\frac{\partial}{\partial \mu} \log \left\{ \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} \exp \left(-\frac{(x_i - \mu)^2}{2\sigma^2} \right) \right\} = 0 \rightarrow \frac{\partial}{\partial \mu} \sum_{i=1}^N \log \left\{ \frac{1}{\sqrt{2\pi}\sigma} \right\} - \frac{(x_i - \mu)^2}{2\sigma^2} = 0 \quad (7.11)$$

Actually taking the derivative, we find

$$\sum_{i=1}^N 0 - \frac{(x_i - \mu)}{\sigma^2} \cdot -1 = 0 \rightarrow \sum_{i=1}^N \frac{(x_i - \mu)}{\sigma^2} = 0 \rightarrow \sum_{i=1}^N (x_i - \mu) = 0 \quad (7.12)$$

The value of μ that satisfies (7.12) is the maximum likelihood estimate, therefore

$$\sum_{i=1}^N x_i = \sum_{i=1}^N \hat{\mu}_{\text{ML}} = N\hat{\mu}_{\text{ML}} \rightarrow \hat{\mu}_{\text{ML}} = \frac{1}{N} \sum_{i=1}^N x_i \quad (7.13)$$

That is, every time we take the average of a set of numbers to find the mean, really what we are doing is finding the maximum likelihood estimate $\hat{\mu}_{\text{ML}}$ of

the true mean μ , assuming that the numbers $\{x_i\}$ are randomly and independently drawn with a fixed variance.

Finding an estimate is, at best, only half of the story, because it is essential for us to know how accurate or meaningful the estimate is. We define the error⁵

$$\text{Estimation Error } \tilde{\theta}_{\text{ML}} = \hat{\theta}_{\text{ML}} - \theta \quad (7.14)$$

For any estimator, there are two basic questions we can have regarding its error:

1. Is there a *systematic* error, is the estimator consistently biased?
2. What is the variance of the *random* (non-bias) part of the error?

The systematic error or bias is just the expectation (the mean) of the error:

$$\text{Estimator Bias} = b = \mathbb{E}[\tilde{\theta}_{\text{ML}}] \quad (7.15)$$

From the definition of the maximum-likelihood estimation of the mean μ in (7.13), we can derive the bias for that estimator as

$$\mathbb{E}[\tilde{\mu}_{\text{ML}}] = \mathbb{E}[\hat{\mu}_{\text{ML}} - \mu] = \mathbb{E}\left[\frac{1}{N} \sum_{i=1}^N x_i\right] - \mu \quad (7.16)$$

From (B.7), since expectation is essentially like an integral, we can switch the order of summation and expectation, so

$$\mathbb{E}\left[\frac{1}{N} \sum_{i=1}^N x_i\right] - \mu = \frac{1}{N} \sum_{i=1}^N \mathbb{E}[x_i] - \mu = \underbrace{\frac{1}{N} \sum_{i=1}^N \mu}_{x_i \text{ has mean } \mu} - \mu = 0 \quad (7.17)$$

So the bias is zero, meaning that the estimation of the mean is *unbiased*, which is, of course, what we want in an estimator. Next, the variance of the estimation error is defined as

$$\text{Estimator Error Variance} = \text{var}(\tilde{\theta}_{\text{ML}}) = \mathbb{E}\left[\left(\underbrace{\tilde{\theta}_{\text{ML}}}_{\text{Error}} - \underbrace{\mathbb{E}[\tilde{\theta}_{\text{ML}}]}_{\text{Bias}}\right)^2\right] \quad (7.18)$$

We can derive the estimation error variance for our estimator of (7.13) as

$$\text{var}(\tilde{\mu}_{\text{ML}}) = \mathbb{E}\left[\left(\underbrace{\frac{1}{N} \sum_{i=1}^N x_i}_{\tilde{\mu}_{\text{ML}}} - \underbrace{\mu}_{\text{Bias}} - 0\right)^2\right] = \frac{1}{N^2} \mathbb{E}\left[\left(\sum_{i=1}^N (x_i - \mu)\right)^2\right] \quad (7.19)$$

⁵ Some texts define error (and therefore bias) the other way around, such that $\tilde{\theta}_{\text{ML}} = \theta - \hat{\theta}_{\text{ML}}$. This leads to the opposite sign on errors and biases, but otherwise does not change anything.

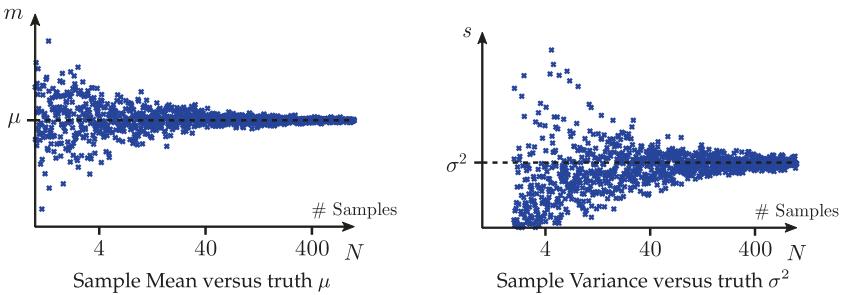


Fig. 7.1. SAMPLE MEAN & VARIANCE: Sample statistics refer to statistics inferred from data. Since data are always subject to some degree of variability and noise, the sample statistics are not exact, rather they approach the true value as more and more data are available, here shown for a mean (left) and variance (right).

In contrast to the transition from (7.16) to (7.17), here the quantity inside the expectation is squared, so we cannot just reverse the order of summation and squaring:

$$\text{var}(\tilde{\mu}_{\text{ML}}) = \frac{1}{N^2} \mathbb{E} \left[\sum_{i=1}^N \sum_{j=1}^N (x_i - \mu)(x_j - \mu) \right] \quad (7.20)$$

$$= \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N \mathbb{E}[(x_i - \mu)(x_j - \mu)] \quad (7.21)$$

$$= \frac{1}{N^2} \sum_{i=1}^N \underbrace{\mathbb{E}[(x_i - \mu)^2]}_{\text{The variance } \sigma^2} + \sum_{i \neq j}^N \underbrace{\mathbb{E}[(x_i - \mu)(x_j - \mu)]}_{= 0, \text{ since samples are independent}} \quad (7.22)$$

$$= \frac{1}{N^2} \cdot N \cdot \sigma^2 = \frac{\sigma^2}{N} \quad (7.23)$$

from which we find that the standard-deviation in the estimation of a mean is σ/\sqrt{N} , as can be seen in Figures 7.1 and 7.2, a result which we used already in (3.11) in Chapter 3, but which we now understand in detail.

The derivation for the maximum-likelihood estimate of the variance is similar, from which we find that

$$\hat{\sigma}_{\text{ML}}^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{\mu}_{\text{ML}})^2 \quad (7.24)$$

The careful reader will observe that this estimator differs slightly from the sample statistic stated in Section B.6. The issue is that the maximum likelihood estimator of (7.24) is biased, and so the unbiased form of (B.49) is often preferred.

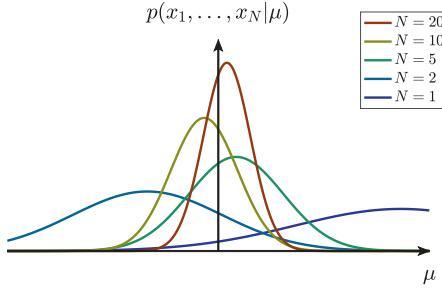


Fig. 7.2. ML LEARNING: We can learn an unknown quantity, here a mean μ , from sample data x_1, \dots, x_N , using *maximum likelihood (ML)*. The uncertainty in μ , seen as the width of the distribution $p(x_1, \dots, x_N | \mu)$, decreases as the number N of samples increases. For $N = 1$ and $N = 2$ samples (left and right blue curves) the estimate of μ is poor, however with more data points (red) our understanding of μ converges significantly.

Both of the mean and variance estimators of (7.13) and (7.24) were derived for simplicity and clarity in the scalar case. The derivation for the estimation of mean and covariance from vector values $\{\underline{x}_i\}$ is similar, leading to

$$\hat{\mu}_{\text{ML}} = \frac{1}{N} \sum_{i=1}^N \underline{x}_i \quad \hat{\Sigma}_{\text{ML}} = \frac{1}{N} \sum_{i=1}^N (\underline{x}_i - \hat{\mu}_{\text{ML}})(\underline{x}_i - \hat{\mu}_{\text{ML}})^T \quad (7.25)$$

Of the three main types of parameters listed on page 154, there remains the estimation of a probability, for example the unknown probability ϑ of some class C . Suppose we have independent observations, each of which may or may not be from class C ; the governing forward model for such events is the Binomial distribution. Suppose we have N independent observations, of which M are observed to come from class C , then

$$\begin{aligned} \mathbf{P}(M \text{ of } N \text{ samples from } C) &= \binom{N}{M} \vartheta^M (1 - \vartheta)^{N-M} \\ &= \frac{N!}{M!(N-M)!} \vartheta^M (1 - \vartheta)^{N-M} \end{aligned} \quad (7.26)$$

Therefore we can formulate the maximum likelihood estimate of probability ϑ as

$$\hat{\vartheta}_{\text{ML}} = \arg_{\vartheta} \max \frac{N!}{M!(N-M)!} \vartheta^M (1 - \vartheta)^{N-M} \quad (7.27)$$

Taking the log likelihood, as we have before,

$$\hat{\vartheta}_{\text{ML}} = \arg_{\vartheta} \max \log \left(\frac{N!}{M!(N-M)!} \right) + M \log \vartheta + (N-M) \log(1 - \vartheta) \quad (7.28)$$

and then differentiating

$$\frac{\partial}{\partial \vartheta} \left\{ M \log \vartheta + (N - M) \log(1 - \vartheta) \right\} = \frac{M}{\vartheta} + \frac{N - M}{1 - \vartheta} \cdot -1 = 0 \quad (7.29)$$

leads to the desired estimator:

$$M(1 - \vartheta) = (N - M)\vartheta \quad \longrightarrow \quad \hat{\vartheta}_{\text{ML}} = \frac{M}{N} \quad (7.30)$$

That is, any time we use the frequency of occurrence, the fraction of observed samples M/N as a measure of a probability, implicitly what we are doing is finding the maximum-likelihood estimate of the underlying probability. Deriving the bias and error variance of the estimator in (7.30) is a detail left to [Appendix D](#).

The reader should not be left with the impression that maximum likelihood is applied exclusively or primarily to linear/Gaussian problems. The uniform distribution of [Example 7.2](#), probably the simplest type of distribution, actually leads to a nonlinear maximum likelihood estimator, derived in [Example 7.2](#); the parameter for exponential distributions can also be derived, as explored in [Problems 7.4, 7.11, and 7.12](#). That is, the maximum likelihood method is in no way limited to or uniquely suited to Gaussian statistics.

If the reader recalls the challenges encountered in selecting model order for polynomial fitting in [Chapter 3](#) or feature dimensionality in [Chapter 5](#), the same issue can appear in parametric models, if there is some flexibility regarding the model complexity.

So in the same way that a given set of measured data may not well be fit by a straight line, and may require a higher-order polynomial, similarly a given set of sample points may not well be modelled by a single Gaussian distribution, and may instead be better represented by the superposition of *multiple* Gaussians, known as a *mixture model*:

$$x \sim \sum_{i=1}^q h_i \mathcal{N}(\mu_i, \sigma_i^2) \quad \text{with normalized weights} \quad \sum_{i=1}^q h_i = 1 \quad (7.31)$$

as illustrated in [Figure 7.3](#). It is important to understand that *any* distribution can be combined as a mixture, the individual elements do not need to be Gaussian, and could instead be uniform as shown in [Figure 7.3](#). Mixtures are easy to think about and visualize in one dimension, however (7.31) does readily generalize to the n -dimensional case as

$$\underline{x} \sim \sum_{i=1}^q h_i \mathcal{N}(\mu_i, \Sigma_i) \quad \sum_i h_i = 1 \quad (7.32)$$

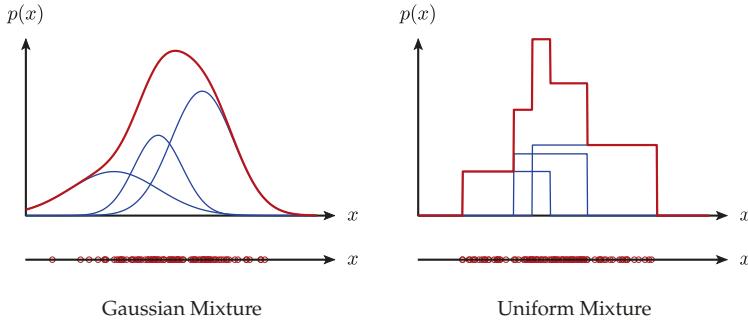


Fig. 7.3. MIXTURE MODELS: Two examples of mixture models, based on (7.31), such that a given distribution $p(x)$, red, is represented as a weighted sum of simple, parameterized distributions (here Gaussian and Uniform, shown in blue). In each case, a random set of points is shown, sampled from $p(x)$. In most cases the challenge is, of course, the other way around: to *infer* the mixture parameters *from* the sample points. Mixture models allow for much greater flexibility in the shape of distributions, but at a cost of additional parameters.

It is also important to understand that a weighted sum of distributions (a mixture model, as in (7.31)) is *not at all* the same as a weighted sum of random variables (not a mixture):

$$\underbrace{x \sim \sum_{i=1}^q h_i \mathcal{N}(\mu_i, \sigma_i^2)}_{\begin{array}{l} \text{A weighted sum of distributions} \\ x \sim \text{Mixture Model} \end{array}} \quad \underbrace{x = \sum_{i=1}^q h_i x_i}_{\begin{array}{l} \text{A weighted sum of random variables} \\ x \sim \text{Single Gaussian} \end{array}} \quad (7.33)$$

since a weighted sum of Gaussian random variables is, itself, just a single Gaussian random variable.

It might be tempting to think that we could place all of the parameters from (7.31) into a single vector of unknowns,

$$\underline{\theta}^T = \{q, h_1, \mu_1, \sigma_1, h_2, \mu_2, \sigma_2, \dots, h_q, \mu_q, \sigma_q\} \quad (7.34)$$

which is then estimated, as in (7.5), from a set of N data points $\{x_i\}$ via maximum likelihood:

$$\hat{\underline{\theta}}_{\text{ML}} = \arg_{\underline{\theta}_{\text{ML}}} \max p(\{x_i\} | \underline{\theta}_{\text{ML}}) \quad (7.35)$$

In fact, the estimation in (7.35) is guaranteed to fail, since (7.35) can be maximized by setting

$$q = N \quad \mu_i = x_i \quad \sigma_i = 0 \quad (7.36)$$

badly overfitting the data by placing a mixture element of infinitesimal width, but fantastic height, at *each* data point, as illustrated in [Example 7.1](#). For the

same reason that we need to limit the order of a polynomial in regression, as discussed in [Chapter 3](#), similarly here we need to limit the number of mixture elements q .

Even given q , the estimation of the mixture parameters remains a relatively complex optimization problem.⁶ One challenge is that we do not know, ahead of time, which data point should be associated with which of the q elements. Essentially this is very much like a clustering problem, which we will see much later in [Chapter 12](#), such that an iterative approach is needed to infer the degree to which each given data point is associated with each of the q elements. One fairly standard approach is shown in [Algorithm 7.1](#), and the use of this method is illustrated in [Lab 7](#), however many other strategies have been developed for such optimization.

Finally, most of this section has examined estimators based on Maximum Likelihood, however the reader will recall that [Section 7.1](#) also stated the form of the Maximum a Posteriori (MAP) estimator in [\(7.7\)](#), [\(7.8\)](#). The derivation of the MAP estimator is somewhat involved and so is unhelpful to write out in detail (see [Further Reading](#)), however there is some value in developing an intuition regarding the behaviour of the MAP estimator.

Suppose we take the same problem as in [Section 7.1](#), such that we have N noisy measurements of an unknown value μ :

$$x_i \sim \mathcal{N}(\mu, \sigma_{\text{Meas}}^2) \quad 1 \leq i \leq N \quad (7.46)$$

where σ_{Meas}^2 reflects the variance of the measurement noise.

From [\(7.13\)](#), we know the resulting maximum likelihood estimator to be

$$\hat{\mu}_{\text{ML}} = \frac{1}{N} \sum_{i=1}^N x_i \quad (7.47)$$

In a Bayesian setting, in addition to the measurements we would also have some sort of *prior* knowledge, additional information not reflected in the measurements. For the purpose of this discussion, suppose that μ is subject to a known distribution,

$$p(\mu) \sim \mathcal{N}(\mu_{\text{Prior}}, \sigma_{\text{Prior}}^2) \quad (7.48)$$

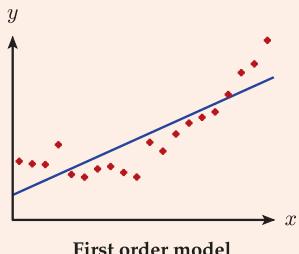
where σ_{Prior}^2 measures the variance (the uncertainty) in the assertion of the prior value μ_{Prior} . As $\sigma_{\text{Prior}} \rightarrow 0$ we are increasingly confident in the prior model, and as $\sigma_{\text{Prior}} \rightarrow \infty$ we are less confident.

The resulting MAP estimator is then a balanced tradeoff between the prior and the measurements, where the balance is found based on the relative accuracy in the prior and measurements, a function of their respective variances:

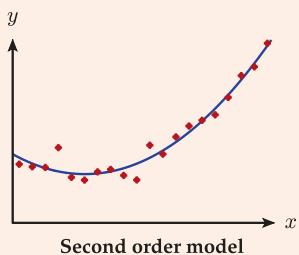
⁶ See [Appendix C](#) for an overview of optimization.

Example 7.1: Learning of Mixture Models

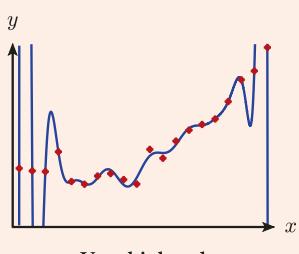
There are deep connections present between the learning of mixture models or non-parametric distributions, and the principles of learning explored in [Chapter 3](#). Note that this example explores two very *different* learning problems: fitting two-dimensional data points (x_i, y_i) (left) sampled from a noisy parabola, and inferring the distribution behind a set of one-dimensional data points z_i (right), which were sampled from a bi-modal distribution.



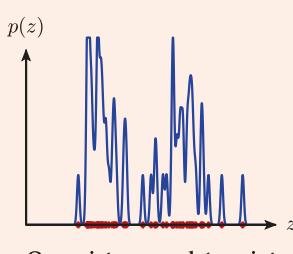
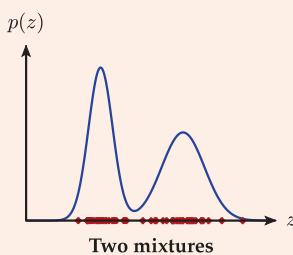
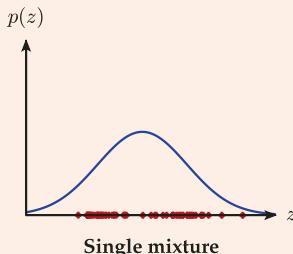
Given data points, a very simple model would be to fit a straight line (left) or a single Gaussian distribution (right). In this case, the model probably has insufficient degrees of freedom to capture the behaviour in the data.



We can consider a higher order model by increasing the degrees of freedom, fitting a parabola (left) or a pair of Gaussians as a mixture (right), an effective fit.



A much higher-order model will fit the data points much more closely, but is now overfitting to the noise (left) or the random sampling variations (right), such that both representations are no longer meaningful.



As we saw in [Chapter 3](#), for every increase in model order there needs to be an expectation of a significant improvement in fit, *more* than just fitting the noise. How to validate a proposed model was summarized in [Example 3.3](#) and will be examined in more detail in [Chapter 9](#).

Further Reading: See [Lab 7](#) for a further exploration of mixture models.

Example 7.2: Nonlinear Maximum Likelihood

We tend to see many examples of linear estimators, such as the mean (7.13) or a probability (7.30), so it may be helpful to consider a nonlinear example.

Suppose random variable x is uniformly distributed, thus

$$x \sim \mathcal{U}(a, b) \quad \rightarrow \quad p(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{Otherwise} \end{cases} \quad (7.37)$$

Suppose we are now given random samples x_1, x_2, \dots, x_N , independently drawn from $p(x)$. The unknown parameters in the distribution for x are the two endpoints, thus

$$\underline{\theta} = \begin{bmatrix} a \\ b \end{bmatrix} \quad (7.38)$$

We would like to find the maximum likelihood estimate for a and b from the observed random samples. Starting with the definition of maximum likelihood in (7.5),

$$\hat{\underline{\theta}}_{\text{ML}} = \arg \underline{\theta} \max p(\{x_i\} | \underline{\theta}) = \arg_{a,b} \max \prod_{i=1}^N p(x_i | a, b) \quad (7.39)$$

If we let

$$x_{\min} = \min\{x_i\} \quad x_{\max} = \max\{x_i\} \quad (7.40)$$

then from (7.37) we can observe that

$$\text{If } a > x_{\min} \quad \rightarrow \quad \prod_{i=1}^N p(x_i | a, b) = 0 \quad \text{since} \quad p(x_{\min} | a, b) = 0 \quad (7.41)$$

$$\text{If } b < x_{\max} \quad \rightarrow \quad \prod_{i=1}^N p(x_i | a, b) = 0 \quad \text{since} \quad p(x_{\max} | a, b) = 0 \quad (7.42)$$

Having $p() = 0$ will certainly not maximize $p()$, so it is clear that we must have

$$\hat{a}_{\text{ML}} \leq \min\{x_i\} \quad \hat{b}_{\text{ML}} \geq \max\{x_i\}, \quad (7.43)$$

in which case

$$\prod_{i=1}^N p(x_i | a, b) = \prod_{i=1}^N \frac{1}{b-a} = \frac{1}{(b-a)^N} \quad (7.44)$$

By inspection, we can see that $p()$ is maximized by minimizing $(b - a)$, where the minimum size of $(b - a)$ is constrained by (7.43), therefore

$$\hat{a}_{\text{ML}} = \min\{x_i\} \quad \hat{b}_{\text{ML}} = \max\{x_i\} \quad (7.45)$$

are the nonlinear maximum likelihood estimates for the endpoints of the unknown uniform distribution.

Algorithm: Gaussian Mixture Models

Goals: Perform Gaussian mixture-model learning to fit to N data points $\{\underline{x}^i\}$

Function Means, Covariances, Weights = **GMM_Learning**($\{\underline{x}^i\}$, $\{\underline{\mu}_j(0)\}$, $\{\Sigma_j(0)\}$, q)

% Initialize default weights to begin
 $w_j^i(0) = 1/q$

% Iterate a predetermined number of times ...

for $k = 1 \dots k_{\max}$ **do**

% Compute overall weights of element j at iteration k

$$h_j(k) = \sum_{i=1}^N w_j^i(k-1)$$

% Update the weighted sample mean and weighted sample covariance for each element j

$$\underline{\mu}_j(k) = \frac{\sum_{i=1}^N w_j^i(k-1) \underline{x}^i}{h_j(k)}$$

$$\Sigma_j(k) = \frac{\sum_{i=1}^N w_j^i(k-1) (\underline{x}^i - \underline{\mu}_j(k)) (\underline{x}^i - \underline{\mu}_j(k))^T}{h_j(k)}$$

% Update the weight w associating data point \underline{x}^i with element j

$$w_j^i(k) = \frac{h_j(k) \mathcal{N}(\underline{x}^i; \underline{\mu}_j(k), \Sigma_j(k))}{\sum_{\tau=1}^q h_\tau(k) \mathcal{N}(\underline{x}_i; \underline{\mu}_\tau(k), \Sigma_\tau(k))}$$

end for

return Means = $\{\underline{\mu}_j(k_{\max})\}$, Covariances = $\{\Sigma_j(k_{\max})\}$, Weights = $\{h_j(k_{\max})\}$

Algorithm 7.1. To learn a mixture model we need to learn the means, covariances and weights characterizing the q individual distribution elements.

$$\hat{\mu}_{\text{MAP}} = \underbrace{\frac{1}{1 + \frac{\sigma_{\text{Prior}}^2}{\frac{1}{N} \sigma_{\text{Meas}}^2}} \mu_{\text{Prior}} + \frac{1}{1 + \frac{\frac{1}{N} \sigma_{\text{Meas}}^2}{\sigma_{\text{Prior}}^2}} \hat{\mu}_{\text{ML}}}_{\begin{array}{l} \text{Relative certainty} \\ \text{of prior} \end{array}} \quad (7.49)$$

For a fixed measurement error variance, we do recover the expected limiting behaviour for very strong and very weak priors:

$$\lim_{\sigma_{\text{Prior}}^2 \rightarrow 0} \hat{\mu}_{\text{MAP}} = \mu_{\text{Prior}} \quad \lim_{\sigma_{\text{Prior}}^2 \rightarrow \infty} \hat{\mu}_{\text{MAP}} = \hat{\mu}_{\text{ML}} \quad (7.50)$$

These behaviours are sketched in Figure 7.4. The left panel illustrates a fairly strong prior (small σ_{Prior}^2) and relatively weak measurements, such that the inclusion of twenty measurements has made almost no difference in the distribution for μ . In contrast the right panel illustrates a weak prior (large σ_{Prior}^2) and relatively accurate measurements, such that the value of μ is essen-

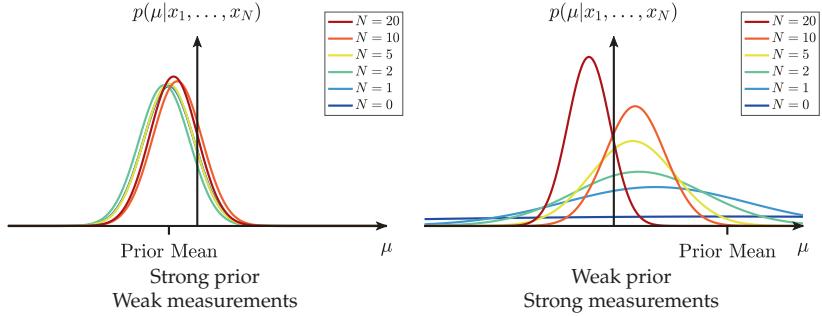


Fig. 7.4. MAP LEARNING: MAP refers to *Maximum a Posteriori*, whereby the random quantity μ is estimated from its *posterior* distribution. In contrast to Figure 7.2, here both the prior knowledge and the measured sample data influence the estimated result. The resulting uncertainty is a function of the uncertainty of the prior model, and the quantity and quality of the measurements. The apparently constant dark blue line (right) is actually Gaussian, but very wide, reflecting the significant uncertainty of the prior.

tially completely unknown based on the prior, but which is rapidly localized (narrower $p(\mu)$) as measurements are accumulated.

Just to ensure that the tradeoff in (7.50) is clear, for a given prior variance we also have two extremes based on the number of measurements:

$$\lim_{N \rightarrow 0} \hat{\mu}_{\text{MAP}} = \mu_{\text{Prior}} \quad \lim_{N \rightarrow \infty} \hat{\mu}_{\text{MAP}} = \hat{\mu}_{\text{ML}} \quad (7.51)$$

7.3 Nonparametric Model Learning

In many pattern recognition problems the parametric approaches of Sections 7.1 and 7.2 will not be helpful if we do not know a model $p(\{\underline{x}_i\} | \underline{\theta})$ (from (7.4)) somehow describing the shape of a class in feature space \mathcal{X} as a function some parameter vector $\underline{\theta}$. The absence of such a model may be due to the highly unusual or irregular shape possessed by a class, a shape which we might not know how to meaningfully parameterize, or because the feature space \mathcal{X} is high-dimensional and we are unable to visualize the class shape and assert a model, or because the nature of the pattern recognition problem does not allow us to assume any particular shape or distribution for the class.

As was already described in Section 4.2, in contrast to the parametric methods of Sections 7.1 and 7.2, we can choose to describe a class *non-parametrically*, *implicitly* in terms of a collection of data points, with no assumption of a model.

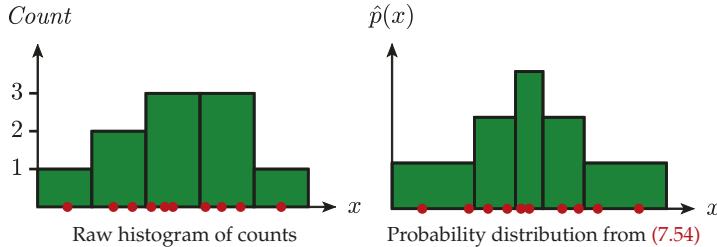


Fig. 7.5. HISTOGRAM ESTIMATION: A classic histogram (left) is found by dividing an axis into equal-sized bins and counting the number of samples (red points) which fall into each bin. To actually estimate a distribution (right) we need to normalize with respect to bin size and the total number of samples, as in (7.54). Note that there is no requirement for all of the bins to have the same size.

Really, there is not a sharp distinction between parametric and non-parametric methods. The reader might think that there must *have* to be a sharp distinction, since we either have a model $p(\underline{x}|\theta)$ or we do not. However parametric models could have any number of degrees of freedom, and the mixture models of (7.31) essentially represent a continuum between parametric and non-parametric, as we saw in [Example 7.1](#), since a mixture model could have very few elements (small q , essentially parametric) or very many elements (large q , essentially data-driven, non-parametric).

We will assume that our N samples are labelled and restrict our attention to a single class; with multiple classes the methods discussed in this section would just be applied multiple times, once per class. To simplify the discussion all of the estimators will focus on the one-dimensional case, however extensions to higher dimensions are straightforward.

7.3.1 Histogram Estimation

A histogram is just a counting of events in regions, with the number of events plotted as the heights of a sequence of bars, as illustrated in [Figure 7.5](#). Such a histogram *does*, in fact, represent the non-parametric estimation of an unknown probability distribution function $p(x)$. A histogram models the unknown distribution as piecewise constant, but otherwise makes no assumptions regarding the unknown shape of a class.

Consider some interval $R = [a, b] \subset \mathbb{R}$; if $p(x)$ is constant over this region, then

$$\underbrace{\mathbf{P}(x \in R)}_{\substack{x \text{ the random} \\ \text{variable}}} = \underbrace{\int_a^b p(x) dx}_{\substack{x \text{ a dummy variable} \\ \text{of integration}}} = p(x) \cdot (b - a) \quad \underbrace{\text{for any } x \in R}_{\substack{x \text{ as a particular value}}} \quad (7.52)$$

The reader hopefully observes the challenge experienced with the notation, in that x is being used in three different ways: as a random variable, as a dummy variable, and as an argument to a function. To distinguish between these roles, let us rewrite (7.52) as

$$\mathbf{P}(x \in R) = \int_a^b p(s) ds = p(\bar{x}) \cdot (b - a) \quad \text{for any } \bar{x} \in R \quad (7.53)$$

which can then be rearranged as

$$\text{For } \bar{x} \in R, \quad p(\bar{x}) = \frac{\mathbf{P}(x \in R)}{|R|} \quad \text{where } |R| = \text{Size of region } R \quad (7.54)$$

Suppose we have a dataset \mathcal{D} of N data points x_1, \dots, x_N sampled from distribution $p(x)$. If M of these points are observed to lie in region R , then from (7.30), the maximum likelihood estimation of a probability, we know that

$$\hat{\mathbf{P}}_{\text{ML}}(x \in R) = \frac{M}{N} \quad (7.55)$$

Combining this with (7.54), we see that the maximum likelihood estimate of the PDF is given by

$$\hat{p}(\bar{x}) = \frac{M}{N \cdot |R|} \quad \text{for } \bar{x} \in R \quad (7.56)$$

where $|R|$ is the size of region R . If we now have a set of non-overlapping regions $\{R_j\}$, and count the number of samples M_j in each bin, then the histogram estimate of the distribution is found as

$$\hat{p}_{\text{Hist}}(\bar{x}) = \frac{M_j}{|R_j| \cdot N} \quad \text{for } \bar{x} \in R_j, \quad (7.57)$$

as plotted in Figure 7.5.

It is important to remember that this derivation was premised on the fact that $p(x)$ is constant within each region, which will generally not be true of the actual underlying distribution; therefore (7.57) is necessarily an approximation. Obviously the histogram method possesses a tradeoff: for fine feature resolution (horizontal axis) we want small-sized regions, whereas for fine resolution in probability (vertical axis) we need large M_j , which requires large-sized regions. The effects of this tradeoff are illustrated in Figure 7.6. As can clearly be seen from the figure, even for a relatively simple, uni-modal distribution like a Gaussian, a large number (more than 1000) of data points is needed to obtain a reasonable non-parametric estimation.

If we were to randomly sample D datasets $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_D$ from the same distribution $p(x)$, then we would generate D different histograms, the differences

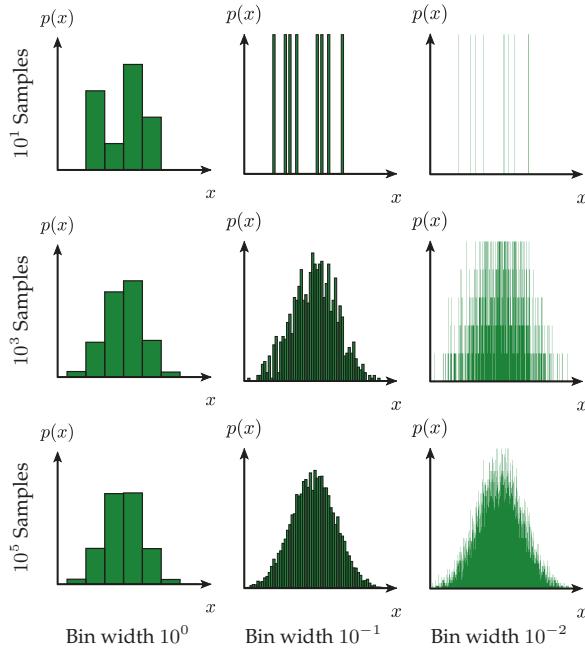


Fig. 7.6. HISTOGRAM ESTIMATION: A histogram (7.57) is found by dividing an axis into bins, and then counting the number of samples which fall into each bin. All nine panels illustrate histograms of the *same* distribution, only the bin size (left to right) and number of samples (top to bottom) vary. A narrower bin width most certainly does not necessarily lead to a superior histogram, as can be seen in the top two rows.

between them due to random sampling. That is, each histogram bar is subject to a distribution, and we can, in principle, ask about the mean and standard deviation of each bar. Since the maximum likelihood probability estimate of (7.30) is unbiased (from (D.57)), therefore

$$\hat{\mathbb{E}}[p_{\text{Hist}}(\bar{x})] = \mathbb{E}\left[\frac{M_j}{|R_j| \cdot N}\right] = \frac{\mathbf{P}(x \in R)}{|R_j|} = \frac{\int_{R_j} p(s) ds}{|R_j|}, \quad (7.58)$$

that is, in each region R_j the histogram estimate \hat{p} is unbiased with respect to the average value of the true distribution over R_j .

Next, the standard-deviation of each histogram bar can be derived from (D.66) and be approximated as

$$\sigma_p \simeq \frac{\sqrt{M_j}}{N|R_j|} \quad \frac{\sigma_p}{\hat{p}} \simeq \frac{1}{\sqrt{M_j}} \quad (7.59)$$

for the absolute and fractional errors, respectively. Three estimated distributions, with error bars, are shown in Figure 7.7.

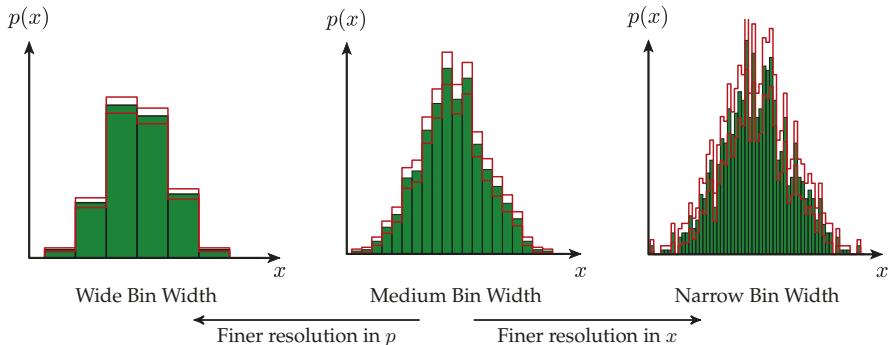


Fig. 7.7. HISTOGRAM UNCERTAINTY: All three distributions were estimated from the same data set, but with different choices of histogram bin size, which affects a tradeoff between resolution in p (larger bin size) and resolution in x (finer bin size). The figure plots the estimated distribution (green) plus/minus one standard-deviation (red), based on (7.59).

There are a few significant limitations in using histograms for estimating probability densities:

1. The estimated distribution is *always* discontinuous, which may be undesirable.
2. A shifting (translation) of the regions changes the estimated distribution, meaning that the estimated distribution is not shift-invariant with respect to the data.
3. Histograms do not generalize well to multiple dimensions. The generalization is simple enough, conceptually, however if we discretize each of the n feature dimensions in \underline{x} into q bins, then the total number of bins in the histogram is q^n , which becomes impractically large for n greater than two or three.

Kernel-based methods, discussed next, resolve these problems and so are generally preferred.

7.3.2 Kernel-Based Estimation

The most obvious flaw in histogram-based estimation was to assume that the distribution $p(x)$ was constant over each region. In fact, the whole idea of partitioning the x -axis into separated regions, and then counting the number of samples which fall into these predefined regions, seems rather rigid, since there are no “natural” or inherent region boundaries associated with the given data.

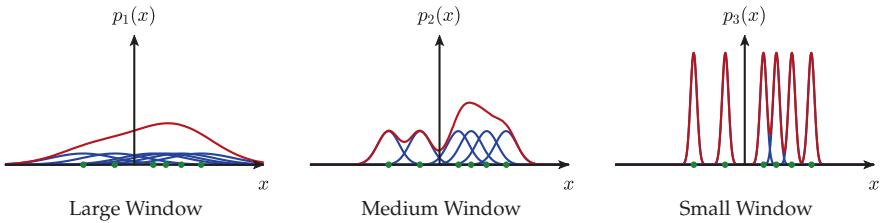


Fig. 7.8. KERNEL ESTIMATION: The kernel estimation of a distribution places a function or kernel (blue) centered at each data point (green), and the resulting weighted-shifted kernels are summed to yield the overall distribution (red). Too wide a kernel (left) smears the distribution; too narrow a kernel (right) overfits the data, resulting in an estimated distribution having a spike at each data point.

Kernel-based estimation takes a different approach, such that every data point x_i implies a degree of heightened probability density in the vicinity of x_i . In other words, having observed x_i , the probability density near to x_i cannot be near to zero. Furthermore, if we see *multiple* samples in a given area, then we would similarly expect the probability density to be correspondingly larger. The kernel⁷ function $\phi()$ describes the shape and extent, in feature space, over which the probability density is raised.

As a result, the kernel-based estimation of the probability density is the sum of the contributions from each of the N data points,

$$\hat{p}_\phi(x) = \frac{1}{N} \sum_{i=1}^N \phi(x - x_i). \quad (7.61)$$

Kernel ϕ controls how each given sample point sample x_i influences the estimated density \hat{p} , a process which is illustrated in Figure 7.8. Since we are computing a probability density, the kernel must be normalized:

$$1 = \int_{-\infty}^{\infty} \hat{p}(x) dx = \int_{-\infty}^{\infty} \frac{1}{N} \sum_{i=1}^N \phi(x - x_i) dx = \int_{-\infty}^{\infty} \phi(x) dx \quad (7.62)$$

Three commonly-used kernels are shown in Figure 7.9; in principle, any smooth, local, non-negative, unimodal function can be chosen.

⁷ The notion of a kernel $\phi()$ is essentially a location-invariant spatial function, a function that has the same shape and form as it is moved from one data point to the next. For readers familiar with signal processing, the kernel plays the role of an impulse response, such that (7.61) can be written as

$$\hat{p}_\phi(x) = \frac{1}{N} \phi(x) * \left(\sum_i \delta(x - x_i) \right), \quad (7.60)$$

the convolution of the kernel ϕ with a train of impulses, one impulse at each data point.

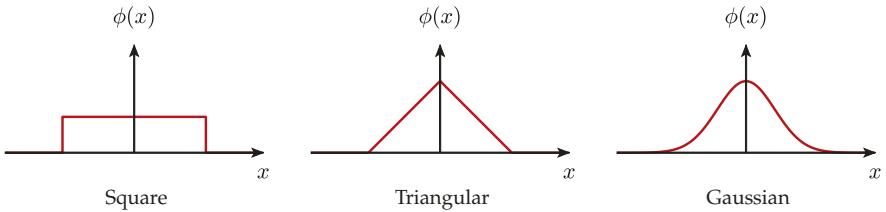


Fig. 7.9. CHOICES OF KERNEL: In principle any normalized non-negative function can be used as the kernel ϕ . However a square window gives rise to discontinuities, and a Gaussian window has infinite extent, so typically a triangular or truncated Gaussian is preferred, being both continuous and local.

Given a selected kernel shape, we may wish to stretch or compress the function, analogous to choosing larger or smaller region sizes $|R_j|$ in a histogram, so (7.61) generalizes to

$$\hat{p}_{\phi,s}(x) = \frac{1}{sN} \sum_{i=1}^N \phi\left(\frac{x - x_i}{s}\right) \quad (7.63)$$

where scaling parameter s controls the degree of expansion of ϕ . If our data points x lie in a multidimensional space, the kernel estimation is essentially unchanged, except that ϕ needs to be a kernel in n dimensions,⁸ and by expanding ϕ by a factor of s in each of n dimensions, normalization requires a division by s^n :

$$\hat{p}_{\phi,s}(\underline{x}) = \frac{1}{s^n N} \sum_{i=1}^N \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right) \quad (7.64)$$

The estimate of (7.64) possesses four important properties:

1. If the window function ϕ is continuous, then so is \hat{p} .
2. The estimated distribution is origin-independent. That is, if all of the sample points are shifted to the right, then the estimated PDF shifts correspondingly, which was *not* true of the histogram method.
3. We can obtain elegant analytical insight into the behaviour of \hat{p} . Taking the expected value of the estimate at some value \bar{x} ,

$$\begin{aligned} \mathbb{E}[\hat{p}_\phi(\bar{x})] &= E_{\{x_i\}} \left[\frac{1}{N} \sum_{i=1}^N \phi(\bar{x} - x_i) \right] = E_x [\phi(\bar{x} - x)] = \int p(x) \phi(\bar{x} - x) dx \\ &= p(\bar{x}) * \phi(\bar{x}) \end{aligned} \quad (7.65)$$

⁸ The scaling s could be different in each dimension, a set of scaling values $\{s_j\}$, similar to the scaled Euclidean distance of (6.6).

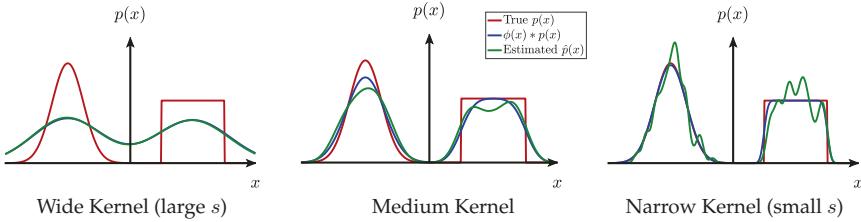


Fig. 7.10. KERNEL CONVOLUTION: The learned distribution \hat{p} (green) will, on average (i.e., in expectation) equal the convolution (blue) of the true distribution (red) with the kernel ϕ , as derived in (7.65). A wide kernel (left) causes smearing via convolution; a narrow kernel (right) acts like a delta function and so causes no smearing (blue and red curves nearly equal), however the variance in \hat{p} increases (as derived in (7.69)) and \hat{p} begins to overfit the data.

So the estimated distribution is, on average, the convolution of the true distribution with the kernel, as illustrated in Figure 7.10. The convolution was derived in (7.65) for one dimension, but remains true in the multidimensional case.

4. In principle, both the histogram and kernel estimators, of (7.54) and (7.63) respectively, are trying to estimate the same distribution $p(\underline{x})$. By setting them equal

$$\hat{p}(\underline{x}) = \frac{M(\underline{x})}{|R(\underline{x})| \cdot N} = \frac{1}{s^n N} \sum_{i=1}^N \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right) \quad (7.66)$$

we can associate the number of points, M , as a weighted sum

$$M(\underline{x}) \simeq \sum_{i=1}^N \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right) / \phi(0) \quad (7.67)$$

and therefore the effective kernel “bin size” as

$$R(\underline{x}) \simeq s^n / \phi(0) \quad (7.68)$$

The reason for drawing the analogy with the histogram case is that this allows us to use (7.59) to infer the error statistics of the kernel estimator:

$$\sigma_p \simeq \frac{\sqrt{M_j}}{N|R_j|} \simeq \frac{\left\{ \phi(0) \cdot \sum_{i=1}^N \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right) \right\}^{1/2}}{N \cdot s^n} \quad (7.69)$$

The kernel approach therefore still possesses a tradeoff, illustrated in Figure 7.10, analogous to that of the histogram approach from Figure 7.6:

Method	What is asserted?	What is measured?
Histogram	Region Size	Number of Samples
Kernel	Kernel Size	Weighted Number of Samples
Neighbourhood	Number of Samples	Region Size

Table 7.1. Both histogram and kernel methods essentially decide on *extent* (region size, kernel size) and then count the number of samples. However there is no reason why this process cannot be the other way around: neighbourhood approaches assert the number of samples, and then determine how large a region is required.

- Too wide a kernel (large s) leads to a growing convolutional error, from (7.65).
- Too narrow a kernel (small s) leads to too few sample points within the region of support of the kernel, leading to growing sampling error, from (7.69).

It is important to distinguish kernel methods from the Gaussian mixture models of Section 7.2 / Example 7.1. Both can underfit the data (too wide a kernel/too few mixture elements) or overfit the data (too narrow a kernel/too many mixture elements), however fundamentally kernels and mixtures are very different approaches:

1. A mixture model is *learned*; that is, the parameter vector $\underline{\theta}$ needs to be estimated from the given data, and depending on the learning strategy it is possible that repeated runs of mixture learning will learn *different* $\underline{\theta}$, and consequently different distributions. In contrast, the non-parametric kernel-based estimation of a distribution follows deterministically and unambiguously from the given data points based on (7.64).
2. Mixture models really are parametric, meaning that if we assume the form of the distribution to be a mixture of two Gaussians, then the estimated PDF really will be *exactly* of the form

$$\hat{p}(\underline{x}) = \alpha \mathcal{N}(\underline{x}; \underline{\mu}_1, \Sigma_1) + (1 - \alpha) \mathcal{N}(\underline{x}; \underline{\mu}_2, \Sigma_2). \quad (7.70)$$

A kernel-based estimator may, with an appropriate choice of and size of kernel, result in an estimated distribution \hat{p} that *looks* similar to a mixture of two Gaussians, however the estimated distribution based on N data points will *always* be the superposition of N kernels.

7.3.3 Neighbourhood-based Estimation

Both the histogram (Section 7.3.1) and kernel (Section 7.3.2) strategies for probability distribution estimation assume an extent, in the form of a histogram bin width or kernel size, and then count the number of samples. That is, in

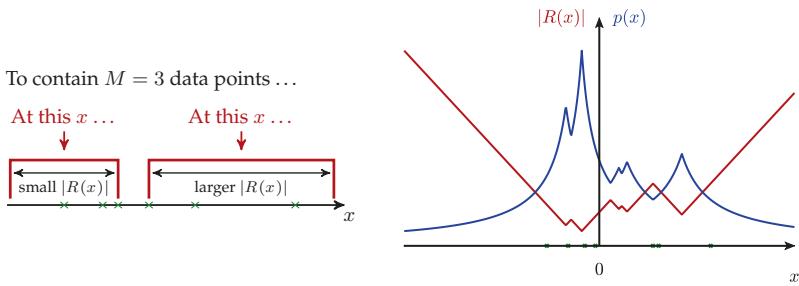


Fig. 7.11. KNN ESTIMATION: The k NN approach seeks to estimate an unknown distribution $p(x)$ by determining, for each value of x , how large a region $R(x)$ would be required to enclose a specified number M of points (left). As a result, from (7.73), where sample points are closely clustered $|R|$ is small and p is high, whereas where sample points are far apart then $|R|$ is large and p is small, here illustrated for $M = 3$ (right).

$$\hat{p}(x) = \frac{M(x)}{|R(x)| \cdot N}, \quad (7.71)$$

for the histogram approach we asserted $R(x)$ and then inferred/counted $M(x)$, however there is no reason why this process cannot be the other way around, as illustrated in Table 7.1, to assert $M(x)$ and then infer $R(x)$. Indeed, since the fractional error from (7.59),

$$\frac{\sigma_p}{\hat{p}} \simeq \frac{1}{\sqrt{M}}, \quad (7.72)$$

is a function *only* of M , the number of samples, there is actually something significantly more natural in specifying M rather than region size $|R|$. The k NN (k Nearest Neighbour) method, clearly building on the distance-based classifier in Section 6.4 of the same name, takes precisely this approach, such that in contrast to (7.71) it is now the number of neighbours (k or M) which is specified. For any point x a region is grown until M points are enclosed, such that it is $R(x)$ which varies with x :

$$\hat{p}_{k\text{NN}}(x) = \frac{M}{|R(x)| \cdot N} \quad (7.73)$$

The method is sketched in one dimension in Figure 7.11.

The k NN estimation is attractive in that only a single parameter M needs to be specified, no kernel shape/size or histogram bins, and the value M is directly associated with the fractional error of (7.72) and therefore there is some degree of rationale in selecting an appropriate M .

On the other hand, the estimated distribution \hat{p} has two serious shortcomings:

1. The estimated \hat{p} is not normalizable, because (from [Appendix D](#)) the tails of the distribution asymptotically decay as $1/x$, meaning that

$$\int_{-\infty}^{\infty} \hat{p}(x) dx = \infty \quad (7.74)$$

2. The estimated \hat{p} is relatively ugly, a very “peaky” distribution (as can be seen in [Figure 7.11](#)), with a peak at the midpoint of every M successive points (for one-dimensional problems).

However it may not matter whether a probability distribution is ugly or is normalizable, depending on our purpose in estimating \hat{p} . If our goal is to plot the estimated distribution for visualization purposes, then almost certainly a kernel approach is preferred. On the other hand, if we need to estimate distributions for the purpose of classification,

$$\hat{p}(x | C_1) \begin{array}{c} \text{Classify } x \text{ as class } C_1 \\ \gtrless \\ \text{Classify } x \text{ as class } C_2 \end{array} \hat{p}(x | C_2), \quad (7.75)$$

as we will see in [Chapter 8](#), then it is only the relative values of $\hat{p}(x | C_1)$ and $\hat{p}(x | C_2)$ that matter, and there is technically no requirement for the distributions to be normalized or smooth. As a result, although the k NN approach is of limited value in producing distributions for visualization, it offers attractive properties, as we saw in [Section 6.4](#), and to which we will return in [Chapter 8](#).

7.4 Distribution Assessment

This chapter has looked at parametric and non-parametric techniques for estimating unknown parameters and distributions from data. How do we assess whether the estimates are any good?

In the case of parametric estimates, given an estimator $\hat{\theta}(\{\underline{x}_i\})$ we can calculate its bias and error covariance:

$$\begin{aligned} \text{Error Bias} \quad b(\hat{\theta}) &= \mathbb{E}[\hat{\theta} - \theta] & \text{Error Covariance} \quad \Sigma_{\hat{\theta}} &= \mathbb{E}[(\hat{\theta} - \theta)(\hat{\theta} - \theta)^T] \end{aligned} \quad (7.76)$$

Even aside from whether the error bias and covariance in (7.76) can be determined, there remains the question of how to interpret the results: how close does $\hat{\underline{\theta}}$ need to be to the true (unknown or random) value of $\underline{\theta}$ in order to be “close”? Clearly not all of the values in $\underline{\theta}$ will contribute to our sense of error in the same way; for example, in estimating the parameters for a one-dimensional Gaussian distribution, an error in mean is not the same as an error in variance.

In principle, it would be convenient to have some sort of distance function $d(\underline{\theta}, \hat{\underline{\theta}})$ as in Section 4.1, however this automatically limits our discussion to parametric estimates, and we may wish to assess the non-parametric estimates just discussed. With further thought, keep in mind that it is really a probability distribution $p(\underline{x}|\underline{\theta})$ which we are estimating, and the estimated parameters $\underline{\theta}$ are mostly a way of describing that distribution. Since our goal is a distribution, as far as a distance metric is concerned, perhaps what we really want is a comparison of distributions,

$$d(p(\underline{x}|\underline{\theta}_1), p(\underline{x}|\underline{\theta}_2)) \quad d(\hat{p}(\underline{x}|\underline{\theta}), p(\underline{x})) \quad (7.77)$$

say in comparing two classes (left), or in comparing an estimated distribution to that of ground truth (right). Indeed, we encounter exactly the same problem in the non-parametric case, where we have only the estimated distribution, with no parameter $\underline{\theta}$:

$$d(\hat{p}(\underline{x}|\{\underline{x}_{1,i}\}), \hat{p}(\underline{x}|\{\underline{x}_{2,i}\})) \quad d(\hat{p}(\underline{x}|\{\underline{x}_i\}), p(\underline{x})) \quad (7.78)$$

So we need some sense of what it means for distributions to be “close” . . . and this is where we run into problems, because there is not one distance measure which is best or optimal. The absence of a uniquely optimal distance measure in comparing distributions is closely analogous to the absence of a single distance for comparing feature vectors in Chapter 4. Indeed, there are *many* (dozens of) distance metrics which have been defined, a few of which are summarized in Table 7.2, where the distance may be formulated in terms of the probability distributions p_1, p_2 or their associated cumulative distributions F_1, F_2

However even with a distance metric in hand, we still have the challenge of interpretation: how small does $d(p_1, p_2)$ need to be in order for the distributions p_1, p_2 to be “close”?

Even more fundamental, what two distributions are we actually comparing? That is, where does $p(x)$ in (7.78) come from? After all, the $p(x)$ in (7.78) was, itself, probably estimated at some point. What we would really like, in both the parametric and non-parametric settings, is to assess the consistency between a given data set $\{\underline{x}_i\}$ and the estimated distribution $\hat{p}(\underline{x})$. That is, we would need a distance metric

$$d(\{\underline{x}_i\}, \hat{p}(\underline{x})) \quad (7.79)$$

$$\begin{aligned}
d(p_1, p_2) &= \int |p_1(x) - p_2(x)| dx && \text{Generalization of Manhattan distance} \\
d(p_1, p_2) &= \left\{ \int (p_1(x) - p_2(x))^2 dx \right\}^{1/2} && \text{Generalization of Euclidean distance} \\
d(p_1, p_2) &= -\log \int \{p_1(x) \cdot p_2(x)\}^{1/2} dx && \text{Bhattacharyya Distance} \\
d(p_1, p_2) &= \int p_1(x) \log(p_1(x)/p_2(x)) dx && \text{Kullback-Leibler Divergence} \\
d(F_1, F_2) &= \left\{ \int |F_1(x) - F_2(x)| dx \right\}^{1/2} && \text{Cramer von Mises} \\
d(F_1, F_2) &= \max_x |F_1(x) - F_2(x)| && \text{Komolgorov-Smirnov}
\end{aligned}$$

Table 7.2. DISTRIBUTION DISTANCE METRICS: A few common ways of comparing the “distance” or dissimilarity between two distributions p_1, p_2 or cumulative distributions F_1, F_2 . It is easy to confirm that each distance function equals zero when the two distributions are identical.

The most obvious step would be to find the likelihood by evaluating the probability density at the sample points:

$$l = \prod_i \hat{p}(\underline{x}_i) \quad (7.80)$$

very similar to how maximum-likelihood was set up, such as in (7.10). That is, maximizing the likelihood in (7.80) allows us to maximize the fit between a distribution \hat{p} and a set of data points $\{\underline{x}_i\}$, however it does *not* tell us how good the resulting fit is. For example, if we look back at Figure 7.8, if we numerically evaluate the likelihood of (7.80) for each of the three estimated densities, we find that

$$\prod_i p_1(x_i) = 3.2 \cdot 10^{-3} \quad \prod_i p_2(x_i) = 2.2 \cdot 10^{-2} \quad \prod_i p_3(x_i) = 5.6 \cdot 10^0 \quad (7.81)$$

That is, as the estimated density overfits the data the likelihood clearly increases, but this is not helpful as a measure of distance to the distribution, since the *true* or correct distribution will almost certainly have a *lower* likelihood than an overfit, peaky distribution like $p_3()$. As a result, very clearly we do *not* wish to maximize (7.80).

Finally, one step more general than (7.79), for the parametric contexts of Sections 7.1 and 7.2, how were we able to use a parametric approach in the first place? How did we actually know what sort of distribution would be appropriate in (7.5)? Ideally we would evaluate the “distance” between a given set of data points $\{\underline{x}_i\}$ and a whole *family* of distributions, for example

$$d(\{\underline{x}_i\}, \text{Gaussian}) \quad d(\{\underline{x}_i\}, \text{Exponential}). \quad (7.82)$$

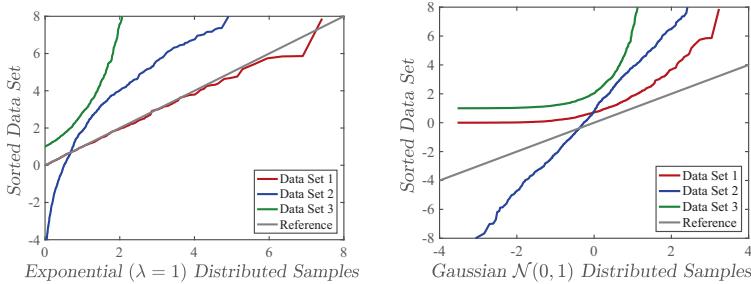


Fig. 7.12. Q–Q PLOTS: Given sample data from an unknown distributions, we can plot the sorted data against data synthesized from a hypothesized distribution, here exponential (left) and Gaussian (right). A fit along the reference $y = x$ (grey line) suggests that the data match the hypothesis, and a straight line suggests a fit to the same family as the hypothesized distribution. As a result, there is strong evidence that data set #1 (red) is exponential with $\lambda = 1$ (left), and data set #2 (blue) is Gaussian, but not zero mean/unit variance (right). The third data set (green) appears to be one-sided, like an exponential, but with significantly heavier tails.

We could define the distance in (7.82) by finding the maximum-likelihood estimate of the parameter(s), and then using (7.79) to find the distance to the resulting distribution:

$$\begin{aligned} d(\{\underline{x}_i\}, \text{Gaussian}) &\equiv d(\{\underline{x}_i\}, \mathcal{N}(\underline{x}; \hat{\mu}_{\text{ML}}, \hat{\Sigma}_{\text{ML}})) \\ d(\{\underline{x}_i\}, \text{Exponential}) &\equiv d(\{\underline{x}_i\}, \mathcal{E}(\underline{x}; \hat{\lambda}_{\text{ML}})) \end{aligned} \quad (7.83)$$

A deep look at methods to define (7.79) goes well beyond the intended scope of this text, however there are two fairly accessible places to start.

First, QQ (Quantile-Quantile) plots are an elegant, intuitive approach to visually compare one-dimensional data points to a family of distributions. Suppose we are given N data points $\{x_i\}$ and a hypothesized distribution $p(x)$. The data points are in no particular order, so it is helpful to sort the data

$$\{\bar{x}_i\} = \text{sort}(\{x_i\}) \quad \text{such that} \quad \bar{x}_1 = \min(\{x_i\}) \quad \dots \quad \bar{x}_N = \max(\{x_i\}) \quad (7.84)$$

The distribution $p()$ will predict the expected value of the order statistic⁹ \bar{x}_i , in that

$$\bar{x}_i \simeq F^{-1} \left(\frac{i + 1/2}{N} \right) \quad (7.85)$$

⁹ Somewhat outside the scope of our discussion here, an *order statistic* describes the statistics of sorted data. So, for example, if $x_i \sim \mathcal{N}(0, 1)$ for $1 \leq i \leq 10$, then x_i is just a zero-mean Gaussian random variable, but $\bar{x}_1 = \min_i(\{x_i\})$ is an order statistic. We will explore its statistics in [Problem 7.14](#).

where $F(x)$ is the cumulative distribution¹⁰ corresponding to $p(x)$. The QQ plot is then a plot of the sorted data points as a function of their predicted locations:

$$\text{plot} \left(F^{-1} \left(\frac{i + 1/2}{N} \right), \bar{x}_i \right) \quad (7.86)$$

for which several examples are shown and interpreted in [Figure 7.12](#).

A second, more quantitative approach can be found, essentially by representing the distribution as a histogram and deducing the statistical fit as a χ^2 problem. Dividing the domain into k regions R_j and counting the number M_j of sample points in each region, as in [\(7.54\)](#), we wish to compare the relative probabilities based on the sample points $\{\underline{x}_i\}$ and the hypothesized distribution $p()$:

$$\begin{aligned} \text{Sample probability of region } R_j &= \frac{M_j}{N} \\ \text{Hypothesized probability of region } R_j &= \int_{R_j} p(x) dx = P_j \end{aligned} \quad (7.87)$$

If the hypothesized distribution is correct, then we know that M_j is binomially distributed with an expected value (from [Appendix D](#)) of NP_j , therefore a plausible distance metric is something like

$$\text{Distance between } p(x) \text{ and } \{\underline{x}_i\} \approx \sum_{j=1}^k (M_j - NP_j)^2 \quad (7.88)$$

This is, in fact, very nearly the classic Pearson χ^2 test, such that

$$d(\{\underline{x}_i\}, p(x)) = \sum_{j=1}^k \frac{(M_j - NP_j)^2}{NP_j} \sim \chi^2(k-1) \quad (7.89)$$

That is, the proposed distance metric has a known χ^2 distribution with $k-1$ degrees of freedom,¹¹ leading to the test of fit

$$d(\{\underline{x}_i\}, p(x)) = \sum_{j=1}^k \frac{(M_j - NP_j)^2}{NP_j} \begin{array}{l} \text{Reject Hypothesis} \\ \gtrless \\ \text{Accept Hypothesis} \end{array} \sim \chi_\alpha^2(k-1) \quad (7.90)$$

for confidence parameter α .

¹⁰ Note that F^{-1} is the inverse function, and *not* the reciprocal. Thus if $u = F(x)$, then $x = F^{-1}(u)$.

¹¹ There are $k-1$ degrees of freedom in the k values M_1, M_2, \dots, M_k because we know that $\sum_j M_j = N$, the total number of points. In other words, in specifying the first $k-1$ values M_1, M_2, \dots, M_{k-1} you know the value of M_k , so it is only $k-1$ parameters which are free.

QQ plots and histograms do not generalize well to the high-dimensional (multi-variate) spaces which we frequently encounter in pattern recognition. The χ^2 test just discussed can be generalized to other attributes of a distribution beyond the histogram bars of (7.87), for example based on the *moments* of (B.12) and (B.13). Two further tests, applicable to high-dimensional data, are developed in some detail in Appendix D.

The preceding discussion has focused entirely on the comparison of probability distributions of a continuous random variable x . There are analogous tests for discrete-valued random variables, not particularly useful to the context of this chapter, but which will be of use to us in assessing the similarity between a target distribution and the actual output of a classifier ensemble in Section 11.4.

Case Study 7: Object Recognition

A great many pattern recognition problems are concerned with recognizing objects, whether recognizing the difference between intact and defective parts, as in Case Study 4, or different faces, as in Example 4.1, recognizing digits or the letters of the alphabet (Example 2.1), or recognizing animals at the zoo, all of which we refer to as *classification* problems, a task of deciding which class a given feature vector \underline{x} belongs to.

Although we will also see non-statistical approaches for characterizing such a classification problem in Chapters 6 and 10, from the statistical inference context discussed in this chapter, the classification problem essentially boils down to the inference of a distribution for each object class, as we saw in (7.1):

$$p(\underline{x} | C_{\text{Object } 1}) \quad p(\underline{x} | C_{\text{Object } 2}) \quad \dots \quad p(\underline{x} | C_{\text{Object } K}) \quad (7.91)$$

The difficulty of the object recognition problem depends on four factors:

- How many different classes are there (how large is K)?
- How many features are there (the dimensionality n of feature vector \underline{x})?
- How variable are samples from a given class (the intra-class variability)?
- How different are samples from different classes (the inter-class separation)?

If there are relatively few objects (small K) to distinguish, and they are clearly very different (large inter-class separation), then probably relatively few features (small n) will suffice to describe the problem, and it may be practical to approach the problem parametrically.

On the other hand, suppose we wish to recognize animals, such that we have classes

$$C_{\text{Cat}} \quad C_{\text{Dog}} \quad C_{\text{Hamster}} \quad C_{\text{Goldfish}} \quad \dots$$

If you do an internet search for “Cat” images (there are *many!*), observe the astounding variety of poses, colours, sizes, actions, and then on hugely varied backgrounds (walls, floors, carpets, couches, …). That is, the intra-class variability is very large. Furthermore there are probably some cat images where you might have difficulty knowing what animal you were looking at, possibly implying a small inter-class separation. For a given cat image y , it is very difficult to imagine what sort of feature (Chapter 5) would lead to a confident assessment of the presence of a cat, as opposed to some other animal.

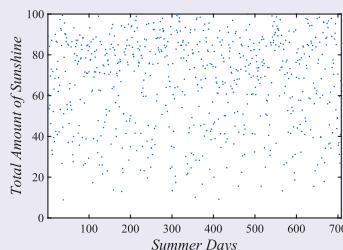
As a result, nearly all *difficult* recognition problems are approached non-parametrically, meaning that we do not need to assume or guess a parametric model, rather our dataset of cat images implicitly describes what *Cat* means. This seems exceptionally convenient: we hardly really need to learn anything, we just let the database of images speak for itself. Indeed, this convenience is true, and is precisely why non-parametric strategies are so widespread in modern pattern recognition. There is a drawback, however: just try to imagine how *many* cat images would be required to characterize nearly every cat type, shape, size, colour, pose, and action. Furthermore, to classify a given input image y we would not only need to compare it to this vast set of cat images, but equally vast sets of dog, hamster, goldfish and all other animal images. That is, there are serious issues to consider of data storage and computational complexity.

Lab 7: Parametric and Nonparametric Estimation

We can easily apply methods of parametric and nonparametric estimation to basic datasets. I have prepared a dataset consisting of total sunshine (incoming solar radiation integrated over a day) for the months of June and July over ten years. By focusing only on summer months, the variations in sunshine are primarily due to weather, and not so much due to the changing length of the day between seasons. The dataset is available for you from [the textbook data site](#):

```
% Load Sunshine data
load Sunshine
num_points = numel(sunshine);
```

The overall behaviour meets some expectations: there is an upper limit to the amount of sunshine (blue skies all day), and incoming radiation cannot be negative, so the data are bounded below at zero.



It is simplest to begin with simple parametric distributions. By computing the mean and standard deviation (formally, we are computing the Maximum Likelihood estimates), we can easily find the Gaussian parametric estimation of the distribution:

```
% Points at which to evaluate
x = (-10:110)';

% find mean and standard deviation
m = sum(sunshine)/num_points;
s = sqrt(sum((sunshine-m).^2)/num_points);
p_gauss1 = normpdf(x,m,s);
```

The distribution is vaguely plausible, but not terribly satisfying.

In particular, the Gaussian has its peak near 65, whereas from the original data the greatest concentration of points is quite clearly closer to 85. Furthermore the original distribution is quite asymmetric, which makes a single Gaussian (which is symmetric) a poor choice as the assumption for a parametric fit.

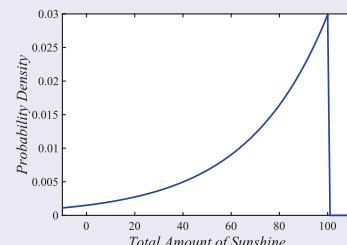
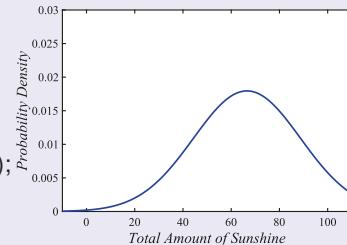
As an alternative we could consider an asymmetric parametric distribution, like a reversed exponential distribution, peaked at 100 and decaying towards zero. From Figure B.1, for a regular exponential we know that the mean of the distribution is located at $1/\lambda$, therefore for a reversed exponential distribution, starting at 100, we know that

$$100 - \text{Mean} = \frac{1}{\lambda}$$

giving us the following distribution:

```
% find lambda for backwards exponential
l = 1/(100-m);
p_exp = (x<=100).* (l*exp(-l*(100-x)));
```

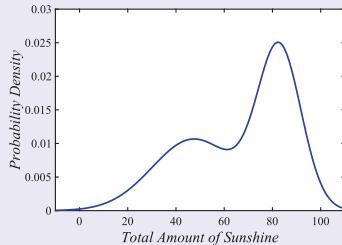
This is probably a better fit than the single Gaussian, however the strong peak right at 100 is not the best fit to the behaviour of the given data. We could, of course, continue to propose different possible parametric distributions, by guessing, however it would be better to move to Gaussian mixtures and nonparametric alternatives.



We start with a two-Gaussian mixture:

```
g2 = fitgmdist(sunshine,2);
p_gauss2 = pdf(g2,x);
```

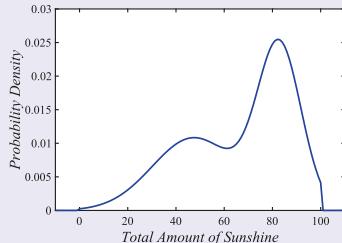
This is starting to look quite promising — the distribution is peaked consistently with the given data, and we can quite clearly see the mixture model process leading to an estimated distribution which is asymmetric and non-Gaussian.



One critique of the two-Gaussian mixture is that it fails to encode known prior knowledge, which is that our distribution is constrained to lie $0 \leq \text{sunshine} \leq 100$. Our mixture model doesn't know anything about this constraint, but we could choose to assert it, essentially creating a custom truncated two-mixture model:

```
% correct at endpoints and re-normalize
p_gauss2_new = p_gauss2;
p_gauss2b(find(x<0)) = 0;
p_gauss2b(find(x>100)) = 0;
p_gauss2b = p_gauss2b / sum(p_gauss2b);
```

Now that is beginning to look like a very plausible representation of the data which we were given.



Clearly, as a final step, we should try a fully nonparametric approach. Indeed, there is strong rationale to *always* compute a nonparametric estimate, even something as simple as a histogram, to at least visualize how the data are distributed, in the absence of any parametric assumption.

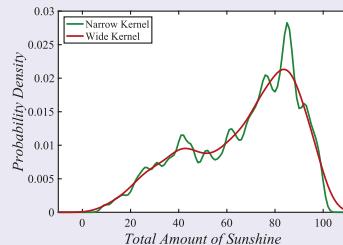
We will apply the kernel method, using two different Gaussian kernels (one narrow, one wide):

```
% get kernel-based distributions
p_wide = kernelPDF(sunshine,[1 0 100],normpdf([-15:15],0.5));
p_narrow = kernelPDF(sunshine,[1 0 100],normpdf([-15:15],0,1.5));
```

The `kernelPDF` function is not printed here, since it does not add insight to the discussion, however it is available to you at the [the textbook data site](#), as is all of the code for all of the labs.

The narrow kernel (green) is clearly overfitting, showing detailed fluctuations driven by the given data points. In contrast, the wider kernel (red) looks remarkably like our two-mixture distribution computed earlier.

Whether we now adopt a two-mixture model or a non-parametric one is something of a personal choice, and may depend what you plan on doing with the model. The two-mixture distribution will be *much* faster to evaluate, whereas the non-parametric approach would adjust easily to accommodating additional data points, if those became available.



Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links — Parametric Methods: [Estimation theory](#), [Bayesian inference](#), [Maximum likelihood estimation](#), [Maximum a posteriori estimation](#), [Mixture model](#), [Expectation-maximization \(EM\) algorithm](#)

Wikipedia Links — Non-Parametric Methods: [Q-Q plot](#), [Order statistic](#), [Kernel density estimation](#), [Multivariate kernel density estimation](#), [Chi-squared test](#), [Kolmogorov-Smirnov test](#), [Anderson-Darling test](#)

The topic of estimation theory quickly becomes quite advanced, and is normally taught at the graduate level. A few accessible places to start would include any of [3, 4, 6, 7]. In particular, my text [4] includes a discussion of Bayesian estimation and derivations of the ML and MAP estimators.

In solving Maximum Likelihood problems, it is relatively frequent that we do *not* know the relationship between the parameters θ and the given data points $\{x_i\}$, in contrast to (7.5). Such a circumstance frequently occurs when there are idealized (but unknown) underlying values $\{z_j\}$, which are key to

characterizing the problem. That is, we are told how our measured noisy data points $\{\underline{x}_i\}$ depend on the underlying values:

$$p(\{\underline{x}_i\} | \{z_j\}) \quad (7.92)$$

and we know how the parameters $\underline{\theta}$ influence the idealized, underlying problem

$$p(\{z_j\} | \underline{\theta}) \quad (7.93)$$

In many such cases the Expectation-Maximization (EM) strategy is employed, which alternates between estimating $\{z_j\}$ and $\underline{\theta}$. A nice illustration and a simple example is offered in [3].

If the reader is interested in further exploring distribution similarity beyond Table 7.2, a comprehensive survey [2] lists 63 (!!) difference metrics, really making clear how there is no objectively right or optimum criterion.

The k NN density estimator is covered very thoroughly in the advanced text [1] of Biau and Bevroye.

Sample Problems

Problem 7.1: Short Answer

Give a short definition of each of the following:

- Maximum Likelihood (ML) estimation
- Maximum a Posteriori (MAP) estimation
- Estimator bias
- Estimator error variance
- Mixture model
- Prior model

Problem 7.2: Short Answer

Offer brief answers to each of the following:

- (a) What are the strengths and weaknesses associated with parametric and non-parametric distributions?
- (b) In which contexts would you prefer parametric over non-parametric learning, and vice-versa?
- (c) Why is the use of Bayes' Rule essential in the development of Bayesian estimators, such as MAP?

- (d) For non-parametric density estimation, what are the issues to keep in mind in selecting ...
- The bin width for a histogram?
 - The choice of kernel and kernel scaling for kernel-based estimation?
 - The choice of k for k NN-based estimation?
- (e) Given the advantages of kernel-based estimation, why do we use other approaches at all? That is, relative to kernel-based approaches, what are the advantages of
- Parametric methods?
 - Histogram estimation?
 - k NN estimation?
- (f) How do we use a QQ-plot to test distribution similarity?

Problem 7.3: Conceptual — Nonparametric Learning

Suppose we have two classes C_A and C_B in a one-dimensional feature space with given data points

$$\mathcal{D}_A = \{0, 1\} \quad \mathcal{D}_B = \{3, 5\} \quad (7.94)$$

- (a) Plot the distributions which result from using kernel estimation with a rectangular window of width 3.
- (b) Plot the distributions which result from using k NN with $k = 2$.

Problem 7.4: Analytical — Parametric Learning

Suppose that we are given a set of independent scalar data points x_1, \dots, x_N obeying a shifted-exponential parametric model:

$$p(x|\alpha, \lambda) = \begin{cases} 0 & x < \alpha \\ \lambda \exp(-\lambda(x - \alpha)) & x \geq \alpha \end{cases}$$

- (a) Solve the ML parameter estimation problem. That is, given x_1, \dots, x_N derive $\hat{\alpha}_{\text{ML}}, \hat{\lambda}_{\text{ML}}$. You may find it helpful to refer to [Example 7.2](#).
- (b) Is $\hat{\alpha}_{\text{ML}}$ biased? Why or why not?

Problem 7.5: Analytical — Parametric Learning

Suppose we have an unknown value x for which we obtain two measurements x_1, x_2 such that

$$x_i = x + v_i$$

where $v_1, v_2 \sim \mathcal{N}(0, 1)$ are zero-mean unit-variance Gaussian random variables.

Let \hat{x}_{ML} be the ML estimate of x , and $\tilde{\sigma}^2$ the corresponding estimation error variance. For what correlation between v_1 and v_2 is $\tilde{\sigma}^2$ minimized?

Problem 7.6: Analytical — Parametric Learning

Suppose that we have two classes, C_1 and C_2 , and feature x . C_1 is uniform, C_2 is two-sided exponential:

$$p(x|C_1) = \begin{cases} b & -a \leq x \leq a \\ 0 & \text{Otherwise} \end{cases} \quad (7.95)$$

$$p(x|C_2) = \beta \exp(-\alpha|x|) \quad -\infty < x < \infty \quad (7.96)$$

- (a) As a function of a , what is the value of b required to normalize $p(x|C_1)$?
- (b) As a function of α , what is the value of β required to normalize $p(x|C_2)$?
- (c) Sketch $p(x|C_1), p(x|C_2)$ for $a = \alpha = 1$.
- (d) We are given N independent samples x_1, x_2, \dots, x_N :
 - (i) Suppose the samples come from class C_1 . Derive the maximum likelihood estimate of unknown parameter a .
 - (ii) Suppose the samples come from class C_2 . Derive the maximum likelihood estimate of unknown parameter α .

Problem 7.7: Analytical — Parametric Learning

- (a) Suppose that we have a random variable x for which we are given two independent samples x_1, x_2 :
- (i) Suppose x is Gaussian: $x \sim \mathcal{N}(\mu, \sigma^2)$, where we know that $\sigma = 1$; thus

$$p(x|\mu) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2}} \quad (7.97)$$

Determine $\hat{\mu}_{\text{ML}}(x_1, x_2)$, the ML estimate of μ in terms of x_1 and x_2 .

- (ii) Suppose x is Uniform: $x \sim \mathcal{U}(a, b)$, thus

$$p(x|a, b) = \begin{cases} 0 & x < a \\ \frac{1}{b-a} & a \leq x \leq (a+b) \\ 0 & x > (a+b) \end{cases} \quad (7.98)$$

Determine $\hat{a}_{\text{ML}}(x_1, x_2), \hat{b}_{\text{ML}}(x_1, x_2)$, the ML estimates of both a and b in terms of x_1 and x_2 .

- (b) Suppose we have two classes C_A and C_B in a one-dimensional feature space with given data points

$$\mathcal{D}_A = \{0, 1\} \quad \mathcal{D}_B = \{3, 5\} \quad (7.99)$$

For both of the cases in part (a), use the ML parameter estimators and sketch the resulting estimated PDFs of clusters A and B .

Problem 7.8: Analytical — Nonparametric Densities

Suppose we have a one-dimensional non-parametric estimation problem for which we are given N samples x_1, \dots, x_N :

- (a) Write down the general form for $\hat{p}_{\text{Hist}}(x)$ for the histogram method.

Show that $\int \hat{p}_{\text{Hist}}(x) dx = 1$; that is, show that the estimated PDF is normalized. Note that you do not need to make any assumptions about the number of bins or bin widths.

- (b) Write down the general form for $\hat{p}_{\phi,s}(x)$ for kernel-based estimation.

Show that $\int \hat{p}_{\phi,s}(x) dx = 1$; that is, show that the estimated PDF is normalized. As for the histogram case, above, you do not need to make any assumptions about the type of window shape or its width.

- (c) Briefly describe how one computes $\hat{p}_{\text{kNN}}(x)$ for the k NN method.

Show that $\int \hat{p}_{\text{kNN}}(x) dx > 1$; that is, show that the estimated PDF is never normalized, regardless of the values of data points x_1, \dots, x_N .

Problem 7.9: Analytical — k NN Normalization

Problem 7.8 had you demonstrate one drawback of k NN density estimation, which is that the PDFs which it produces aren't normalized. It is really *much* worse than that: prove that

$$\int \hat{p}_{\text{kNN}}(x) dx = \infty$$

for *any* PDF estimate produced by k NN, regardless of the data samples x_i .

Problem 7.10: Analytical — k NN Form

Suppose you are given N points $\underline{x}_1, \dots, \underline{x}_N$ in n -dimensional space, and that all of the points are at the origin (i.e., $\underline{x}_i = 0$).

Find the analytical form of $\hat{p}_{\text{kNN}}(\underline{x})$ as a function of \underline{x} , k , n , and N .

Problem 7.11: Comprehensive — Student Grades

Every student knows that exam grades can be rather variable — you have good days and bad days. We will suppose that a student's *intrinsic* ability is represented by an unknown parameter α , where α is measured in percent.

Let us suppose that the actual *measured* grade y , also in percent, that a student receives on an exam is Gaussian distributed around the ability α :

$$p(y | \alpha) = \frac{1}{\sqrt{2\pi}10} \exp \left\{ -\frac{(y - \alpha)^2}{2 \cdot 10^2} \right\}$$

- (a) What is the probability that a student of ability α will get a grade below 50% on an exam? Leave your answer expressed as an integral, since the Gaussian distribution cannot be integrated in closed form.
- (b) Normally we try to get a better idea of a student's ability by having them write several tests. Suppose a student writes N tests, getting grades y_1, \dots, y_N , and the grades are independent.

What is $\hat{\alpha}_{\text{ML}}(y_1, \dots, y_N)$, the Maximum Likelihood estimate of the student's ability?

- (c) What is the probability, as a function of α , that $\hat{\alpha}_{\text{ML}}(y_1, \dots, y_N)$ is below 50%? (Again, expressed as an integral, as in part (a).)
- (d) It is interesting to assess how more and more tests (larger N) affect the precision of the estimated $\hat{\alpha}$. Find the limit (showing your work) of

$$\lim_{N \rightarrow \infty} \mathbf{P}(\hat{\alpha}_{\text{ML}}(y_1, \dots, y_N) < 50)$$

first for the case $\alpha = 49$, and secondly for the case $\alpha = 51$.

Perhaps a Gaussian distribution is not such a good model for how exam grades are distributed, because it seems much more likely that a student would have a bad day, or accidentally do poorly on a question, than to accidentally get a question right. Therefore let us suppose that it is more reasonable to say that the measured grade y can never be higher than one's intrinsic ability α :

$$p(y | \alpha) = \begin{cases} 0 & y > \alpha \\ \frac{1}{10} \exp \left\{ \frac{y-\alpha}{10} \right\} & y \leq \alpha \end{cases}$$

- (e) Draw a sketch of $p(y | \alpha)$ for $\alpha = 75$.
- (f) What is the probability that a student of ability α will receive a grade below 50% on an exam?

- (g) As in part (b), suppose a student writes N tests, getting grades y_1, \dots, y_N , and that the grades are independent.

What is $\hat{\alpha}_{ML}(y_1, \dots, y_N)$, the Maximum Likelihood estimate of the student's ability?

Problem 7.12: Comprehensive — Radioactive Decay

The length of time for the radioactive decay of a given atom is exponential:

$$p(t) = \begin{cases} \lambda e^{-\lambda t} & t \geq 0 \\ 0 & t < 0 \end{cases}$$

We are interested in the estimation of λ , and in using decay rates to distinguish between different types of atomic isotopes.

- (a) The *half-life* τ of an atom is the period of time such that the probability that the atom has decayed is 50%. Determine τ as a function of λ .

Now suppose that we are given measured decay times t_1, \dots, t_N for N atoms, all of the same type (same half-life):

- (b) Derive the ML estimate $\hat{\lambda}_{ML}$.
 (c) The bias of $\hat{\lambda}_{ML}$ is difficult to compute. However, suggest why $\lim_{N \rightarrow \infty} \hat{\lambda}_{ML} = \lambda$.

Suppose that we have two classes of atoms, specifically

$$\begin{aligned} \text{Class } C_1: \text{Uranium 235} &\quad \text{Half life } 7.0 \cdot 10^8 \text{ years} \\ \text{Class } C_2: \text{Uranium 238} &\quad \text{Half life } 4.5 \cdot 10^9 \text{ years} \end{aligned}$$

We are given a sample containing $1 \cdot 10^{17}$ atoms from Class C_1 or C_2 . However we cannot measure the decay time of individual atoms, rather of *any* atom in the entire sample.

- (d) What are λ_1, λ_2 corresponding to the two classes?
 (e) Draw a sketch of the probability distributions of the two classes.

Of course, normally we wouldn't try to classify a sample based on a *single* decay, rather we would wait for a *number* of decays t_1, \dots, t_N . However this makes the problem rather complicated.

On the other hand, if we let $s_N = t_1 + \dots + t_N$, by the central limit theorem because s_N is the sum of a number of independent random variables it is approximately Gaussian, with mean $N \cdot \mathbb{E}[t]$ and variance $N \cdot \text{var}(t)$.

- (f) Draw a sketch of the class distributions $p(s_N | C_1), p(s_N | C_2)$.
 (g) What can you say about class separability as N increases?

Problem 7.13: Computational — QQ Plots

We would like to develop a few QQ plots for distribution comparison. To even set up the problem we need some basic tools to generate data. Given a uniform random value $u \sim \mathcal{U}(0, 1)$, then it can be derived¹² that

$$-\ln(u)/\lambda \sim \mathcal{E}(\lambda) = \text{Exponential Distribution with decay rate } \lambda > 0 \quad (7.100)$$

and

$$u^{-1/(\gamma-1)} \sim \mathcal{H}(\gamma) = \text{Power Law with exponent } \gamma > 1 \quad (7.101)$$

- (a) Synthesize five datasets, each with $N = 20\,000$ points, as follows:

$$\mathcal{D}_1 \sim \mathcal{E}(1) \quad \mathcal{D}_2 \sim \mathcal{E}(2) \quad \mathcal{D}_3 \sim \mathcal{H}(2) \quad \mathcal{D}_4 \sim \mathcal{E}(3) \quad \mathcal{D}_5 \sim \mathcal{N}(1, 1) \quad (7.102)$$

- (b) Develop code to produce QQ plots. Produce three plots, in each one superimposing the QQ plots of all five datasets compared to $\mathcal{D}_1, \mathcal{D}_3, \mathcal{D}_5$, respectively.
(c) Interpret your results from part (b). What do you observe in terms of distribution similarity/dissimilarity?
(d) Try making N , the number of data points, smaller. How small can N be but still allow for a convincing assessment of distribution similarity/dissimilarity?

Problem 7.14: Computational/Analytical — Order Statistics

Given N samples x_1, \dots, x_N from a distribution $p(x)$, then

\bar{x}_i^N represents an order statistic, the random variable corresponding to the i th smallest value in N samples of random variable x .

The distribution for \bar{x}_i^N will clearly depend on the original distribution $p(x)$, however it will definitely *not* be the same as $p(x)$.

We begin analytically, to think more deeply about the problem. Suppose the random values are uniformly distributed, thus $x_i \sim \mathcal{U}(0, 1)$.

- (a) Fairly challenging: Analytically derive and then sketch the probability distribution of the minimum value \bar{x}_1^N , superimposing the distributions corresponding to $N = 1, 2, 3$.

¹² The derivation of the exponential and power-law distributions from be found in Chapter 9 of [5].

From here on we will approach the problem computationally. Let $x \sim \mathcal{N}(0, 1)$ and let $N = 7$. Simulate 100 000 such samples, thus giving you 100 000 samples for each of $\bar{x}_i^1, \dots, \bar{x}_i^7$.

- (b) Find the mean and variance for each \bar{x}_i^7 . What do you observe?
- (c) Really we would like to see the distribution for each of \bar{x}_i^7 . Superimpose the seven distributions corresponding to $\bar{x}_i^1, \dots, \bar{x}_i^7$ based on
- Histogram-based nonparametric distribution estimation (bin size ≈ 0.2)
 - Kernel-based nonparametric distribution estimation with a rectangular kernel (window width 0.2)
 - Kernel-based nonparametric distribution estimation with a Gaussian kernel ($\sigma = 0.2$)

Comment both on the quality of the resulting distribution estimates, and also on what the distributions show about the behaviour of order statistics.

Problem 7.15: Real-World, Open-Ended — Sports

There are a great many statistics present in the domain of professional sports, and similarly a significant interest in applying methods of pattern recognition and machine learning. The applications are many, whether machine learning for real-time recognition of which player is which on a playing field, extracting game statistics for use by coaches to improve team performance, or in research of human biomechanics.

An internet search on machine learning or pattern recognition in sport turns up endless articles, so you would have much to work with. You may wish to select a sport of interest and see what relevant articles you can find, or perhaps the other way around, to select a pattern recognition/machine learning method of interest and then search for ways in which it has been used in sports.

Provide a short summary of the specific pattern recognition methods being used, and what underlying sports-related problem is being addressed.

References

1. G. Biau, L. Devroye, *Lectures on the Nearest Neighbor Method*. (Springer, New York, 2015)
2. S. Cha, Comprehensive survey on distance/similarity measures between probability density functions. *Int. J. Math. Models Methods Appl. Sci.* **1**(4) (2007)

3. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley Interscience, New York, 2009)
4. P. Fieguth, *Statistical Image Processing and Multidimensional Modeling* (Springer, New York, 2010)
5. P. Fieguth, *An Introduction to Complex Systems* (Springer, New York, 2021)
6. S. Kay, *Fundamentals of Statistical Processing, Volume I: Estimation Theory* (Pearson, London, 1993)
7. B. Silverman, *Density Estimation for Statistics and Data Analysis* (Chapman and Hall, New York, 1986)



Statistics-Based Classification

The distance-based classification strategies of [Chapter 6](#) are simple and often effective, but heuristic: the user needs to specify the notion of prototype and a definition of point-to-point distance to be used in assessing the distance from any point to each of the classes. The methods do not involve complicated algorithms, but do require some insight by the user on what approach to take.

The statistical inference methods of [Chapter 7](#) give us far more complete information about a given classification problem, in that an estimated probability distribution $p(\underline{x} | C_\kappa)$ specifies all details regarding the shape/behaviour of class C_κ in feature space. It stands to reason that, given the distributions for each class,

$$p(\underline{x} | C_1), \quad p(\underline{x} | C_2), \quad \dots, \quad p(\underline{x} | C_K) \quad (8.1)$$

there should be principled, systematic approaches to classification which avoid some of the heuristics of [Chapter 6](#), and it is such systematic approaches which will be developed in this chapter.

It should come as no surprise that there is, in fact, a strong parallel between the statistics-based classification of this chapter, and the statistics-based inference of [Chapter 7](#). In particular, the parametric estimation strategies of [\(7.5\)](#) and [\(7.7\)](#)

$$\hat{\underline{\theta}}_{\text{ML}} = \underbrace{\arg_{\underline{\theta}} \max p(\{\underline{x}_i\} | \underline{\theta})}_{\text{Maximum Likelihood (non-Bayesian)}} \quad \hat{\underline{\theta}}_{\text{MAP}} = \underbrace{\arg_{\underline{\theta}} \max p(\underline{\theta} | \{\underline{x}_i\})}_{\text{Maximum a Posteriori (Bayesian)}} \quad (8.2)$$

seek to estimate a vector $\underline{\theta}$ of parameters. The parallel between statistics-based classification (this chapter) and statistics-based inference or estimation ([Chapter 7](#)) is that, in both cases, we wish to use statistical methods to infer some object of interest:

ESTIMATION ([Chapter 7](#)): The object of interest is a parameter vector $\underline{\theta}$, describing a distribution;

CLASSIFICATION (this chapter): The object of interest is the class C associated with a given feature \underline{x} .

Really, the only difference between estimation and classification is that in estimation ([Chapter 7](#)), vector $\underline{\theta}$ contains one or more real values, whereas in undertaking classification, our unknown class C is a single discrete value, as explored further in [Example 8.1](#).

The choice of a non-Bayesian (ML) or Bayesian (MAP) approach depends on the existence of prior information, knowledge of the problem in advance of taking a measurement, as discussed in [Section 7.1](#). For classification, the question comes down to whether we know the probabilities of the various classes:

NON-BAYESIAN: The probability of each class is unknown, there is no prior information. The associated Maximum-Likelihood (ML) classifier is developed in [Section 8.1](#).

BAYESIAN: The relative probabilities of the K classes are given as prior information, leading to the Maximum a Posteriori (MAP) classifier of [Section 8.2](#).

The key practical difference between ML/MAP classification, here, as opposed to inference in [Chapter 7](#), is that the MAP classifier is fairly straightforward to solve, whereas the MAP estimator of [Chapter 7](#) was significantly more involved than ML.

8.1 Non-Bayesian Classification: Maximum Likelihood

Suppose we are given a distribution, such as sketched in [Figure 8.1](#), associated with each class:

$$p(\underline{x}|C_1), \quad p(\underline{x}|C_2), \quad \dots \quad p(\underline{x}|C_K) \quad (8.3)$$

From [Chapter 7](#) you might expect to see parameter vectors, something like

$$p(\underline{x}|C_\kappa) = p(\underline{x}|\underline{\theta}_\kappa) \quad (8.4)$$

such that the distribution for each class C_κ has been learned via some parameter vector $\underline{\theta}_\kappa$. There are several reasons why we would prefer *not* to explicitly refer to a parameter vector $\underline{\theta}$:

1. Writing the distribution for a class in terms of some parameter vector $\underline{\theta}$ presupposes that a parametric method was used to learn the distribution, however a *non-parametric* method could equally well have been used.

Example 8.1: Are we Classifying or Estimating?

In [Chapter 3](#) we saw a number of contexts, particularly in [Example 3.2](#), drawing parallels between the problems of regression and classification. Very similarly, in [Chapter 7](#) we discussed the estimation of a continuous value (normally a parameter or a probability), which is mathematically very similar to the statistical classification of this chapter. In particular, comparing [\(8.6\)](#) and [\(8.7\)](#):

$$\hat{\theta}_{\text{ML}} = \arg_{\theta} \max p(\{\underline{x}_i\} | \theta)$$

θ is a continuous variable

Estimation

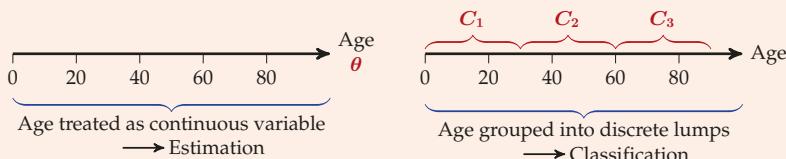
$$\hat{C}_{\text{ML}} = \arg_C \max p(\{\underline{x}_i\} | C)$$

C is a discrete variable

Classification

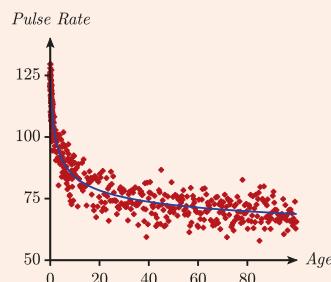
That is, in each case we seek to estimate an unknown (θ or C) from data ($\{\underline{x}_i\}$). Given the similarity between the equations, are estimation and classification actually different problems?

Not really. Both problems are set up the same way and describe the same kind of inference (here maximum likelihood, but other criteria could be used, such as MAP). One can see the fluidity between the two in a great many surveys. Consider a continuous variable, such as a person's age, which could be represented precisely (left), or lumped into broad categories (right):



Since we *know* that age is continuous, what is the incentive for lumping? Treating a continuous quantity (height, age, income) as lumped allows for a much simpler representation (multiple choice) rather than writing down explicit details (your exact height). However the key motivation is that classification is simpler than estimation: classification requires some understanding or description of \underline{x} for each of K classes, whereas estimation requires some understanding or description of \underline{x} continuously for all possible values (infinitely many) of θ .

The problem with lumping a continuous variable is that doing so is artificial, imposing categories where there are not, in fact, any inherent divisions present. Consider, for example, an approximate sampling of human pulse rate (beats per minute) as a function of age. It is very clear that there is some sort of smooth relationship between pulse rate and age, and neither quantity shows any sign of grouping or discretization.



To be sure, there *are* relatively sudden phenomena which can happen as a function of age.

Example continues ...

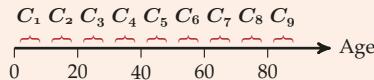
Example 8.1: Are we Classifying or Estimating? (continued)

There are transitions such as puberty or menopause, however these do not take place at any precise, predetermined age. Rather *those* should be the categories, rather than discretization of age. That is, classes such as

$$\mathcal{C} = \{\text{Pre-Puberty, Post-Puberty, Pre-Menopausal, Post-Menopausal}\}$$

would, in fact, have some rationale, since certain aspects of human health and biology do indeed clump based on such classes, however we should be quite cautious in lumping/discretizing continuous values.

We could, of course, try to mimic the continuity of a value by making the lumped discretization finer, something like



In the limit of many such classes, this essentially mimics the non-parametric behaviour of a histogram, in [Section 7.3.1](#), and the same limitations emerge: as the number of classes (K) increases we have increasing computational complexity and, more importantly, a decreasing number of training data points per class, ultimately the same issue which also limited the number of histogram bars.

In practice, although estimation and classification are mathematically very similar, the reason why we appear to treat them so separately is that the methods and algorithms for discrete-value estimation (pattern recognition) end up being quite different from those for continuous-value estimation.

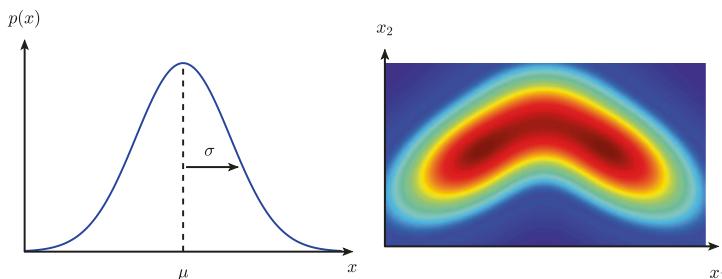


Fig. 8.1. DISTRIBUTIONS: Statistical distributions can be defined for a feature space in one (left), two (right), or higher dimensions.

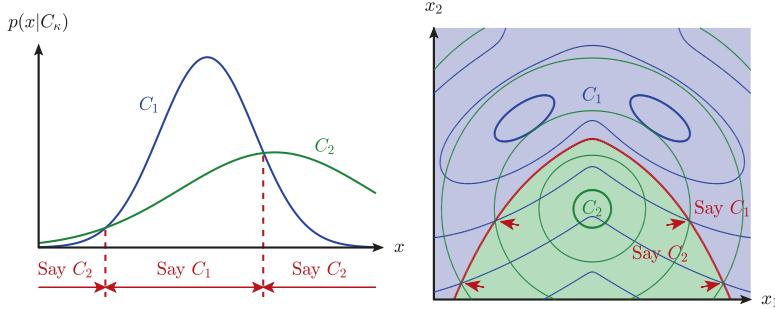


Fig. 8.2. ML CLASSIFICATION: Building on Figure 8.1, the Maximum Likelihood (ML) classifier selects the most likely class for every feature value \underline{x} . That is, for any value of \underline{x} we select that class with the largest probability density. In the two-dimensional case (right), the contours of constant probability density are plotted, and the classification boundary (red curve) can be seen (arrows) to pass through the intersections of the equi-probable contours of the two classes.

2. It is not at all obvious that any parameter vector $\underline{\theta}_\kappa$ needs to be class dependent. Perhaps only a *single* parameter vector was learned which describes all classes.
3. Perhaps learning was not performed at all. Perhaps the distributions $p(\underline{x} | C_\kappa)$ were just given to you.

For all of these reasons, it is cleaner to just refer to the class distributions

$$p(\underline{x} | C_1), \quad p(\underline{x} | C_2), \quad \dots \quad p(\underline{x} | C_K), \quad (8.5)$$

which could equally well be parametric, non-parametric, or from some other source.

Starting with (7.5) for non-Bayesian estimation,

$$\hat{\underline{\theta}}_{\text{ML}} = \arg_{\underline{\theta}} \max p(\{\underline{x}_i\} | \underline{\theta}) \quad (8.6)$$

the equivalent Maximum-Likelihood classifier, written in very much the same form, is

$$\hat{C}_{\text{ML}} = \arg_C \max p(\underline{x} | C). \quad (8.7)$$

Alternatively, we can rewrite (8.7) in a mathematically equivalent (but somewhat simpler) form as

$$\text{Classify } \underline{x} \text{ into class } \hat{C} \quad \text{if} \quad p(\underline{x} | \hat{C}) \geq p(\underline{x} | C_\kappa) \quad \text{for all } 1 \leq \kappa \leq K, \quad (8.8)$$

or, in words:

$$\text{Classify feature } \underline{x} \text{ into that class which makes } \underline{x} \text{ the most likely.} \quad (8.9)$$

This last rule summarizes the ML classifier quite nicely: at any point \underline{x} , evaluate all of the distributions $p(\underline{x}|C_\kappa)$, and classify \underline{x} to the largest distribution at that point, as sketched in Figure 8.2.

In the two-class case, (8.8) simplifies to a likelihood ratio test

$$\frac{p(\underline{x}|C_1)}{p(\underline{x}|C_2)} \stackrel{C_1}{\gtrless} 1 \quad (8.10)$$

Since many distributions include exponential terms, it is frequently convenient to formulate the classifier in terms of log-likelihoods, in which case (8.10) becomes

$$\ln p(\underline{x}|C_1) - \ln p(\underline{x}|C_2) \stackrel{C_1}{\gtrless} 0 \quad (8.11)$$

Finally, if the tests in (8.8), (8.10), (8.11) are met with equality, this implies that \underline{x} lies right on the classification boundary, as shown by the distribution intersections in Figure 8.2.

8.2 Bayesian Classification: Maximum a Posteriori

As in Section 8.1, starting with (7.7) for Bayesian estimation,

$$\hat{\theta}_{\text{MAP}} = \arg_{\theta} \max p(\theta|\{\underline{x}_i\}), \quad (8.12)$$

the equivalent Maximum-Likelihood classifier, written in very much the same form, is

$$\hat{C}_{\text{MAP}} = \arg_C \max \mathbf{P}(C|\underline{x}). \quad (8.13)$$

It is important to observe that the random variable C (the class) is, by definition, discrete, since C_1, C_2, \dots, C_K are separate, individual classes, and a given feature \underline{x} comes from a single class. Therefore we are maximizing over *probabilities* $\mathbf{P}()$ in (8.13), and not over a probability *density*, as in (8.12).

The quantity which we wish to maximize in (8.13), $\mathbf{P}(C|\underline{x})$, is not a quantity we know, but is one which we can infer. From Appendix B.3, Bayes' Rule tells us that

$$p(\underline{x}|C) \mathbf{P}(C) = \mathbf{P}(C|\underline{x}) p(\underline{x}) \quad (8.14)$$

Therefore

$$\mathbf{P}(C|\underline{x}) = \frac{p(\underline{x}|C) \mathbf{P}(C)}{p(\underline{x})} \quad (8.15)$$

Therefore, we can rewrite (8.13) as

$$\hat{C}_{\text{MAP}} = \arg_C \max \mathbf{P}(C|\underline{x}) = \arg_C \max \frac{p(\underline{x}|C) \mathbf{P}(C)}{p(\underline{x})} = \arg_C \max p(\underline{x}|C) \mathbf{P}(C), \quad (8.16)$$

where we are able to drop the $p(\underline{x})$ term in the denominator because $p(\underline{x})$ is not a function of C . That is, *every* choice of C has the *same* denominator, so the selection of C which maximizes $p(\underline{x}|C) \mathbf{P}(C)/p(\underline{x})$ is the same C which maximizes $p(\underline{x}|C) \mathbf{P}(C)$.

As for the ML case in Section 8.1, a mathematically equivalent (but somewhat simpler) form of (8.16) is

$$\begin{aligned} \text{Classify } \underline{x} \text{ into class } \hat{C} &\quad \text{if } p(\underline{x}|\hat{C}) \cdot \mathbf{P}(\hat{C}) \geq p(\underline{x}|C_\kappa) \cdot \mathbf{P}(C_\kappa) \\ &\quad \text{for all } 1 \leq \kappa \leq K, \end{aligned} \quad (8.17)$$

or, in words:

$$\text{Classify } \underline{x} \text{ into the most likely class.} \quad (8.18)$$

This last rule is deceptively simple and seems obvious. It is worth looking more closely at the parallel rule in (8.9): ML makes the feature \underline{x} most likely, whereas MAP makes the class most likely. Comparing (8.8) and (8.17), we have

$$\underbrace{\text{Maximize } p(\underline{x}|C)}_{\text{ML Rule}} \quad \underbrace{\text{Maximize } p(\underline{x}|C) \cdot \mathbf{P}(C)}_{\text{MAP Rule}} \quad (8.19)$$

That is, the two approaches are essentially the same, based on the class shape $p(\underline{x}|C)$, except that the MAP rule further rescales the class shape based on the class prior probability $\mathbf{P}(C)$, the likelihood that a given object or pattern comes from class C , as illustrated in Figure 8.3.

It is important to observe in the figure how

$$\mathbf{P}(C|\underline{x}) \neq p(\underline{x}|C) \cdot \mathbf{P}(C) \quad (8.20)$$

Either side of (8.20) can be maximized to give rise to the same MAP classifier, however that does *not* make the two sides equal, as we know from (8.15). The missing $p(\underline{x})$ term does not affect the MAP classifier, however it *does* most certainly make $\mathbf{P}(C|\underline{x})$ very different from $p(\underline{x}|C) \cdot \mathbf{P}(C)$.

As with the ML classifier, in the two-class case the MAP classifier of (8.17) simplifies to a likelihood ratio test

$$\frac{p(\underline{x}|C_1)}{p(\underline{x}|C_2)} \stackrel{C_1}{\gtrless} \frac{\mathbf{P}(C_2)}{\mathbf{P}(C_1)} \quad (8.21)$$

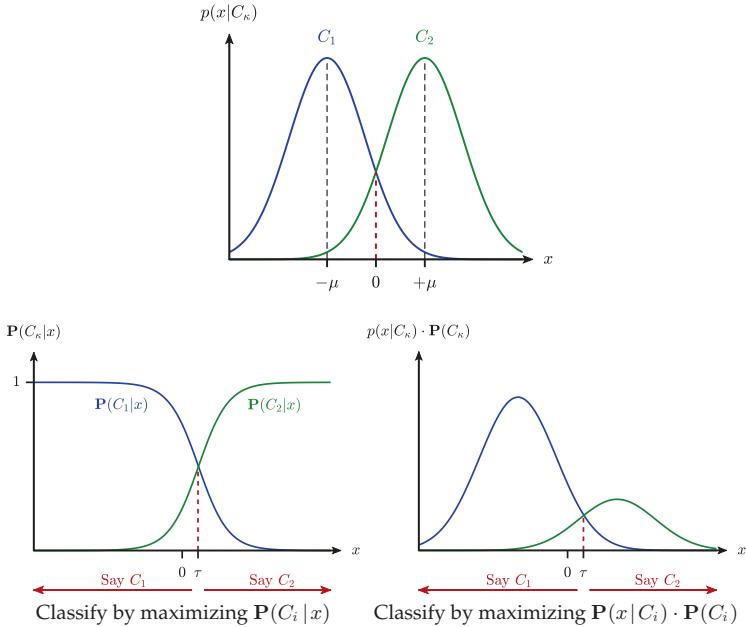


Fig. 8.3. MAP CLASSIFICATION: Building on Figure 8.2, the Maximum a Posteriori (MAP) classifier is subtly different from ML. Rather than selecting the class which makes x most likely (ML), here for each value of x MAP selects the most likely class, based on the posterior likelihood $P(C_i | x)$. By Bayes' Rule, selecting the largest $P(C_i | x)$ (bottom left) is equivalent to selecting the largest $p(x | C_i) \cdot P(C_i)$ (bottom right).

from which it is clear that, in general, if all of the classes are equally likely, $\mathbf{P}(C_\kappa) = 1/K$ for K classes, then the ML and MAP classifiers are identical.

The MAP classifier selects the most likely class, which seems like the obvious best choice, so when would we ever *not* want to use the MAP classifier? There are then a few plausible scenarios in which we might not wish (or be able) to use MAP:

1. We do not know the class distribution $p(\underline{x} | C_\kappa)$, in which case neither ML nor MAP could be used. Most pattern recognition problems do either give the class statistics or give class data, from which the class statistics could be inferred (as in Chapter 7). It is then a choice, whether it is worth inferring the statistics, or whether a non-statistical classifier (such as in Chapter 6) should be used.
2. We do not know the prior class probabilities $\mathbf{P}(C_\kappa)$, which are not required by ML, but are essential in order to apply MAP. Class probabilities can be inferred from data, as we saw from (7.30) in Section 7.2. However in doing so you are making an assumption, which is that the number of data

points given for each class is indicative or reflective of the class likelihood, which may very well *not* be the case. For example, a curated dataset may choose to acquire an equal number of training samples from each class, even though the classes are quite different in likelihood (such as in [Case Study 8](#)).

3. Even if we *do* know both the class distribution (shape) $p(\underline{x} | C_\kappa)$ and prior probability $\mathbf{P}(C_\kappa)$, we may *still* not choose to use MAP. From [\(8.13\)](#), the MAP classifier is choosing the class to maximize the posterior probability, which may not be what we wish our classifier to do, as will be discussed further in [Section 8.5](#).

8.3 Statistical Classification for Normal Distributions

Since normal distributions are a relatively common parametric form for inference (as in [Chapter 7](#)), it is helpful to obtain a deeper understanding in this context. Suppose that each class is distributed as multi-variate normal:

$$p(\underline{x} | C_\kappa) = \mathcal{N}(\underline{x}; \underline{\mu}_\kappa, \Sigma_\kappa) = (2\pi)^{-n/2} \det(\Sigma_\kappa)^{-1/2} e^{-\frac{1}{2}(\underline{x} - \underline{\mu}_\kappa)^T \Sigma_\kappa^{-1} (\underline{x} - \underline{\mu}_\kappa)} \quad (8.22)$$

To keep the analysis as clear as possible, we will focus on the two-class problem, as sketched in [Figure 8.2](#) or [Figure 8.3](#). From [Sections 8.1](#) and [8.2](#), we know that the two-class ML and MAP classifiers can be written in a common form as

$$\frac{p(\underline{x} | C_1)}{p(\underline{x} | C_2)} \stackrel{C_1}{\gtrless} \xi \quad \text{where} \quad \xi_{\text{ML}} = 1 \quad \xi_{\text{MAP}} = \frac{\mathbf{P}(C_2)}{\mathbf{P}(C_1)} \quad (8.23)$$

Substituting in the normal distribution, we have

$$\frac{(2\pi)^{-n/2} \det(\Sigma_1)^{-1/2} \exp(-\frac{1}{2}(\underline{x} - \underline{\mu}_1)^T \Sigma_1^{-1} (\underline{x} - \underline{\mu}_1))}{(2\pi)^{-n/2} \det(\Sigma_2)^{-1/2} \exp(-\frac{1}{2}(\underline{x} - \underline{\mu}_2)^T \Sigma_2^{-1} (\underline{x} - \underline{\mu}_2))} \stackrel{C_1}{\gtrless} \xi \quad (8.24)$$

Cancelling the (2π) terms, moving the determinants, and taking the logarithm of both sides brings us to

$$-\frac{1}{2}(\underline{x} - \underline{\mu}_1)^T \Sigma_1^{-1} (\underline{x} - \underline{\mu}_1) + \frac{1}{2}(\underline{x} - \underline{\mu}_2)^T \Sigma_2^{-1} (\underline{x} - \underline{\mu}_2) \stackrel{C_1}{\gtrless} \ln \left\{ \xi \frac{\det(\Sigma_1)^{1/2}}{\det(\Sigma_2)^{1/2}} \right\} \quad (8.25)$$

It would be nice to not have a leading negative sign, so we can negate both sides, but being careful to reverse the inequalities, so

$$(\underline{x} - \mu_1)^T \Sigma_1^{-1} (\underline{x} - \mu_1) - (\underline{x} - \mu_2)^T \Sigma_2^{-1} (\underline{x} - \mu_2) \stackrel{C_2}{\underset{C_1}{\gtrless}} -2 \ln \left\{ \xi \frac{\det(\Sigma_1)^{1/2}}{\det(\Sigma_2)^{1/2}} \right\} = \eta \quad (8.26)$$

Do keep in mind that the right-hand side, η , can be written in a variety of equivalent ways:

$$\eta = -2 \ln \left\{ \xi \frac{\det(\Sigma_1)^{1/2}}{\det(\Sigma_2)^{1/2}} \right\} = 2 \ln \left\{ \frac{\det(\Sigma_2)^{1/2}}{\xi \det(\Sigma_1)^{1/2}} \right\} = \ln \left\{ \frac{\det(\Sigma_2)}{\xi^2 \det(\Sigma_1)} \right\} \quad (8.27)$$

Let us now examine (8.26) in a few special cases of increasing complexity.

Case I: ML Classifier with Identity Class Covariances

The covariances both equal the identity matrix, thus $\Sigma_1 = \Sigma_2 = I$. Since $\xi_{ML} = 1$ and the covariances are equal, therefore $\eta = 0$ in (8.26). If we multiply out the left-hand side, the quadratic $\underline{x}^T I^{-1} \underline{x}$ terms cancel, leaving us with a linear classifier:

$$-2\underline{x}^T \mu_1 + \mu_1^T \mu_1 + 2\underline{x}^T \mu_2 - \mu_2^T \mu_2 \stackrel{C_2}{\underset{C_1}{\gtrless}} 0 \quad (8.28)$$

Although not obvious,¹ factoring (8.28) gives us

$$-2 \underbrace{\left(\underline{x} - \frac{\mu_1 + \mu_2}{2} \right)^T}_{\text{Offset between } \underline{x} \text{ and midpoint of means}} \underbrace{(\mu_1 - \mu_2)}_{\text{Direction between means}} \stackrel{C_2}{\underset{C_1}{\gtrless}} 0 \quad (8.29)$$

That is, the resulting classifier chooses between C_1 and C_2 on the basis of the sign of the dot product² of the position of \underline{x} relative to the midpoint between the means and the direction between the means. That is, the classification boundary is the right-bisecting plane or hyperplane between the class means, the same as the Mean–Euclidean distance-based classifier (MED), as we saw in Figure 6.16.

Case II: ML Classifier with Equal Class Covariances

Since the class covariances are equal, $\Sigma_1 = \Sigma_2 = \Sigma$, the threshold η of (8.26) still equals zero, as in Case I. The quadratic terms $\underline{x}^T \Sigma^{-1} \underline{x}$ still cancel, so

¹ It may be hard to see how arrive at (8.29) from (8.28), however it is relatively easy to multiply out the terms in (8.29) and validate that it equals (8.28).

² Any vector product $q^T r$ is identical to the dot product $q \bullet r$.

that the classifier is linear (meaning that the classification boundary must be a line/plane/hyperplane), however the behaviour will be clearer if the quadratic terms are left in. Thus we can rewrite (8.26) as

$$(\underline{x} - \mu_1)^T \Sigma^{-1} (\underline{x} - \mu_1) \stackrel{C_2}{\gtrless} (\underline{x} - \mu_2)^T \Sigma^{-1} (\underline{x} - \mu_2) \quad (8.30)$$

which the reader hopefully recognizes as being exactly the Mean–Mahalanobis distance (MMD) of (6.25) in [Chapter 6](#).

Case III: ML and MAP Classifiers with Different Class Covariances

From (8.26), given Gaussian class shapes, the ML and MAP classifiers have the form

$$\underbrace{(\underline{x} - \mu_1)^T \Sigma_1^{-1} (\underline{x} - \mu_1)}_{\text{Mahalanobis distance from } \underline{x} \text{ to } C_1} - \underbrace{(\underline{x} - \mu_2)^T \Sigma_2^{-1} (\underline{x} - \mu_2)}_{\text{Mahalanobis distance from } \underline{x} \text{ to } C_2} \stackrel{C_2}{\gtrless} \frac{\eta}{C_1} \quad (8.31)$$

That is, in the Gaussian case the Mahalanobis, ML, and MAP classifiers are all very similar, differing only in their respective thresholds η . That is, the ML and MAP classifiers are essentially the same as Mahalanobis, just with biases towards one class or the other, based on class covariances and prior probabilities:

$$\begin{aligned} \underbrace{\eta_{\text{Mahal.}} = 0}_{\text{No bias}} & \quad \eta_{\text{ML}} = \underbrace{2 \ln \left\{ \frac{\det(\Sigma_2)^{1/2}}{\det(\Sigma_1)^{1/2}} \right\}}_{\text{Bias toward smaller class}} \\ \underbrace{\eta_{\text{MAP}} = 2 \ln \left\{ \frac{\det(\Sigma_2)^{1/2}}{\det(\Sigma_1)^{1/2}} \right\} + 2 \ln \left\{ \frac{\mathbf{P}(C_1)}{\mathbf{P}(C_2)} \right\}}_{\text{Bias toward smaller class}} & \quad \underbrace{\text{Bias toward more likely class}}_{\text{Bias toward more likely class}} \end{aligned} \quad (8.32)$$

The biases should be clear:

- Both ML and MAP are biased towards smaller classes, since a more compact (smaller standard-deviation) Gaussian will have a higher amplitude, as we can see in [Figure 8.2](#). Therefore at a given number of standard-deviations from the class mean, the more compact distribution will have a higher likelihood, and therefore be chosen.
- MAP is explicitly biased towards more likely classes, since its classification is based on the posterior distribution $\mathbf{P}(C | \underline{x})$, which is also sensitive to class probability, and not just to the shape of the class distribution.

Since the class covariances are not equal, $\Sigma_1 \neq \Sigma_2$, the quadratic terms will *not* cancel, so the two-class classification boundary will, in general, be an ellipsoid or hyperboloid, as was the case for the Mean–Mahalanobis classifier in Chapter 6.

The astute reader will observe that it is possible for the class covariances to differ, $\Sigma_1 \neq \Sigma_2$, and yet their determinants to be equal, $\det(\Sigma_1) = \det(\Sigma_2)$. Referring back to Figure 4.8, observe that

$$\text{Hyper-Volume of Ellipsoid} \propto \prod_{i=1}^n \sqrt{\lambda_i}, \quad (8.33)$$

whereas, from (A.28),

$$\text{Determinant of Covariance } \det(\Sigma) = \prod_{i=1}^n \lambda_i. \quad (8.34)$$

Therefore the size or extent of a class, measured as the hyper-volume of its ellipsoid, is given by $\sqrt{\det(\Sigma)}$, the square-root of the determinant. So the ML and MAP bias in η is actually based on the relative hyper-volumes of the two classes, biased towards the class with the smaller hyper-volume.

In conclusion, if the classes are Gaussian, equally likely, and equal in volume (equal covariance determinant), then all three of the Mahalanobis, ML, and MAP classifiers are the same.

8.4 Classification Error

Given the more complete statistical representation of the classification problem present in this chapter, we are in a better position to quantify the error associated with the ML and MAP classifiers. Recall that probability of error $\mathbf{P}(e)$ was already seen in (3.14), and similarly the class-pairwise error $\mathbf{P}_g(C_j | C_k)$, defined in (3.16) as

$$\mathbf{P}_g(C_j | C_k) = \text{Probability that classifier } g \text{ says } C_j \text{ given a data point from } C_k \quad (8.35)$$

which led to the confusion matrix of (3.17).

Taking the expectation over all \underline{x} , we have

$$\mathbf{P}(e) = \mathbb{E}_{\underline{x}} [\mathbf{P}(e | \underline{x})] = \int \mathbf{P}(e | \underline{x}) p(\underline{x}) d\underline{x} \quad (8.36)$$

Let $\mathcal{R}_\kappa \subset \mathcal{X}$ denote the region of feature space \mathcal{X} in which a given classifier $g()$ selects class C_κ . That is,

$$\mathcal{R}_\kappa = \{\underline{x} \text{ such that } g(\underline{x}) = C_\kappa\} \quad (8.37)$$

Note that the regions $\{\mathcal{R}_\kappa\}$ cover all of \mathcal{X} , therefore (8.36) becomes

$$\mathbf{P}(e) = \int \mathbf{P}(e | \underline{x}) p(\underline{x}) d\underline{x} = \sum_{\kappa=1}^K \int_{\mathcal{R}_\kappa} \mathbf{P}(e | \underline{x}) p(\underline{x}) d\underline{x} \quad (8.38)$$

For any point in \mathcal{R}_κ the classifier says C_κ , therefore the classifier has made an error if the true class was *any* class *other* than C_κ :

$$\mathbf{P}(e) = \sum_{\kappa=1}^K \int_{\mathcal{R}_\kappa} \sum_{j \neq \kappa} \mathbf{P}(C_j | \underline{x}) p(\underline{x}) d\underline{x} \quad (8.39)$$

Minimizing the overall probability of error in (8.39) means minimizing the integrand $\sum_{j \neq \kappa} \mathbf{P}(C_j | \underline{x})$, which seems complicated. However note that

$$\sum_j \mathbf{P}(C_j | \underline{x}) = 1, \quad (8.40)$$

since we are summing over probabilities, therefore

$$\underbrace{\sum_{j \neq \kappa} \mathbf{P}(C_j | \underline{x})}_{\text{Choosing } \kappa \text{ to minimize this ...}} = 1 - \underbrace{\mathbf{P}(C_\kappa | \underline{x})}_{\dots \text{ is the same as choosing } \kappa \text{ to maximize this.}} \quad (8.41)$$

Therefore to minimize the probability of error in (8.39), for each value of \underline{x} we wish to have the classifier choose κ to *maximize* $\mathbf{P}(C_\kappa | \underline{x})$. Since it is the regions $\{\mathcal{R}_\kappa\}$ which control the classifier behaviour, for each value of \underline{x} if κ *maximizes* $\mathbf{P}(C_\kappa | \underline{x})$, then we need to ensure that \underline{x} lies in \mathcal{R}_κ . Maximizing $\mathbf{P}(C_\kappa | \underline{x})$ is, of course, MAP, meaning that MAP is the unique classifier which minimizes the overall probability of error.

Remarkably, asymptotically the k NN classifier of Section 6.4 has the same property. From the derivation in Appendix D, starting with the k NN distribution estimation in Section 7.3.3,

$$\hat{p}_{\text{kNN}}(x | C_\kappa) = \frac{M}{|R_\kappa(x)| \cdot N_\kappa} \quad (8.42)$$

we find that the region size relates to the posterior probability as

$$\frac{1}{|R_\kappa(x)|} \propto \mathbf{P}(C_\kappa | x), \quad (8.43)$$

meaning that choosing κ to minimize $|R_\kappa|$, that is choosing the class κ whose k th point is closest (i.e., k NN) is equivalent to maximizing $\mathbf{P}(C_\kappa | \underline{x})$ which is, of course, the MAP rule. That is, asymptotically³ k NN converges to MAP and minimizes the overall probability of error. Furthermore the derivation also shows that nearest neighbour (i.e., $k = 1$) will asymptotically have an error no worse than twice that of MAP.

The expression for error in (8.39) may seem a little complex at first glance. Applying Bayes' rule converts the equation into terms which we are more likely to know:

$$\mathbf{P}(e) = \sum_{\kappa=1}^K \int_{\mathcal{R}_\kappa} \sum_{j \neq \kappa} p(\underline{x} | C_j) \mathbf{P}(C_j) d\underline{x} \quad (8.44)$$

for which a simple example is illustrated in Example 8.2. The expression further simplifies in the two-class case:

$$\mathbf{P}(e) = \underbrace{\int_{\mathcal{R}_1} p(\underline{x} | C_2) \mathbf{P}(C_2) d\underline{x}}_{\text{Classifier says } C_1, \text{ so integrate over the likelihood that } \underline{x} \text{ came from } C_2} + \underbrace{\int_{\mathcal{R}_2} p(\underline{x} | C_1) \mathbf{P}(C_1) d\underline{x}}_{\text{Classifier says } C_2, \text{ so integrate over the likelihood that } \underline{x} \text{ came from } C_1} \quad (8.45)$$

which is illustrated in Figures 8.4 and 8.5.

It is important to understand that although MAP is guaranteed to minimize the probability of classification error, (8.44) can be used to evaluate the probability of error for *any* classifier, whether MAP, ML, or something else. However in order to compute $\mathbf{P}(e)$ in (8.44) we do need to know the statistics of the class shapes $p(\underline{x} | C_\kappa)$ and the class prior probabilities $\mathbf{P}(C_\kappa)$. That is, we can compute the probability of error for other classifiers, however in order to do so we need the same information which we would have needed to infer the optimal MAP classifier.

As is shown in Figure 8.6, $\mathbf{P}(e)$ can be computed for any classifier, for any number of classes, and for any class distribution shape. A few remarks to keep in mind with regards to classification error:

- For the MAP classifier, if any class is guaranteed, meaning that there is some class C_κ such that $\mathbf{P}(C_\kappa) = 1$, then the probability of error $\mathbf{P}_{\text{MAP}}(e) = 0$ (as shown in the bottom half of Figure 8.5), since MAP will choose the guaranteed class.
- In general, as shown in Figure 8.5, the probability of error can be a function of the class prior probabilities $\mathbf{P}(C_\kappa)$, but that does not need to be the

³ For large k , large N , and small k/N .

Example 8.2: Classifier Probability of Error

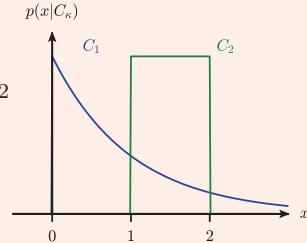
Let us consider a simple example to explore the assessment of classifier error. Suppose we have two classes, one exponential and one uniform:

$$p(x|C_1) = \begin{cases} 0 & x < 0 \\ e^{-x} & x \geq 0 \end{cases} \quad p(x|C_2) = \begin{cases} 0 & x < 1 \\ 1 & 1 \leq x \leq 2 \\ 0 & x > 2 \end{cases}$$

The ML classifier is very easy to identify, by inspection, from the figure:

Say $C_1 \quad x < 1$ or $x > 2$

Say $C_2 \quad 1 \leq x \leq 2$



If a point x came from C_2 it is guaranteed to be classified as C_2 , thus $\mathbf{P}_{\text{ML}}(e|C_2) = 0$, so errors will appear only for those samples from C_1 lying $1 \leq x \leq 2$:

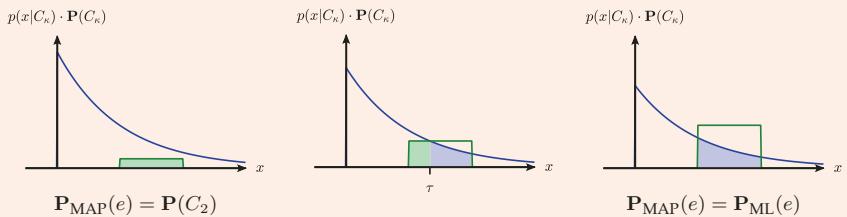
$$\mathbf{P}_{\text{ML}}(e|C_1) = \int_1^2 p(x|C_1) dx$$

The ML probability of error therefore evaluates to

$$\mathbf{P}_{\text{ML}}(e) = 0 + \mathbf{P}(C_1) \cdot \int_1^2 e^{-x} dx = \mathbf{P}(C_1)(e^{-1} - e^{-2})$$

The MAP classifier is more complicated, since the behaviour of the classifier (i.e., the classification boundaries) can be a function of the class prior probability. For compactness, let $\mathbf{P}(C_1) = \mathbf{P}_1$, $\mathbf{P}(C_2) = 1 - \mathbf{P}_1$; then the MAP classifier can be divided into three categories of behaviour:

If $(1 - \mathbf{P}_1) \leq \mathbf{P}_1 \cdot e^{-2}$ If $\mathbf{P}_1 \cdot e^{-2} \leq (1 - \mathbf{P}_1) \leq \mathbf{P}_1 e^{-1}$ If $(1 - \mathbf{P}_1) \geq \mathbf{P}_1 e^{-1}$



The first and third cases have a straightforward classifier error; it is the middle case which needs some attention. We can find the classification boundary τ as

$$\mathbf{P}_1 \cdot e^{-\tau} = (1 - \mathbf{P}_1) \rightarrow \tau = \ln \frac{\mathbf{P}_1}{1 - \mathbf{P}_1}$$

So for the middle case, the MAP probability of error evaluates to

$$\mathbf{P}_{\text{MAP}}(e) = (1 - \mathbf{P}_1) \int_1^\tau 1 dx + \mathbf{P}_1 \int_\tau^2 e^{-x} dx = (1 - \mathbf{P}_1)(\tau - 1) + \mathbf{P}_1(e^{-1} - e^{-\tau})$$

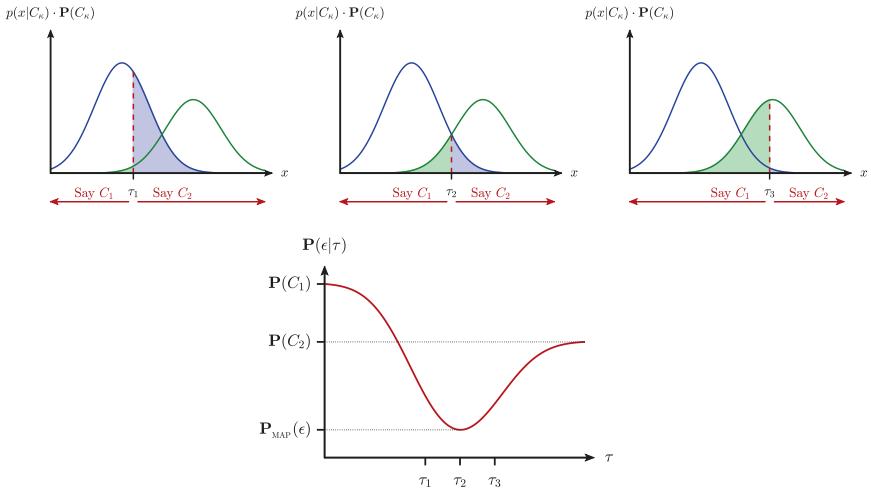


Fig. 8.4. CLASSIFICATION ERROR: Each value of the classification threshold τ (top) implies a classifier having some probability of error, given by the shaded areas. The integral of the shaded areas yields the probability of error $P(e)$ (bottom). The MAP classifier, here corresponding to classification boundary τ_2 , always has the lowest $P(e)$ of any classifier. In this problem, for very negative τ the MAP classifier always says C_2 , thus the probability of error is simply $P(C_1)$; similarly for very large τ the probability of error must converge to $P(C_2)$.

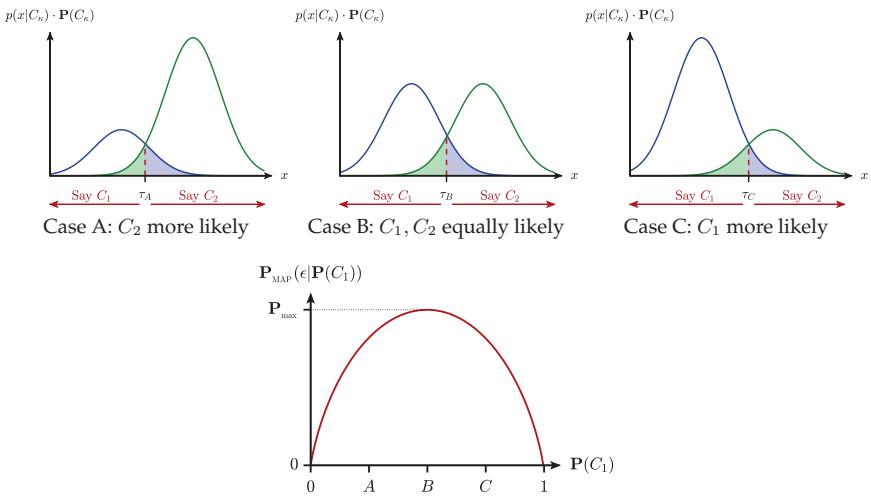


Fig. 8.5. MAP CLASSIFICATION ERROR: The MAP probability of error will clearly be a function of the class prior probabilities, for which three cases are shown (top). The resulting $P_{MAP}(e)$ is plotted (bottom); at the endpoints one of the classes is guaranteed ($P(C_i) = 1$), therefore $P_{MAP}(e)$ must go to zero.

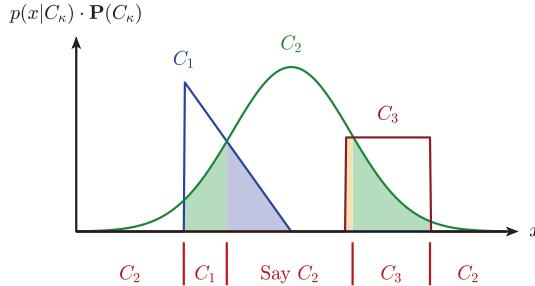


Fig. 8.6. CLASSIFICATION ERROR: The approach to calculating $\mathbf{P}(e)$ is the same, regardless of the type of classifier, the number of classes, or the shape of the class distributions. In all cases the error is the integral over those feature values where a class is *not* chosen.

case: the probability of error *can* be constant, unchanging as the prior probabilities change, as can be seen in [Problem 8.3](#).

- Although the prior probabilities $\mathbf{P}(C_\kappa)$ are needed in order to compute the error, the classifier itself may (if Bayesian) or may not (if non-Bayesian) be a function of $\mathbf{P}(C_\kappa)$. So the classification boundary τ in [Figure 8.5](#) for MAP moves as $\mathbf{P}(C_\kappa)$ changes, whereas the ML classification boundary would be at a fixed location.
- The Normal distribution cannot be integrated in closed form, meaning that there is no equation or analytical solution to the integral of the shaded tails in [Figure 8.5](#). Clearly we know that the entire Gaussian is normalized,

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\left(\frac{(x-\mu)^2}{2\sigma^2}\right)} dx = 1 \quad (8.46)$$

however this equation does not help us to integrate just a *part* of the Gaussian. There are two standard functions, $erf()$ and $Q()$, which have been defined for this purpose:

$$erf(\alpha) = \frac{2}{\sqrt{\pi}} \int_0^\alpha e^{-x^2} dx = \int_{-\alpha}^\alpha \mathcal{N}(x; 0, 1/\sqrt{2}) dx \quad (8.47)$$

which is the integral between $-\alpha$ and $+\alpha$ for a zero-mean Gaussian having a standard deviation of $1/\sqrt{2}$, and

$$Q(\alpha) = \frac{1}{\sqrt{2\pi}} \int_\alpha^\infty e^{-\frac{x^2}{2}} dx = \int_\alpha^\infty \mathcal{N}(x; 0, 1) dx = \frac{1}{2} (1 - erf(\alpha/\sqrt{2})) \quad (8.48)$$

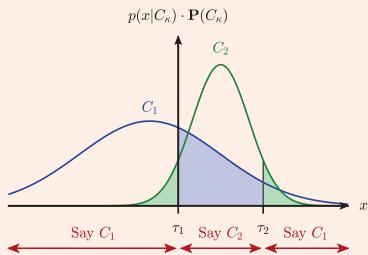
the integral of the tail of a Gaussian distribution, the integral from α out to infinity for a zero-mean unit-variance Gaussian.

Example 8.3: Classifier Probability of Error for Gaussian Statistics

We would like to evaluate the probability of error for a simple example involving Gaussian distributions. Consider the two-class problem as shown, where

$$C_1 \sim \mathcal{N}(\mu_1, \sigma_1^2) \quad C_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$$

There are classification boundaries at τ_1 and τ_2 . Very clearly this is *not* the MAP classifier. However we have emphasized that the error can be computed for *any* classifier, not just ML or MAP, so the sense that τ_1 and τ_2 are sub-optimally chosen is not a concern in this example, since our focus is on understanding how to represent tail-integrals of Gaussian distributions.



From (8.45) and from the illustration, it should be clear that

$$\mathbf{P}(e) = \underbrace{\int_{-\infty}^{\tau_1} p(x|C_2)\mathbf{P}(C_2) dx}_{\text{Left-tail integral}} + \underbrace{\int_{\tau_1}^{\tau_2} p(x|C_1)\mathbf{P}(C_1) dx}_{\text{Interval integral}} + \underbrace{\int_{\tau_2}^{\infty} p(x|C_2)\mathbf{P}(C_2) dx}_{\text{Right-tail integral}} \quad (8.49)$$

Because Gaussian tails are not closed-form integrable, function $Q()$ in (8.48) was defined as a right-tail integral, so the first two terms in (8.49) need a little extra work. In all three cases we undertake the change of variable as shown in (8.55), and we need to keep in mind that $Q()$ is an integral over distribution $p(x|C_\kappa)$, and so the probability $\mathbf{P}(C_\kappa)$ needs to be factored out:

$$\underbrace{\int_{-\infty}^{\tau_1} p(x|C_2)\mathbf{P}(C_2) dx}_{\text{Left tail}} = \mathbf{P}(C_2) \left\{ 1 - \underbrace{\int_{-\infty}^{\tau_1} p(x|C_2) dx}_{\text{Right tail}} \right\} \\ = \mathbf{P}(C_2) \left\{ 1 - Q \left(\frac{\tau_1 - \mu_2}{\sigma_2} \right) \right\} \quad (8.50)$$

$$\underbrace{\int_{\tau_1}^{\tau_2} p(x|C_1)\mathbf{P}(C_1) dx}_{\text{Interval}} = \mathbf{P}(C_1) \left\{ \underbrace{\int_{\tau_1}^{\infty} p(x|C_1) dx}_{\text{Right tail}} - \underbrace{\int_{\tau_2}^{\infty} p(x|C_1) dx}_{\text{Right tail}} \right\} \\ = \mathbf{P}(C_1) \left\{ Q \left(\frac{\tau_1 - \mu_1}{\sigma_1} \right) - Q \left(\frac{\tau_2 - \mu_1}{\sigma_1} \right) \right\} \quad (8.52)$$

$$\underbrace{\int_{\tau_2}^{\infty} p(x|C_2)\mathbf{P}(C_2) dx}_{\text{Right tail}} = \mathbf{P}(C_2) Q \left(\frac{\tau_2 - \mu_2}{\sigma_2} \right) \quad (8.53)$$

Example continues ...

Example 8.3: Classifier Probability of Error for Gaussian Statistics (continued)

Pulling the terms together, the resulting expression for the classification probability of error is

$$\begin{aligned}\mathbf{P}(e) &= \mathbf{P}(C_2) \left\{ 1 + Q\left(\frac{\tau_2 - \mu_2}{\sigma_2}\right) - Q\left(\frac{\tau_1 - \mu_2}{\sigma_2}\right) \right\} \\ &\quad + \mathbf{P}(C_1) \left\{ Q\left(\frac{\tau_1 - \mu_1}{\sigma_1}\right) - Q\left(\frac{\tau_2 - \mu_1}{\sigma_1}\right) \right\}\end{aligned}$$

Between these two functions, it is Q which is much more convenient, being the tail integral of a unit-variance Gaussian. Typically the distribution being integrated is *not* zero-mean and unit-variance, so a simple change of variable is needed to normalize a given problem. Suppose we are given the tail integral

$$\int_{\alpha}^{\infty} \mathcal{N}(x; \mu, \sigma^2) dx = \int_{\alpha}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx. \quad (8.54)$$

If we make a change of variable $z = \frac{x-\mu}{\sigma}$, then the integral becomes normalized and can be expressed in terms of Q :

$$\int_{\frac{\alpha-\mu}{\sigma}}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{z^2}{2}} dz = Q\left(\frac{\alpha-\mu}{\sigma}\right). \quad (8.55)$$

The use of Q is illustrated in [Example 8.3](#).

8.5 Other Statistical Classifiers

In [Section 8.4](#) we saw that MAP is the unique classifier which is guaranteed to minimize the probability of classification error. We might suppose, then, that we would *always* wish to find the MAP classifier, since it is always the best (most accurate) classifier.

There are problems, however, where we do *not* want to minimize the overall error. Suppose we have a two-class problem

$$C_1 : \text{Patient is Healthy} \quad C_2 : \text{Patient has rare Disease X} \quad (8.56)$$

Because the disease is rare, the prior probabilities are rather different:

$$\mathbf{P}(C_1) = 99\% \quad \mathbf{P}(C_2) = 1\% \quad (8.57)$$

Then a classifier which completely ignores the lab test result y and declares every patient to be healthy,

$$g(y) = C_1 \quad \leftarrow \quad \text{Declare every patient to be healthy} \quad (8.58)$$

is obviously a stupid/meaningless classifier, and yet has an “accuracy” of 99%.

We need to introduce the concept of *risk* $R_{\kappa j}$, a measure of cost or badness, such that

$$R_{\kappa j} = \text{Cost or Risk of Classifying } \underline{x} \text{ as } C_\kappa \text{ when } C_j \text{ is actually true.} \quad (8.59)$$

where normally there is no cost to being correct: $R_{\kappa\kappa} = 0$. If the costs are highly asymmetric, that is

$$R_{12} \gg R_{21} \quad (8.60)$$

then MAP may very well not be the correct choice, since MAP will not be taking these costs into account in classifying. If the above discussion feels mathematically abstract or possibly irrelevant, consider the following costs:

$$\begin{aligned} R_{21} &= \text{Cost of informing a patient that they have a life-threatening illness,} \\ &\quad \text{but in fact they are healthy.} \\ R_{12} &= \text{Cost of informing a patient that they are healthy,} \\ &\quad \text{but in fact they have an undiagnosed life-threatening illness.} \end{aligned} \quad (8.61)$$

Neither of these outcomes is desirable, of course, however to consider these two scenarios as “equivalent” (as MAP would do) is seriously problematic. We can develop the optimal Bayes-Risk classifier as a generalization of the MAP classifier of (8.44). In particular, the expected⁴ risk is

$$\mathbb{E}[R] = \sum_{\kappa=1}^K \sum_j R_{\kappa j} \mathbf{P}(C_\kappa | C_j) \mathbf{P}(C_j) \quad (8.62)$$

or, written more in the style of (8.44),

$$\mathbb{E}[R] = \sum_{\kappa=1}^K \int_{\mathcal{R}_\kappa} \sum_j R_{\kappa j} p(\underline{x} | C_j) \mathbf{P}(C_j) dx \quad (8.63)$$

In contrast to (8.44), here the summation is over all possible j , to allow the possibility of a non-zero cost for correct classification R_{jj} . In the usual event that $R_{jj} = 0$ for all j , then both (8.62) and (8.63) are unchanged if the summation for j excludes κ , as before in (8.44).

⁴ “Expected” in the sense of statistical *expectation*, as discussed in Appendix B.2.

The optimal Bayes-risk classifier is that classifier minimizing $\mathbb{E}[R]$. Applying Bayes' rule to (8.63) to return to posterior form,

$$\mathbb{E}[R] = \sum_{\kappa=1}^K \int_{\mathcal{R}_\kappa} \sum_j R_{\kappa j} \mathbf{P}(C_j | \underline{x}) p(\underline{x}) dx, \quad (8.64)$$

minimizing $\mathbb{E}[R]$ means that for every value of \underline{x} we need to select κ in order to minimize

$$\sum_j R_{\kappa j} \mathbf{P}(C_j | \underline{x}), \quad (8.65)$$

essentially a risk re-weighted version of MAP.

One significant difficulty in applying the Bayes-risk approach is that it can be politically problematic or ethically challenging to assign values to the risks in (8.61), since we need to think about the relative "costs" of different kinds of errors. In such cases the very different formulation of the Neyman-Pearson criterion may be preferred:

$$\text{Find classifier } g \text{ to minimize } \mathbf{P}_g(e | C_1) \text{ such that } \mathbf{P}_g(e | C_2) \leq \alpha_2 \quad (8.66)$$

That is, to minimize the probability of incorrectly declaring the patient to be ill, with a false-negative⁵ diagnosis probability no worse than α_2 .

Or, the classifier can be constrained the other way around:

$$\text{Find classifier } g \text{ to minimize } \mathbf{P}_g(e | C_2) \text{ such that } \mathbf{P}_g(e | C_1) \leq \alpha_1 \quad (8.67)$$

minimizing the probability of incorrectly declaring the patient to be healthy, with a false-positive diagnosis probability no worse than α_1 .

The significant advance is that (8.66) and (8.67) are formulated entirely in terms of probabilities which are easily understood by most patients or doctors.

Case Study 8: Medical Assessments

Medical tests are a particularly common, widespread, and important category of pattern recognition/hypothesis testing⁶ problems. In particular, let us again consider the basic medical testing question of (8.56) from Section 8.5:

$$C_1 : \text{Patient is Healthy} \quad C_2 : \text{Patient has rare Disease X} \quad (8.68)$$

⁵ Concepts of *false-positive* and *false-negative* are defined more thoroughly in the discussion of classifier evaluation in Section 9.2.

⁶ Hypothesis testing is discussed in Case Study 4 and Section 9.2.

Because the disease is rare, the prior probabilities are rather different:

$$\mathbf{P}(C_1) = 99\% \quad \mathbf{P}(C_2) = 1\% \quad (8.69)$$

Most diseases are actually *significantly* less common than 1% across the general population, however we'll stick with the 1% number for convenience.

To complete the problem setup, in addition to defining the classes, we also need to define the pattern recognition measurement y : the lab test for Disease X, which we will assume is classified on the basis of a single threshold τ :

$$g(y) = \begin{cases} C_1 & y < \tau \leftarrow \text{Patient is declared healthy} \\ C_2 & y > \tau \leftarrow \text{Disease X is declared present} \end{cases} \quad (8.70)$$

With the problem in place, we now have several issues with which to contend:

1. What is our objective? What criterion are we trying to minimize?
2. What can we assume about our classes? Are all healthy people alike?
3. The challenge associated with highly asymmetric data.
4. The counter-intuitive behaviour of Bayes' Rule.

Let us now treat each of these in turn.

What criterion are we trying to minimize?

The question of risk versus the overall probability of classification error was just discussed in [Section 8.5](#), and from that discussion it should be clear that probability of error alone is a problematic metric for medical tests with highly asymmetric risk. We can characterize the performance of the lab test in [\(8.70\)](#) using a confusion matrix [\(3.19\)](#) of probabilities:

		Lab Test Claims ...	
		Patient Healthy	Disease X Present
Patient's True Health	Healthy	$P(\text{Test says Healthy} \text{Healthy})$	$P(\text{Test says Disease X} \text{Healthy})$
	Disease X	$P(\text{Test says Healthy} \text{Disease X})$	$P(\text{Test says Disease X} \text{Disease X})$

(8.71)

The confusion matrix is actually not so convenient for two classes. Each row in the matrix must sum to one, since each row just reflects all of the test probabilities for a given underlying state of health. So of the four values in the confusion matrix, we actually have only two degrees of freedom; although

there are a great many possible criteria,⁷ for medical lab tests it is common to characterize the test accuracy in terms of its

$$\text{Sensitivity} = \frac{\text{True Positive}}{\text{All Positive}} \quad (8.72)$$

the fraction of positive test results which are correct, and

$$\text{Specificity} = \frac{\text{True Negative}}{\text{All Negative}} \quad (8.73)$$

the fraction of all negative test results which are correct.

Because accidentally failing to detect an ill person is far worse than accidentally declaring a healthy person as ill, it is the specificity which we wish to control. That is, in the Neyman-Pearson criteria of [Section 8.5](#), it is the version in [\(8.66\)](#), setting the specificity (say, to 99%) and then maximizing the sensitivity.

What can we assume about our classes?

For the statistical classification methods of this chapter, we would expect to be given⁸ the class statistics

$$p(y|C_1) \quad p(y|C_2) \quad (8.74)$$

This is a hypothesis testing problem, as was discussed in [Case Study 4](#). In most hypothesis tests, we have a good understanding of $p(y|C_1)$, since this is the default/expected case (fair coin, fair die, correctly-shaped part, clean water), whereas it is quite unpredictable what “not normal” (unfair coin, loaded die, defective part, unclean water) looks like, and so we do not necessarily have a model for $p(y|C_2)$.

It is not at all clear, however, that medical testing behaves this way. Although it is true that all people in C_1 (“Healthy”) do not have Disease X, they may have any other number of *other* medical conditions. That is, C_1 is not necessarily a monolithic, tightly-controlled class.

It is also true that people either have Disease X (C_2) or do not (C_1), which would seem to be a sharp, binary distinction. However many diseases start very mild, and slowly become more severe, so that, indeed, the *degree* of Disease X may very well lie on a continuum, as discussed in [Examples 3.2](#) and [8.1](#), and not nearly be the sharp binary distinction implied by [\(8.74\)](#).

Decisions therefore need to be made whether we have an estimation or a classification problem, and secondly whether we have meaningful statistics for one or both classes.

⁷ As will be explored in more detail in [Table 9.1](#) in [Section 9.2](#).

⁸ Or we might learn/infer the statistics, as in [Chapter 7](#).

The challenge associated with highly asymmetric data.

Whether we are inferring class statistics, performing estimation, or learning the classifier in (8.70), we will need labelled data. The problem in most medical diagnosis problems is that

$$\mathbf{P}(C_1) \gg \mathbf{P}(C_2) \quad (8.75)$$

which, in and of itself, is not so much of a problem, however the consequence is that

$$\text{Number of data points from class } C_1 \gg \text{Number of data points from class } C_2 \quad (8.76)$$

This is known as the *class imbalance problem*, discussed in further detail in [Section 9.1](#). Although there can be many subtle issues associated with imbalanced classes, largely these boil down to two:

1. There may not be enough samples of Disease X (class C_2) to adequately learn the class statistics or to characterize the class shape.

This issue may be addressed by data augmentation/data resampling, or by choosing a classifier with fewer degrees of freedom, which can therefore be robustly learned with fewer data points.

2. The overwhelming number of samples from healthy people (class C_1) may implicitly cause classifiers to be biased towards C_1 (think of the behaviour of NN or k NN, for example).

This issue may be addressed by re-balancing the class data (here, removing labelled data from C_1), or by asserting class-dependent criteria, such as Neyman-Pearson, rather than a single overall behaviour (such as k NN).

The counter-intuitive behaviour of Bayes' Rule.

Even if we have adequate data and can design an effective classifier, it remains a significant challenge to meaningfully interpret the results. In particular, *Bayes' Rule*, which is taught in any basic course on statistics and is easy enough to derive, has a counter-intuitive behaviour which challenges even experts.

The challenge is known as the [Base Rate Fallacy](#), an issue which was very widely promoted by G. Gigerenzer in the area of medical diagnosis, for example in

P. Sedlmeier, G. Gigerenzer, "Teaching Bayesian reasoning in less than two hours," *Journal of experimental psychology* (130), 2001

or a more lengthy discussion in one of his texts, such as [2]. That is, suppose we are given the medical diagnosis setup of this case study, such that

$$\mathbf{P}(C_1) = 99\% \quad \mathbf{P}(C_2) = 1\% \quad \text{Specificity} = \text{Sensitivity} = 90\% \quad (8.77)$$

That is, our medical test gives the correct result nine times out of ten.

Now, given a patient of whom we know nothing else (no additional information), if the medical test says that Disease X is present, what is the likelihood that they are ill? Most people, *including* medical diagnosis specialists, would say that, because the “medical test gives the correct result nine times out of ten,” we would expect a likelihood around 90% that the person is ill.

From Bayes’ Rule (B.18), in general, for discrete events A, B we know that

$$\mathbf{P}(A|B) = \frac{\mathbf{P}(B|A)\mathbf{P}(A)}{\mathbf{P}(B)} \quad (8.78)$$

Therefore, in the context of our problem,

$$\begin{aligned} & \mathbf{P}(\text{Person has Disease X} | \text{Test declares Disease X}) \\ &= \frac{\mathbf{P}(\text{Test declares Disease X} | \text{Person has Disease X})\mathbf{P}(\text{Person has Disease X})}{\mathbf{P}(\text{Test declares Disease X})} \\ &= \frac{90\% \cdot 1\%}{99\% \cdot 10\% + 1\% \cdot 90\%} = 8.3\% \end{aligned} \quad (8.79)$$

How is this even remotely possible?! The test is correct 9 times out of 10, however a person declared ill has only an 8% chance of actually having the disease?

This is the Base Rate Fallacy, staring right at us.

The test statistics are not Bayesian: the test sensitivity and specificity know *nothing* about prior probabilities. The sensitivity and specificity both express probabilities *given healthy* and *given ill* people, but that is very different from a statistic *given a positive test result*. Turning around the conditioning, from conditioning on the state of health, to conditioning on the test outcome, *that* is Bayes Rule, and it is Bayes Rule which introduces the prior probabilities.

Figure 8.7 breaks this down graphically, to (hopefully) demystify the fallacy and make the situation more intuitive.

In practice, the situation is unlikely to quite be this way, because people are going to the doctor and having a test undertaken *because* there are symptoms. That is, we don’t normally prescribe tests of rare diseases to people at random. As a result,

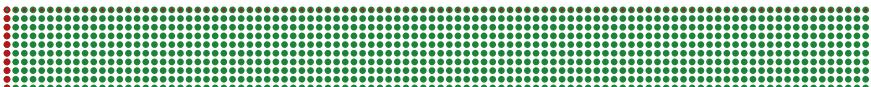
Suppose we begin with 1000 people:



Given the prior probabilities, 1% of the people are ill (red), the rest are not (green):



Our medical test gets 90% of the tests right (solid), and 10% incorrect (bi-colour):



So if we now focus only on those people who were declared ill (red centres):

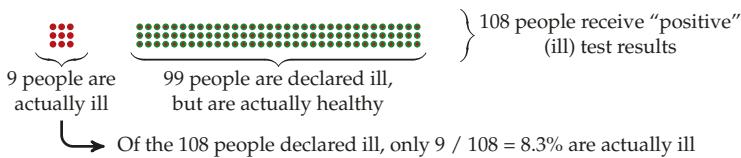


Fig. 8.7. BASE RATE FALLACY: Suppose there is a rare disease, affecting 1% of the population, for which we are given a medical test with 90% specificity and 90% sensitivity (90% accuracy for both ill and healthy people). Nevertheless, a given person who tests ill has a probability of only 8.3% of being ill!, a result which is highly counterintuitive, until broken down explicitly, as shown by the dots in the figure.

$$\mathbf{P}(C_2) = 1\% \quad \text{but} \quad \mathbf{P}(C_2 | \text{Symptom A, B, C, ...}) \gg 1\% \quad (8.80)$$

which clearly changes the prior statistics, and therefore lessens the degree to which (8.79) would lead to a surprising result.

Lab 8: Statistical and Distance-Based Classifiers

Section 8.3 discussed the behaviour of the ML and MAP classifiers in the case that the class distributions are Gaussian. In this lab we would like to implement the classifiers and see how they compare to the Mahalanobis distance of Chapter 6.

We begin with a function that does all of the work: it computes the relative Mahalanobis distance of (8.26), and then determines the Mahalanobis, ML, and MAP classification boundaries by comparing the distance to the τ_{Mahal} , τ_{ML} , τ_{MAP} of (8.32):

```
function h = two_class_test( xgrid, ygrid, mean1, cov1, prob1, mean2,
  cov2, prob2)
  % compute distance from every grid point to class means
  pts = [xgrid(:) ygrid(:)];
```

```

pts1 = pts - mean1(:)';
pts2 = pts - mean2(:');

% compute relative Mahalanobis distance
reldist = sum((pts1 * inv(cov1)) .* pts1,2) - sum((pts2 * inv(
cov2)) .* pts2,2);
reldist = reshape(reldist, size(xgrid));
tau_mahal = 0;
tau_ml = 2*log(sqrt(det(cov2)/det(cov1)));
tau_map = tau_ml + 2*log(prob1/prob2);

% plot classification boundaries
clf
contour(xgrid, ygrid, reldist, tau_mahal*[1 1], 'r');
hold on
contour(xgrid, ygrid, reldist, tau_ml*[1 1], 'g');
contour(xgrid, ygrid, reldist, tau_map*[1 1], 'b');

% plot class ellipses
ellipse_plot( mean1, cov1, '—', 'k' );
ellipse_plot( mean2, cov2, '—', 'k' );
end

```

where the *ellipse_plot* function is the same one which was already defined in [Lab 5](#). With the test in place, we will call it four times with different choices of covariances and prior probabilities:

```

% All examples on same grid
[X,Y] = meshgrid( 0:0.1:8, 0:0.1:6 );

% Equal covariances and Equal probabilities
two_class_test( X, Y, [3 2], [4 0; 0 2], 0.5, [5 4], [4 0; 0 2], 0.5 );

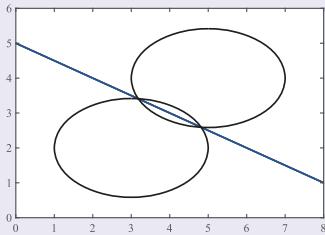
% Different covariances with Equal probabilities
two_class_test( X, Y, [3.5 2.5], [7 0; 0 3], 0.5, [5 4], [2 0; 0 1], 0.5 );

% Different covariances with Unequal probabilities
two_class_test( X, Y, [3.5 2.5], [7 0; 0 3], 0.7, [5 4], [2 0; 0 1], 0.3 );
two_class_test( X, Y, [3.5 2.5], [7 0; 0 3], 0.3, [5 4], [2 0; 0 1], 0.7 );

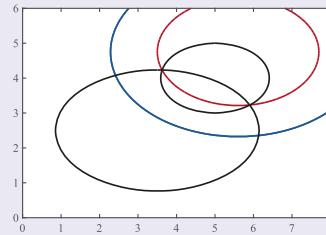
```

... which give rise to the following results. In all cases the MMD classification boundary is red, ML is green, and MAP is blue. In the first

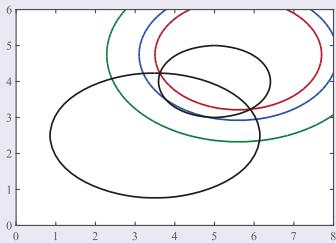
two examples some classifiers have identical classification boundaries, so not all three colours are seen.



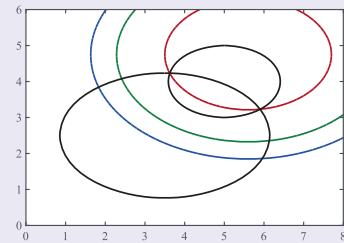
Equal covariances and equal probabilities
Manalanobis, ML, MAP all the same



Different covariances with equal probabilities
ML and MAP (blue) biased in favour of smaller class



Different covariances with unequal probabilities
The larger (bottom left) class is more likely



The smaller (upper right) class is more likely

We observe the behaviours derived analytically in [Section 8.3](#):

- If the covariances are equal the classifier is linear;
- If the classes are equally likely then ML and MAP are the same;
- For unequal covariances, relative to MMD, the ML and MAP classifiers will offer a larger decision region to the more compact class;
- For unequal class probabilities, relative to ML, the MAP classifier will offer a larger decision region to the more likely class.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links: [Probabilistic classification](#), [Statistical classification](#),
[Evaluation of binary classifiers](#), [Q-function](#), [Error function](#)

This chapter did rely on a solid understanding of basic elements of probability theory, which are reviewed in [Appendix B](#). Very accessible starting points for probability and statistics include the texts by Brase & Brase [1], Kiemele [4], Haigh [3], and Urdan [5].

Sample Problems

Problem 8.1: Short Answer

Give a short definition of each of the following:

- ML Classification
- MAP Classification
- The Gaussian tail function $Q()$
- Risk versus error
- Neyman-Pearson criterion
- Base Rate Fallacy

Problem 8.2: Short Answer

Offer brief answers to each of the following:

- (a) When might you want/need to use the ML classifier instead of MAP?
- (b) For a set of N Gaussian clusters in n dimensions, under what conditions will the MMD, ML, and MAP classifiers all yield the same classifier?
- (c) Draw a simple sketch which argues persuasively why the MAP classifier minimizes the probability of classification error.
- (d) The ML classifier is not a function of the class prior probabilities $\mathbf{P}(C_\kappa)$, so how is it that $\mathbf{P}_{\text{ML}}(e)$, the probability of error for the ML classifier, can be a function of $\mathbf{P}(C_\kappa)$?

Problem 8.3: Conceptual — MAP

Each of the following statements is either true or false. If true, argue why; if false, develop a convincing counterexample.

- (a) The MAP classification boundaries always depend on the class probabilities $\mathbf{P}(C_\kappa)$.
- (b) There will always be some set of values for the class probabilities $\mathbf{P}(C_\kappa)$ such that $\mathbf{P}_{\text{MAP}}(e) = \mathbf{P}_{\text{ML}}(e)$.
- (c) The Maximum Likelihood probability of error $\mathbf{P}_{\text{ML}}(e)$ is never a function of the class probabilities $\mathbf{P}(C_\kappa)$.
- (d) For every possible classifier, regardless of its type, the probability of error $\mathbf{P}(e)$ must always be a function of the class probabilities $\mathbf{P}(C_\kappa)$.

Problem 8.4: Conceptual — Bayesian Statistics

Given three equally likely classes, each obeying a Gaussian distribution:

$$C_A \sim \mathcal{N}(0, 4) \quad C_B \sim \mathcal{N}(5, 2) \quad C_C \sim \mathcal{N}(10, 4) \quad (8.81)$$

- (a) Draw $p(x|A)$, $p(x|B)$, $p(x|C)$ on the same plot. Label the maximum value of each function.
- (b) Now sketch $\mathbf{P}(A|x)$, $\mathbf{P}(B|x)$, $\mathbf{P}(C|x)$ together on a new plot. You do not need to show calculations, but do carefully label your figure.

Problem 8.5: Conceptual — Gaussian Statistics

Suppose we have two classes, C_A , C_B , both of them Gaussian.

For *each* of the four cases below, sketch an axis and show the relative positions of all MED, ML, and MAP decision boundaries. The boundaries do not need to be drawn to scale, but the *relative* positions are important.

Case	μ_A	μ_B	σ_A	σ_B	$\mathbf{P}(C_A)$	$\mathbf{P}(C_B)$
(i)	-2	1	1	1	1/2	1/2
(ii)	-1	1	1	2	1/2	1/2
(iii)	-1	1	2	1	1/3	2/3
(iv)	-1	1	1	1	1/10	9/10

Problem 8.6: Analytical — ML Classification

Refer back to [Problem 5.7 on page 111](#), in which a two-dimensional problem having two classes, parameterized by b , is transformed into a one-dimensional feature y :

- (a) What is the 1-D ML classifier for these two classes? In other words what is the ML classifier, as a function of b , for the two classes given the feature y ?
- (b) Suppose that the two classes are equally likely; that is,

$$\mathbf{P}(C_1) = \mathbf{P}(C_2) = 0.5$$

What is the probability of classification error $\mathbf{P}(e)$ for the ML classifier, as a function of b ?

- (c) Draw a sketch of $\mathbf{P}(e)$ versus b .

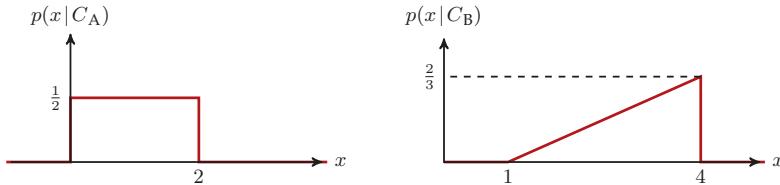


Fig. 8.8. The two class distributions for **Problem 8.8**.

Problem 8.7: Analytical — Parameter Dependencies

Suppose we are given two one-dimensional Gaussian classes:

$$p(x|C_A) \sim \mathcal{N}(\mu_A, \sigma_A^2) \quad p(x|C_B) \sim \mathcal{N}(\mu_B, \sigma_B^2) \quad (8.82)$$

where $\mu_A < \mu_B$, and the prior probability of class C_A is P_A .

- (a) Suppose $\mu_A < \tau < \mu_B$, where τ is the classification boundary between clusters A and B. What is $P(e)$, the probability of classification error, in terms of $P_A, \mu_A, \mu_B, \sigma_A, \sigma_B, \tau$?
- (b) For each of the variables $P_A, \mu_A, \mu_B, \sigma_A, \sigma_B, \tau$, as the variable is increased, does $P(e)$ necessarily increase, decrease, stay unchanged, or would its change depend on specific parameter values?

Problem 8.8: Analytical — MAP Classification

Given two classes, C_A and C_B as shown in [Figure 8.8](#), where C_A has a uniform distribution and C_B has a triangular distribution. The prior probability of class C_A is P_A .

- (a) Write down explicit forms (i.e., equations) for $p(x|C_A), p(x|C_B)$. Compute the means μ_A, μ_B and the variances σ_A^2, σ_B^2 .
- (b) Let τ represent the MAP classification boundary between C_A and C_B . Determine the dependence of τ on P_A and draw a sketch of τ versus P_A .
- (c) Give a rough sketch of the MAP probability of error, $P_{MAP}(e)$ as a function of P_A .

Problem 8.9: Analytical — Classification Error

[Problem 5.8](#) parts (b) and (d) each proposed a single feature for two equally likely Gaussian classes. For each of these two features,

- (a) Compute the class means and variances in the resulting feature space.
- (b) Find the MAP classifier in this feature space.
- (c) Compute the probability of classification error $P_{MAP}(e)$.

Problem 8.10: Analytical — Classification Error

In Appendix D there is an analytic derivation presented for the probability of classification error for Euclidean-NN, relative to MAP, for the two-class ($K = 2$) case.

Now, generalize that argument for the case of three classes ($K = 3$).

Problem 8.11: Analytical — Classification & Inference

Consider two equally likely classes, both characterized by a uniform distribution:

$$p(x | C_A) \sim \mathcal{U}(0, 2) \quad p(x | C_B) \sim \mathcal{U}(2, 4) \quad (8.83)$$

- (a) Draw a labelled sketch of the two distributions.
- (b) What is the mean and variance of each class?
- (c) Find the MED and MAP classifiers and their probabilities of error, $\mathbf{P}_{\text{MED}}(e)$, $\mathbf{P}_{\text{MAP}}(e)$.
- (d) Now suppose we don't know the class distributions. Instead, we'll do parameter estimation and *assume* that the distributions are Gaussian. Assume that we observe a lot of data, so that we estimate the means μ_A , μ_B and the variances σ_A^2 , σ_B^2 exactly.

Find the MAP classifier. Under our assumption that the distributions are Gaussian, what *would* $\mathbf{P}_{\text{MAP}}(e)$ be?

- (e) A bit more challenging, suppose that we correctly know that the class distributions are uniform with a width of 2.0. From each class we observe only *one* randomly sampled data point: x_A , x_B . So our estimated class distributions will be uniform, width of 2.0, centered on the random samples x_A and x_B .

What is the MED classifier, as a function of x_A , x_B ?

Of course, x_A , x_B are actually random. Sketch the distribution for the value (location) of the MED classification boundary.

- (f) For the MED classifier which you determined as a function of x_A , x_B in part (e), sketch the distribution for $\mathbf{P}_{\text{MED}}(e)$, where the probability of error is based on the *actual* cluster distributions.

Finally, a bit harder, if we didn't know the *actual* cluster distributions, to infer $\mathbf{P}_{\text{MED}}(e)$ we would need to do so based on the *estimated* cluster distributions. Sketch the distribution for $\mathbf{P}_{\text{MED}}(e)$ based on the estimated distributions.

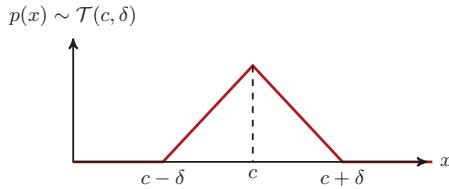


Fig. 8.9. The triangular distribution $\mathcal{T}(c, \delta)$ for Problem 8.12.

Problem 8.12: Analytical — Classification

A random variable x obeys the triangular distribution, $x \sim \mathcal{T}(c, \delta)$, if its distribution takes the shape as shown in Figure 8.9.

- (a) If $x \sim \mathcal{T}(c, \delta)$, find μ_x, σ_x the mean and the standard deviation of x , in terms of c and δ .

- (b) Suppose that we have two classes, C_A and C_B , where

$$\begin{aligned} p(x|C_A) &\sim \mathcal{T}(10, 1) \\ p(x|C_B) &\sim \mathcal{T}(10, 3) \end{aligned}$$

(i) Draw a sketch of the two distributions.

(ii) Draw a sketch of $\mathbf{P}_{\text{MAP}}(e)$ as a function of \mathbf{P}_A .

- (c) Suppose that we have two classes, C_A and C_B , where

$$\begin{aligned} p(x|C_A) &\sim \mathcal{T}(3, \delta) & \mathbf{P}_A = \frac{1}{2} \\ p(x|C_B) &\sim \mathcal{T}(5, \delta) & \mathbf{P}_B = \frac{1}{2} \end{aligned}$$

Draw a sketch of $\mathbf{P}_{\text{ML}}(e)$ as a function of δ .

Problem 8.13: Comprehensive — Human Neurons

Human neurons (you may also wish to take a look at Example 11.2) communicate by firing electric signals repeatedly over time. For certain neurons the time x between successive firings is exponential:

$$p(x|C_1) = \begin{cases} e^{-t} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (8.84)$$

In other nerve cells, there is a certain recovery time which is required before the cell can fire again, so the time interval is delayed:

$$p(x|C_2) = \begin{cases} 2e^{-2(t-1)} & x \geq 1 \\ 0 & x < 1 \end{cases}$$

We can try to classify a neuron into one of these two classes by measuring the inter-firing time x . Unless otherwise stated, assume that the two types of neurons are equally likely: $\mathbf{P}(C_1) = \mathbf{P}(C_2) = 0.5$.

- (a) Draw a sketch of the two classes.
- (b) Find the mean and variance for each of the two classes.
- (c) Write down the MAP classifier *as a function of $\mathbf{P}(C_1)$.*
- (d) Find the MED and MMD classifiers.
- (e) Assume that the two classes are equally likely. Find expressions for the probabilities of error $\mathbf{P}_{\text{MED}}(e)$ and $\mathbf{P}_{\text{MAP}}(e)$. Which error is smaller?
- (f) Carefully redraw your sketch from (a), but now on your sketch compare $\mathbf{P}_{\text{MED}}(e)$, $\mathbf{P}_{\text{MMD}}(e)$, and $\mathbf{P}_{\text{MAP}}(e)$. Based on your sketch, approximately how similar or different are the performances of the three classifiers?

Problem 8.14: Analytical — Multi-Class Classification

Suppose that we have two classes, C_1 and C_2 :

$$C_1 \sim \mathcal{U}\left(-\frac{1}{2}, \frac{1}{2}\right) \quad C_2 \sim \mathcal{U}(-1, 1) \quad (8.85)$$

- (a) Give an equation for each of the two class distributions.
- (b) Draw a sketch of the two classes.
- (c) Find the mean and variance for each of the two classes.
- (d) Write down the MMD, ML, and MAP classifiers *as a function of $\mathbf{P}(C_1)$.*
- (e) Compute $\mathbf{P}_{\text{ML}}(e)$, $\mathbf{P}_{\text{MAP}}(e)$ as a function of $\mathbf{P}(C_1)$.
- (f) Let us generalize the problem. We now have K classes, C_1, \dots, C_K , where each class C_κ is uniformly distributed over $-2^{\kappa-2} < x < 2^{\kappa-2}$.
 - (i) Draw a sketch of this new problem.
 - (ii) What is the ML classifier?
 - (iii) Determine $\mathbf{P}_{\text{ML}}(e)$ as a function of $\mathbf{P}(C_1), \mathbf{P}(C_2), \dots, \mathbf{P}(C_K)$.
 - (iv) Suppose that all of the classes are equally likely. Does $\mathbf{P}_{\text{ML}}(e)$ increase with K , decrease with K , or stay constant? What is $\lim_{K \rightarrow \infty} \mathbf{P}_{\text{ML}}(e)$?

Problem 8.15: Comprehensive — Multi-Dimensional Classification

Suppose that we have two classes, C_1 and C_2 in feature space \mathcal{X} :

$$C_1 \sim \mathcal{U}(1, 3) \quad C_2 \sim \mathcal{N}(3, 1) \quad (8.86)$$

- (a) Sketch $p(x|C_1), p(x|C_2)$.
- (b) For this part only, suppose that $\mathbf{P}(C_1) = \mathbf{P}(C_2) = 0.5$.
 - (i) What is the MAP classifier?
 - (ii) Compute $\mathbf{P}_{\text{MAP}}(e)$.
- (c) Now we generalize part (b):
 - (i) What is the MAP classifier as a function of $\mathbf{P}(C_1)$?

- (ii) Sketch $\mathbf{P}_{\text{MAP}}(e)$ as a function of $\mathbf{P}(C_1)$.
- (iii) For comparison purposes, also sketch $\mathbf{P}_{\text{MED}}(e)$ as a function of $\mathbf{P}(C_1)$ on the same plot with $\mathbf{P}_{\text{MAP}}(e)$.

(d) Let us generalize the problem, further, to 2D:

We now have a 2D feature space $\underline{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, where

$$p(\underline{x} | C_i) = p(x_1 | C_i) \cdot p(x_2 | C_i)$$

where x_1 and x_2 have the same distributions as before (that is for class C_1 , x_1 and x_2 are both uniform, and for class C_2 they are both Gaussian, as before).

- (i) Draw a sketch of the shapes of classes C_1 and C_2 .
- (ii) Assume $\mathbf{P}(C_1) = \mathbf{P}(C_2) = 0.5$: Determine (give equations for) the MED and MAP classifiers and draw a sketch.
- (e) **Difficult:** Suppose the idea of part (d) is generalized even further to n dimensions. For a fixed $\mathbf{P}(C_1)$, argue persuasively whether $\mathbf{P}_{\text{MAP}}(e)$ is an increasing, decreasing, fixed, or irregular function of n .

Problem 8.16: Analytical — Multidimensional Gaussian

Suppose we have a problem with $K = 2^n$ equally-likely Gaussian-distributed classes in n dimensions. The class means are

$$\mu_k = \begin{bmatrix} \pm 1 \\ \pm 1 \\ \vdots \end{bmatrix}$$

In other words, the means are

$$\begin{aligned} n = 1 \text{ Dimension: } \mu_1 &= \begin{bmatrix} -1 \end{bmatrix} \quad \mu_2 = \begin{bmatrix} 1 \end{bmatrix} \\ n = 2 \text{ Dimensions: } \mu_1 &= \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad \mu_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \mu_3 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \mu_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (8.87) \\ &\text{etc.} \end{aligned}$$

All of the covariances are the same: $\Sigma_i = I$

- (a) Start in 1D: Sketch $p(x | C_1)$ and $p(x | C_2)$. Sketch the ML classifier, and compute $\mathbf{P}_{\text{ML}}(e)$.
- (b) Continue to 2D: Draw a sketch of the four class means with their unit standard deviations. Sketch the ML classifier, and compute $\mathbf{P}_{\text{ML}}(e)$.
- (c) Generalize to n D: Compute $\mathbf{P}_{\text{ML}}(e)$, the ML probability of classification error for the 2^n classes in n dimensions.

Problem 8.17: Analytical — Multidimensional Gaussian

Suppose that we have just two classes in n dimensions, $\Sigma_0 = \Sigma_1 = I$, such that

$$\mu_0 = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \mu_1 = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (8.88)$$

- (a) Sketch the problem in 1D, compute $\mathbf{P}_{\text{ML}}(e)$ as a function of $\mathbf{P}(C_0)$.
- (b) Sketch the problem in 2D, compute $\mathbf{P}_{\text{ML}}(e)$ as a function of $\mathbf{P}(C_0)$.
- (c) Generalize to n D (but no sketch!): compute $\mathbf{P}_{\text{ML}}(e)$ as a function of $\mathbf{P}(C_0)$.

Problem 8.18: Numeric/Computational — Classification Error

[Appendix D](#) derived the result that the NN-Euclidean classifier for $K = 2$ classes would have an expected probability of error no worse than twice that of MAP. This result was then generalized to the $K = 3$ class case in [Problem 8.10](#).

In this problem, we would like to explore this question numerically/- computationally. We begin with definitions for uniform and Gaussian classes:

Uniform Distributions:	$C_1 \sim \mathcal{U}(-\alpha, -\alpha + 1)$	$C_2 \sim \mathcal{U}(\alpha, \alpha + 1)$	$C_3 \sim \mathcal{U}(0, 1)$
Gaussian Distributions:	$C_1 \sim \mathcal{N}(-\alpha, 1)$	$C_2 \sim \mathcal{N}(\alpha, 1)$	$C_3 \sim \mathcal{N}(0, 1)$

We will undertake a total of four tests, based on two or three classes, and based on uniform or Gaussian distributions; all four tests are parameterized by a single parameter α . Two-class tests will be based on C_1 and C_2 only.

- (a) Find the analytical $\mathbf{P}_{\text{MAP}}(e)$ for each of the four scenarios, as a function of α . You could, in principle, deduce these computationally, however the analytical approach is not very difficult.
- (b) Develop a program to generate sample points from the classes. You will need two sets of data: one for training (on which the NN classifier is based) and one for testing. The number of points N is for you to decide.⁹

⁹ Essentially, there will be experimental variability in the evaluated $\mathbf{P}_{\text{NN}}(e)$ from one run to the next. By doing a number of runs (say 10) for a given value of N , you can compute the standard deviation of the computed values of $\mathbf{P}_{\text{NN}}(e)$. If the standard deviation is too large to allow you to reach a meaningful conclusion regarding $\mathbf{P}_{\text{NN}}(e)$, then you need to increase the number of points.

- (c) Develop a program to solve for the NN probability of error. At each testing point you need to see which point in all of the training data is the closest.
- (d) Plot $\mathbf{P}_{\text{MAP}}(e)$ and $\mathbf{P}_{\text{NN}}(e)$ as a function of α for each of the four tests. What do you observe?

The ambitious student could easily generalize this problem in a variety of ways:

- Extend your code to numerically find the MAP probability of error, to allow more complex scenarios to be evaluated, where it would be inconvenient to work out the error analytically
- Increase the parameterization, such that both means and variances can vary
- Allow for variations in class prior probabilities
- Explore the problem in higher dimensions
- Explore the problem for more than three classes
- Allow greater problem variation by having means and variances be random, rather than predetermined
- Develop efficient code to allow a greater number of runs or more points per class, in order to have tighter bounds on the probability estimates

Problem 8.19: Real World, Open-Ended — Robustness

Any Gaussian distribution has infinitely-long tails, by definition, and so is almost certainly not an exact model for almost any real-world/real-data problem, which must be bounded (since nothing real can be infinite).

Ironically, in practice the problem is actually the other way around. Although the Gaussian tails are infinitely long, analytically, in actual fact the tails are very thin (rare). Keeping in mind the equation for the Gaussian distribution,

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (8.89)$$

as x moves away from its mean, the exponential term decays as e^{-x^2} , which is *very* fast. To better visualize this, if you had randomly selected a $\mathcal{N}(0, 1)$ value every *second* since the beginning of the universe (14 billion years), it is unlikely that you would yet have had a *single* value greater than $8\frac{1}{2}$. That is what we mean by thin tails.

In practice, most measurements and everyday phenomena are associated with outliers — the occasional wildly-different measurement, test result, or freak weather event. All such phenomena are poorly represented by Gaussian (Normal) distributions, because their tails are so thin that they predict outliers to occur far less frequently than what we actually observe.

Aspects of robustness were discussed in [Example 3.1](#), outlier behaviour in the context of prototypes in [Figure 6.12](#), and an overview of heavy-tail statistics can be found in [Appendix B.5](#).

Your task: Look up robust methods, as they relate to pattern recognition/machine learning. This might be robustness in learning, robustness in validation and testing, or strategies for pruning outliers. Prepare an overview of the key concerns associated with outliers and robustness, and one or two of the approaches which have been developed in response.

References

1. C. Brase, C. Brase, *Understanding Basic Statistics* (Cengage Learning, Boston, 2012)
2. G. Gigerenzer, *Rationality for Mortals: How People Cope with Uncertainty* (Oxford University Press, Oxford, 2008)
3. J. Haigh, *Probability: A Very Short Introduction* (Oxford Academic, Oxford, 2012)
4. M. Kiemele, *Basic Statistics*. (Air Academy, Colorado Springs, 1997)
5. T. Urdan, *Statistics in Plain English* (Routledge, Milton Park, 2010)



Classifier Testing and Validation

The question of data was already discussed at some length in [Chapter 3](#), but mostly at a conceptual level, in terms of overfitting/underfitting. Now that we have had several chapters actually developing classification algorithms, we need to get much more specific about how to work with data: how do we properly validate a classifier, how do we determine overfitting, and what are the pitfalls to avoid?

9.1 Working with Data

A dataset \mathcal{D} of measured points $\mathcal{D} = \{\underline{y}_i\}$ for some class C_κ is understood to be randomly drawn from a (typically unknown) distribution $p(\underline{y} | C_\kappa)$. In the ideal case, we frequently refer to the data as being IID (independent, identically-distributed), meaning that all of the data are drawn from the *same* distribution (identical), and that if you *did* know distribution $p(\underline{y} | C_\kappa)$, then the given data $\underline{y}_1, \dots, \underline{y}_{N-1}$ would give you no additional information about \underline{y}_N than what you already knew from $p(\underline{y} | C_\kappa)$ (independent).¹

However in being given a set of data points, these may be subject to any number of limitations:

Limited Data: Based on learning from data in [Example 3.3](#), the discussion of inference in [Section 7.2](#), and particularly the derivation of the estimation error variance of σ^2/N in [\(7.23\)](#), it should be clear that estimating and validating model parameters requires data, and too few data points (too small $N = |\mathcal{D}|$) simply leads to poor models. More generally, to avoid

¹ More formally, independence is defined in [\(B.25\)](#) in [Section B.4](#).

overfitting we need N to be much greater than the number of degrees of freedom in any proposed model.

Given limited data, we have a number of possible responses:

CHANGE THE PROBLEM: Is it possible to reconsider the pattern recognition problem being solved? Are there similar classes which might be combined?, or, since the process of labelling data is expensive/time consuming, would it be possible to extend the data significantly by considering semi-supervised learning, as discussed in [Section 12.3](#)?

ADAPT FROM ANOTHER PROBLEM: Can we undertake learning in a closely-related context, and then transfer or apply this learning to our original context?, a process known as *Domain Adaptation* or *Transfer Learning*. We could consider such adaptation in a variety of ways:

- If there is another dataset $\bar{\mathcal{D}}$, with the same definition of measurements and classes, then we can learn the classifier from that dataset with no further adaptation required.
- If there is a closely-related dataset $\bar{\mathcal{D}}$, then we can learn the classifier from that dataset and nudge/tweak/fine-tune the classifier on the original (smaller) dataset \mathcal{D} . Only a modest amount of re-learning is undertaken on \mathcal{D} to avoid overfitting. Note that such an approach is possible only for classifiers for which an iterative optimization is used, and not a closed-form end-to-end classifier such as Mean-Mahalanobis.
- If there is an alternate dataset $\bar{\mathcal{D}}$ with the same measurements as in \mathcal{D} but possibly different classes, one could consider learning the feature extraction ([Chapter 5](#)) from $\bar{\mathcal{D}}$, leading to a reduced-dimensional feature space \mathcal{X} , and then learning only the relatively few parameters in the classifier based on \mathcal{D} .

CHANGE THE MODEL: Since the key concern is to have the number of data points significantly larger than the degrees of freedom, can we reduce the number of degrees of freedom, for example by using a parametric model rather than a nonparametric one, by using a lower-dimensional feature space, or by using a simpler parametric model having fewer parameters?

CREATE MORE DATA: Also known as *Data Augmentation*, in some cases it can be possible to synthesize additional data. Essentially one uses the given N data points to statistically resample additional points; this approach is particularly widely-used in classifier validation, to be described further in [Section 9.3](#), or in working with images and video, in which case a variety of class-invariant transformations can be proposed, as described in [Example 9.1](#), which allows for significant data creation.

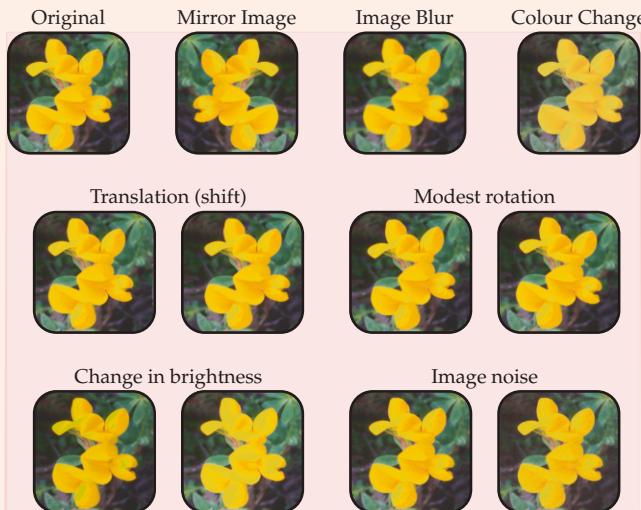
Example 9.1: Data Augmentation

Data Augmentation refers to increasing the size of a dataset, particularly for classifiers which require very large amounts of data in order to train. But how can one reliably create additional data from given data, without accidentally creating false or unreliable data?

Data augmentation relies on the user to identify *invariants*, types of behaviour to which we would like the classifier to be insensitive, any transformation of a feature which truly does not change the class to which the feature vector belongs. For example, in classifying plant species, if $C(I)$ is the true class associated with image I , then C is invariant to transformation $t()$ if

$$C(I) = C(t(I)) \quad \text{For every image } I$$

Given an image of a wildflower, we would expect the image classification to be unchanged under any of the following transformations:



All of the images, above, look essentially identical to the human eye, yet *very* different in raw image form, which is precisely what makes these effective from a data augmentation perspective. The augmented data are also fully labelled, since they all inherit the class label of the original image.

Even in a context where the image is fairly constrained, such as face images for biometric purposes (as in [Case Study 2](#)), the imaging system will seek to have the face centered in the image with no rotation, however even the most rigid face imaging cannot preclude a small head rotation of 1 to 2 degrees, or a translation by a few pixels, or slight changes in skin tone due to sun exposure. Since all of the invariants can be considered, together, and not just in isolation, the combinatorics lead to nearly endless permutations, and correspondingly nearly unlimited augmented data.

Dependent data: The given data are normally assumed to be independent; if they are not, then they are *dependent*² in some way.

Having dependent data is not inherently problematic: it is still perfectly legitimate to learn/classify on the basis of such data, and the measured data y_i remain random samples of $p(y|C_\kappa)$. However there are two potential issues here.

First, dependencies between data points may imply inadequate mixing, such that (for example) the first half of the dataset tends to have samples from one half of a class, and the second half of the dataset has samples from the other half. This issue is easily addressed by mixing/randomly permuting the order of data points, or by using multi-fold cross-validation (Section 9.3), to ensure that the training/testing/validation sets are statistically equivalent.

Secondly, the effective amount of information may be reduced, and therefore the *effective* number of data points is fewer than what is offered in the dataset. As a very simple example, consider the dataset \mathcal{D} consisting of $2N$ data points:

$$\mathcal{D} = [\underline{x}_1, \underline{x}_1, \underline{x}_2, \underline{x}_2, \dots, \underline{x}_N, \underline{x}_N]$$

in which each of N points is just included twice. Very clearly the *effective* number of data points, the actual amount of information provided, is only N points, even though $2N$ points were provided.

Dependence does not necessarily explicitly need quantifying. Cross-validation, as described in Section 9.2, will still properly assess a given classifier. However it is important to understand that the required dataset size $|\mathcal{D}|$ may be larger than had been expected to reach a given degree of accuracy.

Noisy Data: Nearly every dataset will be subject to some sort of uncertainty or variability:

$$\text{Ideal Distrib. } p_{\text{Ideal}}(y|C_\kappa) \xrightarrow{\text{Plus Noise}} \text{Noisy Distrib. } p_{\text{Noisy}}(y|C_\kappa) \quad (9.1)$$

Statistically, it does not really matter whether the variability is due to actual measurement noise, or to within-class variability inherent to the definition of the class: either source of variability leads to a degree of ambiguity in the definition of the class in measurement or feature space, and normally then requires more data to properly characterize.

² In the English vernacular people might refer to such data as *correlated*, i.e., having some relationship. Strictly speaking, as described in Section B.4, correlation is only a measure of *linear* relationship, such that data could be dependent but *uncorrelated*.

In principle, noise is not an issue, as long as it is consistent. That is, if a classifier is learned on the basis of dataset

$$\mathcal{D} \sim p_{\text{Learning}}(\mathcal{Y} | C_\kappa) = p_{\text{Noisy}}(\mathcal{Y} | C_\kappa) \quad (9.2)$$

and then later applied to classify measurements

$$\mathcal{Y} \sim p_{\text{Use Case}}(\mathcal{Y} | C_\kappa) = p_{\text{Noisy}}(\mathcal{Y} | C_\kappa) \quad (9.3)$$

then the learning and later use are consistent.

In contrast, *any* inconsistency between the distributions at learning and later use, whether due to noise or some other effect, means that the pattern recognition has been approached in an apples-and-oranges kind of way: the problem that was trained upon is different from the one later presented to the classifier.

Outliers: A dataset which is subject to outliers is essentially just a special case of the *Noisy Data* case just discussed. That is, $p_{\text{Noisy}}(\mathcal{Y} | C_\kappa)$ could be described by a Gaussian noise process (typical random noise, not characterized by outliers), or by a power-law/heavy-tail noise process (which very readily leads to outliers).

The key concern with outliers is that their variability is typically so great that they can be expected to overlap other classes, and for datasets subject to outliers it is imperative to have a classifier which is robust to outliers, as was illustrated in [Figure 6.12](#).

Classifiers which are particularly *sensitive* to outliers are Nearest Neighbour and Hard-Margin Support Vector Machines, since both of these classifiers can have their decisions significantly influenced by a single data point. Classifiers which are generally *robust* to outliers are k NN for k sufficiently large, Soft-Margin Support Vector Machines, ML/MAP (assuming that the underlying statistical distribution accurately reflects the outlier behaviour), and low-order parametric methods (such as Mahalanobis) which resist overfitting.

Not Randomly Sampled: We may *believe* that we have an appropriate match between learning and later use, on the basis of a consistency in definition between dataset \mathcal{D} and the measurement y to be classified. For example, for face recognition,

$$\mathcal{D} = \{\text{Dataset of Face Images}\} \quad y = (\text{Measured Face Image}) \quad (9.4)$$

However, even given a consistent definition it is very easy to have a dataset

$$\mathcal{D} \sim p_{\text{Learning}}(\mathcal{Y} | C_\kappa) \quad (9.5)$$

which is biased relative to the later use case, such that

$$p_{\text{Learning}}(y|C_\kappa) \neq p_{\text{Use Case}}(y|C_\kappa). \quad (9.6)$$

Written out like this, it seems obvious that there is a mismatch between learning and later use, however that is only because (9.6) is written explicitly in terms of the mismatched distributions.

In practice, such a bias can be *much* more subtle, because the distributions are normally not, in fact, known. Rather, we are given a dataset, which may give the appearance of being comprehensive, particularly if very large, however very typically the person or organization who compiled the dataset is *different* from the person or organization undertaking the training and validation of the classifier, which may furthermore be different from the person or organization later using the classifier. Particularly with databases compiled from the Internet, it is very easy to have biases based on gender, geography, skin colour, age, education, or income.

Avoiding such biases has been a significant aspect in statistics, particularly with regards to biases in surveys, polls, and census taking. In principle, given sufficient use-case samples y_i , one might attempt a statistical test on the match of the distributions underlying \mathcal{D} and $\{y_i\}$, as discussed in Section 7.4. In any event, there is a certain moral responsibility for the designer of a classifier to undertake due diligence to ensure that subtle discriminatory biases of the form of (9.6) are not present, either by understanding the process by which a dataset \mathcal{D} was formed or gathered, or with a critical examination the data used in testing and validation, possibly with data resampling for under-representation, as discussed below in the context of class imbalance.

Nonstationarily Sampled: A statistical concept, such as a distribution, is said to be nonstationary with respect to some variable, such as location or time, if the model is a function of this variable. In practical terms, for pattern recognition this can be an issue if the distribution underlying a class drifts over time, such that

$$p(y|C_\kappa, t_{\text{Learning}}) \neq p(y|C_\kappa, t_{\text{Use}}) \quad (9.7)$$

As a simple example, a face recognition biometric (as in Case Study 2) will be trained on a person at some point in time, and as the person ages certain aspects of the biometric may begin to drift, eventually leading to a misclassification. In general, we can consider one of two strategies:

1. The extraction of features x robust to drift, remaining constant over time, making the problem time-stationary;
2. Periodic re-training and re-validation of the classifier.

Temporal drifts are not just due to aging; there are many social phenomena, such as smart-phone use by pedestrians, which could lead to very different models of pedestrian awareness now as opposed to twenty years ago.

All of the preceding discussion was focused on a single class; how well learning or inference associated with one class could be accomplished based on data.

There is one final issue *between* classes: an asymmetry, known as the *class imbalance problem*, already introduced in [Section 8.5](#). Whereas [Section 8.5](#) considered an asymmetry in risk or cost, a further asymmetry is that of data: if class C_1 is far more likely than C_2 , then it stands to reason that we may have far more training/testing/validation data for C_1 than for C_2 .

There is extensive literature on the class imbalance problem [1, 2], and the solutions broadly fall into three categories: changing the data, changing the objective, or changing the classifier.

1. Change the Data: That a data imbalance leads to problems can be quickly appreciated in considering the application of the k NN classifier of [Section 6.4](#). Given N_κ data points in class C_κ , in considering a suitable value of k we typically wish

- $k \gg 1$ to avoid overfitting, but
- $k \ll N_\kappa$ to preserve a sensitivity to class structure.

However if $N_1 \gg N_2$ there may simply be no suitable value of k :

- A value of k suitable for C_2 will overfit C_1 , and
- A value of k suitable for C_1 will underfit and lose all structure of C_2 .

If a given class is under-represented in a dataset, then the dataset can be resampled to balance the representation:

DATA OVERSAMPLING: We can resample a class having fewer data points to increase its representation in the data set. Given N data points $\{y_1, \dots, y_N\}$, an additional data point y_{N+1} could be sampled in a number of ways:

- Sample a point at random from the dataset and duplicate:

$$y_{N+1} = y_r \quad \text{Index } r \text{ uniformly chosen randomly from } 1, \dots, N$$

It might seem that having multiple data values at the *identical* location in measurement or feature space might not be useful, since it is just the same data point, repeated one or more times. However there are a number of classifiers that *do* respond to repeated data, such as k NN.

The advantage of random duplication is that no assumptions are made regarding the shape of the class or the distribution of its data.

- Sample a point at random from the dataset and add noise:

$$y_{N+1} = y_r + \mathcal{N}(\underline{0}, \Sigma) \quad \text{Random index } r, \text{ Covariance } \Sigma$$

where Σ reflects the scaling behaviour of the measurement or feature space, as discussed in [Section 6.1](#). Adding noise does spread the points around, but requires Σ to be specified: if Σ is too small, we are really no further ahead of just duplicating points, and if Σ is too large then we may be inappropriately changing the shape of the class.

- Randomly interpolate between two proximate points in the dataset:

$$y_{N+1} = \alpha y_r + (1 - \alpha) y_{r'} \quad \text{Random index } r, \text{ Uniform random weight } 0 \leq \alpha \leq 1$$

such that index r is chosen at random, and then index r' is chosen at random among the k points closest to y_r . This approach is, in most cases, preferable to adding random noise, since selecting k is significantly simpler than specifying a covariance Σ .

Clearly if a given set of N points needs to be increased to βN points, then the above resampling needs to take place $(\beta - 1)N$ times.

DATA UNDERSAMPLING: Rather than increasing the number of data points for the under-represented class, we can *decrease* the number of points for the over-represented class, by repeatedly removing points at random until a representation parity objective is met.

Removing data points does have the disadvantage of having fewer data points for learning, but with a significant advantage is that no assumptions are made regarding any class shape.

In practice, we would not necessarily need to eliminate data points completely, rather just randomly select a subset of points for training, with the remaining point still in place for testing and validation, with the overall process undertaken repeatedly, as described for cross validation in [Section 9.3](#).

2. Change the Objective:

In seeking the best classifier, what objective are we attempting to optimize?

Repeating the example from [Section 8.5](#), suppose there is a rare medical condition, such that it is 99% likely that a given patient is healthy, and only 1% likely that they are ill. A classifier which *always* says “healthy” will achieve a classification accuracy of 99%, and yet accomplish nothing

whatsoever. That is, the overall accuracy (or probability of classification error) can be a very poor measure of performance, particularly when classes are highly imbalanced.

There are other objectives which are far more robust to imbalances in class likelihoods. In particular, the Bayes Risk, Balanced Accuracy, and Neyman-Pearson objectives, all discussed in [Section 9.2](#), offer alternatives to minimizing the probability of error.

- 3. Change the Classifier:** Almost certainly it would be preferable to learn a classifier based on a context which has been adjusted to lead to fair training. That is, we would prefer to learn a *new* classifier based on a rebalanced dataset or an appropriately chosen objective function.

As a poor second choice, we could modify an already-learned classifier by adjusting its thresholds, either directly adjusting the decision boundary in feature space, or indirectly via a discriminant threshold, for example adjusting threshold τ in [Figure 8.3](#) to more strongly favour the less likely class.

9.2 Classifier Evaluation

So far in this text, the criteria by which classifiers have been assessed has focused on a confusion matrix ([Chapter 3](#)) or probability of error ([Chapter 8](#)). However, as was already raised in the preceding section, there are many objectives by which classifier performance could be measured.

Given a classifier g , the corresponding confusion matrix from [\(3.19\)](#),

$$\begin{array}{c} & \text{Is classified as ...} \\ & C_1 \quad C_2 \quad \cdots \quad C_K \\ \text{True Class} & \begin{array}{|c|cccc|} \hline & C_1 & C_2 & \cdots & C_K \\ \hline C_1 & S_g(C_1|C_1) & S_g(C_2|C_1) & \cdots & S_g(C_K|C_1) \\ \vdots & \vdots & \vdots & & \vdots \\ C_K & S_g(C_1|C_K) & S_g(C_2|C_K) & \cdots & S_g(C_K|C_K) \\ \hline \end{array} \end{array} \quad (9.8)$$

is actually quite comprehensive, having all of the information regarding how a classifier $g()$ performs as a function of class C_κ . It is the *interpretation* of the confusion matrix and the formulation of a single scalar objective from the confusion matrix that benefits from further discussion, and *all* of the following metrics are just ways of understanding the performance implied by the values contained in the confusion matrix.

Although there are many multi-class ($K > 2$) problems, a large fraction of detection problems are binary ($K = 2$) hypothesis tests:

- Medical Test: Does the patient have disease X? (Yes/No)
- Radar: Is the airplane detected? (Yes/No)
- Biometrics: Is person Y looking at the camera? (Yes/No)

In this context we prefer to talk about classes C_0 and C_1 ,

$$\begin{aligned} C_0 &: \text{Negative / Quiescent / Default / Background} \\ C_1 &: \text{Positive / Active / Exception / Detection} \end{aligned} \quad (9.9)$$

such that (9.8) becomes a fairly modest 2×2 grid:

		Is classified as ...	
		C_0	C_1
True Class	C_0	$S(C_0 C_0)$	$S(C_1 C_0)$
	C_1	$S(C_0 C_1)$	$S(C_1 C_1)$

(9.10)

which is, almost universally, articulated in terms of the “Negative” and “Positive” classes as

		Is classified as ...	
		$C_0 = C_{\text{Negative}}$	$C_1 = C_{\text{Positive}}$
True Class	$C_0 = C_{\text{Negative}}$	True Negative (TN)	False Positive (FP)
	$C_1 = C_{\text{Positive}}$	False Negative (FN)	True Positive (TP)

(9.11)

The four types of outcomes (TN/FP/FN/TP) are interpreted graphically in Figure 9.1.

The question still remains: what function of the TN/FP/FN/TP values should be optimized in selecting a classifier? For example, the probability of classification error can be expressed as

$$\mathbf{P}(e) = \frac{\text{FN} + \text{FP}}{\text{TN} + \text{FP} + \text{FN} + \text{TP}} \quad (9.12)$$

However, as we know from Section 8.5, the probability of classification error may not be a suitable metric for certain types of problems. Indeed, as shown in Table 9.1, a great many different criteria have been formulated.

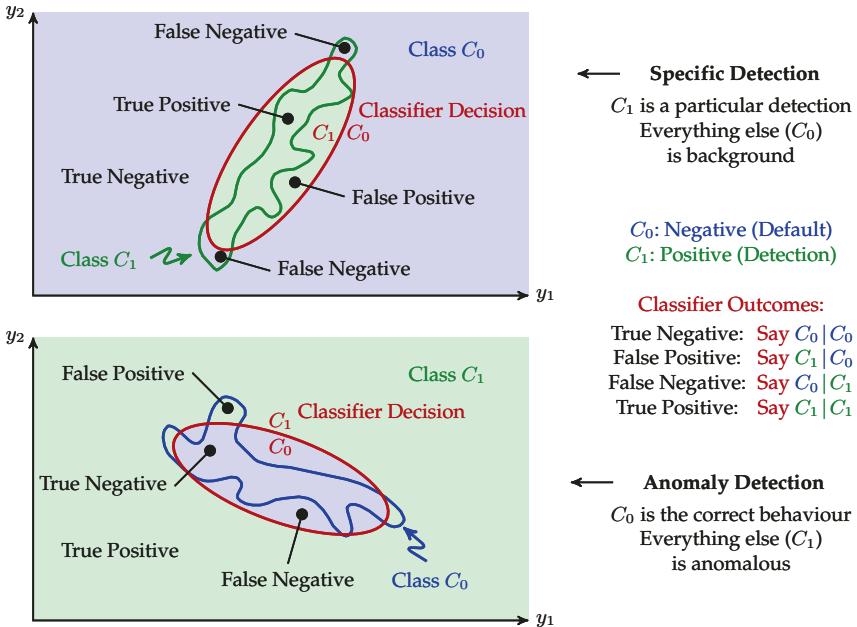


Fig. 9.1. DETECTION: In most detection problems there is some *thing* (disease in a patient, airplane on a radar screen, crack in a steel pipe) about which we are asking whether it is present (C_1) or not (C_0). Words like “Positive” and “Negative” need to be understood in the sense of making a detection, and not in the sense of whether the outcome is desirable or not. Both of the sketches (top, bottom) have the same interpretations, it is mostly a matter of whether we have a particular detection (an airplane), and everything else is clutter (top), or whether we have a particular notion of behaviour (no defect), and everything else is anomalous (bottom). In both cases a simple classifier (red) is proposed, leading to the four outcome permutations, as shown.

The interested reader can look up the individual criteria in [Table 9.1](#), as needed. Only a few will be examined here in more detail:

- The overall classification accuracy has already been discussed in [Section 8.5](#), where it was emphasized that there are circumstances in which accuracy is *not* necessarily the criterion to optimize. Hopefully seeing accuracy as only one of many criteria in [Table 9.1](#) helps to emphasize this point.
- The balanced accuracy is a simple criterion, seeking to minimize the sum of class-specific errors

$$\text{Maximizing Balanced Accuracy} = \text{Minimizing } \sum_{\kappa} \mathbf{P}(e | C_{\kappa}) \quad (9.13)$$

$$\begin{array}{ll} \text{True Negative (TN)} = S(C_0 | C_0) & \text{False Positive (FP)} = S(C_1 | C_0) \\ \text{False Negative (FN)} = S(C_0 | C_1) & \text{True Positive (TP)} = S(C_1 | C_1) \end{array}$$

$$\text{Negative Sample (N)} = \text{TN} + \text{FP} \quad \text{Positive Sample (P)} = \text{FN} + \text{TP}$$

$$\text{Accuracy} = \frac{\text{TN} + \text{TP}}{\text{N} + \text{P}} = 1 - \mathbf{P}(e) \quad \text{Error} = \frac{\text{FN} + \text{FP}}{\text{N} + \text{P}} = \mathbf{P}(e)$$

$$\text{Sensitivity, Recall, or True Positive Rate (TPR)} = \frac{\text{TP}}{\text{P}} = 1 - \text{FNR} = 1 - \mathbf{P}(e | C_1)$$

$$\text{Specificity, Selectivity, or True Negative Rate (TNR)} = \frac{\text{TN}}{\text{N}} = 1 - \text{FPR} = 1 - \mathbf{P}(e | C_0)$$

$$\text{Miss Rate or False Negative Rate (FNR)} = \frac{\text{FN}}{\text{P}} = 1 - \text{TPR} = \mathbf{P}(e | C_1)$$

$$\text{Fall Out or False Positive Rate (FPR)} = \frac{\text{FP}}{\text{N}} = 1 - \text{TNR} = \mathbf{P}(e | C_0)$$

$$\text{Positive Predictive Value (PPV)} = \frac{\text{TP}}{\text{TP} + \text{FP}} = 1 - \text{FDR} = \mathbf{P}(\text{True Class is } C_1 | \text{Classify as } C_1)$$

$$\text{Negative Predictive Value (NPV)} = \frac{\text{TN}}{\text{TN} + \text{FN}} = 1 - \text{FOR} = \mathbf{P}(\text{True Class is } C_0 | \text{Classify as } C_0)$$

$$\text{False Discovery Rate (FDR)} = \frac{\text{FP}}{\text{FP} + \text{TP}} = 1 - \text{PPV} = \mathbf{P}(\text{True Class is } C_0 | \text{Classify as } C_1)$$

$$\text{False Omission Rate (FOR)} = \frac{\text{FN}}{\text{FN} + \text{TN}} = 1 - \text{NPV} = \mathbf{P}(\text{True Class is } C_1 | \text{Classify as } C_0)$$

$$\text{Balanced Accuracy (BA)} = \frac{\text{TPR} + \text{TNR}}{2} \quad \text{F1 Score (F}_1\text{)} = 2 \cdot \frac{\text{PPV} \cdot \text{TPR}}{\text{PPV} + \text{TPR}}$$

$$\text{Informedness} = \text{TPR} + \text{TNR} - 1 \quad \text{Markedness} = \text{PPV} + \text{NPV} - 1$$

$$\text{Precision} = \text{Positive Predictive Value (PPV)}$$

Table 9.1. PERFORMANCE CRITERIA: Quite a number of criteria have been proposed, all based on the same four numbers in the 2×2 confusion matrix. Clearly there is significant redundancy/similarity between some of these.

That is, we treat the probability of error from each class as equally important, regardless of how likely or unlikely the class.

The Maximum Likelihood classifier of [Section 8.1](#) optimizes the balanced accuracy, since for each feature value x the ML classifiers selects that class which maximizes the likelihood of x , thereby minimizing the sum of all other classes, an argument which is made more precise in [Appendix D](#).

- The F_1 measure is the harmonic (multiplicative) mean of *precision* and *recall*:

PRECISION: The fraction of positive classifications which are correct;

RECALL: The fraction of positive samples which are correctly classified;

such that $0 \leq F_1 \leq 1$, where a score of 1 represents a perfect classifier. Since it is not inherently obvious (or necessarily correct) that precision and recall are of equal importance in a given classification problem, the generalization

$$F_\beta = (1 + \beta^2) \frac{Precision \cdot Recall}{(1 + \beta^2)Precision + Recall} \quad (9.14)$$

has been proposed, but which then requires the relative weight β to be supplied.

Many of the criteria in [Table 9.1](#) are not intended as optimization objectives; for example, the meaningless classifier

$$g(y) = C_1 \quad \left\{ \begin{array}{l} \text{Always say } C_1 \end{array} \right. \rightarrow \text{Sensitivity} = 1 \quad (9.15)$$

has a *perfect* sensitivity of 1.0, so maximizing sensitivity on its own is clearly not a helpful objective. Rather, the various criteria are frequently used as additional assessments, once a given classifier has been learned.

The confusion matrices of [\(9.10\)](#) and [\(9.11\)](#) actually have only two degrees of freedom, since

$$\mathbf{P}(\text{TN}) + \mathbf{P}(\text{FP}) = 1 \quad \mathbf{P}(\text{FN}) + \mathbf{P}(\text{TP}) = 1 \quad (9.16)$$

therefore it is particularly convenient to visualize classifier performance as a plot in two dimensions, comparing two criteria in pairs, as shown in [Figure 9.2](#).

The so-called Receiver Operating Characteristic (ROC), a name dating back to the early days of classification in the context of detecting radar returns, compares the true positive and false positive rates. Along the diagonal lie the classifiers which ignore the measurement y , for example any of the random classifiers

$$g(y) = \begin{cases} C_0 & \text{With probability } \theta \\ C_1 & \text{With probability } 1 - \theta \end{cases} \quad (9.17)$$

These are not useful classifiers, since the input measurement is ignored, however it is helpful to understand their location in the ROC diagram. There can be certain cases where a stochastic (not deterministic) classifier, which depends on y but also possesses random elements, can be useful, as explored further in [Problem 9.4](#).

Below the diagonal are the classifiers which are wrong more often than they are right; that is, they perform *worse* than just flipping a coin at random. There is a limit, however, in terms of how bad a classifier g can be, since we can easily construct the reversed classifier \bar{g} from g :

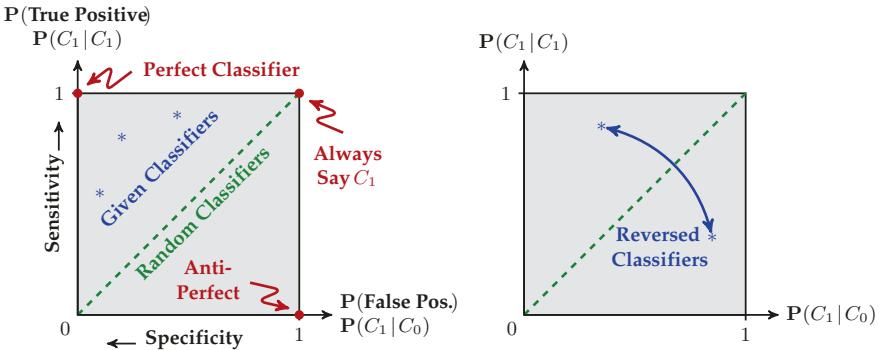


Fig. 9.2. The performance of a two-class classifier can be summarized by its sensitivity (true positives) and specificity (true negatives), left. The advantage of such a plot over a confusion matrix is that *many* classifiers can be compared simultaneously, by plotting their respective sensitivities and specificities. The *random* classifiers, choosing C_0 and C_1 at random with some probability, lie along the green dashed line. In general, very poor classifiers lie along the green dashed line, and *not* in the bottom right corner: the anti-perfect classifier in the bottom right, which is *always* wrong, is actually quite remarkable, since just saying the *opposite* of that classifier gives you a classifier which is *always* right. That is, there is a symmetry (right) above and below the green line, so that it is really only the points above the line that realistically matter.

$$\bar{g}(y) = \begin{cases} C_0 & \text{If } g(y) = C_1 \\ C_1 & \text{If } g(y) = C_0 \end{cases} \quad (9.18)$$

That is, given an anti-perfect classifier g , which is *always* wrong, we could easily construct the *perfect* classifier \bar{g} , which is *always* right. Because of this symmetry, shown in Figure 9.2, we normally consider only the upper-left half of the diagram.

Relative to the confusion matrices of (9.10) and (9.11), the most significant aspect of an ROC plot is that it can simultaneously compare *many* classifiers, as shown in Figure 9.3, whereas a confusion matrix specifies the performance for only a single classifier. It is for us to decide what we would like to plot on the False Positive – True Positive axes of the ROC plot:

- The curve traced out by a single parameterized classifier $g(y|\theta)$ as scalar parameter θ is varied;
- An ensemble of all possible classifiers (left panel of Figure 9.3)
- The upper-left (optimal) envelope of an ensemble of possible classifiers (right panel of Figure 9.3), the only classifiers of interest out of a given ensemble.

Given the ROC, whether a single curve $g(y|\theta)$ or a family of parameterized classifiers $g(y|\theta)$, how do we select the best classifier? This question is closely

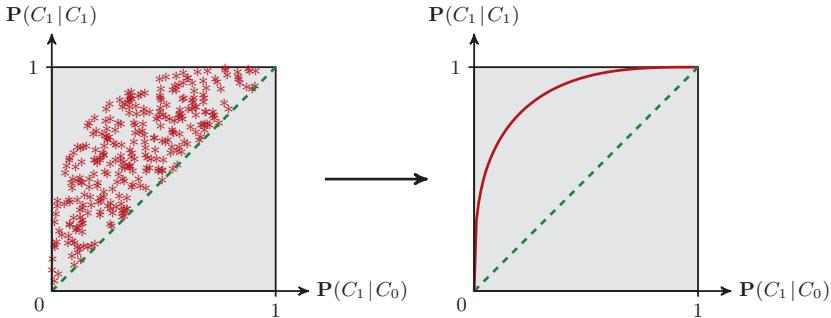


Fig. 9.3. Given a large ensemble of all possible classifiers (left), the only classifiers of interest to us are those on the upper-left envelope (right), since all classifiers *inside* the envelope are inferior to some point *on* the envelope, regardless of the performance criterion. The ROC (Receiver Operating Characteristic) curve, right, fully characterizes the family of classifiers available to us in a given problem.

related to the multi-objective optimization³ of Section C.4; that is, we have *two* objectives, to maximize sensitivity and to maximize specificity. Each point on the ROC will represent a classifier offering some tradeoff between sensitivity and specificity. It is therefore impossible to talk about the “best” classifier without first specifying a criterion/objective function/loss function to be optimized.

We have already seen several choices of classifier:

MAP: MAP minimizes $\mathbf{P}(e)$, the overall probability of error. Thus the minimization loss is

$$\mathcal{L}_{\text{MAP}} = \mathbf{P}(e) = \mathbf{P}(C_0)\mathbf{P}(e|C_0) + \mathbf{P}(C_1)\mathbf{P}(e|C_1) \quad (9.21)$$

ML/BALANCED: As we saw from (9.13), the ML classifier optimizes the balanced accuracy criterion of Table 9.1, thus the corresponding minimization loss is

$$\mathcal{L}_{\text{ML}} = \frac{1}{2}\mathbf{P}(e|C_0) + \frac{1}{2}\mathbf{P}(e|C_1) \quad (9.22)$$

BAYES RISK: The Bayes Risk classifier of Section 8.5 seeks to minimize the overall cost or risk of classification. In the two-class case considered in the ROC, the expected risk of (8.62) becomes

³ Optimization and concepts such as objective and loss functions are discussed in Appendix C.

Example 9.2: Receiver Operating Characteristics Examples

Suppose we have a one-dimensional problem, as in [Chapter 8](#), in which the class statistics $p(y|C_0)$, $p(y|C_1)$ are known, as plotted.

We might have been told to use ML or MAP, however in the absence of any particular direction we might be better off visualizing the ROC. We can construct the family of classifiers, based on decision threshold θ , as

$$g(y|\theta) = \begin{cases} C_0 & y < \theta \\ C_1 & y > \theta \end{cases} \quad (9.19)$$

leading to the ROC, as shown, computed using the Q function of [\(8.48\)](#). The endpoints of the curve correspond to

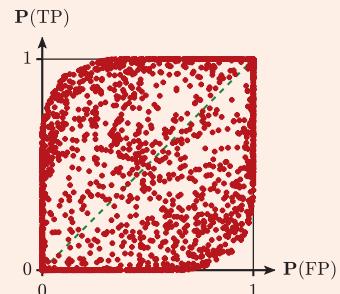
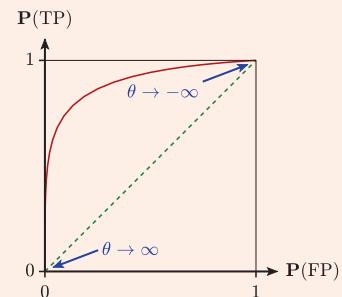
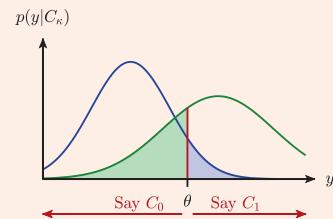
$$\begin{aligned} \theta \rightarrow -\infty : g(y|\theta) \text{ always says } C_1 \\ \theta \rightarrow +\infty : g(y|\theta) \text{ always says } C_0 \end{aligned} \quad (9.20)$$

The ROC is neutral, showing all possible classifiers; the ROC does *not* tell us which classifier is the optimum one, since optimality can only be claimed with respect to a selected criterion.

To better illustrate the distribution of a family of classifiers as had been shown in [Figure 9.3](#), suppose we consider the TwoArc dataset of [Lab 3](#). We could consider a wide variety of possible classifiers, but we will try something very simple, just a straight line separating the two classes. That is, vector $\underline{\theta}$ needs to parameterize all possible lines, for example

$$\underline{\theta} = \begin{bmatrix} x_{\text{Origin}} \\ y_{\text{Origin}} \\ \text{Angle} \end{bmatrix} .$$

2000 lines were selected at random, and for each the False-Positive and True-Positive statistics were found, resulting in the 2000 points in the ROC plot, as shown. We can see that classifiers exist which detect approximately two-thirds of class C_1 without *any* False-Positives, or that *all* of class C_1 can be detected with approximately one-third of False-Positives. That is, there is a tradeoff articulated in the figure, and there is no perfect classifier among the classifiers tested, which makes sense, because the classes are not linearly separable: there is *no* straight line which can separate the TwoArc classes.



$$\text{Minimizing } \alpha\mathbf{P}(e|C_0) + (1 - \alpha)\mathbf{P}(e|C_1) \quad 0 \leq \alpha \leq 1$$

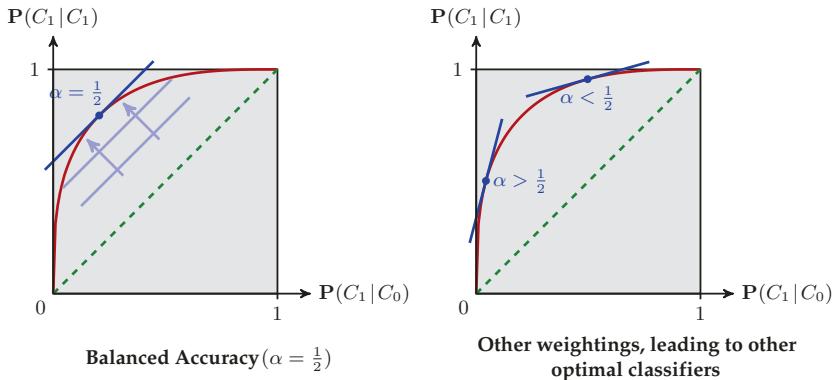


Fig. 9.4. ROC OPTIMIZATION: The optimum classifier will be a function of how we treat the relative significance of $\mathbf{P}(e|C_0)$ and $\mathbf{P}(e|C_1)$, essentially in the same way that MAP biases in favour of more likely classes, and the Bayes-Risk classifier biases to avoid the error of greater cost or risk. The contour of constant weighted errors or weighted risks is a straight line (blue) in the ROC domain, therefore the optimum classifier is then the tangent point (blue dot) of the ROC (red) with the constant-risk contour, as shown.

$$\begin{aligned} \mathbb{E}[R] &= R_{00}\mathbf{P}(C_0|C_0)\mathbf{P}(C_0) + R_{01}\mathbf{P}(C_0|C_1)\mathbf{P}(C_1) + \\ &\quad R_{10}\mathbf{P}(C_1|C_0)\mathbf{P}(C_0) + R_{11}\mathbf{P}(C_1|C_1)\mathbf{P}(C_1) \end{aligned} \quad (9.23)$$

$$\begin{aligned} &= R_{00}(1 - \mathbf{P}(C_1|C_0))\mathbf{P}(C_0) + R_{01}\mathbf{P}(C_0|C_1)\mathbf{P}(C_1) + \\ &\quad R_{10}\mathbf{P}(C_1|C_0)\mathbf{P}(C_0) + R_{11}(1 - \mathbf{P}(C_1|C_1))\mathbf{P}(C_1) \end{aligned} \quad (9.24)$$

$$\begin{aligned} &= (R_{10} - R_{00})\mathbf{P}(C_1|C_0)\mathbf{P}(C_0) + (R_{01} - R_{11})\mathbf{P}(C_0|C_1)\mathbf{P}(C_1) + \\ &\quad \text{constant} \end{aligned}$$

Ignoring the constant term, which does not affect the optimization, we have

$$\mathcal{L}_{\text{BayesRisk}} = (R_{10} - R_{00})\mathbf{P}(C_0)\mathbf{P}(e|C_0) + (R_{01} - R_{11})\mathbf{P}(C_1)\mathbf{P}(e|C_1) \quad (9.25)$$

All three cases result in an objective to minimize a weighted combination of the two errors

$$\mathcal{L} = \alpha\mathbf{P}(e|C_0) + (1 - \alpha)\mathbf{P}(e|C_1) \quad (9.26)$$

The contours of constant objective (that is, the contours of constant classifier “goodness”) are straight lines in the ROC domain, with a slope of $\alpha/(1 - \alpha)$. The optimal classifier, for any of MAP/ML/Bayes-Risk or other linear criteria, is where the line of slope $\alpha/(1 - \alpha)$ is tangent to the ROC envelope, as shown in Figure 9.4.

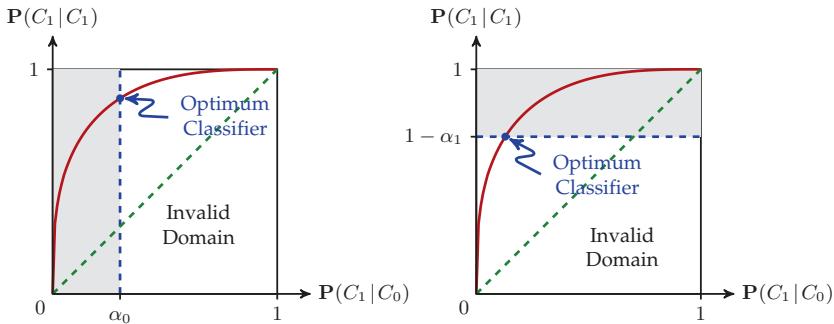


Fig. 9.5. NEYMAN-PEARSON: The ROC offers an intuitive approach to visualizing the constrained Neyman-Pearson classifiers of (8.66) and (8.67). Rather than minimizing the probability of error or a Bayes risk, the Neyman-Pearson approach specifies an upper bound on one class error (that of C_0 , left, or of C_1 , right) and then seeks the classifier maximizing the performance on the other class (C_1 , left; C_0 , right).

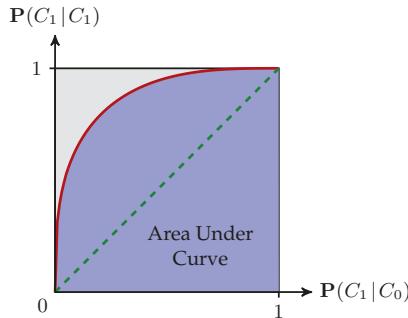


Fig. 9.6. In principle the ROC concept allows us to specify measures of quality for the entire *family* of classifiers implied by the ROC envelope, and not just the quality of a single classifier. For example the AUC, the integrated area under the ROC curve, is one such measure. Its drawback, however, is that not all points along the ROC are necessarily of interest in a given context, so the AUC may not measure the performance of the classifier that matters to us.

We are not limited to linear combinations of class errors. In particular, the Neyman-Pearson criteria of Section 8.5 can be particularly clearly articulated in the ROC domain, as shown in Figure 9.5, in which case we wish to place a hard constraint on the error for one class, and then find the classifier minimizing the error for the other class.

Finally, the ROC also allows measures of quality to be defined for the whole ROC curve; that is, articulating a metric of quality for the whole family of classifiers associated with the ROC. The simplest and most obvious of these is the AUC, the area under the curve, as plotted in Figure 9.6. The advantage

of such a metric is that we do not need to decide on a specific classifier in order to have an assessment of one classification approach versus another; the obvious disadvantage is the metric will be sensitive to the entire ROC curve, including those parts of the curve that may be entirely irrelevant to the context of interest.

9.3 Classifier Validation

Given a metric by which to evaluate a classifier, how do you use data to do so? That is, *selecting* a metric, the subject of the preceding section, is a very different task from *evaluating* a metric.

[Chapter 3](#) already raised the question of validation, particularly the overview in [Example 3.3](#) of the use of data in learning and validation, and the use of wrapper techniques in [Example 3.4](#). The reader may wish to review those examples before proceeding.

The remainder of this section will proceed from the simplest to more advanced strategies for validation, per the overview presented in [Figure 9.7](#).

1. Simplistic Train and Test: We could, ignoring the advice of [Chapter 3](#), just use the *same* dataset \mathcal{D} both for training and testing. Since the programming overhead to implement Holdout or Jackknifing is so modest, really using the same data for training and testing can never be defended, except as a superficial exploration in cases where it is known from the size $|\mathcal{D}|$ of the dataset and the number of degrees of freedom in the model that overfitting is unlikely to occur.

2. Holdout / Basic Validation: The most basic non-overfitting validation is to separate the given data \mathcal{D} into separate training $\mathcal{D}_{\text{Train}}$ and testing $\mathcal{D}_{\text{Test}}$ sets. Although overfitting is avoided, two problems remain:

- Both training and testing are undertaken with only a fraction of the data, and both are therefore less accurate compared with cross-validation, below.
- Certain data points are used *only* for training, and other data points *only* for testing, which can lead to biases.

3. q -fold Cross-Validation: If the given dataset is divided into q equal parts,

$$\mathcal{D} = \{\mathcal{D}_1 \cup \mathcal{D}_2 \cup \dots \cup \mathcal{D}_q\}$$

then the drawbacks of the holdout method can be eliminated by performing holdout validation q times, such that in iteration r , \mathcal{D}_r is removed from training and used for testing:

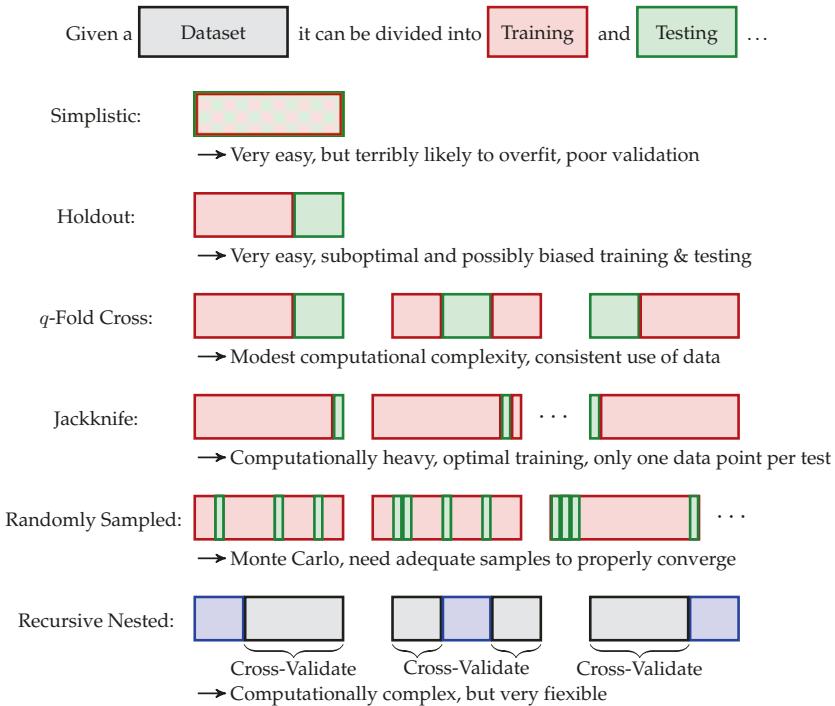


Fig. 9.7. CROSS-VALIDATION: An overview of validation and cross-validation, with training data shown in red and testing data in green. The methods shown here are representative of the main approaches, but many variations are possible on each of these. If some classifier parameter is learned through the cross-validation, then to avoid overfitting there needs to be *another* (recursive) layer of validation, here illustrated as 3-Fold Cross Validation (bottom), but any validation method could be applied recursively.

Training based on $\{\mathcal{D} \setminus \mathcal{D}_r\}$ and Testing based on \mathcal{D}_r

The performance metric (probability of error, balanced accuracy, or anything else) is computed for each iteration r , and then averaged over the q iterations. Compared to basic holdout, the q -fold cross-validation will have q times greater computational complexity, but with significant benefits:

- Each training is undertaken based on $(q - 1)/q$ (almost all) of the data, Thus training is close to optimal. Averaged over the q iterations *every* data point is used for testing, therefore a more accurate assessment than holdout.
- Every data point is used consistently: $(q - 1)$ times for training, and once for testing, greatly reducing biases.

4. Jackknife / Leave-one-out Cross-Validation: One of the most clever ideas in validation is known as leave-one-out or jack-knifing, the extreme limit of q -fold cross-validation in which $q = N$, the number of data points. In each iteration the leave-one-out approach says to perform training on $N - 1$ data points (all but one), and then to use the the *single* remaining data point for testing, to assess the learned model's degree of successful generalization. What is remarkable is that given N data points, leave-one-out is able to use $N - 1$ points for training *and* (over all iterations) N points for testing, all without overfitting or compromising the test assessment, but with a few drawbacks:

- The computational burden is N times as great as basic holdout. If N is small (a small dataset), the computational burden is modest, and the ability to maximize the number of points for training is highly attractive. For larger N the benefit of using $N - 1$ points for training is not so great, so q -fold cross-validation (with $q \ll N$) is preferred.
- Clearly a single data point reveals very little, in terms of assessment, and certain performance metrics (for example, involving the computation of a variance) require multiple test data points, in which case q -fold cross-validation is recommended.

The leave-one-out process is illustrated in Figure 9.8 and Algorithms 9.1 and 9.2.

The leave-one-out cross-validation is a special case of leave- r -out cross-validation, in which $N - r$ points are used for training, and r for testing. In principle, to avoid biases, all

$$\binom{N}{r} = \frac{N!}{r!(N-r)!} \quad (9.27)$$

permutations of r from N points need to be considered, a number of permutations so large as to be unrealistic for all but the smallest datasets. In general, if $r > 1$ cross-validation is required, it would be far more realistic to use q -fold cross validation (above) or random sampling (below).

5. Randomly Sampled Cross-Validation: If it is desired to use nearly all of the data points for training,

- Leave-one-out requires N training iterations;
- q -fold cross-validation requires N/q training iterations (and where q is small).

That is, both of the preceding approaches would be computationally quite heavy.

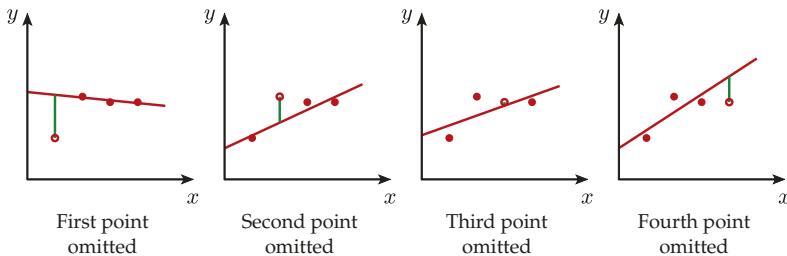


Fig. 9.8. LEAVE ONE OUT: Since straight-line fitting (regression) is a very intuitive and easily visualized idea, this figure illustrates leave-one-out cross-validation for regression. The Leave-One-Out approach maximizes the data available for training by testing on only one point. A data point (open circle) is removed from the training data, learning (red solid line) is based on the remaining data (filled circles), and the generalization error (green line) is based on the misfit between the learning and that data point which was *not* used in training. The process is then repeated, successively, over all data points. The algorithm is presented in [Algorithm 9.1](#).

Algorithm: Leave-One-Out Validation: Regression

Goals: Perform leave-one-out validation to fit N data points (x_i, y_i)

Function RMS_Error = LOO_Regression($\{x_i\}, \{y_i\}$)

ErrorSum = 0

for $j = 1 : N$ do

$f() = \text{Learn}(\{x_i, y_i | i \neq j\})$
ErrorSum += $(y_j - f(x_j))^2$

Learn fit on all points except (x_j, y_j)

Assess learning on omitted point

end for

RMS_Error = sqrt(ErrorSum/ N)

Algorithm 9.1. Leave-one-out validation, as illustrated in [Figure 9.8](#). The misfit in learning is assessed via root-mean-square error, however other penalty functions could have been used. The classification counterpart is shown in [Algorithm 9.2](#).

Although the number of iterations for leave- r -out, from [\(9.27\)](#), is impossibly many, we could use only a tiny fraction of those permutations, sampling r points at random, a so-called Monte-Carlo⁴ approach. Although an average over *all* permutations [\(9.27\)](#) is required to completely eliminate any bias, in practice only a modest number M of iterations of random sampling is needed to adequately reduce the bias to near zero. The result is attractive:

- Only $M \ll N$ training iterations are needed.
- Each training uses nearly all $(N - r)$ of the N data points for training.
- $r > 1$ points can be used for testing.

⁴ Monte Carlo methods are a broad class of approaches, whereby random sampling replaces exhaustive or systematic sampling. One simple example is explored in [Problem 9.6](#).

Algorithm: Leave-One-Out Validation: Classification

Goals: Perform leave-one-out validation to fit N data points (x_i, c_i)

Function Classification_Error = LOO_Classification($\{x_i\}, \{c_i\}$)

```

    Errors = 0
    for  $j = 1 : N$  do
         $g() = \text{Learn}(\{x_i, c_i | i \neq j\})$       Learn classifier on all points except  $(x_j, c_j)$ 
        if  $g(x_j) \neq c_j$  then
            Errors ++
        end if
    end for
    Classification_Error = (Errors/ $N$ )

```

Algorithm 9.2. The principle of leave-one-out is easy to understand in the context of linear regression, as in [Figure 9.8](#) and [Algorithm 9.1](#), however the algorithm for classification is similarly straightforward. Note that the performance metric which is computed in this example is the probability of classification error, but other metrics could be used instead.

6. Recursively Nested Cross-Validation: With reference to the learning in [Example 3.3](#), and particularly the wrapper approaches of [Example 3.4](#), suppose we have an unknown parameter θ , somehow part of the classification process, which we wish to optimize:

- Given θ , classifier $g(\underline{x} | \theta)$ is learned from the training data and the loss function \mathcal{L} , the selected performance metric, is assessed based on any of the cross-validation methods just discussed.
- The loss $\mathcal{L}(\theta)$ can be a function of θ , since different values of θ may lead to differently performing classifiers. In principle, we might estimate θ by finding the value minimizing the loss function:

$$\hat{\theta} = \arg_{\theta} \min \mathcal{L}(\theta) \quad (9.28)$$

- We cannot use the testing data to assess $\hat{\theta}$, since it was chosen based on \mathcal{L} , which was computed from the testing data. That is, the learned value of $\hat{\theta}$ may be overfitting the dataset.
- So now we need *another* testing dataset, *separate* from the classifier testing, in order to test how well $\hat{\theta}$ was learned, and whether it has overfit or not. This principle, whereby a hierarchy or sequence of test sets is used to progressively assess the next layer of learning, is the basis of nested cross-validation.

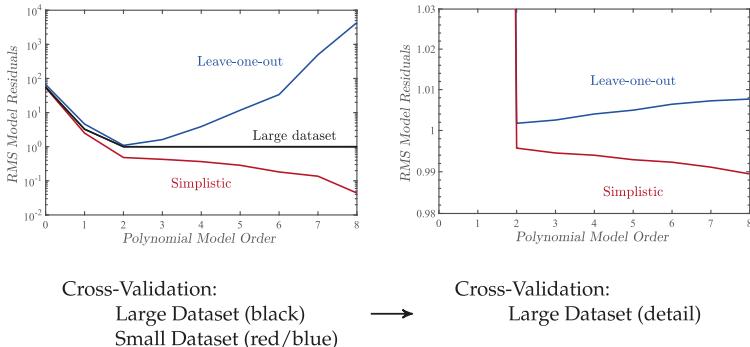


Fig. 9.9. CROSS-VALIDATION: The claim was made that simplistic testing and training leads to overfitting, and therefore cross-validation should be used. Building on the simple regression example of Figure 9.8 and the polynomial fitting problem of Figure 3.2, we consider fitting a polynomial to a set of points, which consists of a parabola plus unit-variance noise. The figure shows the cross-validated error (blue) and the simplistic error (red). Since unit-variance noise was added, a root-mean-square error below one implies a model which is overfitting to noise. For a very small dataset of only 10 data points (left: red/blue) the difference is striking: the simplistic test reports improved fit as polynomial order is increased, whereas cross-validation correctly identifies that the fit becomes *poorer* beyond the correct order of two. For a larger dataset the degree to which higher order polynomials fit the measurement noise is much smaller, since there are so many data points, so the cross-validated and simplistic curves are essentially identical (left: black). However the same overfitting phenomenon can still be seen when sufficiently zoomed in (right).

There is significant latitude in how each layer is cross-validated; that is, any cross-validation approach can be used, and in principle the approach could vary from one layer to the next.

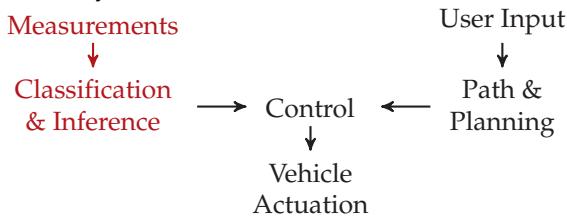
The whole rationale of performing cross-validation was to assess overfitting in learning. Figure 9.9 shows the result of applying leave-one-out to both small and large datasets for polynomial fitting. For both datasets and both simplistic and cross-validation assessment, the root-mean-square residual errors decrease rapidly to the correct model order of two. However beyond order two the simplistic test, which tests on the same data as were used for training, continues to report a drop in error (naively suggesting a better model), whereas cross-validation correctly identifies an *increase* in the error, meaning that the model is in fact generalizing more *poorly* for orders above two.

Case Study 9: Autonomous Vehicles

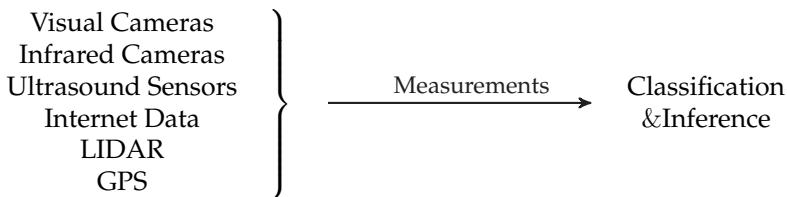
Long the domain of science fiction, the confluence of high-accuracy GPS, electric vehicles, ubiquitous high-speed internet, graphic processors (GPUs), cheap sensors, and huge improvements in machine learning, has led to the emergence of autonomous or self-driving / self-navigating vehicles.

There have been articles on nearly endless topics, such as the business opportunities (pizza delivery, driverless taxis), the impact on infrastructure, and how such vehicles could reshape cities and transportation. However, very much aligned with the topic of this chapter, there is clearly a huge testing / validation component to the development of autonomous vehicle systems! If there is an accident involving an autonomous vehicle, who carries the responsibility or the liability — the automation designer, the car manufacturer, or the person who owns (but is not driving) the vehicle?

With mechanical system or electrical sensor failures it is easier to diagnose and to pinpoint fault, compared to trying understand how a mountain of computer code and gigantic non-parametric classifiers reached some autonomous decision. Clearly measurement and classification are only one part of an autonomous vehicle system,



however a very large part of the validation challenge does stem from the measurement / classification parts of the system. A big part of the challenge stems from the sheer size of the measurement space \mathcal{Y} :



The processing of all this data is challenging, to be sure, however managing such data flows is part of the overall engineering design specifications, the number of frames per second to process for each camera etc. The *real* challenge of the huge measurement space lies in validation: with a space so vast, how can you possibly know how the classifier will respond to all possible inputs? At first glance, “*all possible inputs*” seems unnecessarily broad; after all, human

drivers do not encounter “all” possible inputs,⁵ rather the usual sequence of inputs which a human driver experiences: lane markings, road signs, and other road users.

Collecting data is not hard: we need cars, outfitted with the same sensors as autonomous vehicles, but then driven by humans. We can also perform data augmentation, as in [Example 9.1](#), by taking input datasets and applying car movement (translations, rotations) or simulating weather conditions (fog, rain, or snow). The problem is that the data collection and augmentation mostly produce regular / everyday data, such that the data do not include rare (problem) events, since serious accidents are very rare, and essentially impossible to simulate via data augmentation. As a result, the dataset will typically have very few serious outliers, those rare but important challenging circumstances where the classification decision can make the difference between continued driving and a major accident.

This challenge is related, but not quite identical, to the class-imbalance problem of [Section 9.1](#). Our classification / recognition problem does consist of a fair number of classes,

$$\mathcal{C} = \{\text{Pedestrian, Bicycle, Vehicle, Stop Sign, Do Not Enter, Railroad Crossing, ...}\} \quad (9.29)$$

however all of these classes will appear commonly enough in driving datasets, and we can choose to gather test data along road segments where such signs are known to occur. So the imbalance is not specifically one of a class being rare, rather more of a circumstance-imbalance or measurement-outlier-imbalance problem.

Now if a classifier is developed, based on *years* of driving data, and is robust to car movement (translation, rotation) and weather (fog, rain, snow), how is this not adequate?

1. Years of data is not necessarily all that thorough a test: There are over 1 billion cars in the world, thus a few car-years of data is only an infinitesimal fraction of the situations encountered by 1 billion cars, worldwide, *every day*. The proposed data augmentation will not introduce those highly unusual events, never seen in the data, and on which the classifier will never have been trained *nor* tested.
2. Autonomous vehicles would need to be robust not only to their surroundings, but also to deliberate attacks. As a highly complex and computerized system, it is easy to imagine attempts to hack or corrupt the system in some

⁵ Although who knows how many human-driver accidents are indeed caused by outliers, whether some sort of optical illusion, confused perception, or some highly unusual and unanticipated combination of pedestrians, cyclists, and road signs.

way. Clearly there could be hacks based on viruses etc., well outside the scope of this text, however suppose that there was a glaring sensitivity discovered in the classifier, such that a fairly simple pattern of bright and dark spots, for example, could lead the classifier to completely misunderstand a road sign, leading to an accident.

Although providing every possible input, or a long list of unusually challenging ones, may be impractical, there are other ways robustness might be tested. For example, for any measured input y associated with some class C , we can ask

What is the smallest perturbation \bar{y} such that $g(y + \bar{y}) \neq g(y) = C$ (9.30)

The size of this minimum perturbation can offer a measure of robustness to disturbance (whether deliberate or accidental).

Next, although there are many classes, we can simplify the analysis and validation by focusing on one class at a time. For example, for every *Stop* sign, we really only care whether the classifier recognized *Stop*, and not so much what other sign it saw instead. That is, we have the much simpler set of classes

$$\mathcal{C} = \{\text{Background / Unimportant, Stop Sign, Anything Else}\} \quad (9.31)$$

or possibly even the two-class case

$$\mathcal{C} = \{\text{Not Stop Sign, Stop Sign}\} \quad (9.32)$$

in which case the two-class and ROC analysis of [Section 9.2](#) can be applied.

By performing a set of two-classes analyses (each sign versus everything else), we can begin to better understand how such a large classifier might behave in practice.

Lab 9: Leave-One-Out Validation

[Lab 6](#) applied the Mahalanobis classifier to the Iris dataset, however the classifier assessment, via confusion matrix and overall probability of error, was based on the *same* data as for classifier training. That is, testing and training data were the same (the “Simplistic” approach of [Figure 9.7](#)), meaning that those error assessments were, in general, underestimated, and with no opportunity to assess classifier overfit-

ting. In this lab we would like to use the leave-one-out technique of [Section 9.3](#) to more correctly / objectively test a classifier.

Because this chapter focuses on validation, and not on the details of creating a classifier, it will be cleanest to assume the existence of a few helper functions:

```
% Return a distance-based classifier
function G = learn_classifier( classifier_type , training_points ,
    training_labels )
    % Possible uses:
    % learn_classifier( 'Mahalanobis', points, labels )
    % learn_classifier( 'Euclidean', points, labels )
    % learn_classifier( 'NN', points, labels )

% Return a vector of labels, one per point
function C = classify_points( classifier , points )

% Strip one or more points from a dataset
function Dataset_Remaining = remove_points( Dataset, indices_to_remove )
```

Given these helper functions, the resulting cross-validation is remarkably easy — precisely the reason that it should routinely be used:

```
load Iris

% Helper functions assume data points in columns
IrisData = IrisData';
K = 3;

% Loop over classification method
for method = {'Mahal', 'NN'},

    % Loop over number of dimensions to use
    for dims = 1:4,
        % Reset confusion matrix
        cm = zeros(K);

        % Wrapper for Jackknife, omit i'th data point
        for i=1:size(IrisData,2),
            IrisRemoved = remove_points( IrisData, i );
            G = learn_classifier( method{1}, IrisRemoved(1:dims,:) );
            IrisRemoved(5,:);
            C = classify_points( G, IrisData(1:dims,i) );
            cm(IrisData(5,i),C) = cm(IrisData(5,i),C) + 1;
        end

        % Extract overall probability of error from the confusion matrix
```

```

    perr = 1 - sum(diag(cm))/sum(cm(:))
end
end

```

We don't really have any need to show a large number of confusion matrices. Let us begin with the overfit and cross-validated probabilities of classification error:

Classifier	Validation	# Dimensions			
		$n = 1$	$n = 2$	$n = 3$	$n = 4$
Mean–Mahalanobis	Simplistic	28.0%	19.3%	4.0%	2.0%
Mean–Mahalanobis	Jackknife	28.0%	19.3%	4.7%	2.7%

We do observe that the cross-validated approach has probabilities of error greater than or equal to those of simplistic, but only very slightly. The reason is clear: the Mean–Mahalanobis classifier has a fairly modest parameterization. It cannot carefully fit the details of the class shape, since it is limited to fitting an ellipsoid, a fairly restrictive shape which limits overfitting.

The picture is *very* different, even shockingly so, if we apply Nearest Neighbour:

Classifier	Validation	# Dimensions			
		$n = 1$	$n = 2$	$n = 3$	$n = 4$
Nearest Neighbour	Simplistic	0.0%	0.0%	0.0%	0.0%
Nearest Neighbour	Jackknife	41.3%	29.3%	6.7%	4.0%

Since Nearest Neighbour essentially memorizes all of the data points, unless multiple data points from different classes are located at *exactly* the same point in feature space, a simplistic (and badly overfit) assessment of Nearest Neighbour is guaranteed to find *no* mis-classifications, thus a probability of error of zero. In striking contrast, the jackknife cross-validated results show Nearest Neighbour to perform rather poorly, significantly more poorly than Mean–Mahalanobis.

We can see the difference laid out more completely in the confusion matrices for the $n = 2$ dimensional case:

Mean–Mahalanobis		Nearest Neighbour	
Simplistic	Jackknife	Simplistic	Jackknife
Classified	Classified	Classified	Classified
Truth	Truth	Truth	Truth
$\begin{bmatrix} 49 & 1 & 0 \\ 0 & 34 & 16 \\ 0 & 12 & 38 \end{bmatrix}$	$\begin{bmatrix} 49 & 1 & 0 \\ 0 & 34 & 16 \\ 0 & 12 & 38 \end{bmatrix}$	$\begin{bmatrix} 50 & 0 & 0 \\ 0 & 50 & 0 \\ 0 & 0 & 50 \end{bmatrix}$	$\begin{bmatrix} 49 & 1 & 0 \\ 0 & 32 & 18 \\ 0 & 25 & 25 \end{bmatrix}$
$P(e) = 19.3\%$	$P(e) = 19.3\%$	$P(e) = 0.0\%$	$P(e) = 29.3\%$

The jackknife cross-validation was applied here because the code is very simple (a single for-loop wrapper) and because there are few data points. In more realistic settings with large datasets we would probably prefer q -fold cross-validation.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links — Data Resampling: [Resampling](#), [Jackknife resampling](#), [Bootstrapping](#), [Data augmentation](#)

Wikipedia Links — Evaluation Metrics: [Precision and recall](#), [F-score](#), [Sensitivity and specificity](#), [Receiver operating characteristic](#)

Wikipedia Links — Classifier Validation: [Cross-validation](#), [Jackknife resampling](#)

Sample Problems

Problem 9.1: Short Answer

Give a short definition of each of the following:

- Data augmentation
- Outlier
- Nonstationarity
- Class imbalance problem
- Confusion matrix
- Receiver operating characteristic
- Neyman-Pearson criterion
- Jackknifing

Problem 9.2: Short Answer

Offer brief answers to each of the following:

- (a) What is data augmentation, why do we do it, and what are a few strategies for undertaking it?

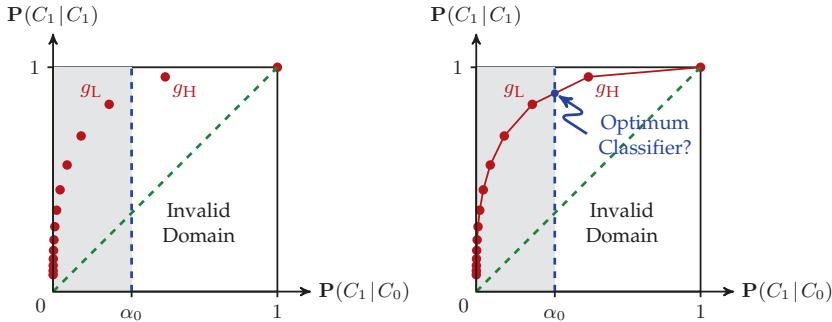


Fig. 9.10. DISCRETE-VALUED ROC: If our measurement is discrete-valued, as in Problem 9.4, then the ROC of Figure 9.5 becomes a sequence of discrete points (left). In the context of Neyman-Pearson, interpreting such an ROC becomes a little unusual. The Neyman-Pearson constraint α_0 is bracketed by classifiers g_L and g_H . The latter classifier, g_H , is invalid, so presumably g_L is the optimal solution. In actual fact, weirdly, a *better* classifier than g_L can be accomplished by a *randomized* test (right), essentially interpolating (solid red line) between g_L and g_H .

- (b) What is the class imbalance problem, and what are a few approaches to address it?
- (c) What does a Receiver Operating Characteristic (ROC) plot allow us to interpret, as opposed to numerical criteria such as accuracy, precision, or recall?
- (d) Explain clearly how it is that cross-validation allows us to infer the degree to which a given classifier is overfitting its training data or not.

Problem 9.3: Conceptual — Error

Suppose we are given a problem with K classes in a $m = 3$ dimensional measurement space:

$$y | C_\kappa \sim \mathcal{N}(\mu_\kappa, \Sigma_\kappa) \quad 1 \leq \kappa \leq K \quad (9.33)$$

We are also given proposed one- and two-dimensional feature extractors

$$x^{(1)} = f_1(y) \quad x^{(2)} = f_2(y) \quad (9.34)$$

- (a) In general, what can we say about $\mathbf{P}_{\text{MAP}}(e)$ in the original three-dimensional measurement space, relative to $\mathbf{P}_{\text{MAP}}(e)$ in the one- and two-dimensional feature spaces?
- (b) Does your answer in (a) depend on the choices of μ_κ and Σ_κ or on the selection of feature extractors $f_1(), f_2()$?

Problem 9.4: Conceptual / Analytical — Error

For nearly every problem in this text, the measurements have been vectors of real values,⁶ $y \in \mathbb{R}^m$. However it is possible for a measurement to be discrete (an integer), particularly in the context of counting.

In particular, consider [Problem 7.12](#). In assessing radioactive decay using a Geiger counter, normally we will *not* be given the decay times, rather the *number* of decays (clicks) over some time interval. There are a great many related counting problems, such as the number of cars on a road per hour, the number of people arriving at a grocery store per hour etc.

Back to the two classes from [Problem 7.12](#):

$$\begin{aligned} \text{Class } C_1: & \text{ Uranium 235 Half life } 7.0 \cdot 10^8 \text{ years} \\ \text{Class } C_2: & \text{ Uranium 238 Half life } 4.5 \cdot 10^9 \text{ years} \end{aligned}$$

We are given a sample containing $1 \cdot 10^{17}$ atoms, all from either Class C_1 or C_2 . You can easily work out the *rate* of decays (clicks per second) for each class. Let y be the *number* of decays (an integer) measured over 10 seconds in which case, as derived in [Appendix D](#), y will be Poisson distributed. We can define a classifier

$$g(y) = \begin{cases} C_1 & y > \tau \\ C_2 & y \leq \tau \end{cases} \quad (9.35)$$

- (a) Write the distribution $y|C_1$ and $y|C_2$ for each of the classes.
- (b) Sketch the ROC corresponding to the classifier in (9.35). Since y is an integer, the ROC will be a series of points, as illustrated in [Figure 9.10](#), rather than a curve. Rather than sketching, you may wish to plot the ROC, computationally, based on the distributions in (a).

From [Figure 9.10](#), if the Neyman-Pearson criterion α_0 lies between two classifiers, then presumably we need to select classifier g_L , since the next classifier, g_H , does not satisfy the criterion. On the other hand, presumably g_L is not optimal, since a classifier interpolated partway between g_L and g_H would have a higher sensitivity and still meet criterion α_0 .

What is an interpolated classifier? Essentially one selects between two classifiers at random:

$$\text{Given } u \in \mathcal{U}(0, 1) \quad g_{\text{Interp}}(y) = \begin{cases} g_L(y) & u \leq \beta \\ g_H(y) & u > \beta \end{cases} \quad (9.36)$$

where $0 \leq \beta \leq 1$ controls the relative weights of g_L and g_H in forming g_{Interp} .

⁶ With significant exceptions under vector embedding in [Section 5.2](#) and [Chapter 13](#).

- (c) Propose one or two values of α_0 and identify the corresponding interpolated optimal Neyman-Pearson classifiers.

Problem 9.5: Analytical — ROC

Suppose we have a simple one-dimensional, symmetric, two-class Gaussian problem

$$x|C_1 \sim \mathcal{N}(-\alpha, 1) \quad x|C_2 \sim \mathcal{N}(\alpha, 1) \quad (9.37)$$

In [Chapter 6](#) or [Chapter 8](#) a specific classifier would be found, based on some criterion such as minimizing distance or error. In contrast, for an ROC plot we wish to explore the tradeoff of a whole *family* of classifiers,

$$g(x) = \begin{cases} C_1 & x \leq \tau \\ C_2 & x > \tau \end{cases} \quad (9.38)$$

Thus we have a *problem* parameterized by α , and a *classifier* parameterized by τ . Note that we do *not* need to know the class prior probabilities in order to generate an ROC plot.

- (a) Derive $\mathbf{P}(e|C_1)$ and $\mathbf{P}(e|C_2)$ as a function of α and τ
- (b) Draw a rough sketch, superimposing the ROC plots for $\alpha = \frac{1}{2}, 1, 2$.
- (c) Compute and plot, numerically, the ROC plots for $\alpha = \frac{1}{2}, 1, 2$.

Problem 9.6: Computational — Monte Carlo Methods

Monte Carlo methods are a broad, fascinating range of approaches which use *randomness* to answer problems which do not appear to have anything necessarily random about them.

For example, as sketched in [Figure 9.11](#), a random variable uniformly placed in a square has a probability of $\pi/4$ of lying within the inscribed circle. The key idea is that we can *test* whether a point lies within the circle without knowing π , since

$$x_1, x_2 \sim \mathcal{U}(-1, 1) \longrightarrow (x_1, x_2) \text{ lies in the circle if } x_1^2 + x_2^2 \leq 1 \quad (9.39)$$

Therefore if N points are uniformly dropped into a square, at random, we can count the number, M , which lie inside the circle, and then from [Section 7.2](#) on [page 157](#) we can estimate the probability as

$$\mathbf{P}(\text{Point in Circle}) = \frac{\pi}{4} \quad \hat{\mathbf{P}}(\text{Point in Circle}) = \frac{M}{N} \quad \longrightarrow \quad \hat{\pi} = \frac{4M}{N} \quad (9.40)$$

Develop a program to undertake the Monte Carlo random sampling, as just described, to estimate π . Run your code repeatedly to plot the standard deviation of the estimation error as a function of N .

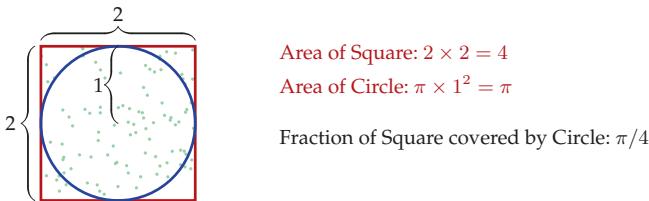


Fig. 9.11. How to know the value of π : By dropping random points (light green) within a square, and seeing what fraction lies within the circle, leads to a simple Monte Carlo approach to estimating π , as discussed in [Problem 9.6](#).

To be sure, this is *far* from the most efficient way of computing the value of π , since the error bars around $\hat{\pi}$ converge as $1/\sqrt{N}$. Many other methods related to the computing of π can be found [online](#).

Problem 9.7: Numeric/Computational — Jackknife and Multi-Fold Validation

From a programming perspective, Jackknife and multi-fold validation are exceptionally simple to implement — essentially just a single for-loop wrapper around an existing classifier learning.

Suppose we have two classes

$$C_1 \sim \mathcal{N}(-1, 2) \quad C_2 \sim \mathcal{N}(1, 2) \quad (9.41)$$

Find or implement several classifiers subject to varying degrees of overfit. A good suggestion might be to try NN (highly overfit), 2NN, and MED (very limited overfit).

For each of the three classifiers, assess the classifier probability of error based on

- The training set itself
- Jackknifing based on the training set
- 5-Fold Cross-Validation based on the training set
- Based on an additional test set

Compare the results of the four validations on each of the three classifiers. What do you observe?

Problem 9.8: Real World, Open-Ended — Validation

Novel methods of classifier learning can introduce challenges to validation. The criteria of [Section 9.2](#) are still of interest, however the strategies of [Section 9.3](#) may be more difficult to apply.

Prepare a short overview of some current state-of-the-art concepts or challenges in the field of validation. A few possible starting points:

- Validation for Distributed or Federated Learning: If a classifier was learned in a distributed sense, over a number of datasets in different places, what does it mean to validate? What dataset is to be used for testing, if there was no central dataset to begin with?
- Validation for Very Large Datasets: Suppose a classifier is developed for video analysis. The datasets can be enormously large, since each individual datum (a video) could possibly be a whole movie, and thus many GB in size. How does one test systematically / comprehensively when individual data points are large, making it difficult to test large numbers?
- Validation for Very Large Classifiers: Suppose a large neural network classifier ([Chapter 11](#)) has been learned. Is validation affected by whether we have access to the network internals (white-box) or just the overall inputs and outputs (black-box)?

References

1. M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera. A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. *IEEE Trans. Systems, Man, and Cybernetics C* **42**(4) (2012)
2. M. Mazurowski, M. Buda, A. Maki, A systematic study of the class imbalance problem in convolutional neural networks. *Neural Netw.* **106** (2018)



Discriminant-Based Classification

Earlier chapters of this text developed two types of classifiers:

1. Distance-based classifiers of [Chapter 6](#), which required the selection of a distance metric and a prototype;
2. Statistics-based classifiers of [Chapter 8](#), which required the inference or knowledge of class distributions and (in the Bayesian case) prior probabilities.

In all cases, whether statistical or non-statistical, whether Bayesian or non-Bayesian, we eventually arrive at some sort of function, the actual classifier, which (in the binary/two-class case) can be written in the form

$$h(\underline{x}) \begin{array}{c} \text{Classify as } C_1 \\ \gtrless \\ \text{Classify as } C_2 \end{array} 0 \quad \xleftarrow{\text{Equivalent}} \quad g(\underline{x}) = \begin{cases} C_1 & h(\underline{x}) > 0 \\ C_2 & h(\underline{x}) < 0 \end{cases} \quad (10.1)$$

That is, we have a relationship between a so-called discriminant function $h(\underline{x}) \in \mathbb{R}$ and its associated classifier $g(\underline{x}) \in \mathcal{C} = \{C_1, C_2\}$. Because of the equivalence from (10.1), both $h()$ and $g()$ will be referred to as a “discriminant,” and whether we are referring to the deciding function or its resulting classifier should be clear from the notation and context. The classification or decision boundary of $g(\underline{x})$ is found as that set of points satisfying $h(\underline{x}) = 0$, as illustrated in [Figure 10.1](#).

We have seen classifiers written in discriminant form already. For example, rearranging the terms in (6.25) gives us the two-class Mahalanobis classifier

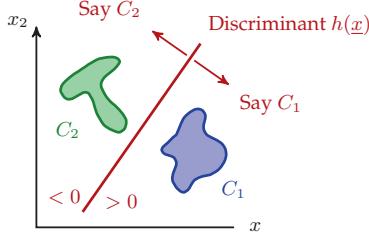


Fig. 10.1. DISCRIMINANTS: A discriminant is just a classifier, but learned without necessarily any particular regard to notions of distance, prototypes, or probability distributions. The sign of discriminant $h()$ naturally separates the feature space into two halves, allowing classification into C_1 and C_2 .

$$h_{\text{Mahal}}(\underline{x}) = -(\underline{x} - \mu_1)^T \Sigma_1^{-1} (\underline{x} - \mu_1) + (\underline{x} - \mu_2)^T \Sigma_2^{-1} (\underline{x} - \mu_2) \begin{array}{l} \xrightarrow{\text{Say } C_1} \\ \gtrless 0 \\ \xrightarrow{\text{Say } C_2} \end{array} \quad (10.2)$$

or similarly from (8.21) the two-class MAP classifier

$$h_{\text{MAP}}(\underline{x}) = \frac{p(\underline{x}|C_1)}{p(\underline{x}|C_2)} - \frac{\mathbf{P}(C_2)}{\mathbf{P}(C_1)} \begin{array}{l} \xrightarrow{\text{Say } C_1} \\ \gtrless 0 \\ \xrightarrow{\text{Say } C_2} \end{array} \quad (10.3)$$

The question of this chapter is whether we can *directly* learn $h(\underline{x})$. That is, do we really need distance metrics and prototypes or probability distributions, when in the end we always end up with some sort of discriminant $h()$? Perhaps we can save ourselves a great many assumptions and mathematical derivations by starting right at the end, by directly finding the discriminant function (or, equivalently, the classification boundary).

We are essentially back to feature extraction, as in [Chapter 5](#), seeking a simplified feature

$$x' = h(\underline{x}|\theta) \quad (10.4)$$

such that the given classes are separable in feature x' . Indeed, the language of discriminants was already used in [Chapter 5](#), in the sense of Fisher's discriminant of [Figure 5.6](#). However, aside from certain heuristics (minimizing class size or maximizing overall spread), [Chapter 5](#) did not identify systematic ways of learning discriminants, in general. As a result, what we need to solve is a parameter estimation problem, of [Section 7.1](#), in that we wish to learn a discriminant function

$$h(\underline{x}|\theta) \begin{array}{l} \xrightarrow{\text{Classify as } C_1} \\ \gtrless 0 \\ \xrightarrow{\text{Classify as } C_2} \end{array} \quad (10.5)$$

by optimizing over parameter θ , which somehow controls the choice of discriminant.

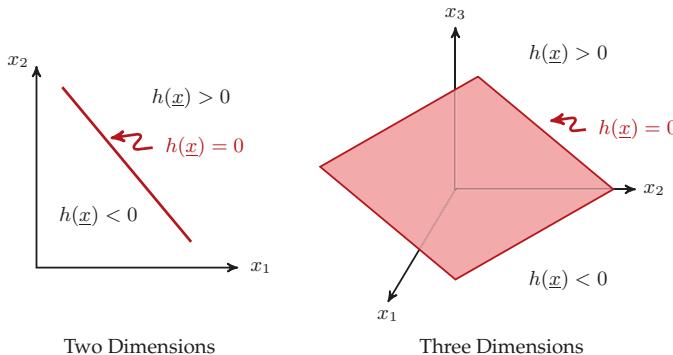


Fig. 10.2. LINEAR DISCRIMINANTS: A linear discriminant is a linear function $h()$ of feature \underline{x} . The resulting classification boundary, corresponding to $h(\underline{x}) = 0$, is then a straight line in 2D (left) or a plane in 3D (right). The sign of $h(\underline{x})$ determines on which side of the line or plane a given point \underline{x} is located.

So what form of discriminant might we like to learn? The Mahalanobis classifier (10.2) is quadratic, as were the ML/MAP classifiers in the Gaussian case (as developed in Section 8.3), so the quadratic case is already well explored. The ML/MAP classifiers in the *non-Gaussian* case could be essentially *any* function, which starts to feel very open-ended, and the reader will recall some of the overfitting pitfalls associated with learning high-order models in Chapter 3.

So, to avoid overcomplicating the discussion we are motivated to start with the simplest possible case: a *binary* discriminant based on a *linear* function. The generalization to multiple discriminants, nonlinear discriminants, and having more than two classes will be addressed later in this chapter.

10.1 Linear Discriminants

A linear function of feature x takes the form

$$\theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n + \theta_{n+1} \quad (10.6)$$

which we can write as a discriminant $h()$ parameterized by $\underline{\theta}$ as

$$h(\underline{x} | \theta) = \underline{w}^T \underline{x} + w_o \quad \theta = \begin{bmatrix} \underline{w} \\ w_o \end{bmatrix} \quad (10.7)$$

In two dimensions, the boundary implied by the linear discriminant can be found as

$$w_1x_1 + w_2x_2 + w_o = 0 \quad \rightarrow \quad x_2 = -\frac{w_1x_1 + w_o}{w_2}, \quad (10.8)$$

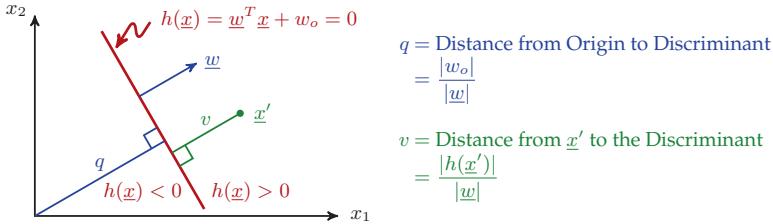


Fig. 10.3. HYPERPLANE GEOMETRY: Any hyperplane (here shown as a line in two feature dimensions) is characterized by its normal vector \underline{w} and offset w_o . The normal vector points in the direction in which $h(\underline{x}) > 0$.

the equation for a line. Similarly, in three dimensions, we recognize

$$w_1x_1 + w_2x_2 + w_3x_3 + w_o = 0, \quad (10.9)$$

as the equation for a plane, as shown in Figure 10.2. In n dimensions we will refer to the linear discriminant boundary as a hyperplane of infinite extent, separating the feature space \mathcal{X} into two halves. Since it is exceptionally difficult to visualize geometry in high dimensional spaces, we will normally choose to visualize a hyperplane as a flat plane in three dimensions.

Every line/plane/hyperplane is characterized by its normal vector \underline{w} and offset w_o . The normal vector is at right angles to the hyperplane and points in the positive direction, meaning that the half of \mathcal{X} pointed to by \underline{w} will satisfy $h(\underline{x}|\underline{\theta}) > 0$, as shown in Figure 10.3. That is, the sign of $h(\underline{x}|\underline{\theta})$ is our classifier, telling us which side of the hyperplane we are on; how far a point \underline{x}' is from the hyperplane, in Euclidean distance, can be derived as

$$d_E(\underline{x}', \{h(\underline{x}) = 0\}) = \frac{|h(\underline{x}')|}{|\underline{w}|}. \quad (10.10)$$

It needs to be emphasized that a single linear discriminant is a very basic classifier. As shown in Figure 10.4, two classes can be correctly classified by a linear discriminant only if the classes are *linearly separable*, which is clearly not the case in all problems, for example the Iris classes of Lab 2. However even in constraining our attention to linear discriminants, the challenge is still far from trivial:

- Given data for two classes, it is quite difficult to know whether they are linearly separable (particularly in cases of high-dimensional data);
- Given data for two classes known to be linearly-separable, it can be difficult to find the separating hyperplane.

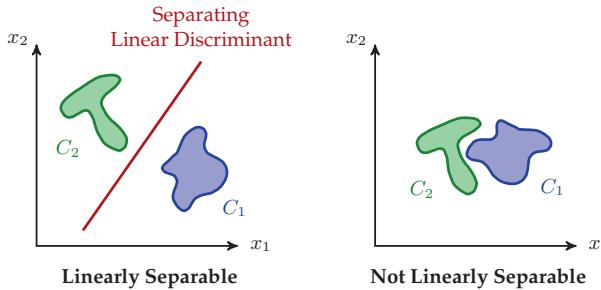


Fig. 10.4. LINEAR SEPARABILITY: Two classes are said to be *linearly separable* if a linear discriminant (a hyperplane) is able to perfectly separate the classes, with *all* of the C_1 data points on the positive side of the hyperplane, and *all* of the C_2 data points on the negative side. Linear separability seems like a trivially simple idea in two dimensions, as sketched here, but is actually very difficult to validate for high-dimensional problems.

10.2 Discriminant Model Learning

Since it is the sign of $h(\underline{x} | \underline{\theta})$ which classifies, as in Figure 10.1, clearly our objective in learning an effective discriminant $h(\underline{x} | \underline{\theta})$ (by learning an effective choice of parameter $\underline{\theta}$) is to place the hyperplane *between* the two classes.

In the parameter estimation context of Chapter 7 we had a complete statistical description of the problem, and could therefore talk about the *optimal* parameter choice in the ML or MAP sense. Here we have no statistical description of any kind; rather we have N labelled training data for the two classes,

$$(\underline{x}_i, c_i)_{i=1,\dots,N} \quad \text{where} \quad c_i \in \{C_1, C_2\} \quad (10.11)$$

and we assume a parameterized form for the discriminant $h(\underline{x} | \underline{\theta})$, for now assumed to be linear, but which could also possibly be nonlinear. We have two items to figure out before obtaining a proposed choice of $\underline{\theta}$:

THE OBJECTIVE FUNCTION: What does it mean for a value of $\underline{\theta}$ to be “good” or “bad”? We need some sort of penalty, often referred to as an *objective* or *loss function*¹ $\mathcal{L}()$ such that we have some rationale by which the “best” parameter can be selected:

$$\underline{\theta}_{\text{Optimum}} = \arg_{\underline{\theta}} \min \mathcal{L}(\underline{\theta}) \quad (10.12)$$

So we need to propose the objective $\mathcal{L}()$ to assert.

THE OPTIMIZATION STRATEGY: Given an objective function $\mathcal{L}()$ to minimize, how will the minimum be found? ... analytically vs. computationally,

¹ An introduction to optimization is provided in Appendix C.

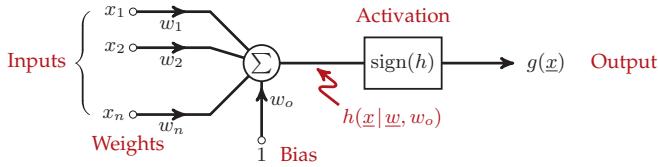


Fig. 10.5. The PERCEPTRON was developed as a very basic model of a human neuron. A set of inputs x_j is multiplied by weights w_j , summed, and offset by a bias w_o to produce linear discriminant h . Neurons are either on or off, like a binary classifier, so the discriminant is passed through a nonlinear activation function (here the sign function of Figure 10.6), to produce the resulting classification g . See the further discussion in Example 11.2.

exactly vs. approximately, iteratively vs. directly, a generic optimization algorithm vs. an approach taking advantage of known properties of the problem? Discussing all of the permutations of the options here is far beyond the scope of this text, however the options do serve to illustrate the difficulty in how to decide what algorithm might be used to minimize the selected objective $\mathcal{L}()$.

We begin with the objective function. What is it that we wish $\mathcal{L}()$ to accomplish? Well, we would like a hyperplane which effectively separates the data points in the two given classes, as described in (10.1) and Figure 10.1. That is, we would like to *minimize* the number of points on the *wrong* side of the hyperplane from both C_1 and C_2 :

$$\mathcal{L}_{\text{Count}}(\underline{\theta}) = -(\# \text{ Points on Correct Side}) + (\# \text{ Points on Incorrect Side}). \quad (10.13)$$

If the two classes are linearly separable (Figure 10.4) then \mathcal{L} can be minimized to $-N$; if the two classes are not linearly separable then the minimum possible value will be some integer larger than $-N$.

We can implement (10.14) by understanding that it is the sign of h which determines whether a given point is on the correct or incorrect side:

$$\mathcal{L}_{\text{Count}}(\underline{\theta}) = - \underbrace{\sum_i \delta(c_i, C_1) \text{sign}(\underbrace{h(\underline{x}_i | \underline{\theta})}_{< 0 \text{ if bad}})}_{\# \text{ bad} - \# \text{ good points in } C_1} + \underbrace{\sum_i \delta(c_i, C_2) \text{sign}(\underbrace{h(\underline{x}_i | \underline{\theta})}_{> 0 \text{ if bad}})}_{\# \text{ bad} - \# \text{ good points in } C_2} \quad (10.14)$$

where $\delta()$ is the standard delta function, as in (3.3).

The representation in (10.14) is correct, but feels a bit cluttered, since we have a separate sum for each class, and then a delta function to test each data point. It would be more elegant to just have a single sum over all of the data

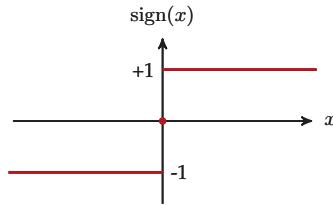


Fig. 10.6. The SIGN FUNCTION looks innocent enough ... a very easy idea, just keeping the sign (positive or negative) of its input argument x . However $\text{sign}()$ is nonlinear; even worse, it is discontinuous, making it challenging to optimize if present in an objective function.

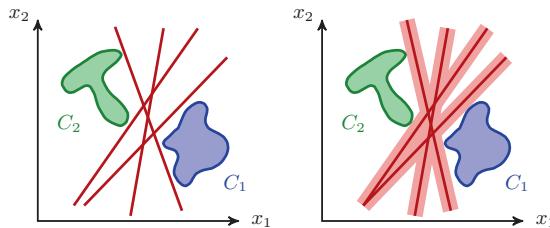


Fig. 10.7. OPTIMAL SEPARABILITY: These two classes linearly separable, therefore a separating hyperplane exists. In fact, there are *infinitely* many different hyperplanes, of which four are shown (left), however presumably not all of these hyperplanes are equally desirable, in terms of leading to a robust classifier. Asserting a margin (right) *does* push the hyperplane away from the class, but still does nothing to find the *best* hyperplane.

points, and to ask the appropriate question of each datum. If we create a set of constants \bar{c}_i ,

$$\bar{c}_i = \begin{cases} 1 & \text{if } c_i = C_1 \\ -1 & \text{if } c_i = C_2 \end{cases} \quad (10.15)$$

which keep track of whether data point \underline{x}_i comes from C_1 or C_2 , then (10.14) can be significantly simplified to

$$\mathcal{L}_{\text{Count}}(\underline{\theta}) = - \underbrace{\sum_i \text{sign}(\bar{c}_i \cdot h(\underline{x}_i | \underline{\theta}))}_{\# \text{ bad} - \# \text{ good points overall}} \quad (10.16)$$

This definition of $\mathcal{L}_{\text{Count}}$ in (10.16) is legitimate, and was the basis for the *Perceptron* of Figure 10.5, one of the first neuron-based approaches in pattern recognition. However $\mathcal{L}_{\text{Count}}$ has two limitations or disadvantages:

1. $\mathcal{L}_{\text{Count}}$ is not easy to optimize (see Appendix C.2), because the $\text{sign}()$ function (Figure 10.6) is discontinuous.

2. $\mathcal{L}_{\text{Count}}$ is not necessarily terribly effective, as sketched in [Figure 10.7](#): $\mathcal{L}_{\text{Count}}$ only seeks to have all of the training data on the correct side of the hyperplane, but has no opinion about how close or far the points may be from the hyperplane.

We can try to address the latter point, the possible closeness of the hyperplane to the training data, by asserting a margin $\gamma > 0$, the minimum distance from the hyperplane to a data point. Recall that

$$\bar{c}_i \cdot \frac{h(\underline{x}_i | \underline{\theta})}{|\underline{w}|} \quad (10.17)$$

is the signed distance from a point \underline{x}_i to the hyperplane (where a positive distance means correctly classified). Our margin requirement therefore asserts that

$$\bar{c}_i \cdot \frac{h(\underline{x}_i | \underline{\theta})}{|\underline{w}|} \geq \gamma \rightarrow \bar{c}_i \cdot h(\underline{x}_i) \geq |\underline{w}| \gamma \rightarrow \bar{c}_i \cdot h(\underline{x}_i) - |\underline{w}| \gamma \geq 0 \quad (10.18)$$

such that [\(10.16\)](#) can be modified as

$$\mathcal{L}_{\text{Count}}(\underline{\theta}) = - \sum_i \text{sign}(\bar{c}_i \cdot h(\underline{x}_i | \underline{\theta}) - |\underline{w}| \gamma) \quad \underline{\theta} = \begin{bmatrix} \underline{w} \\ w_o \end{bmatrix} \quad (10.19)$$

The assertion of a margin will indeed push the hyperplane away from the class data ([Figure 10.7](#)), however

1. The objective function *still* does not give us the *best* hyperplane, just one that is at least γ away from the classes (if possible), as shown in [Figure 10.7](#).
2. We now have *another* parameter, γ , to learn or assert. In a high-dimensional feature space, where the classes are not easily visualized, it may be very hard to know how to select an appropriate γ .

On top of this, the objective $\mathcal{L}_{\text{Count}}$ is still discontinuous, as had been mentioned before, which makes gradient-based optimization strategies inapplicable. A very different approach is to assert a quadratic objective without inequalities, since in that case it is possible to derive the optimal solution analytically. Starting again with [\(10.17\)](#), we would like the signed distance from a point \underline{x}_i to a hyperplane to exceed some margin γ :

$$\bar{c}_i \cdot \frac{h(\underline{x}_i | \underline{\theta})}{|\underline{w}|} > \gamma \quad (10.20)$$

However the inequality is problematic, since it makes the optimization challenging. Although slightly odd, suppose that some value $\gamma > 0$ represents an idealized distance from the class points to the hyperplane, such that we desire

$$\bar{c}_i \cdot \frac{h(\underline{x}_i | \underline{\theta})}{|\underline{w}|} = \gamma \rightarrow \bar{c}_i \cdot h(\underline{x}_i | \underline{\theta}) - |\underline{w}| \gamma = 0 \quad (10.21)$$

We could penalize the average square of the difference, how far each point was from the target distance γ :

$$\mathcal{L}_{\text{Quad}}(\underline{\theta}) = \frac{1}{N} \sum_{i=1}^N (\bar{c}_i \cdot h(\underline{x}_i | \underline{\theta}) - |\underline{w}| \gamma)^2 \quad (10.22)$$

The *magnitude* of vector \underline{w} does not affect the geometry of the hyperplane, so the value of γ does not² actually influence the optimization. Therefore we can solve

$$\mathcal{L}_{\text{Quad}}(\underline{\theta}) = \frac{1}{N} \sum_{i=1}^N (\bar{c}_i \cdot h(\underline{x}_i | \underline{\theta}) - 1)^2 \quad (10.23)$$

This is a MSE (mean-squared error) approach, which allows for an analytical closed-form solution [6], as derived in [Appendix D](#) and demonstrated in [Example 10.1](#). The attractiveness of the MSE method is that it is simple, fast, produces a hyperplane for both linearly separable and *non*-separable classes, and always³ finds a solution (that is, never gets stuck or fails to converge). The significant limitation or disadvantage is that there is absolutely no guarantee that the resulting hyperplane actually lies between the classes, even if they are linearly separable. That is, although the MSE hyperplane is optimal with respect to $\mathcal{L}_{\text{Quad}}$, it is clearly *not* the optimal linear discriminant in the sense of classification performance.

So what *is* the optimal linear discriminant? Given the data points from two classes, presumably the best separation is that hyperplane which is *furthest* from any point, in other words that one with the greatest margin. So essentially the margin idea of [Figure 10.7](#) was good, except that the criterion in [\(10.19\)](#) was specified the wrong way around. That is, [\(10.19\)](#) essentially proposed

Assert a margin size, then find a hyperplane minimizing the number of bad points,

as had been illustrated in [Figure 10.7](#), whereas really we should be optimizing as

From all hyperplanes having no bad points, select the one with the largest margin,

² Admittedly this seems unusual. Clearly the resulting solution *does* scale with γ , it is just that the scaling does not affect the orientation or location of the hyperplane.

³ The MSE produces a solution as long as the number of data points (N) is at least as large as the number of dimensions (n), and that the data points are spread in all directions, meaning that there is no $(n - 1)$ -dimensional hyperplane on which all of the points lie. See also [Figure B.5](#).

Example 10.1: Mean-Squared Error Discriminant

This example builds on the earlier discussion in [Example 3.2](#), which already established a relationship between classification and regression.

The optimization objective proposed in [\(10.23\)](#) was quadratic, asserting a penalty on the square of the degree to which

$$\bar{c} \cdot h(\underline{x} | \underline{\theta}) = 1$$

fails to hold. From the derivation in [Appendix D](#) we know that such problems possess a single, unique minimum which can be found analytically. In particular, in general, given a set of equations

$$y_i = q_i^T \underline{\theta} \quad (10.24)$$

linear in unknown parameters $\underline{\theta}$, when subject to a quadratic penalty the optimal estimate can be found as

$$\text{Optimal Estimate } \hat{\underline{\theta}} = (Q Q^T)^{-1} Q \underline{y} \quad (10.25)$$

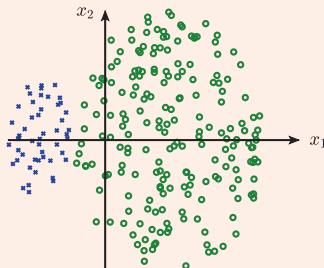
For our linear discriminant, each given equation is

$$1 = \bar{c}_i [\underline{x}_i^T \ 1] \begin{bmatrix} \underline{w} \\ w_o \end{bmatrix} \quad \underline{\theta} = \begin{bmatrix} \underline{w} \\ w_o \end{bmatrix} \quad (10.26)$$

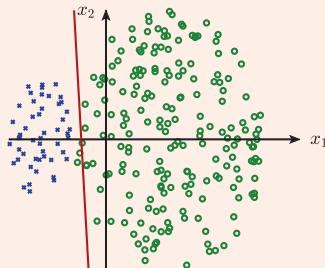
where [\(10.26\)](#) was reorganized from the usual $(\underline{w}^T \underline{x} + w_o)$ to be written in the same form as [\(10.24\)](#). Consequently the solution from [\(10.25\)](#) follows from defining Q and \underline{y} :

$$Q^T = \begin{bmatrix} \bar{c}_1 \underline{x}_1^T & \bar{c}_1 \\ \vdots & \vdots \\ \bar{c}_N \underline{x}_N^T & \bar{c}_N \end{bmatrix} \quad \underline{y} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \quad (10.27)$$

Suppose we have show two clusters, as shown:



Input Data Points



Resulting Discriminant

The resulting estimated parameter is $\hat{\underline{\theta}}^T = [-0.30 \ -0.02 \ -0.23]$, meaning that the

Example continues ...

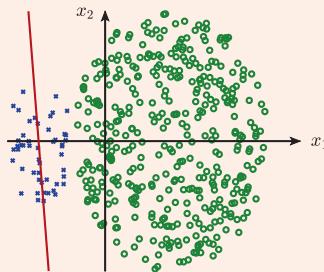
Example 10.1: Mean-Squared Error Discriminant (continued)

proposed discriminant is

$$h(\underline{x} | \hat{\theta}) = (-0.30x_1 - 0.02x_2 - 0.23) \quad \text{thus} \quad \underline{w} = \begin{bmatrix} -0.30 \\ -0.02 \end{bmatrix} \quad (10.28)$$

and we can interpret the normal vector as pointing sideways (large x_1 component, small x_2 component) and correctly pointing to the left, towards C_1 (blue).

The resulting discriminant (previous page) is actually very good, passing very close to the optimal location, with only very few points on the incorrect side of the discriminant. However that fit is a bit of a coincidence, and will normally *not* be expected from MSE. For example, doubling the number of green data points, from $N = 200$ before to $N = 400$ here, shows a far poorer discriminant, even though the problem is just as separable as before.



The MSE derivation in [Appendix D](#) claimed that exactly the same equations could be used to perform linear regression. Given a set of data points (x_i, y_i) and a model $y = mx + b$ for a line, our measurements become

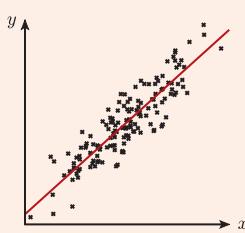
$$y_i = mx_i + b \quad \rightarrow \quad y_i = [x_i \ 1] \underline{\theta} \quad \underline{\theta} = \begin{bmatrix} m \\ b \end{bmatrix} \quad (10.29)$$

which allows us to formulate Q and apply [\(10.25\)](#).

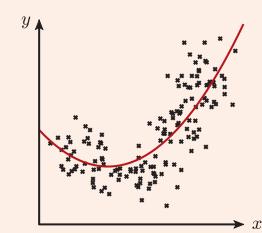
Indeed, the MSE derivation had also claimed that the same approach could perform quadratic regression, the fitting of a parabola. This *sounds* nonlinear, and a parabolic model $y = \alpha x_i^2 + \beta x_i + \gamma$ is nonlinear in x , but is still linear in the parameters, which is what matters. Therefore the measurement model

$$y_i = \alpha x_i^2 + \beta x_i + \gamma \quad \rightarrow \quad y_i = [x_i^2 \ x_i \ 1] \underline{\theta} \quad \underline{\theta} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \quad (10.30)$$

is still linear in $\underline{\theta}$, allowing [\(10.25\)](#) to be used, yet again.



The first-order regression from [\(10.29\)](#) (left) and the second-order regression from [\(10.30\)](#) (right), along with the two preceding linear discriminants, were *all* based on [\(10.25\)](#), an attractively flexible formulation.



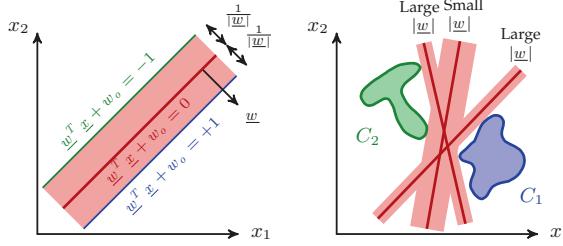


Fig. 10.8. MAXIMIZED MARGIN: Building on [Figure 10.7](#), the best hyperplane is the one which has the greatest margin. If the data points are constrained to lie outside of the margin ($|\underline{w}^T \underline{x} + w_o| \geq 1$), then the size of the margin is $1/|\underline{w}|$, left. The solution offering the greatest margin is then that with the smallest $|\underline{w}|$, right.

as illustrated in [Figure 10.8](#). To develop the corresponding optimization objective, $\mathcal{L}_{\text{Margin}}$, we begin by asserting a constraint

$$\bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) \geq 1 \quad \text{for all } i. \quad (10.31)$$

Since the distance from a point \underline{x}_i to the hyperplane is $|h(\underline{x}_i | \underline{w}, w_o)|/|\underline{w}|$, requiring that $\bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) \geq 1$ is equivalent to forcing each data point to be a distance of at least $1/|\underline{w}|$ from the hyperplane. That is, $1/|\underline{w}|$ is the margin (the distance to the closest point), which we would like to maximize, which we can accomplish by *minimizing*⁴ $|\underline{w}|$, leading to

$$\begin{aligned} \text{Minimize } \mathcal{L}_{\text{Margin}}(\theta) &= \mathcal{L}\left(\begin{bmatrix} \underline{w} \\ w_o \end{bmatrix}\right) = |\underline{w}| \\ \text{such that } \bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) &\geq 1 \quad \text{for all } i \end{aligned} \quad (10.32)$$

This is referred to as a constrained optimization problem, and objective $\mathcal{L}_{\text{Margin}}$ is widely used as the basis for many methods in pattern recognition, particularly the Support Vector Machine of the following section.

The application of [\(10.32\)](#) to a few data points is illustrated in [Figure 10.9](#). Given many training data subject to the inequality constraint $\bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) \geq 1$, there may be as few as only two points which are actually *on* the margin, meaning those points where $\bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) = 1$, known as *support vectors*, indicated by arrows in the right panel of [Figure 10.9](#). If all *other* points from the classes were to be removed (those points shown lightly shaded in [Figure 10.9](#)), then the solution to [\(10.32\)](#) would change not at all; that is, only

⁴ It may seem confusing that here we are minimizing $|\underline{w}|$, and yet in the discussion deriving [\(10.23\)](#) we had declared that the magnitude of $|\underline{w}|$ did not affect the result. Both statements are correct, but the problem contexts are different. The magnitude $|\underline{w}|$ does not, indeed, affect the definition of the hyperplane, however it *does* affect the definition of *distance* to the hyperplane, which matters greatly in asserting a margin, such as in [\(10.32\)](#).

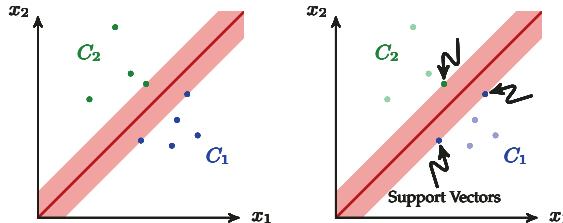


Fig. 10.9. SUPPORT VECTORS: Objective $\mathcal{L}_{\text{Margin}}$ of (10.32) seeks the largest possible margin separating two classes (left). However only very few points will actually lie on the margin, and those are the only points, the *support vectors* (arrows), which are needed to define the hyperplane (right).

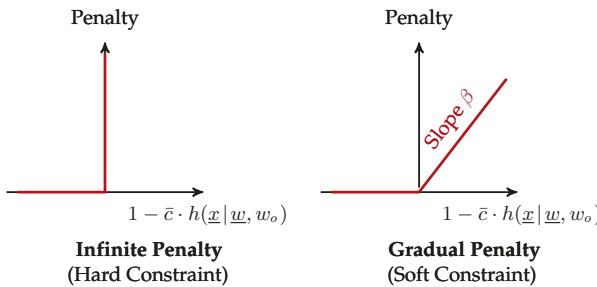


Fig. 10.10. HARD VS. SOFT CONSTRAINTS: A hard constraint, left, requires that the constraint be satisfied, which may mean that no solution exists (for example, for two classes which are not linearly separable). A soft constraint, right, brings flexibility into the objective function, such that there is a *preference* for the constraint to be satisfied, but not a requirement.

the support vectors are needed to completely characterize the margin and optimal hyperplane.

The support vectors of Figure 10.9 are essentially equivalent to the idea of point pruning of Figures 6.18 and 6.19 in Section 6.4.

The classes do need to be linearly separable for (10.32) to work, since the inequality constraint is a so-called *hard* constraint that *must* be satisfied, meaning that it must be possible to find a hyperplane to have all of the points from C_1 and C_2 on the positive and negative sides of the hyperplane, respectively. If two classes are not linearly separable, then (10.32) will simply fail to produce a solution. An alternative, sketched in Figure 10.10 allows for a constraint to be preferred, but not required. In particular, the constraint from (10.32) says

$$\bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) \geq 1 \quad \rightarrow \quad 1 - \bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) \leq 0 \quad (10.33)$$

We can incorporate this requirement into the objective function, such that a penalty term is included proportional to (controlled by β) the degree to which

a point fails to meet the margin requirement (i.e., the degree to which it is above zero):

$$\text{Penalty} = \beta \cdot \max\{0, 1 - \bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o)\} \quad (10.34)$$

We finally have a comprehensive objective,

$$\text{Minimize } \mathcal{L}_{\text{Soft-Margin}} \left(\begin{bmatrix} \underline{w} \\ w_o \end{bmatrix} \right) = |\underline{w}| + \frac{\beta}{N} \sum_i \max\{0, 1 - \bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o)\} \quad (10.35)$$

which is guaranteed to select a separating hyperplane if the classes are linearly separable, and which produces a credible compromise if the classes are not separable. We will address the question of non-separable classes further in the following section.

10.3 Nonlinear Discriminants

The restriction of our attention to linear discriminants was a convenient assumption in [Sections 10.1](#) and [10.2](#) because of the ability to formulate the geometry of distance from a point to the hyperplane and the analytical solution formulated for the mean-square error approach.

As we saw in [Figure 10.4](#)⁵ if two classes are not linearly separable, then we have two options:

1. Use a linear classifier and accept a higher probability of classification error;
2. Use a *non-linear* classifier.

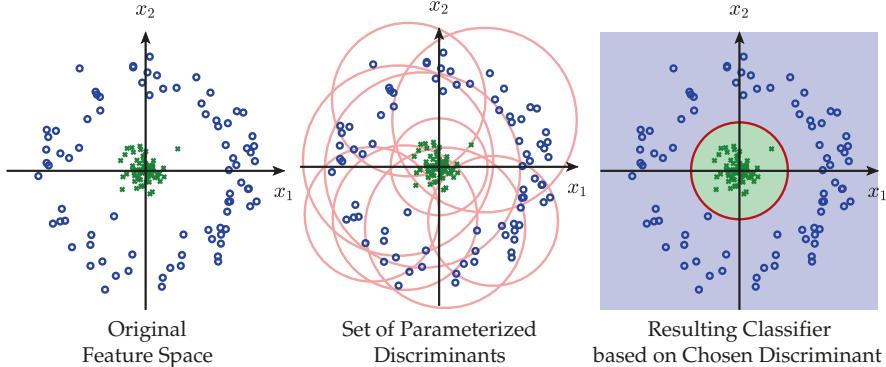
Linear discriminants having been addressed in detail in the previous section, we now turn our attention to nonlinear discriminants.

As was discussed at the beginning of this chapter, we have already seen examples of nonlinear classifiers, such as the quadratic Mahalanobis in [\(10.2\)](#) or the ML/MAP (which could be linear, quadratic, or more generally nonlinear depending on the statistical distributions) in [\(10.3\)](#). To be sure, the NN or k NN classifiers of [Chapter 6](#) are also nonlinear, however they are not readily written in discriminant form, and so are not as helpful in the present context.

Given two classes in feature space \mathcal{X} which are not linearly separable, we have two basic strategies to develop a nonlinear discriminant:

⁵ Really [Figure 10.4](#) is not so unique. We have seen non-separability in many other examples throughout this text, such as [Lab 2](#), [Figure 3.5](#), [Lab 4](#), or [Figure 7.11](#).

Nonlinear Classification via Nonlinear Parameterized Discriminant:



Nonlinear Classification via Nonlinear Feature-Space Transformation:

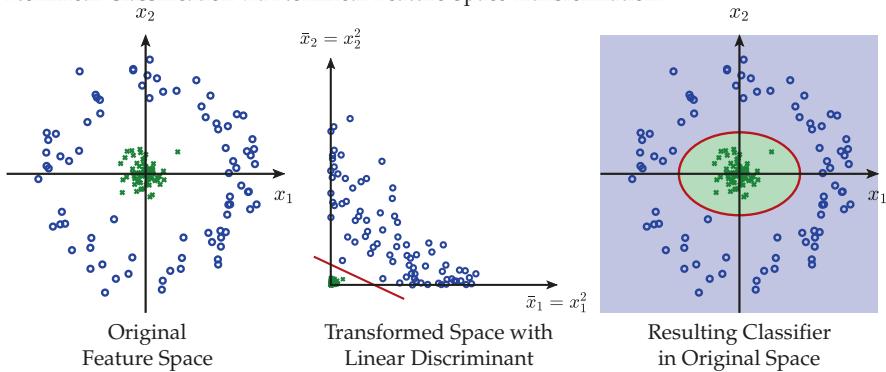


Fig. 10.11. Building on the example from [Figure 5.10](#), we can imagine developing a nonlinear discriminant-based classifier by proposing either a nonlinear parametric discriminant (top), or by applying a *linear* discriminant to a nonlinear transformation (bottom).

1. We parameterize a nonlinear discriminant $h(\underline{x}|\theta)$, as in [\(10.5\)](#), an approach which is illustrated in the top half of [Figure 10.11](#). The Mahalanobis classifier is essentially in this form, in that it is a parameterized nonlinear function:

$$h(\underline{x}|\theta) \text{ is quadratic (nonlinear)} \quad \theta = \begin{bmatrix} \text{Mean} \\ \text{Covariance} \end{bmatrix} \quad (10.36)$$

In [Figure 10.11](#) a circular discriminant was chosen, however clearly any number of different nonlinear functions could have been chosen, and indeed this is the problem: it is not at all obvious *what* sort of nonlinear parameterization to employ. Classification boundaries can have all sorts of complicated forms, particularly in high-dimensional spaces. Clearly NN/kNN/ML/MAP can all generate highly complex decision boundaries,

however they do not give us any further insight in terms of how to select a nonlinear form for $h()$.

2. We nonlinearly transform the feature space

$$\underline{x} \in \mathcal{X} \xrightarrow{\varphi()} \bar{x} = \varphi(\underline{x}) \in \bar{\mathcal{X}} \quad (10.37)$$

and then apply a *linear* discriminant in the transformed space $\bar{\mathcal{X}}$, as illustrated in the bottom half of [Figure 10.11](#). In that figure the features were squared, therefore a straight line in the transformed domain maps back to an ellipse in the original domain, however a non-quadratic transformation would have led to some other (non-elliptical) shape.

The finding of a linear discriminant (in the transformed space) we know how to do from [Section 10.2](#), so it is the nonlinear transformation $\varphi()$ which needs to be inferred.

It would seem that the inference of a nonlinear transformation $\varphi()$, in the latter case, is just as unobvious as the inference of a nonlinear discriminant $h()$ in the former, so that we are no better off than inferring a nonlinear discriminant directly. It turns out that this claim is incorrect: nonlinear feature transformations are used almost universally in Support Vector Machine (SVM) classifiers.

It is true, of course, that in general we do indeed *not* know specifically which nonlinear transformation will allow two given classes to become linearly separable. As a result, we probably need a rather *large* number of nonlinear functions

$$\bar{x}_j = \varphi_j(\underline{x}) \quad \underline{x} \in \mathbb{R}^n \rightarrow \bar{x} \in \mathbb{R}^{\bar{n}} \quad \bar{n} \gg n \quad (10.38)$$

That is, we are normally projecting into a *much* higher dimensional space. We are still seeking a linear discriminant, so the soft-margin approach of [\(10.35\)](#) applies, but now in the transformed domain:

$$\text{Minimize } \mathcal{L}_{\text{Soft-Margin}} \left(\begin{bmatrix} \bar{w} \\ \bar{w}_o \end{bmatrix} \right) = |\bar{w}| + \frac{\beta}{N} \sum_i \max\{0, 1 - \bar{c}_i \cdot (\bar{w}^T \varphi(x_i) + \bar{w}_o)\} \quad (10.39)$$

where the notation \bar{w} and \bar{w}_o of the hyperplane parameters emphasizes that these too apply in the transformed domain.

The computational complexity of [\(10.39\)](#) becomes much more difficult as \bar{n} increases, which seems frustrating, because the underlying problem (the number of data points and their feature dimensionality) is not changing. Even worse, there are families of nonlinear transformations which are *infinite* dimensional, in which case [\(10.39\)](#) cannot be solved at all.

Remarkably, the infinite-dimensional nonlinear-transformed problem *is* solvable. The details of the optimization techniques to solve such problems are outside of the scope of interest in this text, however there are three key concepts which allow the transformed problem to be solved:

SUPPORT VECTORS: Recall from [Figure 10.9](#) that the data points that actually characterize the problem, the support vectors, are few in number and each of dimensionality n . That is, even if the problem is transformed into an infinite-dimensional space, the actual degrees of freedom underlying the problem is relatively modest, which we can try to exploit by formulating the *dual* optimization problem.

DUAL PROBLEM: Certain optimization problems can be written in two equivalent ways, known as the primal (original) and dual (based on Lagrange multipliers). The dual does not optimize over \underline{w} , rather it optimizes over weights $\{\alpha_i\}$ from which \underline{w} can be found:

$$\underline{w} = \sum_{i=1}^N \alpha_i \bar{c}_i \varphi(\underline{x}_i) \quad (10.40)$$

where $\alpha_i = 0$ if $\varphi(\underline{x}_i)$ is away from the margin, and $\alpha_i \neq 0$ if $\varphi(\underline{x}_i)$ lies exactly on the margin (is a support vector). That is, we have two striking conclusions:

- The unknown \underline{w} which we wish to solve for is a linear function of the support vectors
- Although \underline{w} may be *infinite* dimensional, the actual number of unknowns $\{\alpha_i\}$ in the dual problem is N , the number of data points in the original problem.

The key is that we do not ever need to compute or store \underline{w} or \underline{x} explicitly. All that we need, to solve the optimization problem and then to implement the resulting classifier, are terms like $\underline{w}^T \underline{x}$ and $\underline{x}^T \underline{x}'$: dot products of (possibly infinite-dimensional) vectors.

KERNEL TRICK: It may sound daunting, computing the dot product of infinite-dimensional vectors, however an infinite-length dot product is just like an infinite series, such as

$$\left[1 \quad \frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8} \quad \dots \right] \bullet \left[1 \quad 1 \quad 1 \quad 1 \quad \dots \right] = 2 \quad (10.41)$$

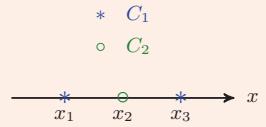
or

$$\left[1 \quad \frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8} \quad \dots \right] \bullet \left[1 \quad \frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8} \quad \dots \right] = \frac{4}{3} \quad (10.42)$$

These two examples are fairly trivial, but serve to illustrate the point: there is no inherent problem in taking dot products of infinite-length vectors, the question is whether we are able to evaluate the dot product directly.

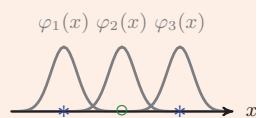
Example 10.2: Radial Basis Functions

How is it that classes which are arbitrarily overlapping can become linearly separable under a nonlinear transformation? Suppose we have three points from two classes, as shown. The two classes are clearly *not* linearly separable, since no point along x separates C_1 and C_2 .

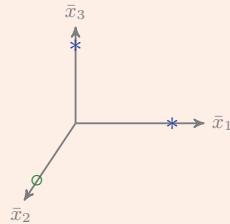


However suppose we create a family of nonlinear transformations $\varphi_j(x)$, more or less an indicator function such that

$$\varphi_i(x) = \begin{cases} 1 & x = x_i \\ \approx 1 & x \approx x_i \\ 0 & |x - x_i| \gg 0 \end{cases}$$



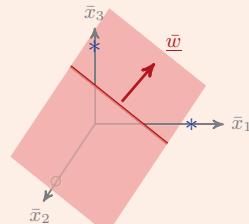
The actual shape of the function doesn't really matter, however it is easiest to think of them as very local Gaussian functions, centered on each data point, as shown.



Given the nonlinear transformations $\bar{x}_j = \varphi_j(x)$, it is easy to plot the locations of our three given data points in the new transformed space $\bar{\mathcal{X}}$.

Now, it can be very hard to visualize a high-dimensional space, and how a hyperplane behaves in that space. But it turns out that we can create a hyperplane to separate the points *any* way we want. That is, given N data points $\underline{x}_1, \dots, \underline{x}_N$, there are 2^N ways of grouping these into two classes, and we can find a hyperplane for *each* possible grouping based on the normal vector

$$\underline{w}^T = [s_1 \ s_2 \ \dots \ s_N] \quad s_i = \begin{cases} 1 & \text{Classify } \underline{x}_i \text{ as } C_1 \\ 0 & \text{Classify } \underline{x}_i \text{ as } C_2 \end{cases}$$



in which case the linear discriminant $\underline{w}^T \varphi(\bar{x}) - \frac{1}{2}$ leads to the desired hyperplane.

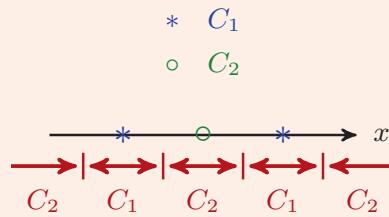
In this example, we would like the first and third data points to be classified as C_1 , so we select normal vector

$$\underline{w}^T = [1 \ 0 \ 1] \quad \underline{w}^T \varphi(\bar{x}) - \frac{1}{2} \stackrel{C_1}{\geq} 0 \stackrel{C_2}{<} 0$$

Example continues ...

Example 10.2: Radial Basis Functions (continued)

and obtain the discriminant, as plotted. Back in the original domain, this linear discriminant is now *non-linear*, giving us the classifier as shown, which successfully separates the classes. Placing a local Gaussian/indicator function at each data point essentially reproduces the nearest-neighbour classifier, clearly overfitting the data. In practice we would prefer to use a wider $\varphi()$ to avoid memorizing individual data points, reduce overfitting, and better generalize.



That is, are there choices of nonlinear transformation families $\{\varphi_j()\}$, such that the dot product $\varphi(\underline{x}) \bullet \varphi(\underline{x}')$ can be evaluated directly as a function of *finite*-dimensional \underline{x} ? The answer is *yes*, and is known as the *kernel trick*, such that

$$\bar{x} \bullet \bar{x}' = \varphi(\underline{x}) \bullet \varphi(\underline{x}') \equiv \varphi(\underline{x})^T \varphi(\underline{x}') = \Phi_\varphi(\underline{x}, \underline{x}') \in \mathbb{R} \quad (10.43)$$

That is, $\Phi_\varphi(\underline{x}, \underline{x}')$ equals or evaluates the scalar dot product that *would* result from applying the nonlinear transformation φ to finite-length vectors $\underline{x}, \underline{x}'$ from our original problem domain.

Kernels Φ_φ are known for a variety of nonlinear families; by far the most widely used are the polynomial and radial basis function (RBF) transformations, both of which have very simple kernels,

$$\begin{aligned} \Phi_{\text{Polynomial}}(\underline{x}, \underline{x}') &= (\underline{x}^T \underline{x}' + c)^d && \text{Constant } c, \text{ Polynomial order } d \\ \Phi_{\text{RBF}}(\underline{x}, \underline{x}') &= \exp\left(-\frac{(\underline{x} - \underline{x}')^T (\underline{x} - \underline{x}')}{2\sigma^2}\right) && \text{Kernel size } \sigma \end{aligned} \quad (10.44)$$

which are discussed in [Example 10.2](#), illustrated in [Lab 10](#), and which are widely available in any numerical library offering the SVM classifier.

Kernels will be discussed further and somewhat more intuitively in the context of clustering, in the Kernel K-Means method of [Section 12.1.2](#).

10.4 Multi-Class Problems

We started this chapter by developing an objective function to learn linear discriminants, and then generalized this to nonlinear classification by means

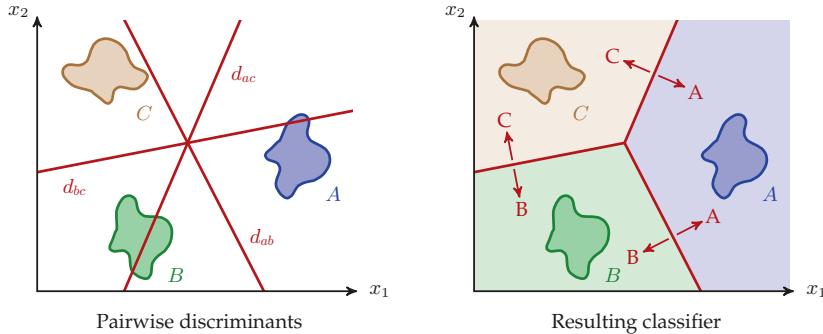


Fig. 10.12. ONE-VERSUS-ONE: We can learn a discriminant for *each* pair of classes, left. The resulting classifier, right, is based on a majority vote by all of the discriminants.

of a nonlinear transformation, realized implicitly via a kernel. This section will generalize discriminants further, by exploring how they might be used in problems involving more than two classes ($K > 2$).

By definition, using discriminants (whether linear or nonlinear) to discriminate between more than two classes clearly requires multiple discriminants, since a single discriminant can only divide a space \mathcal{X} into two: the positive and negative sides of the discriminant. That is, the $K > 2$ classifier will somehow be some combination of multiple discriminants; since classifier combination is the subject of [Chapter 11](#), we will keep the discussion here relatively brief.

Although a great many approaches can be proposed, broadly the ways in which a two-class method like SVM is applied to $K > 2$ classes falls into one of three categories:

1. **One versus One:** Given K classes, we have $K(K - 1)/2$ pairs of classes, so it is straightforward (if not possibly computationally demanding, for large K) to learn a discriminant (whether SVM or otherwise) d_{C_i, C_j} for each possible pair of classes. Given a feature point \underline{x} , each discriminant $d_{C_i, C_j}(\underline{x})$ is evaluated, and \underline{x} is classified as that class having the greatest number of votes among the $K(K - 1)/2$ discriminants, as illustrated in [Figure 10.12](#).

The obvious drawbacks of this approach are the computational complexity, growing quadratically in K , and the possibility of undecided regions, where there is more than one class with the greatest number of votes, just as we encountered with k NN in [Figure 6.20](#).

2. **One versus All:** Given K classes, it is straightforward to learn K discriminants $\{d_{C_i, C_i}\}$, each one learning to separate a given class C_i from data

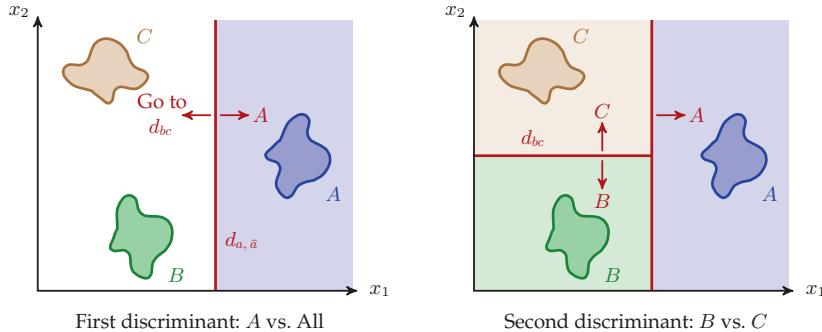


Fig. 10.13. ONE-VERSUS-ALL: In contrast to Figure 10.12, rather than learning discriminants for each pair of classes, we can learn a single discriminant to separate one class from *all* others (left). The process illustrated here is sequential, in that having separated A from (BC) , we then need a discriminant to separate B from C (right).

points taken from all⁶ of the $(K - 1)$ other classes $C_{\bar{i}}$. The resulting classifier can then be based on a winner-takes-all approach, whereby the most positive discriminant declares the selected class,

$$\text{Classify } \underline{x} \in C_i \text{ if } d_{C_i, C_{\bar{i}}}(\underline{x}) > d_{C_j, C_{\bar{j}}}(\underline{x}) \text{ for all } j \neq i \quad (10.45)$$

This does, of course, require that the scaling of the discriminants be learned in a consistent way, so that not just the sign of the discriminant, but also its magnitude, is meaningfully comparable.

3. Decision Tree: To employ the learned discriminants in the usual binary fashion, and not based on the largest magnitude, we could also proceed sequentially, one class at a time: proceed as

$$\begin{aligned} \text{If } d_{C_1, C_1} > 0 \text{ then classify as } C_1 \\ \text{otherwise if } d_{C_2, C_2} > 0 \text{ then classify as } C_2 \\ &\quad \text{otherwise ...} \end{aligned} \quad (10.46)$$

as illustrated in Figure 10.13. The method is simple, but has the limitation that it is not remotely obvious that the first test should be of C_1 ; instead it might be some other class that could be separated most reliably from the others. This sequential approach will be developed further and more systematically under *Decision Trees* in Section 11.3.

A further critique of the approaches in (10.45) and (10.46) is that both require the evaluation of K discriminants whereas, in principle, we should need

⁶ There will almost certainly be significant class-imbalance issues present, since the $K - 1$ classes in $C_{\bar{i}}$ would be expected to have $K - 1$ times as many data points as in C_i . The *class imbalance problem* was discussed in Section 9.1.

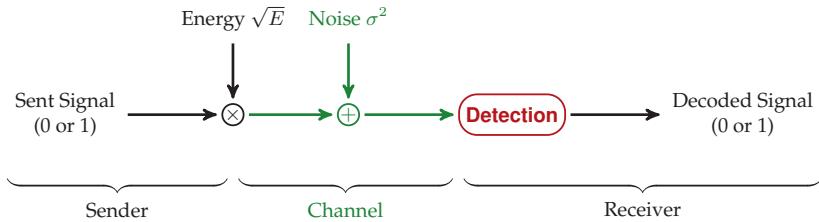


Fig. 10.14. BINARY COMMUNICATIONS CHANNEL: A signal is given some energy and sent across a communications channel (wire, fibre, microwave) in which it is subject to some amount of noise. At the receiving end the noisy signal needs to be classified by a detector, red, as zero or one, fundamentally a pattern recognition problem.

only $\lceil \log_2 K \rceil$ binary tests in order to choose among K classes. We could learn fewer discriminants in this way, however each discriminant would need to separate *groups* of classes from one another. Such an approach would require a degree of clumping or cluster analysis, along the lines of hierarchical clustering in [Section 12.1.4](#).

Case Study 10: Digital Communications

Communications theory is a vast discipline, involving the sending/receiving of information over communication channels. The explosion of the Internet and the sheer number of devices, their geographic spread, their bandwidth expectations, and the heterogeneity of contexts in which communications is expected — your wristwatch with your smartphone, your wifi-router with your fridge, adjacent cars on a smart highway — all make clear the tremendous expectations our society places on sending information over a communications channel.

A great deal of communications has to do with protocols, handing off a device (a cell phone, say) from one communications tower to another, however such protocols are not what interest us here. Instead, the very basic question of how two devices send information to one another, once a communications link has been defined and established, is the problem quite central to pattern recognition.

[Figure 10.14](#) sketches a very basic model of a communications channel, whereby a symbol (a binary digit) is selected to be sent, where the binary value is multiplied by a signal of energy⁷ E . The communications channel — the medium (wire, fibre, air) through which the signal propagates through the

⁷ The figure shows multiplication by \sqrt{E} , since signal energy is proportional to the square of amplitude.

physical space separating the sender and the receiver — typically corrupts the sent signal in some way (otherwise communications would be pretty easy!). In practice there can be many subtle signal effects, such as nonlinearities in optical fibres or multipath echoes in microwave systems, however we will assume only the presence of additive noise to corrupt the transmitted signal, a common model in wired communications.

That is, the statistics of the received (measured) signal y obeys

$$p(y|C_0) = \mathcal{N}(y; 0, \sigma^2) \quad p(y|C_1) = \mathcal{N}(y; \sqrt{E}, \sigma^2) \quad (10.47)$$

This is a very simple one-dimensional two-class problem, which we can easily solve using the ML classifier from [Chapter 8](#):

$$y \begin{matrix} C_1 \\ \gtrless \\ C_0 \end{matrix} \frac{\sqrt{E}}{2} \quad (10.48)$$

From [\(8.55\)](#), the corresponding probability of received-symbol error, assuming the two transmitted digits to be equally likely, is

$$\mathbf{P}(e) = 2 \cdot \frac{1}{2} \cdot Q\left(\frac{\sqrt{E}}{2\sigma}\right) \quad (10.49)$$

That is, the probability of communications error decays monotonically with \sqrt{E}/σ . For a fixed level of noise σ in a given communications channel, the critical issue then is the separation \sqrt{E} between adjacent signals.

Of course, communicating one binary digit at a time is a significant limitation. Instead of just sending two signals (0 or 1), in many cases we would be better off communicating *multiple* bits simultaneously. The obvious extension of [Figure 10.14](#) and [\(10.47\)](#) would be to select signals of K different amplitudes, such that

$$p(y|C_\kappa) = \mathcal{N}(y; \kappa \cdot \sqrt{E}, \sigma^2) \quad 0 \leq \kappa < K \quad (10.50)$$

In practice this ends up being a poor design, since the different signals require very different amounts of energy, and the average energy-per-bit for a given probability of error is significantly inferior to other designs.

The reader may wonder why communications theory is appearing in a chapter on discriminants. Well, in order to improve energy-per-bit we need to increase the dimensionality of the problem, varying the amplitude of *multiple* signals, ideally orthogonal, such as sine and cosine, or sinusoids of different frequencies. To communicate one of 16 possible symbols (i.e., 4 bits), we could arrange 16 symbols on a 4×4 grid on two dimensions, as shown in [Figure 10.15](#), leading to a fairly simple set of discriminants.

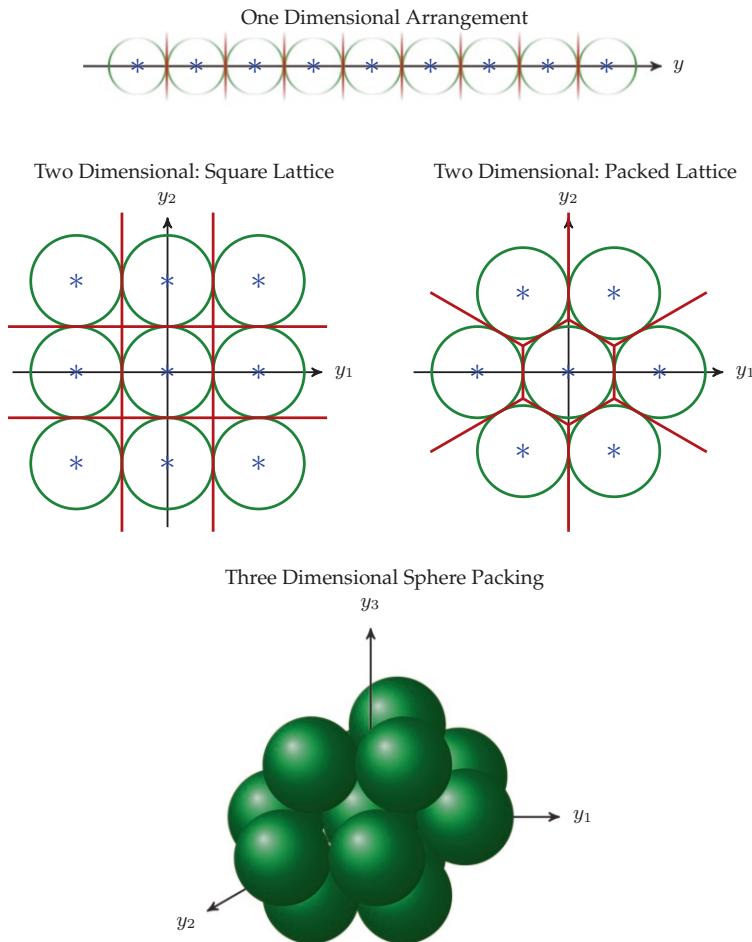


Fig. 10.15. COMMUNICATIONS: The most obvious way to encode one of K symbols would be to modulate the amplitude of a signal, top. Since power grows as the square of amplitude, the power efficiency becomes poorer as K grows, so higher-dimensional packings are preferred. In all cases the green circle/sphere of radius 1 indicates the contour of constant probability, and the red discriminants show the decision boundaries. The packing is fairly intuitive up to three dimensions, however in practice communication systems operate in far higher dimensional spaces. For illustration purposes, to keep individual diagrams nicely symmetric, the number of symbols K varies from one diagram to another; the four illustrations here are not meant to be directly comparable one to another.

However anyone who has played with coins on a table will know that you can pack coins more compactly on a hexagonal grid rather than a square grid, also shown in [Figure 10.15](#), meaning that we can reduce average energy while still maintaining the same signal separation, however the choices of discriminants become more complicated.

The efficiency can be pushed further, of course. Oranges or marbles, stacked into a pyramid, pack even more tightly in three dimensions than coins in two dimensions, with further improvements in even higher dimensions, but similarly with further challenges in efficient decoding.

Lab 10: Discriminants

A basic linear discriminant is a straightforward concept for well-separated classes, so it is really the case of classes which are slightly overlapping or not linearly separable which would be of greater interest. We will use a noisy version of the two-arc dataset, since the noise helps to reveal whether a given classifier is overfitting.

```
load TwoArc
```

```
% add modest noise
```

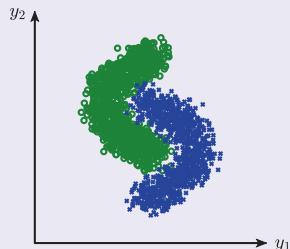
```
p1 = p1 + 0.1 * randn(size(p1));
p2 = p2 + 0.1 * randn(size(p2));
```

```
% axis ranges reasonable for this data set
```

```
xr = [2.5:0.05:7.5];
yr = [1:0.05:6];
```

```
% plot data points
```

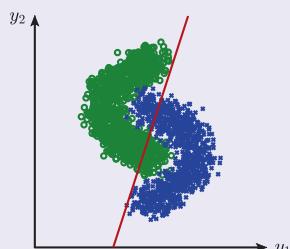
```
clf
plot(p1(1,:), p1(2,:), 'og');
hold on;
plot(p2(1,:), p2(2,:), 'xb');
```



We'll start with the MSE-based discriminant of [Section 10.2](#). The reader is encouraged to follow the detailed mathematical description of [Example 10.1](#) to understand the following code:

```
% set up matrices for MSE
```

```
Qt = [p1' ones(size(p1,2),1) ; -p2' -ones(size(p2,2),1)];
z = ones(size(Qt,1),1);
```



```
% Compute the optimal estimate
est = inv(Qt'*Qt)*Qt'*z;
```

The MSE discriminant is roughly as expected: we obtain a fairly plausible result with very few lines of code. The analytical derivation associated with the MSE relies on the model being linear, so it is not possible to generalize the MSE approach to a nonlinear discriminant and still retain an analytical solution.

The alternative is, of course, the support vector machine (SVM). If we do *not* offer a nonlinear transformation, then the SVM hard-margin objective of (10.32) *must* fail, because there is simply no linear discriminant which is able to perfectly separate the classes:

```
% combine points and create vector denoting the class
pts = [p1'; p2'];
c = [ones(size(p1,2),1) ; -ones(size(p2,2),1)];
```

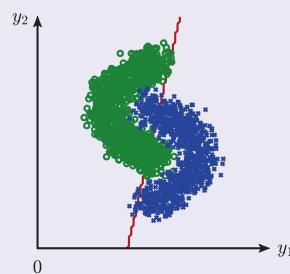
```
% compute hard-margin linear SVM
SVMHard = fitcsvm(pts,c,'KernelFunction','linear','BoxConstraint',Inf,
    'ClassNames',[−1,1]);
```

The specification of the SVM will vary slightly from one package to another; in Matlab it is the “BoxConstraint” parameter which is essentially β of Figure 10.10, controlling how hard the margin is asserted, and $\beta = \infty$ should make the margin hard. In practice, many SVM packages will automatically vary parameters to try to find a solution, so in fact the above code *does* return a discriminant, but only by weakening BoxConstraint. In general, hard-margin optimizations take much longer, computationally, than soft-margin.

Instead, the soft margin SVM of (10.35), now deliberately asserted, can produce a result fairly similar to that of MSE ...

```
% compute soft-margin linear SVM
SVMSoft = fitcsvm(pts,c,'KernelFunction',
    'linear','BoxConstraint',10,
    'ClassNames',[−1,1]);
```

```
% obtain grid to sample over
[X,Y]=meshgrid(xr,yr);
meshpts = [X(:) Y(:)];
l = predict(SVMSoft,meshpts);
clf
[~, h] = contour
(X,Y,reshape(l,numel(yr),numel(xr)),[0 0]);
hold on
```



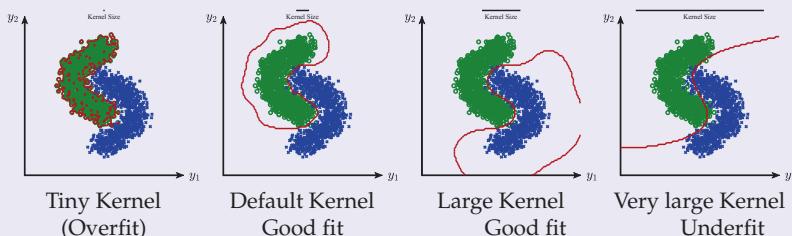
```
plot(p1(1,:),p1(2,:),'og');
plot(p2(1,:),p2(2,:),'xb');
```

However the real power of SVM is with the use of a nonlinear transformation, which allows the optimization objective associated with a linear discriminant to give rise to a nonlinear classifier, a very clever idea. Building on the intuition from [Example 10.2](#), we now assert a radial-basis-function (RBF) kernel. The only parameter is the kernel size, σ ; we can instruct Matlab to automatically select a kernel size, but then we do three additional passes to try larger and smaller kernels:

```
kernelfactor = [1 0.1 3 10];
for i=1:numel(kernelfactor),
    % compute non-linear SVM based on RBF kernels
    if (i==1),
        SVMRBF = fitcsvm(pts,c,'KernelFunction','rbf', 'KernelScale','auto',
        ...
                    'ClassNames',[-1,1]);

    % extract kernel scale from first run
    ks = SVMRBF.KernelParameters.Scale;
    else
        SVMRBF = fitcsvm(pts,c,'KernelFunction','rbf', 'KernelScale',ks*
        kernelfactor(i), ...
                    'ClassNames',[-1,1]);
    end
    l = predict(SVMRBF,meshpts);
    clf
    [~, h] = contour(X,Y,reshape(l,numel(yr),numel(xr)),[0 0]);
    hold on
    plot(p1(1,:),p1(2,:),'og');
    plot(p2(1,:),p2(2,:),'xb');
end
```

The classification results, with the kernel size shown as a black bar at the top of each panel:



As described in [Example 10.2](#), a very small (local) kernel size more or less gives each data point its own dedicated dimension, making

overfitting very likely (left). As the kernel size grows (to the right) each dimension “senses” multiple closely-spaced data points, making overfitting less likely. If the kernel is *too* large (right) then the discriminant is no longer able to effectively separate the classes, the details having been blurred by the coarse kernel scale.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links: [Linear Discriminant Analysis](#), [Mathematical Optimization](#), [Mean squared error](#), [Support Vector Machine](#), [Kernel Method](#)

There are a great many books on pattern recognition and machine learning in general, all of which address aspects of linear discriminants and support vector machines [1, 3, 5, 9, 10, 12].

For texts focussing more specifically on kernel methods and support vector machines, the reader may wish to begin with [4, 11].

Except for kernel methods and support vector machines, which are able to solve meaningful problems using a single discriminant, most current work based on discriminants is focused on the use of *multiple* discriminants, which are learned and then scored/voted in some way, as will be discussed in [Chapter 11](#).

Optimization became central to the discussion of discriminants, particularly in [Section 10.2](#), since what distinguishes one discriminant from another, or what makes a given discriminant “good”, is on the basis of an objective function to be optimized. The principle of optimization is discussed in [Appendix C](#), with further reading in [2, 7, 8].

Sample Problems

Problem 10.1: Short Answer

Give a short definition of each of the following:

- (a) Discriminant
- (b) Linear Separability

- (c) Perceptron
- (d) Support Vector
- (e) Kernel Trick
- (f) Radial Basis Function

Problem 10.2: Short Answer

Offer brief answers to each of the following:

- (a) Why are we interested in linear discriminants? What do they have to offer that is different from statistics-based or distance-based methods?
- (b) Although most of the discriminants we have seen in this chapter are linear, linearity is not actually a requirement. Give an example of a nonlinear discriminant.
- (c) Summarize, conceptually, the difference between a Soft-Margin objective and a Hard-Margin objective.
- (d) Explain how a support-vector-machine (SVM) is able to use a linear discriminant to solve a classification problem which is not linearly separable.
- (e) A single discriminant divides a space into two parts, and is therefore naturally suited to problems having $K = 2$ classes. Briefly describe how linear discriminants can be applied to problems having $K > 2$ classes.

Problem 10.3: Conceptual — Discriminants:

In [Problem 10.2](#) we proposed that a discriminant could be nonlinear. In this question we would like to make this more concrete:

- (a) Identify (sketch) two classes which cannot be linearly separated, and
- (b) Identify (give equation) a nonlinear discriminant which separates the two classes.

Problem 10.4: Conceptual — Separability

Suppose that two million points are chosen, at random, in two-dimensional (x, y) coordinate space. Assume that points cannot overlap (there are no exact repetitions).

Show that, no matter how closely together these points may be chosen, it is always possible to find a straight line so that it does not touch (directly pass through) any of the points, and splits the set exactly in half, i.e., one million on each side.

Problem 10.5: Comprehensive — Discriminants

Recall that a linear discriminant $\underline{w}^T \underline{x} + w_o$ defines a hyperplane which separates classes C_1 and C_2 . Let us define the *quality* Q of a discriminant to be the smallest signed distance (positive for correct side, negative for incorrect side) from the hyperplane to any point in C_1 or C_2 ; i.e.,

$$Q(\underline{w}, w_o) = \min_{\underline{x}' \in C_1 \cup C_2} (\text{signed distance from } \underline{x}' \text{ to the line } \underline{w}^T \underline{x} + w_o = 0) \quad (10.51)$$

If the line $\underline{w}^T \underline{x} + w_o = 0$ fails to separate C_1 and C_2 then $Q(\underline{w}, w_o) < 0$. A larger Q is clearly better than a small one.

Suppose we are given two clusters C_1, C_2 with four data points each:

$$C_1 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\}$$

$$C_2 = \left\{ \begin{bmatrix} -1 + \alpha \\ \alpha \end{bmatrix}, \begin{bmatrix} -2 + \alpha \\ \alpha \end{bmatrix}, \begin{bmatrix} -1 + \alpha \\ 1 + \alpha \end{bmatrix}, \begin{bmatrix} -1 + \alpha \\ -1 + \alpha \end{bmatrix} \right\}$$

where $\alpha = 2$ is a scalar.

- (a) What is the “best” linear discriminant. That is, what are the \underline{w}, w_o which maximize $Q()$? A proof is not required, however do draw a sketch and write down \underline{w}, w_o .
- (b) Let the discriminant normal vector \underline{w} be given by the difference of the class sample means. What is \underline{w} ? Sketch the best discriminant having this normal vector.
- (c) Compute \underline{w} using Fisher’s discriminant. What is \underline{w} ? Sketch the best discriminant having this normal vector.
- (d) Let \underline{w} be the principal component of the total sample; that is, let \underline{w} be the eigenvector corresponding to the largest eigenvalue of the total sample covariance. Find \underline{w} . Sketch the best discriminant having this normal.
- (e) Compute and sketch the discriminant that the MSE method proposes for these classes.
- (f) Suppose that α can vary, such that we are interested in the problem as a function of α . Sketch the best quality, the maximum value of $Q(\underline{w}, w_o | \alpha)$ as a function of α .

Problem 10.6: Comprehensive — Discriminants

We are given two classes C_A, C_B with four data points each:

$$\text{Class A: } \underline{x}_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \quad \underline{x}_2 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \quad \underline{x}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \underline{x}_4 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

$$\text{Class B: } \underline{x}_5 = \begin{bmatrix} 4 \\ 1 \end{bmatrix} \quad \underline{x}_6 = \begin{bmatrix} 6 \\ 1 \end{bmatrix} \quad \underline{x}_7 = \begin{bmatrix} 4 \\ 3 \end{bmatrix} \quad \underline{x}_8 = \begin{bmatrix} 6 \\ 3 \end{bmatrix}$$

We are interested in developing a classifier based on these data.

- (a) Sketch a figure showing the two classes.
- (b) Let us write the linear discriminant as usual, that is

$$\underline{w}^T \underline{x} + w_o$$

Suppose we write \underline{w} as

$$\underline{w} = \begin{bmatrix} 1 \\ \beta \end{bmatrix}$$

What is the range of values for β that allows the data points of the above two clusters to be linearly separated?

- (c) Use Fisher's method to find a feature vector \underline{w} .
- (d) Fisher's method from part (c) only identifies a feature direction \underline{w} , not an actual classifier. Using whatever method you like, develop a classifier for classes C_A, C_B based on Fisher's feature \underline{w} .

Problem 10.7: Numeric/Computational — Support Vector Machines

[Lab 10](#) showed the application of Support Vector Machines of [Section 10.3](#) to a pair of curved classes which were not linearly separable.

In this question, similarly develop a support vector machine to find a classifier for the concentric-class problem of [Figure 10.11](#). You can follow the approach developed in [Lab 10](#), or you are welcome to develop your own implementation.

For the input data, you can synthesize the two classes based on the statistics in [Problem 4.8](#), or you can choose a different approach.

Experiment with at least two different kernels, and report on the quality of the resulting classifiers.

Problem 10.8: Real World, Open-Ended — Finance

Pattern Recognition and Machine Learning have broad application in many areas of finance. In particular, finance involves decision making

(classification) on the basis of large amounts of data, much of which may be irrelevant (recall feature extraction and selection).

For many problems in finance, the line between regression and classification (as in [Example 3.2](#)) may be somewhat blurred:

- Is the question *whether* to buy a stock or not (binary, classification), as opposed to *how much* of a stock to buy (continuous, regression)?
- Is the question *whether* to offer a loan to a given individual (binary), as opposed to the *probability of default* (continuous), which then leads to associated issues, such as the time period or interest rate on the loan.

Do a bit of a search on pattern recognition, machine learning, and finance. This field, as a whole, will be much too broad, so select a specific topic of interest. Articulate clearly what elements of pattern recognition are present in the topic, what methods are being employed, and what challenges are present.

References

1. C. Bishop, *Pattern Recognition and Machine Learning* (Springer, New York, 2011)
2. S. Boyd, L. Vandenberghe, *Convex Optimization* (Cambridge University Press, Cambridge, 2004)
3. M. Deisenroth, A. Faisal, C. Ong, *Mathematics for Machine Learning* (Cambridge University Press, Cambridge, 2020)
4. N. Deng, Y. Tian, C. Zhang, *Support Vector Machines: Optimization Based Theory, Algorithms, and Extensions* (Chapman and Hall, London, 2012)
5. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley Interscience, New York, 2009)
6. P. Fieguth, *Statistical Image Processing and Multidimensional Modeling* (Springer, New York, 2010)
7. M. Fischetti, *Introduction to Mathematical Optimization* (Independent, Chicago, 2019)
8. M. Kochenderfer, T. Wheeler, *Introduction to Mathematical Optimization* (MIT Press, Cambridge, 2019)
9. K. Murphy, *Machine Learning: A Probabilistic Perspective* (MIT Press, Cambridge, 2012)
10. S. Rogers, M. Girolami, *A First Course in Machine Learning* (Chapman and Hall, London, 2016)
11. J. Shawe-Taylor, N. Cristianini, *Kernel Methods for Pattern Analysis* (Cambridge University Press, Cambridge, 2004)
12. A. Webb, K. Copsey, *Statistical Pattern Recognition* (Wiley, Chichester, 2011)



Ensemble Classification

We have seen a wide variety of approaches to classification, as summarized in [Table 11.1](#). In this chapter, we would like to explore whether we can combine *multiple* classifiers in order to outperform any single classifier. That is, all of the details of individual classifier learning were discussed in [Chapter 6](#) through [Chapter 10](#), classification strategies which are now part of our toolbox, so that we can treat classifiers as a black box, with the internal details less important, and focus on how to combine them.

If we are thinking of learning *multiple* classifiers to be combined, presumably we are contemplating multiple relatively *simple* classifiers as an alternative to complex or computationally heavy classifiers. As a result, from the list in [Table 11.1](#) we will primarily be motivated to explore some combination of multiple parametric distance-based classifiers ([Section 6.3](#)) and/or linear discriminants ([Section 10.1](#)).

Classifier Type	Example	Summary of Issues
Distance-Based — Parametric	Mahalanobis	Typically limited to simple class shapes
Distance-Based — Nonpar. Statistical	NN, kNN ML, MAP	Very general, but computationally slow Rely on class distributions being known
Discriminant — Linear	Linear SVM	Very simplistic (single hyperplane)
Discriminant — Nonlinear	Kernel SVM	Very general, may be hard to optimize

Table 11.1. An overview of the classifiers from preceding chapters, as a reminder. In this chapter we are considering ensembles of classifiers, so it is helpful to understand the individual classifiers as building blocks.

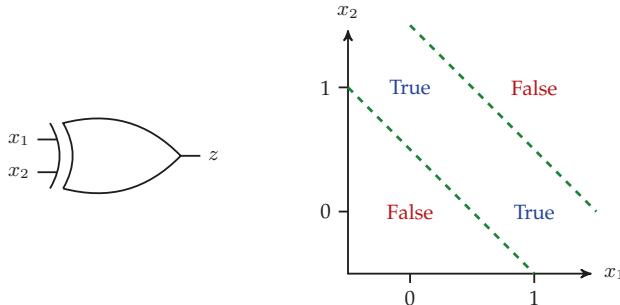


Fig. 11.1. XOR PROBLEM: Although an XOR (exclusive-or) logic gate is a very elementary piece of circuitry (left), it leads to a decision space (right) which cannot be solved via a single linear discriminant (try it!), since two boundaries (green dashed lines) are required. Clearly other nonlinear classifiers could solve this problem (NN, Mahalanobis), but not any single linear one.

The XOR (exclusive-or)¹ problem of Figure 11.1 was a classic challenge in pattern recognition and represents an excellent point to start our discussion. In particular, in contrast to other basic logic operations (OR, AND, NOT, NOR, NAND), all of which can be realized as a single linear discriminant (that is, using a single Perceptron, as in Figure 10.5), the XOR operation of Figure 11.1 is not linearly separable and therefore cannot be represented via a single Perceptron, a limitation which significantly derailed neural-network research in the early years of artificial intelligence.

It is relatively easy to see, from Figure 11.1, that *three* linear discriminants could perform the XOR function: two discriminants per the dashed green lines in Figure 11.1, and one more discriminant to combine the first two (see Problem 11.3). It is important to understand, however, the crucial role played by the nonlinear activation function, the thresholding of the linear discriminant:

$$\text{Input } \underline{x} \quad \text{Discriminant } h(\underline{x}) = \underline{w}^T \underline{x} + w_o \quad \text{Classifier } g(\underline{x}) = \begin{cases} 1 & h(\underline{x}) > 0 \\ 0 & h(\underline{x}) < 0 \end{cases} \quad (11.1)$$

That is, we need to quite clearly distinguish between the discriminant h , which is a *linear* function of \underline{x} , and the resulting classifier, $g(\underline{x})$, which is *not*. The fundamental issue is that a linear function of a linear function is, still, a linear function. Suppose we construct a hierarchy of layers, such that linear

¹ In logic there are concepts of *inclusive-or* and *exclusive-or*. In common parlance, “or” as an instruction usually means exclusive-or: you can have a cookie *or* a slice of cake (but not both). In responding to some signal, “or” is typically understood as inclusive-or: you can drive through the intersection if the light turns green *or* the police officer waves you through *or* both.

discriminants $h_{l\xi}$ on layer l are fed into linear discriminants $h_{(l+1)\zeta}$ on the next layer:

$$\underline{x} \rightarrow \begin{bmatrix} h_{11}(\underline{x}) \\ h_{12}(\underline{x}) \\ \vdots \end{bmatrix} = \underline{x}^{(1)} \rightarrow \begin{bmatrix} h_{21}(\underline{x}^{(1)}) \\ h_{22}(\underline{x}^{(1)}) \\ \vdots \end{bmatrix} = \underline{x}^{(2)} \rightarrow \dots \rightarrow \underline{x}^{(q)} \quad (11.2)$$

Then, as derived in [Appendix D](#), after q layers and a great many functions $h_{l\xi}$, the final result is *still* just a linear function of the input:

$$\underline{x}^{(q)} = A\underline{x} + \underline{b} \quad \text{For some matrix } A, \text{ vector } \underline{b} \quad (11.3)$$

That is, regardless how many layers of functions we add, we don't actually accomplish anything new or different. The whole network is still just a single linear function, and therefore can produce only a single linear discriminant, therefore *cannot* solve the XOR problem. The nonlinearity in (11.1) comes from the thresholding of $h \gtrless 0$, and so it is *classifiers* which we need to layer, and *not* discriminants.

11.1 Combining Classifiers

Having established that it is *classifiers* which we wish to combine, and not *discriminants*, the obvious next question is how to go about combining classifiers.

Given a set of classifiers $\{g_j(\underline{x})\}$, the most obvious way of combining them is by voting,

$$g_{\text{Combined}}(\underline{x}) = \arg_{\kappa} \max \sum_j \delta(\kappa, g_j(\underline{x})) \quad (11.4)$$

That is, finding that class that appears the most frequently (obviously, in practice, also needing some rule for tie-breaking). In principle the k NN-Global classifier, of [Section 6.4](#), was already of this form, accepting each of the k closest points as one vote in the schema of (11.4).

In all of the classifiers which we have seen, in addition to selecting the preferred class, in principle the classifier will also have an opinion of how strongly that class is supported, allowing a score to be computed,

$$s_{\kappa,j}(\underline{x}) = \text{Support for class } C_{\kappa} \text{ from classifier } g_j \text{ at feature point } \underline{x} \quad (11.5)$$

For example, for the statistics-based classifiers of [Chapter 8](#), we could propose

$$s_{\kappa}(\underline{x}) = p(\underline{x} | C_{\kappa}) \quad s_{\kappa}(\underline{x}) = P(C_{\kappa} | \underline{x}) \quad (11.6)$$

for ML and MAP, respectively. Similarly for the distance-based classifiers of [Chapter 6](#), since a small distance needs to correspond to a high score, we could propose

$$s_\kappa(\underline{x}) = \exp(-d(\underline{x}, C_\kappa)) \quad \text{or} \quad s_\kappa(\underline{x}) = \frac{\exp(d(\underline{x}, C_\kappa))}{\sum_t \exp(d(\underline{x}, C_t))} \quad (11.7)$$

for a simple or normalized score, respectively. In principle we could also formulate a score for discriminant-based classifiers. Given a discriminant² d_{C_+, C_-} which discriminates between class sets C_+, C_- and has no opinion regarding the classes in C_o , where

$$\mathcal{C} = \{C_+, C_-, C_o\} \quad (11.8)$$

then this discriminant's support for class κ can be scored as

$$s_\kappa(\underline{x}) = \begin{cases} 1/(1 + \exp(-h(\underline{x}))) & \kappa \in C_+ \\ 1/(1 + \exp(+h(\underline{x}))) & \kappa \in C_- \\ \frac{1}{2} & \kappa \in C_o \end{cases} \quad (11.9)$$

In those cases where a classification result is provided and the internal operations of the classifier (the distances, the likelihoods, the discriminants) are inaccessible to us, then the score defaults to just supporting the classification result:

$$s_{\kappa,j}(\underline{x}) = \delta(C_\kappa, g_j(\underline{x})) \quad (11.10)$$

Given a set of classifiers $\{g_j(\underline{x})\}$ and their respective scores $s_{\kappa,j}(\underline{x})$, we can choose to classify based on the single highest score,

$$\text{Winner Takes All: Classify } \underline{x} \text{ as } C_\kappa \text{ where } \kappa = \arg_t \max \left\{ \max_j s_{t,j}(\underline{x}) \right\} \quad (11.11)$$

but which has the significant drawback of being sensitive to outlying classifiers, allowing a single (possibly mislead) classifier to outvote all others. More robust would be

$$\text{Most Supported Class: Classify } \underline{x} \text{ as } C_\kappa \text{ where } \kappa = \arg_t \max_j \sum_j s_{t,j}(\underline{x}) \quad (11.12)$$

So we see that we can combine a number of classifiers, but why does this help? We begin with the concept of a *weak learner*. For a K class problem, a randomly-guessing classifier will satisfy

² A slightly more general notation is needed here, consistent with and building on [Section 10.4](#). Since a given discriminant just implies a boundary with two sides, it can choose to support one set of classes C_+ on the positive side, as opposed to another set of classes C_- on the negative side, and a third set C_o on which it renders no opinion.

$$\mathbf{P}(\text{Say } C_\kappa \mid \text{Truth is } C_\kappa) = \frac{1}{K} \quad (11.13)$$

Suppose, then, that we have a $K = 2$ class problem and a classifier g every so slightly better than random guessing:

$$\mathbf{P}_g(\text{Say } C_1 \mid \text{Truth is } C_1) = \mathbf{P}_g(\text{Say } C_2 \mid \text{Truth is } C_2) = \frac{1}{2} + \epsilon = \rho \quad (11.14)$$

If (a big “if”, to be further discussed ...) we have q independent such classifiers, then the overall behaviour is described by a binomial distribution³

$$\mathbf{P}_g(\tau \text{ classifiers out of } q \text{ say } C_1 \mid \text{Truth is } C_1) = \binom{q}{\tau} \rho^\tau (1 - \rho)^{q - \tau} \quad (11.15)$$

We know that τ must be an integer, since it is a count of classifiers reaching a certain conclusion, however we can approximate the binomial distribution as a Normal distribution over continuous τ :

$$p_g(\tau) = \mathcal{N}(q\rho, q\rho(1 - \rho)) \quad (11.16)$$

Then, it is more convenient to think about the *fraction* of classifiers, τ/q , which are selecting the correct class:

$$p_g(\tau/q) = \mathcal{N}(\rho, q\rho(1 - \rho)/q^2) = \mathcal{N}(\rho, \rho(1 - \rho)/q) \quad (11.17)$$

That is, on average a fraction ρ of the classifiers will be correct, and with a variance that shrinks as q increases.

Given q classifiers, we make an error if the majority are not selecting the correct class, meaning that

$$\tau < \frac{q}{2} \rightarrow \frac{\tau}{q} < \frac{1}{2} \quad (11.18)$$

That is, the probability that the combined q classifiers make an error is

$$\mathbf{P}_g(e \mid C_1) = \mathbf{P}_g(\tau/q < \frac{1}{2}) = Q\left(\frac{\rho - \frac{1}{2}}{\rho(1 - \rho)/q}\right) = Q\left(\frac{\epsilon \cdot q}{\rho(1 - \rho)}\right) \quad (11.19)$$

where $Q()$ is the integral of the Gaussian tail, from (8.48), so

$$\lim_{x \rightarrow \infty} Q(x) = 0 \quad \text{and} \quad \frac{\epsilon}{\rho(1 - \rho)} > 0 \quad \text{therefore} \quad \lim_{q \rightarrow \infty} Q\left(q \frac{\epsilon}{\rho(1 - \rho)}\right) = 0 \quad (11.20)$$

That is, a classifier g , arbitrarily close to random guessing, can achieve arbitrarily *perfect* performance ($\mathbf{P}(e) \rightarrow 0$) if there are sufficiently many classifiers being combined.

³ Essentially like flipping q coins, except that the coins are slightly biased towards heads or tails, not quite 50-50.

The intuition is simple: as long as the classifiers are better than guessing, then as the number of classifiers increases it becomes less and less likely that a *majority* of classifiers can coincidentally vote the wrong way.

In the $K > 2$ class case, the derivation is significantly more complicated, because in not choosing the correct class there is a *variety* of other classes to choose from, however the conclusion is the same: a weak learner, performing better than random guessing, performs ever better as q , the number of classifiers being combined, increases.

So, are we done? All we need to do is choose q sufficiently large? Actually there are two problems:

COMPUTATIONAL: Applying q classifiers to a feature vector is q times as much computational complexity, and possibly also q times as much storage requirement, as applying one classifier. In practice, the increase in computational complexity would be far *less* than q , because we should be comparing

Computational Complexity of one very capable classifier (e.g., nonlinear Kernel SVM)	versus	Computational Complexity of q very simple (weak) classifiers (e.g., linear discriminants)
--	--------	---

Computational complexity *may*, at times, be an issue, however the real problem is the second one ...

INDEPENDENCE: How do we find q *independent* classifiers? The preceding derivation with the binomial distribution relied on independent classifiers, meaning that the likelihood of error of one classifier must be unrelated to the likelihood of error of the next. In practice, classifier independence is actually quite a difficult condition to ensure, broadly leaving us with two strategies:

1. Generate non-independent classifiers, but very *many* of them, since for a given desired accuracy you will require more non-independent classifiers than independent ones;
2. Devise clever strategies for learning independent or near-independent classifiers.

Throwing endless classifiers at a problem is, in general, not very successful. There are, however, quite clever strategies which have been developed, and which are the subject of the next two sections.

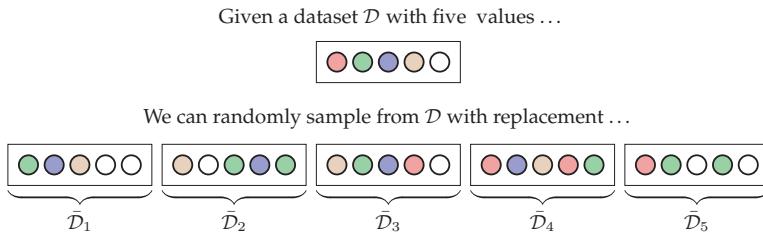


Fig. 11.2. BOOTSTRAP: From one dataset \mathcal{D} we can randomly (independently) resample an essentially indefinite number of bootstrapped datasets $\bar{\mathcal{D}}_1, \bar{\mathcal{D}}_2, \dots$

11.2 Resampling Strategies

Trying to adjust or modify a classifier learning strategy to end up with independent classifiers is very difficult. In almost all cases, we continue to use the same classifier learning methods of the preceding chapters, and achieve some degree of independence by modifying the *dataset*.

Given a dataset \mathcal{D} consisting of $N = |\mathcal{D}|$ data points, in [Chapter 9](#) (particularly [Section 9.3](#)) we saw ways, such as q -fold cross-validation, to divide \mathcal{D} into separate pieces for training and testing. In contrast the *Bootstrap*, a classic method of data analysis widely used in estimation (as illustrated in [Example 11.1](#)), takes a very different approach, constructing S separate datasets from \mathcal{D} , rather than dividing \mathcal{D} into pieces.

As shown in [Figure 11.2](#), we can construct S datasets, $\bar{\mathcal{D}}_1, \dots, \bar{\mathcal{D}}_S$ each consisting of \bar{N} data points, by sampling from \mathcal{D} *with replacement*, meaning that a given point in \mathcal{D} could be sampled multiple times in constructing $\bar{\mathcal{D}}_i$, and similarly other points in \mathcal{D} might not be sampled at all in $\bar{\mathcal{D}}_i$. Sampling with replacement is essential, since this is what makes each dataset $\bar{\mathcal{D}}_i$ essentially independently sampled from the distribution underlying \mathcal{D} . Although typically the resampled datasets are the same size as the original ($N = \bar{N}$), this is not a requirement, and \bar{N} could be larger or smaller than N .

Bagging (Bootstrap Aggregation): Given that we can create any number⁴ of datasets, the most obvious idea is then to train a classifier on each one:

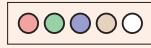
$$\mathcal{D} \xrightarrow{\text{Bootstrap}} \{\bar{\mathcal{D}}_i\} \xrightarrow[\text{Learning}]{\text{Classifier}} \{g_i(\underline{x})\} \xrightarrow{\text{Voting}} g_{\text{Bagging}}(\underline{x}) \quad (11.21)$$

where the voting could be weighted/scored or just a plurality vote among the classifiers, as discussed in [Section 11.1](#).

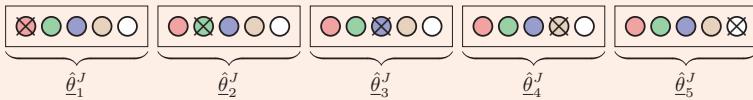
⁴ A dataset consisting of N unique values can be randomly sampled, with replacement, in N^N different ways. So for N of any realistic size, N^N is inconceivably large, meaning that we can create essentially indefinitely many bootstrapped datasets.

Example 11.1: Bootstrap and Estimation

Suppose we would like to estimate some parameter vector $\underline{\theta}$ from a given dataset \mathcal{D} :



We can compute $\hat{\underline{\theta}}(\mathcal{D})$, but how good is this estimate? Suppose we are unable to analyze the estimator mathematically; can we determine its bias or error covariance numerically? Certainly a *single* estimate $\hat{\underline{\theta}}(\mathcal{D})$ cannot say anything about bias or variability. We could find jackknifed datasets, and find an estimate $\hat{\underline{\theta}}_i^J$ from each one:



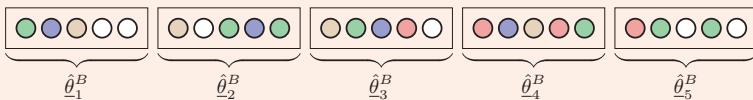
But note how similar the jackknifed datasets are — each pair is essentially identical to one another except for two values, which means that the estimates $\{\hat{\underline{\theta}}_i^J\}$ will be rather similar to one another, almost certainly *more* similar than if we had estimated $\hat{\underline{\theta}}$ from separate datasets (for example, if multiple datasets had been given to us).

Finding the mean of the jackknifed estimates $\hat{\underline{\theta}}_*^J = \frac{1}{N} \sum_{i=1}^N \hat{\underline{\theta}}_i^J$ leads to

$$\text{Bias}^J(\hat{\underline{\theta}}(\mathcal{D})) = (N-1)(\hat{\underline{\theta}}(\mathcal{D}) - \hat{\underline{\theta}}_*^J) \quad \text{Cov}^J(\tilde{\underline{\theta}}(\mathcal{D})) = \frac{N-1}{N} \sum_i (\hat{\underline{\theta}}_i^J - \hat{\underline{\theta}}_*^J)(\hat{\underline{\theta}}_i^J - \hat{\underline{\theta}}_*^J)^T$$

where Bias and Covariance are multiplied by $(N - 1)$, relative to their usual definitions, because of the significant similarity between jackknifed estimates.

In contrast, we could apply bootstrapping (Figure 11.2), randomly sampling S times from \mathcal{D} with replacement:



In contrast to jackknifing, note how significantly more different each of the bootstrapped datasets is from the others, meaning that the estimates $\{\hat{\underline{\theta}}_i^B\}$ will be much more variable. Indeed, because the bootstrap datasets are independently sampled, the variability among the estimates would be expected to be the same as on a new dataset, so the equations for bias and error covariance are not scaled as with jackknifing:

$$\text{Bias}^B(\hat{\underline{\theta}}(\mathcal{D})) = (\hat{\underline{\theta}}(\mathcal{D}) - \hat{\underline{\theta}}_*^B) \quad \text{Cov}^B(\tilde{\underline{\theta}}(\mathcal{D})) = \frac{1}{N} \sum_i (\hat{\underline{\theta}}_i^B - \hat{\underline{\theta}}_*^B)(\hat{\underline{\theta}}_i^B - \hat{\underline{\theta}}_*^B)^T$$

where $\hat{\underline{\theta}}_*^B = \frac{1}{N} \sum_{i=1}^N \hat{\underline{\theta}}_i^B$ is the mean of the bootstrapped estimates. A significant strength of the bootstrapping approach is that the number of synthesized datasets, S , can be chosen to be less than N (for computational reasons) or greater than N (for statistical accuracy), a flexibility that jackknifing cannot offer.

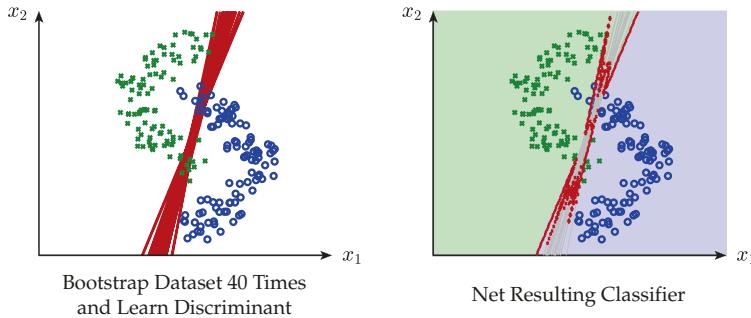


Fig. 11.3. BAGGING: The bagging approach to classifier learning applies the bootstrap method to sample independent datasets (Figure 11.2). However the datasets all follow the same distribution and so lead to very similar discriminants (left) which, when combined (right) are not a significant improvement over a single discriminant.

The idea is elegant and straightforward, however although the bootstrapped datasets $\{\bar{\mathcal{D}}_i\}$ are independently drawn, they are all drawn from the *same* underlying (unknown) distribution, and consequently the classifiers $\{g_i(\underline{x})\}$ are typically very similar, as shown in Figure 11.3. That is, the classifiers do *not* have independent errors, and therefore do not produce the richness/diversity of classifiers needed to create a strong learner from an ensemble of weak ones.

Bagging is sometimes applied to decision trees of Section 11.3, since decision tree learning can lead to significant variations with only minor perturbations in the dataset (usually not a desirable characteristic), which can make the decision tree more amenable to the dataset-to-dataset variations produced by bootstrapping.

Boosting: The limitation of bagging is that the datasets are, although independently sampled, still too similar, and lead to very similar classifiers.

Boosting takes a very different approach, sampling datasets and learning classifiers *dependent on* the behaviour of earlier classifiers.

Suppose we begin with a $K = 2$ class problem with given dataset \mathcal{D} . Construct \mathcal{D}_1 by selecting, at random (without replacement), approximately one third of the points from \mathcal{D} :

→ Train classifier $g_1(\underline{x})$ from \mathcal{D}_1

If classifier $g_1(\underline{x})$ perfectly separates the two classes, then nothing more needs to be done; however in every case of practical interest $g_1(\underline{x})$ will be imperfect. We wish \mathcal{D}_2 to be as informative as possible, so we have $g_1(\underline{x})$ classify all points in $\mathcal{D} \setminus \mathcal{D}_1$ and sample into \mathcal{D}_2 , at random, between points correctly and incorrectly classified by $g_1(\underline{x})$:

→ Train classifier $g_2(\underline{x})$ from \mathcal{D}_2

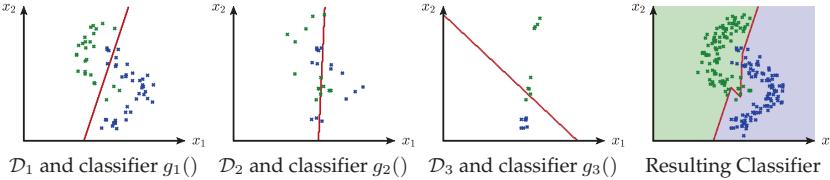


Fig. 11.4. BOOSTING: The principle of Boosting is to start with some dataset \mathcal{D}_1 , and then to construct further datasets, and associated classifiers, which seek to address perceived limitations of earlier classifiers. Here MSE was used to learn the discriminants.

Finally classifier $g_3(\underline{x})$ is specifically learned as a tie-breaker between $g_1(\underline{x})$ and $g_2(\underline{x})$. Let \mathcal{D}_3 consist of all points in $(\mathcal{D} \setminus \mathcal{D}_1) \setminus \mathcal{D}_2$ which are *differently* classified by $g_1(\underline{x})$ and $g_2(\underline{x})$:

→ Train classifier $g_3(\underline{x})$ from \mathcal{D}_3

The process is illustrated in [Figure 11.4](#), where the MSE method of [Section 10.2](#) is used. Clearly linear-SVM could have been used to learn a discriminant, however that somewhat defeats the purpose of ensemble classifiers, which is to learn *simple* classifiers. If you are going to the trouble of setting up SVM, then it probably makes more sense to use a nonlinear kernel and learn a sophisticated classifier. As it turns out, however, linear-SVM actually did not produce boosted results any better than those based on MSE, as we will see in [Figure 11.5](#).

The limitation of both MSE and SVM is that, in seeking a “good” discriminant, there is too little variability to explore very different classifiers. This then limits the ability for the boosting approach to combine the learned g_1, g_2, g_3 classifiers to look very different from a single discriminant. That is, the uninspiring results in [Figures 11.4](#) and [11.5](#) have much more to do with the method of classifier learning, than the principle of boosting.

To demonstrate this, suppose we attempt to boost *random* discriminants, found by choosing a point at random from each class (at each of the three learning stages). There are many variations possible, however here are three very basic strategies:

1. Randomly select one pair of points;
2. Randomly select pairs of points until the resulting discriminant performs better than random guessing (satisfies the weak-learner condition);
3. Randomly select some number (say 20) of pairs of points, and keep the best (clearly not optimal, just the best of 20 choices).

The results are shown in [Figure 11.5](#), and begin to reveal what boosting is able to do with far greater variability in learning, particularly (weak-random and best-random) when the chosen discriminant is at least plausibly good. The very last result in [Figure 11.5](#) applies bagging; in sharp contrast to [Figure 11.3](#),

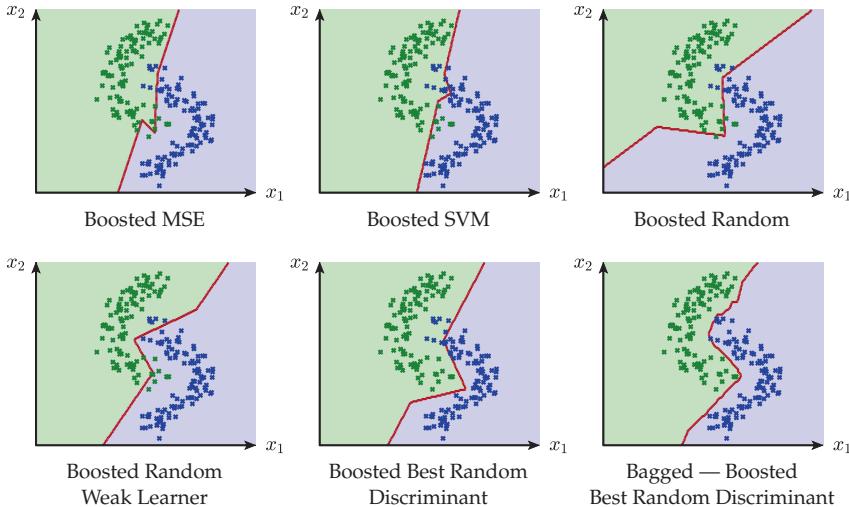


Fig. 11.5. BOOSTING: Building on Figure 11.4, the approach to learning the individual classifiers has a significant impact on the boosted result. Both MSE and SVM seek a *single*, good discriminant, so these methods tend to produce very similar classifiers in each of the boosting stages, limiting the ability for boosting to actually produce something different. In contrast, choosing points at random leads to a far richer set of discriminants which boost/bag quite successfully.

the much greater variability here of the underlying classifiers leads to a very plausible classifier, all based on very fast learning (random point pairs) and a classifier built up of a set of efficient discriminants.

In the same spirit of the wrapper methods of Example 3.4, boosting can be applied in a nested/recursive way, such that all (or some) of g_1, g_2, g_3 are, themselves, boosted classifiers. In principle such a recursion could be applied multiple times, with increasing flexibility to accommodate unusual class shapes, but also at an increased risk of overfitting.

AdaBoost (Adaptive Boosting): A great many variations on boosting have been proposed; by far the most widely used is AdaBoost (Adaptive Boosting), which allows for more than three classifiers, and which is also much more explicit (Algorithm 11.1) in its implementation.

Once boosting has been understood, the AdaBoost procedure is quite straightforward, and consists of three parts:

1. At each iteration j there is a set of weights $w_{j,i}$ (a distribution) over the dataset \mathcal{D} , indicating the importance of each data point \underline{x}_i (that is, the importance that it be classified correctly), based on the performance of

Algorithm: AdaBoost – Adaptive Boosting

Goals: Learn a set of S classifiers $\{g_j(\underline{x})\}$ based on dataset $\mathcal{D} = \{\underline{x}_i, c_i\}$

Function Classifiers $\{g_j\} = \text{AdaBoost}(\mathcal{D}, S)$

```

 $N = |\mathcal{D}|$ 
 $w_{1,i} = 1/N$  Initialize data weights uniformly across all data points
for  $j = 1 : S$  do
    Option 1: Train classifier  $g_j$  to minimize  $\sum_i w_{j,i}(1 - \delta(c_i, g_j(\underline{x}_i)))$ 
    Option 2: Train classifier  $g_j$  based on dataset  $\mathcal{D}_j$  sampled from  $\mathcal{D}$  based
    on  $\{w_{j,i}\}$ 
     $e_j = \sum_i w_{j,i}(1 - \delta(c_i, g_j(\underline{x}_i)))$  Compute weighted classifier error
     $\alpha_j = \frac{1}{2} \ln((1 - e_j)/e_j)$  Compute classifier weight
     $\bar{w}_{j+1,i} = w_{j,i} \cdot \exp[\alpha_j(1 - 2 * \delta(c_i, g_j(\underline{x}_i)))]$  Update data weights
     $w_{j+1,i} = \bar{w}_{j+1,i} / \sum_i \bar{w}_{j+1,i}$  Normalize data weights
end for

```

Algorithm 11.1. The boosting method is fairly intuitive, however there is some ambiguity in constructing the three datasets, and no guidance on how to add additional classifiers. AdaBoost is much cleaner by being much more specific about what datasets to use for learning, and is in no way limited to three classifiers.

previous classifiers on this same data point. The more previous classifiers g_1, \dots, g_{j-1} have classified a point correctly, the lower its present weight.

2. Each classifier g_j is trained, biased by the weights $w_{j,i}$, either by modifying the objective function of the classifier, or by presenting a dataset \mathcal{D}_j formed by sampling from \mathcal{D} based on $w_{j,i}$.
3. Each classifier g_j has an associated weight α_j , based on how well classifier g_j was able to address its given task.

The resulting classifier is then a combination of the $\{g_j\}$ ensemble. If the two classes are understood to numerically evaluate to -1 and $+1$, then the weighted classifier could be written as

$$g_{\text{AdaBoost}}(\underline{x}) = \text{sign}\left(\sum_j \alpha_j g_j(\underline{x})\right) \quad (11.22)$$

More generally, we would use the scored voting approach of (11.12).

AdaBoost has been found to be particularly effective when applied to decision-tree classifiers of Section 11.3.

Stacking: The preceding methods of Bagging, Boosting, and AdaBoost all sought to create an ensemble of homogeneous weak learners; that is, all three methods were not specific about what sort of underlying classifier should be learned, but then had an explicit, deterministic approach for combining the

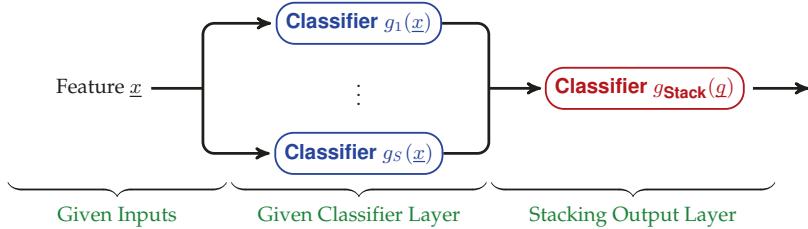


Fig. 11.6. STACKING is much more open-ended than the related bagging, boosting, and adaboost. In particular, preceding methods focused on how the classification layer (blue) should be learned, and then had a deterministic rule for combining (red). In contrast, Stacking accepts a set of classifiers (blue) as given, and the key job of the stacking approach is how to learn the output classifier (red).

classifiers via a specific rule, whether evenly weighted (bagging, boosting) or adaptively and unevenly weighted (adaboost).

Stacking is a very different approach, and more of an overall philosophy, illustrated in Figure 11.6, rather than a particular method. Stacking accepts S classifiers $g_1 \dots g_S$, possibly heterogeneous and very differently learned, and then learns another classifier, g_{Stack} , essentially as a wrapper, such that g_{Stack} learns the strengths and weaknesses of the given classifiers, producing a strong overall classifier.

The main consideration is how to allocate data for training. Given a dataset \mathcal{D} , if that dataset is used for training the given classifiers $\{g_i\}$, then we are not likely to produce an effective stacking classifier g_{Stack} if it, too, is trained on \mathcal{D} , because of overfitting. That is, we wish for g_{Stack} to have a realistic assessment of the classification ability of the input classifiers. We could therefore divide the dataset

$$\mathcal{D} = \{\mathcal{D}_{\text{Input}}, \mathcal{D}_{\text{Stack}}\}$$

so that the input and stacking classifiers are separately trained. Obviously using only half of the data may be undesirable, in which case we can undertake the equivalent of q -Fold Cross-validation from Section 9.3, repeatedly training input classifiers on most of the data (the $q - 1$ folds) and then learning the stacking classifier on the remaining fold, but at an increase in computational effort for training.

In the same way that boosting was able to be applied recursively, in a multi-layered fashion, in principle the same can be done with stacking, such that rather than a single classifier g_{Stack} , we could have a set of classifiers $\{\bar{g}_j\}$ seeking improve the classification of the input set of classifiers $\{g_j\}$, with the final output assessment g_{Stack} applied to the preceding layer:

$$\underline{x} \rightarrow \text{Classifiers } \{g_j(\underline{x})\} \rightarrow \text{Stacking Layer } \{\bar{g}_j(q)\} \rightarrow \text{Stacking Output } \{\bar{g}_{\text{Stack}}(\bar{q})\} \quad (11.23)$$

This might seem to be getting overcomplicated, and not at all clear how all of these classifiers should be learned and optimized. In fact, such a multi-layered set of classifiers is *precisely* how large (deep) neural networks are formed. So while it is helpful to have developed the conceptual picture of how stacking built on ideas of ensemble-based classifiers, it is not particularly helpful to attempt to formulate the stacking approach in more detail, since it is deep networks, to be discussed in [Section 11.4](#), which are more capable, more systematic in how they learn, and in far greater widespread use.

11.3 Sequential Strategies

The resampling-based ensemble classifiers from [Section 11.2](#) offer highly attractive ways of resampling independent datasets and of combining multiple classifiers. The single greatest limitation was identifying promising weak-learners to be combined. In particular, the linear discriminants from [Chapter 10](#) are very good as *single* discriminants, but do not have the variety, variability, or the appropriate criteria for working well as an ensemble.

Furthermore, except for the preliminary discussion in [Section 10.4](#), most of the methods we have seen (MSE, SVM, Boosting, AdaBoost) all assume the $K = 2$ class problem, and surely we need better support for multi-class contexts.

Let us return to the two-arc dataset, which we have seen repeatedly in [Figures 11.3, 11.4, and 11.5](#), and which has largely stymied past attempts to learn discriminant ensembles. Surely it cannot be that difficult ... how would we design a set of discriminants by hand to solve this problem? Almost certainly we would progress sequentially, classifying certain parts of the problem, as illustrated in [Figure 11.7](#), progressively making the problem easier, yielding a combined classifier known as a *Decision Tree*. It is attractive that the multi-class case, also sketched in [Figure 11.7](#), can also be approached by precisely the same logic and approach.

So what was the limitation in choosing linear discriminants in [Chapter 10](#)? After all, we derived a number of objective functions to be optimized, in order to find the *best* discriminant.

The issue is that all of the linear discriminants in [Chapter 10](#) were finding the best *single* discriminant, whereas in decision trees a given discriminant has the luxury, so to speak, of knowing that there are more discriminants still to come. This significantly changes the optimization objective. Rather than a

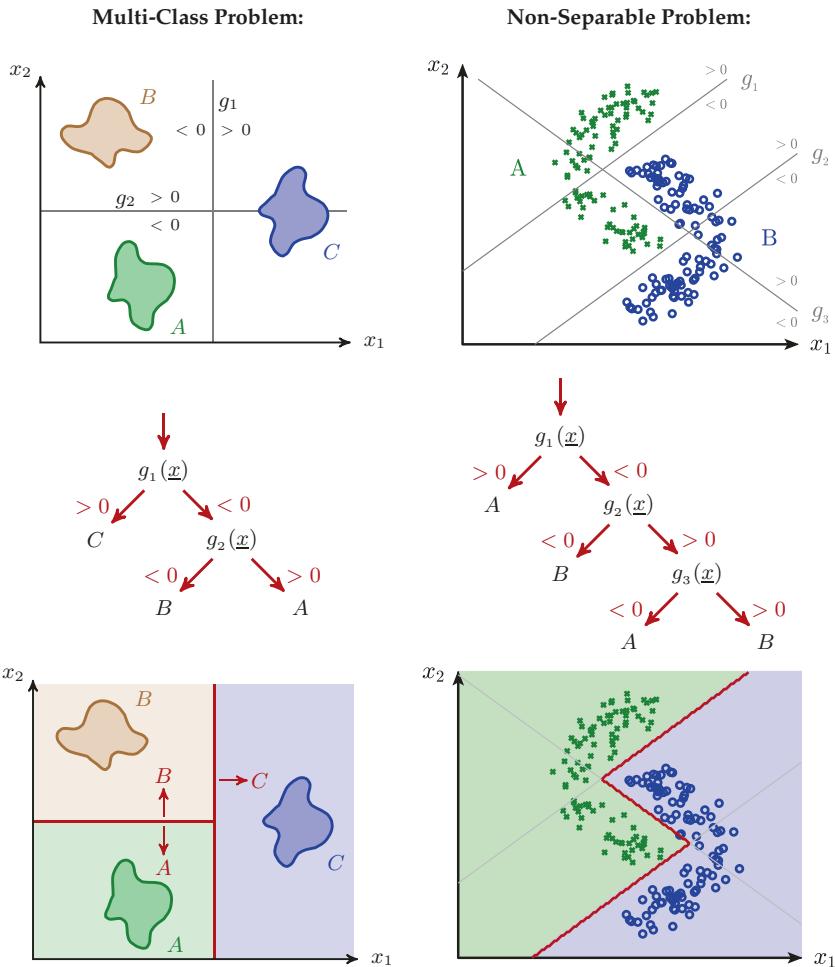


Fig. 11.7. DECISION TREES: Given a multi-class problem (left) or a non-separable problem (right), how might we go about effectively solving them using one or more binary classifiers? The obvious answer would seem to be some sort of sequential approach, known as a decision tree (middle), leading to an effective classification (bottom).

global objective, like the overall classification error of (10.13) or the maximum margin of (10.32), we can state more of a local objective:

Find a linear discriminant which most greatly simplifies the classification problem

That is, given an overall dataset \mathcal{D} , which some discriminant h will divide into two parts, we want to maximize how much h is able to reduce the problem complexity:

$$\hat{h} = \arg_h \max \underbrace{\mathbb{E}[\text{Complexity}(\mathcal{D})]}_{\text{Overall complexity of problem before } h} - \underbrace{\mathbb{E}[\text{Complexity}(\mathcal{D} | h > 0)]}_{\text{Problem complexity on positive side of } h} - \underbrace{\mathbb{E}[\text{Complexity}(\mathcal{D} | h < 0)]}_{\text{Problem complexity on negative side of } h} \quad (11.24)$$

In other words, we really do not need that h do a good job of classifying \mathcal{D} ; what we need is for h to significantly simplify the problem, so that we will not need too many *more* discriminants to complete the classification task.

Presumably what we need, then, is some measure of complexity. The most widely used approach for decision trees is based on *Information Theory*, which was introduced in the [Further Reading to Chapter 5](#). Very briefly, given a dataset \mathcal{D} consisting of samples from each of K classes, the *entropy* associated with \mathcal{D} is

$$\text{Entropy}(\mathcal{D}) = - \sum_{\kappa=1}^K \mathbf{P}(C_\kappa) \log_2(\mathbf{P}(C_\kappa)) \quad (\text{in units of bits/sample}) \quad (11.25)$$

So the entropy measures the number of bits per sample, on average, to describe a dataset. If a discriminant h divides \mathcal{D} into two pieces, \mathcal{D}_1 and \mathcal{D}_2 , the remaining complexity is the weighted entropy

$$\text{Entropy}(\mathcal{D} | h) = \frac{|\mathcal{D}_1|}{|\mathcal{D}|} \text{Entropy}(\mathcal{D}_1) + \frac{|\mathcal{D}_2|}{|\mathcal{D}|} \text{Entropy}(\mathcal{D}_2) \quad (11.26)$$

as illustrated in [Figure 11.8](#).

The entire premise is that a single discriminant does *not* (and is not expected to) solve the problem, so then each of the resulting problems, \mathcal{D}_1 and \mathcal{D}_2 , is further subdivided $\mathcal{D}_{1,1}, \mathcal{D}_{1,2}, \mathcal{D}_{2,1}, \mathcal{D}_{2,2}$ etc., by further discriminants. There is an obvious stopping condition, which is when any piece of the problem has

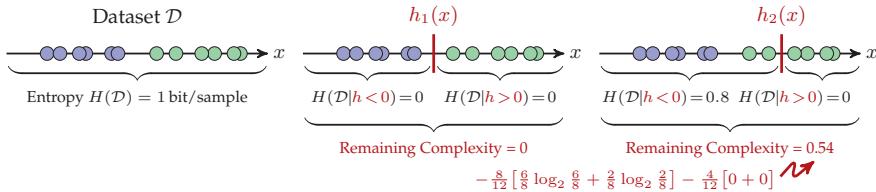


Fig. 11.8. ENTROPY: A simple example can illustrate what entropy reveals about the choice of discriminant. Given a dataset \mathcal{D} having six points in each of two classes (left), we could propose two discriminants, h_1 and h_2 , to divide \mathcal{D} . The original problem has equal numbers of blue and green points, and therefore has an entropy (the information complexity) of one bit per sample. Discriminant h_1 divides the problem perfectly, leaving an entropy of zero (i.e., no remaining uncertainty); h_2 divides the problem imperfectly, leaving a non-zero entropy.

Entropy() equal to zero, meaning that the remaining problem at that point in the tree has elements from only one class, what we would refer to as a leaf or terminating node of the tree. As we saw in both illustrations in Figure 11.7, the tree does not need to be symmetric: at a given partitioning, one side may have an entropy of zero (leaf node), and the other side may require further splitting.

Clearly the above process, by which the problem is successively split until every region or sub-problem has an entropy of zero, guarantees that *every* training point is classified correctly. This process should (hopefully, at this point) make us concerned about overfitting. There are a number of ways in which overfitting can be limited:

LIMIT TREE DEPTH: One can assert the maximum number, T , of successive classifiers that will be permitted, such that the splitting process is terminated at that point. This does, of course, require significant understanding into the complexity of the problem domain ahead of time, and it may be unrealistic to choose T well.

ALLOW IMPERFECT LEAVES: In the same way that the soft-margin approach of (10.35) allowed for some points on the wrong side of the discriminant hyperplane, or the Neyman-Pearson test of Figure 9.5 explicitly allows for some specified fraction of error for some class, similarly it would be straightforward to specify some upper limit $P(e|C_\kappa) \leq \beta_\kappa$ for each class. Keep in mind that β_κ is a threshold for each tree leaf, not the overall problem as a whole: each learned discriminant would still seek to maximize the entropy difference of (11.26), which would frequently be associated with $P(e|C_\kappa) = 0$ at most of the tree leaves.

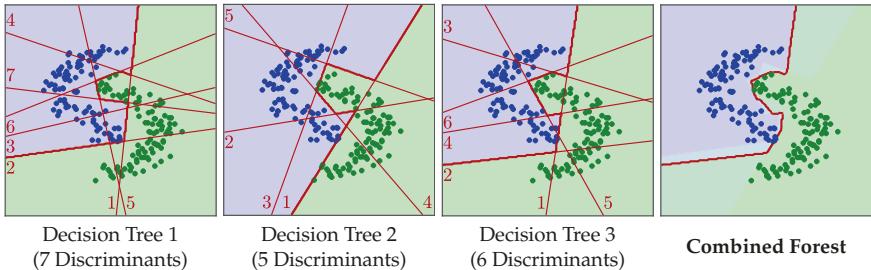


Fig. 11.9. DECISION TREES are easily able to solve the two-arc problem, which we have repeatedly seen, by using an entropy-based criterion on a sequence of discriminants. For a given training set, not only the specific discriminants, but even the shape or size of the decision tree (as revealed here by the *number* of discriminants) can vary. The resulting random forest, right, is a very credible solution, based on only a few very simple operations (one dot product per decision).

BUILD AND PRUNE: Allow the decision tree to be generated freely, with no constraints, leading to an overfit classifier. Then use any of the cross-validation techniques of [Section 9.3](#) to test leaf nodes, one at a time, and compare the classifier performance with and without that particular discriminant, and keep removing discriminants as long as the performance is improved.

We therefore have a relatively simple, conceptually elegant, and very flexible approach to classification ([Figure 11.9](#)), easily capable of handling multiple ($K > 2$) classes, and classes which are not linearly separable. A few comments and variations:

- Decision Trees do not have to be based on linear discriminants. Any simple, easily-learned binary classifier can be used.
- Decision trees offer significant flexibility. Nearly all methods in this text relied on a real-valued measurement vector $y \in \mathbb{R}^m$, whereas a decision tree just requires some way of partitioning a dataset, which can be undertaken equally well given real, discrete, or even cardinal (unordered) values.
- Decision trees have a major advantage of being intuitive and interpretable. That is, the resulting decision trees, such as sketched in [Figure 11.7](#), are easily explained and graphically displayed in a way that is easy for people to interpret.
- Indeed, there are many circumstances (particularly those involving cardinal measurements) in which the decision tree algorithm is constrained to base each decision on only a single measurement. This has the distinct advantage that all decisions are based on measurements with understood units, rather than a hyperplane in n dimensions which may combine measurements with incompatible units in some counterintuitive way.

- The entropy criterion of this section is based only on whether data points are lying on the correct side of each discriminant. However the same principles of hard or soft margins, as discussed in [Section 10.2](#), can similarly be applied here.

It is often cited as a limitation or disadvantage that decision trees can be very non-robust, in that a small change in the training data can lead to a large change in the resulting decision tree. In all cases, if properly learned, the resulting final classifiers will be very similar; the issue is that the actual sequence of decisions, the decision tree itself and its associated interpretation, can vary with small perturbations in the data. Essentially this is nothing more than saying that there are multiple decision trees which give rise to very nearly the same classifier. In fact, the sensitivity of decision trees to the underlying dataset makes decision trees exceptionally *well* suited to bagging, a combined approach known as a *random forest*.

Random Forest:

In [Section 11.2](#), Bagging and Boosting sought to improve weak learners on the basis of resampling the dataset, to lead to *differently* (ideally independently) learned classifiers. In practice, the discriminant methods of [Chapter 10](#) led to only small variations (and thus *not* independently learned) in classifiers from one sampled dataset to another, which limited the benefit anticipated from Bagging and Boosting.

Decision trees, as just discussed, can be quite sensitive to the training data, and so exhibit a desirable degree of variability which leads to more effective recombining in bagging. The random forest method is, then, very straightforward: Given a dataset \mathcal{D} , sample with replacement S times to produce datasets $\mathcal{D}_1, \dots, \mathcal{D}_S$. From each a decision tree is learned, and the resulting S trees are combined via majority vote.

There are two techniques which can be used to encourage further diversity in decision-tree learning:

1. How are the individual binary discriminants found? We may not want the computational complexity of an exhaustive, global optimizer, so a sub-optimal stochastic (random) Monte Carlo method could be proposed, as discussed and illustrated in [Lab 11](#). Because decision trees are already sensitive to subtle variations in the training data, they also exhibit significant variation based on suboptimal discriminants.
2. If the method of discriminant learning does not exhibit sufficient diversity, even in light of dataset resampling via bagging, then it is common, for each decision, to sample at random from the measurements or features. This does *not* mean sampling from the training data points; instead, to

actually *reduce* the number of measurements or features by randomly sampling, without replacement.⁵ The rationale is that, for certain classification problems, one or two features may dominate the decision process, making successive trees very similar, and reducing the benefit of bagging. By sampling among measurements or features we encourage a greater degree of feature assessment variability. Given m measurement dimensions or n feature dimensions, typically one might sample \sqrt{m} measurement or \sqrt{n} feature dimensions, at random, for each decision.

Although much of pattern recognition has been come to be dominated by large neural networks, to be described in [Section 11.4](#), there are certain key advantages that decision trees/random forests have:

- Interpretability: Each decision, each tree, and the process of combining them into a forest are all conceptually straightforward.
- Complexity: Each decision is just a single dot product, therefore each decision tree is just a sequence of dot products and if-statements.
- Flexibility: The random forest is built upon some number of decisions, each of which is just a binary separation of a measurement or feature. For most of our discussion we have assumed this to be a linear discriminant from [Chapter 10](#),

$$h(\underline{x}) \begin{array}{c} \text{Decision 1} \\ \gtrless \\ \text{Decision 2} \end{array} 0 \quad (11.27)$$

The decision could equally well be based on some thresholded distance, as in [Chapter 6](#),

$$d(\underline{x}, \underline{z}) \begin{array}{c} \text{Decision 1} \\ \gtrless \\ \text{Decision 2} \end{array} \tau \quad (11.28)$$

However really *any* binary decision could be applied, which includes cardinal (non-ordered) data. For example, if some measurement space

$$\mathcal{Y} = \{\text{Maple, Oak, Beech, Pine, Cedar}\} \quad (11.29)$$

describes a cardinal measurement y , then we can formulate a binary question

$$\begin{array}{ll} \text{Is } y \text{ in ...} & \{\text{Maple, Oak, Beech}\} \rightarrow \text{Decision 1} \\ & \{\text{Pine, Cedar}\} \rightarrow \text{Decision 2} \end{array} \quad (11.30)$$

That is, we can create a decision based on partitioning set \mathcal{Y} into two subsets.

⁵ It would not make sense to sample dimensions *with* replacement: We do not want a given dimension to show up more than once, since that just creates unnecessary correlation.

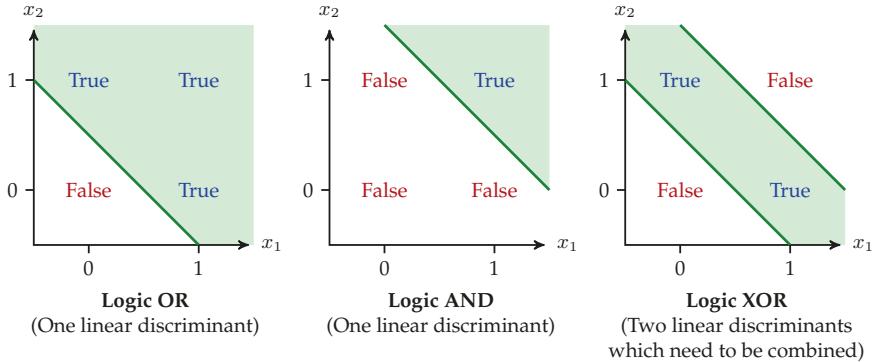


Fig. 11.10. BASIC LOGIC OPERATIONS: A single discriminant (and, therefore, a single neuron) is able to mimic an OR or AND operation, but not XOR, which requires two hyperplanes. Two neurons would not be enough to represent XOR, since the two hyperplanes (one neuron each) require one *further* neuron to combine the first two (see Problem 11.3).

11.4 Nonlinear Strategies

The single perceptron of [Section 10.2](#) is really a very limited classifier ([Figure 11.10](#)), able to realize only a single linear discriminant. Any classifier of significance necessarily needs to combine an ensemble of discriminants, as has been discussed throughout this chapter. In principle the neuron model of [Figure 11.11](#) allows for a connection to be asserted between any two neurons N_α and N_ξ , allowing for arbitrary topologies, however it is much simpler to focus our attention to what are known as multi-layer feedforward networks, as illustrated in [Figure 11.12](#). Here the neurons are organized into layers, whereby the inputs to neurons at some layer are limited to neuron outputs from the preceding layer. All of the details of neuron layers and weights are internal to the network, meaning that as a pattern recognition tool it can be helpful to view such a network as complex nonlinear function,

$$\text{Input Vector } y \rightarrow \begin{pmatrix} \text{Nonlinear} \\ \text{Neural Network} \end{pmatrix} \rightarrow g(y) \text{ Discriminant Output} \quad (11.31)$$

For now we will have to accept some ambiguity regarding what g actually represents: g could be real-valued, a continuous nonlinear function of the inputs, or it could be discrete, acting more like a classifier.

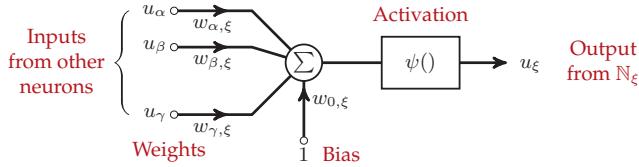


Fig. 11.11. NEURON NOTATION: Building on Figure 10.5, here the notation for the inputs, weights, and output for a single neuron N_ξ .

11.4.1 Neural Network Learning

The most fundamental question is how we might train such a network, how to perform learning on the weights $\{w_{\alpha,\beta}\}$. Here we are doing supervised learning,⁶ meaning that training data $\{\underline{y}_i, \bar{g}_i\}$ are available which implicitly specify the input–output relationship we would like the network to learn. That is, for each input vector \underline{y}_i which is provided to the network input, we have a target value \bar{g}_i for the discriminant outputs in (11.31).

Suppose our network has L layers $\mathbb{L}_1, \dots, \mathbb{L}_L$, such that $\xi \in \mathbb{L}_l$ means that neuron N_ξ appears in layer l . The network inputs appear at the input layer \mathbb{L}_1 , and the network outputs emerge from the output layer \mathbb{L}_L . For notational convenience, as shown in Figure 11.12, the neurons N_j in the output layer are indexed to match the network outputs g_j .

We begin the learning process at the output layer. The learning process is based on gradient descent, such that given a loss function \mathcal{L} and an unknown parameter (weight) w we move the weight (from iteration to iteration) in the direction opposite to the gradient,

$$w^{t+1} = w^t - a \frac{\partial \mathcal{L}(w)}{\partial w} \quad (11.34)$$

where a controls the rate of weight adjustment relative to the size of the gradient. In practice we will of course have *many* weights, not just the one weight implied by (11.34).

For each network input y_i we have an associated output *target* \bar{g}_i , which we then compare to the *actual* network output $g(y_i)$, so we can propose a mean-squared loss function

$$\mathcal{L}(\{w_{\alpha,\beta}\}) = \frac{1}{2} \sum_i \sum_{j \in \mathbb{L}_L} \underbrace{(g_j(y_i | \{w_{\alpha,\beta}\}))}_{j\text{th network output}} - \underbrace{(\bar{g}_{ij})^2}_{j\text{th target}} = \frac{1}{2} \sum_i \sum_{j \in \mathbb{L}_L} (u_j - \bar{g}_{ij})^2 \quad (11.35)$$

⁶ Unsupervised network learning will be discussed in Section 12.2.

Example 11.2: Biological and Artificial Neurons

Figure 11.11 illustrated the Perceptron model of a neuron, in which we compute a weighted sum of a set of inputs, followed by an activation function. The simplest activation function is just a step (on/off) threshold response:

$$\text{Neuron Output} = \begin{cases} \text{On (High)} & \text{Weighted sum} > \text{Threshold} \\ \text{Off (Low)} & \text{Weighted sum} < \text{Threshold} \end{cases} \quad (11.32)$$

Such a threshold is discontinuous. To make the learning/optimization more reliable and straightforward, in practice continuous neuron-activation models are preferred, such as

$$\text{Sigmoid}(x) = \frac{1}{1 + \exp(-x)} \quad \text{or} \quad \text{ReLU}(x) = \max(0, x) \quad (11.33)$$

plus many other variations.

A biological neuron similarly consists of inputs (dendrites), weights, summation, activation, and output (axon): that is, it would appear to be qualitatively similar in structure to the artificial neuron, and with a calculation which is also similar.

There are, however, very significant differences:

FUNCTION: There are research groups seeking to model and simulate brain function, however the overwhelming majority of large artificial neural networks are simply enormous classifier ensembles, which train well for certain pattern recognition objectives, and which make no particular attempt to actually mimic biological brain function.

TIME: The analytical neuron input-output model of (11.37) is instantaneous, with no sense of time. Similarly a large network of neurons is also mathematically represented as instantaneous, even if the underlying computation actually does take time. Artificial networks *have* been developed with temporal elements (delays, memory, etc.), however the basic neuron discriminant model incorporates no time. The biological neuron, as a living cell, is clearly very much a function of time.

SIGNAL REPRESENTATION: In electrical circuits it is typically convenient to represent signal strength by an amplitude, normally a voltage. In biological neurons, the neuron is indeed performing a calculation similar to that of the Perceptron of **Figure 10.5**, however the signal strength is represented by the *frequency* of pulses, and not the amplitude.

SIGNAL PROPAGATION: Biological (clearly including human) neurons use electricity, pumping sodium and potassium ions to create a voltage difference, having a signal propagation speed *much* slower than what most people would think. In unmyelinated (uncoated) axons the signal speed is only approximately 1 m/s, and in myelinated (protected/coated) axons of roughly 100 m/s, a tiny fraction of the speed of propagation of electrical signals of roughly 200 000 km/s.

Example continues ...

Example 11.2: Biological and Artificial Neurons (continued)

To be fair, even in a wire, the electrons typically move on the order of 1 mm/s (yes, only a few meters per *hour*!). How is it, then, that an electrical signal moves at two-thirds of the speed of light? The process is highly analogous to a garden hose — the water molecules actually move relatively slowly, but when you turn on the tap the water almost instantly comes out the far end, since the pressure wave travels *much* faster than the water molecules themselves. Similarly an electrical potential (“pressure”) travels down a wire *much* faster than the individual electrons.

There are many [Biological neuron models](#), particularly the [Hodgkin-Huxley model](#).

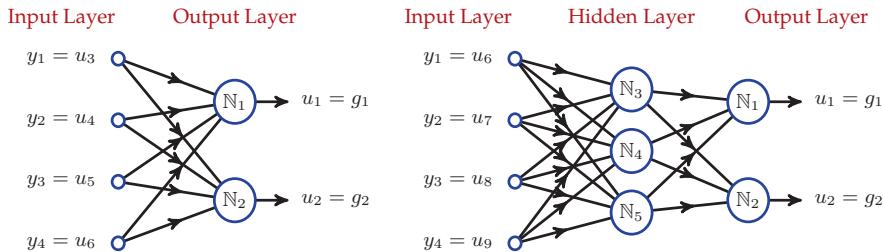


Fig. 11.12. NETWORK TOPOLOGY: How might an ensemble of neurons, each as in Figure 11.11, be organized? A single layer of neurons, left, all based on the same inputs, generate a set of discriminants (one per neuron), however in order to further *combine* these discriminants in some way requires a second layer, right, such that the outputs of some neurons become the inputs to others. Clearly more general topologies of three, four, or more layers of neurons can be constructed. The overall network performs a nonlinear transformation of input vector \underline{y} to output discriminants $\{g_j\}$.

where the sum is taken over all of the training data indices i , and over all of the neuron labels j in the output layer \mathbb{L}_L .

A given neuron \mathbb{N}_ξ from Figure 11.11 is characterized or defined by

$$\mathbb{N}_\xi = \left\{ \underbrace{\{w_{\alpha,\xi}\}}_{\text{Neuron weights}}, \underbrace{\alpha \in \mathbb{I}_\xi}_{\text{Neuron connections}} \right\}, \quad (11.36)$$

its weights and the list \mathbb{I}_ξ of neuron outputs, normally from the preceding layer, connected to the inputs of \mathbb{N}_ξ . We can therefore express the output of \mathbb{N}_ξ as

$$u_\xi = \psi(h_\xi) = \psi(w_{0,\xi} + \sum_{\alpha \in \mathbb{I}_\xi} w_{\alpha,\xi} u_\alpha) \quad (11.37)$$

where h is the linear discriminant which is passed through the nonlinear activation function ψ to produce the output u .

Recall that our goal is to learn the weights $w_{\alpha,\xi}$. From (11.34) we know that we need the gradient of \mathcal{L} , which we can express more simply via chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{\alpha,\xi}} = \frac{\partial \mathcal{L}}{\partial u_\xi} \cdot \frac{\partial u_\xi}{\partial h_\xi} \cdot \frac{\partial h_\xi}{\partial w_{\alpha,\xi}} \quad (11.38)$$

Although we will later see other choices of activation function, the derivation proceeds quite elegantly if the sigmoid activation is chosen,

$$\psi(h) = \frac{1}{1 + e^{-h}} \quad \rightarrow \quad \frac{\partial \psi(h)}{\partial h} = \psi(h)(1 - \psi(h)) \quad (11.39)$$

The three terms of (11.38) can then be further simplified as

$$\frac{\partial \mathcal{L}}{\partial w_{\alpha,\xi}} = \underbrace{(u_\xi - \bar{g}_\xi) \cdot u_\xi(1 - u_\xi)}_{\text{No dependence on } \alpha} \cdot u_\alpha \equiv u'_\xi u_\alpha \quad (11.40)$$

That is, u'_ξ is just a scalar measuring the sensitivity of the gradient of the loss \mathcal{L} to neuron output u_α . As a result, we can write the iterative weight update from (11.34) as

$$w_{\alpha,\xi}^{t+1} = w_{\alpha,\xi}^t - au'_\xi u_\alpha \quad (11.41)$$

Determining the gradients for the output layer seemed straightforward enough, since the weights in the output layer directly affect the network outputs, and so the dependence of each output on the various weights is quite simple. In contrast, in looking at Figure 11.12 it would seem that learning weights *internal* to the network would be *much* more difficult:

- A single internal weight influences *all* of the network outputs, not just one output.
- There are many paths connecting a given internal weight and a given network output, where the number of paths explodes combinatorially as the number of layers increases.
- The influence of a given weight passes through multiple nonlinearities (activation functions), one per layer, presumably making its influence on the network outputs very hard to ascertain.

With these concerns in mind, let us repeat the derivation of (11.38) through (11.41), but now for a weight $w_{\alpha,\xi}$ associated with \mathbb{N}_ξ , a neuron on an *internal* layer l .

In order to learn/update an internal weight we still need the gradient of \mathcal{L} , as before:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial w_{\alpha,\xi}}}_{\text{Hard}} = \frac{\partial \mathcal{L}}{\partial u_\xi} \cdot \frac{\partial u_\xi}{\partial h_\xi} \cdot \frac{\partial h_\xi}{\partial w_{\alpha,\xi}} = \underbrace{\frac{\partial \mathcal{L}}{\partial u_\xi}}_{\text{Hard}} \cdot u_\xi(1 - u_\xi) \cdot u_\alpha \quad (11.42)$$

The latter two terms are internal to neuron N_ξ , and so remain the same as in (11.40); it is the first term, which now spans one or more layers, which is more difficult. We can use chain rule, again, to express the first term in terms of the neurons at the *next* layer:

$$\frac{\partial \mathcal{L}}{\partial u_\xi} = \sum_{\beta \in \mathbb{L}_{l+1}} \frac{\partial \mathcal{L}}{\partial u_\beta} \cdot \frac{\partial u_\beta}{\partial h_\beta} \cdot \frac{\partial h_\beta}{\partial u_\xi} \quad (11.43)$$

where β indexes over the neurons on the next layer. Again, here the latter two terms are straightforward, and the first we do not yet know:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial u_\xi}}_{\partial \mathcal{L}/\partial u \text{ at layer } l} = \sum_{\beta \in \mathbb{L}_{l+1}} \underbrace{\frac{\partial \mathcal{L}}{\partial u_\beta}}_{\partial \mathcal{L}/\partial u \text{ at layer } l+1} \cdot u_\beta(1 - u_\beta) \cdot w_{\beta,\xi} = \sum_{\beta \in \mathbb{L}_{l+1}} u'_\beta w_{\beta,\xi} \quad (11.44)$$

That is, the unknown gradient of \mathcal{L} at layer l could recursively be written in terms of the gradient of \mathcal{L} at layer $l + 1$. However we already know $\partial \mathcal{L}/\partial u$ at the output layer, therefore we can propagate the gradient back, layer-by-layer, the essence of what is known as the *backpropagation algorithm*. The needed gradient in (11.42) therefore becomes

$$\underbrace{\frac{\partial \mathcal{L}}{\partial w_{\alpha,\xi}}}_{\begin{array}{c} \text{Again, no dependence on } \alpha \\ \text{ } \end{array}} = \sum_{\beta \in \mathbb{L}_{l+1}} u'_\beta w_{\beta,\xi} \cdot u_\xi(1 - u_\xi) \cdot u_\alpha \equiv u'_\xi u_\alpha \quad (11.45)$$

The iterative update for internal network weights is therefore completely unchanged from (11.41):

$$w_{\alpha,\xi}^{t+1} = w_{\alpha,\xi}^t - a u'_\xi u_\alpha \quad (11.46)$$

It is this recursive, layer-by-layer backpropagation approach, not subject to any combinatorial explosion, which is the key idea that allows for incredibly deep networks, with over 100 layers, to be possible.

The power of a neural network is that, for a sufficiently large number of layers, the network is a universal function approximator, meaning that essentially *any* input–output relationship $y \rightarrow g(y)$ can be learned, in theory. In practice, learning a given input–output relationship may be difficult for any number of reasons:

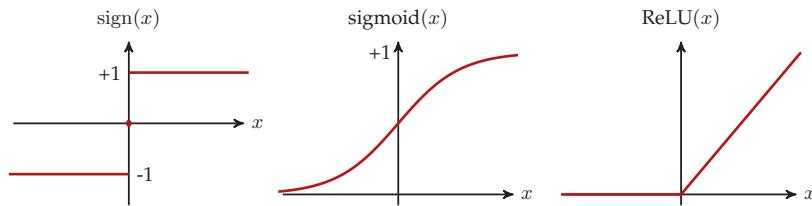


Fig. 11.13. Three very common nonlinear activation functions used in neural networks.

- Overfitting, due to too little training data or too large a network (every network weight represents a degree of freedom, so a large network requires a commensurately large amount of training data);
- Underfitting, due to too few training iterations;
- Poor training convergence because of poorly chosen learning parameters (such as a , in (11.46), however in practice there are also other parameters to choose);
- Poor network architecture (too few or too many neurons per layer).

11.4.2 Deep Neural Network Classifiers

The preceding discussion derived network learning, and how it is that deep (many layer) networks are realistically trainable. There are a few further steps and decisions which need to be made, in practice, in order for a network to act as a classifier.

Choice of Activation Function

The perceptron in Section 10.2 was based on the sign-function activation of Figure 10.6, in order to produce a binary classification decision at the neuron output. Similarly network learning in the preceding section was derived based on a sigmoid activation function. In practice, many networks are based on Rectified Linear Units (ReLU), as plotted in Figure 11.13:

- The ReLU function is continuous, whereas the sign function is not.
- ReLU is an exceptionally simple computation, *much* faster than a sigmoid.
- The ReLU naturally leads to network sparsity, in that its output will frequently be exactly zero, leading to neurons and weights which can be entirely removed.
- In practice, ReLU seems to lead to more effective network training than networks based on sigmoid functions.

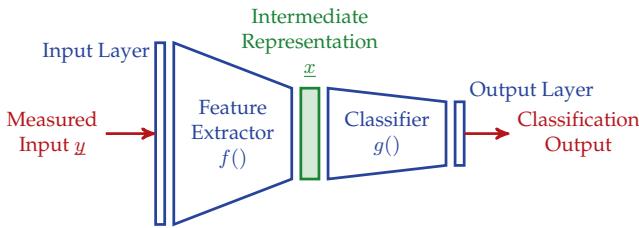


Fig. 11.14. A network does not literally need to be divided, as shown here, however frequently it is conceptually helpful to think of the initial network layers are primarily involved in feature extraction, and the latter layers as primarily involved in feature classification.

Network Feature Extraction

Most of the classifiers developed in this text were discussed with the assumption that feature exaction, as discussed in [Chapter 5](#), had already taken place. In contrast, in neural network based classification, because the network is understood to be a universal function approximator, there is no rationale in undertaking feature-extraction analysis in advance of the network, since any needed feature extraction (whether linear or nonlinear) can be inferred as part of network learning.

In most networks (also depending on details of the chosen network architecture) there is evidence that initial layers tend to fulfill more of a feature-extraction role, with latter layers performing more classification-related actions, as illustrated in [Figure 11.14](#). Although the network structure itself is not necessarily deliberately distinct between the feature-extraction and classification portions,⁷ the distinction may be helpful in planning network architecture, such as number of neurons as a function network layer.

One area where the feature–classification distinction *is* useful is in domain adaptation or transfer learning, as discussed in [Section 9.1](#). Suppose that our problem of interest has only a modest amount of data available, \mathcal{D} , but that there is a very large related dataset, $\bar{\mathcal{D}}$, having similar attributes, but differing in the details of how many classes, which classes etc.⁸ One could then propose the following strategy:

- Learn a whole network (features and classifier) from large dataset $\bar{\mathcal{D}}$.
- Freeze the feature-extraction part of the network, such that further learning can update only classifier weights.
- Learn the network classifier layers based on smaller dataset \mathcal{D} .

⁷ With the significant exception of convolutional neural networks (CNNs), which have convolutional (local filter) layers for feature extraction, followed by fully-connected layers (as in [Figure 11.12](#)) for classification.

⁸ For example, a large dataset $\bar{\mathcal{D}}$ of dog images might very well lead to *features* similar to those from a dataset \mathcal{D} of cat images, but clearly having very different *classes*.

Network Output Layer as Classifier

Following (11.31) there was some ambiguity regarding the network output $g(y)$, whether the output was a feature, a discriminant, or an actual discrete class. It is now time to resolve this ambiguity and to decide more clearly what it is that the network produces at its output layer.

Network Output as Feature: With reference to Figure 11.14, we could imagine that the entire network is *only* the feature-extraction part, such that the classifier is not present, and that the output layer is a feature vector \underline{x} . One could then apply any of the classifiers in this text — NN, k NN, SVM etc. — to that output layer to perform the actual classification.

There are two circumstances where there might be rationale to having a network output a feature vector:

1. In those cases where the network training is capable of learning excellent feature extraction, but where the classifier is learned poorly, perhaps because the classes have highly unusual shapes and distortions that k NN might accommodate, but that may be difficult to successfully train in a network.
2. In those cases where we wish to undertake transfer learning, as we have just discussed, we could train the whole network (feature extraction and classification) on large dataset $\bar{\mathcal{D}}$. We would then preserve (transfer) only the feature-extraction portion of the network, giving us the nonlinear features $f(y)$, and subsequently learn a classifier (NN, k NN, SVM, etc.) based on our actual (typically smaller) dataset \mathcal{D} .

Network Output as Classifier: Rather than output a feature vector, more commonly we wish to train a network which accomplishes the entire task, all the way from measurements to classification. Although we understand that a network will be *inclined* to separate into feature-extraction and classification portions, as in Figure 11.14, a sharp distinction will typically not take place, and will not matter in any event, since we are interested in training the network end-to-end, as a whole.

How the output layer is interpreted is *necessarily* and directly related to the loss function by which the network was trained. That is, whatever interpretation you wish to place on output $g(y)$ must be consistent with the behaviour encoded by loss function \mathcal{L} .

For example, the perceptron of Figure 10.5 had a thresholding activation function, such that its output $g(y) \in \{-1, +1\}$. In principle we could generalize this to the K -class problem by expecting a single network output $g(y)$ whose ideal target value is the class index:

$$\text{If } \underline{y} \text{ comes from class } C_\kappa \rightarrow \bar{g}(\underline{y}) = \kappa \quad (11.47)$$

so that given a dataset $\mathcal{D} = \{\underline{y}_i, c_i\}$ we could assert a loss function

$$\mathcal{L}(\text{Network}) = \sum_i \left(g(\underline{y}_i) - c_i \right)^2 \quad (11.48)$$

with the corresponding actual classification based on the network output as

$$\text{Classify as } \hat{C} = \begin{cases} C_1 & g(\underline{y}) < 1.5 \\ C_2 & 1.5 < g(\underline{y}) < 2.5 \\ \vdots & \vdots \\ C_{K-1} & K - 1.5 < g(\underline{y}) < K - 0.5 \\ C_K & K - 0.5 < g(\underline{y}) \end{cases} \quad (11.49)$$

In practice, having a network learn to tune a single output to have a near-integer value based on the class label does not work so well. Instead, a K -class problem (for $K > 2$) is almost always approached by having K outputs $g_1(), \dots, g_K()$, such that

$$\text{If } \underline{y} \text{ comes from class } C_\kappa \rightarrow \bar{g}_j(\underline{y}) = \begin{cases} 1 & j = \kappa \\ 0 & j \neq \kappa \end{cases} \quad (11.50)$$

That is, our network is taught to produce K indicator functions, one output for each class, and typically we would then take the largest output as showing the greatest support for that class:

$$\text{Classify } \underline{y} \text{ as } \hat{C} = \arg_\kappa \max g_\kappa(\underline{y}) \quad (11.51)$$

Since the network gives us one output per class, frequently we may wish to interpret the size of the output as a *score* (as in [Section 11.1](#)) or probability of the class being correct. For this purpose a *SoftMax* normalization is widely used,

$$g_\kappa^{\text{SoftMax}} = \frac{e^{\gamma g_\kappa(\underline{y})}}{\sum_{j=1}^K e^{\gamma g_j(\underline{y})}} \rightarrow 0 \leq g_\kappa^{\text{SoftMax}} \leq 1 \quad \sum_\kappa g_\kappa^{\text{SoftMax}} = 1 \quad (11.52)$$

which produces a vector g^{SoftMax} whose values sum to one and where each value lies between zero and one, regardless whether the elements in g were positive, negative, large, or small.

Types of Loss Functions

For every input training vector \underline{y} , we have the associated network output values $\{g_j(\underline{y})\}$ and the corresponding target (ideal) values $\{\bar{g}_j\}$. So far in this

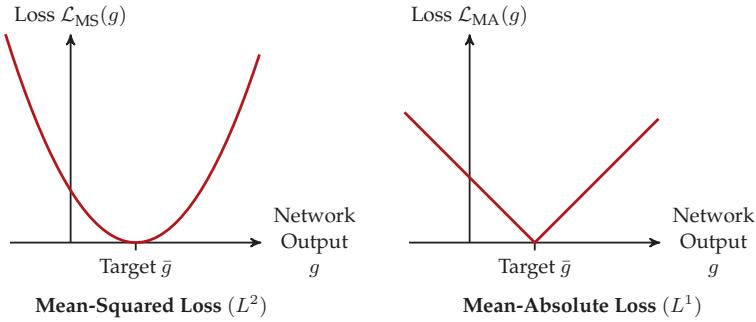


Fig. 11.15. LOSS FUNCTIONS describe how deviations from an ideal target value \bar{g} are penalized, whether based on a quadratic (left) or absolute difference (right).

text, in discussing an optimization objective or network loss we have mostly focused on the mean-squared⁹ loss

$$\mathcal{L}_{MS}(\{g_j(y)\}) = \sum_j (g_j(y) - \bar{g}_j)^2, \quad (11.53)$$

illustrated in [Figure 11.15](#). The squared-difference penalizes outliers (far from the target) very heavily, which can lead to an undesirable sensitivity to such outliers. As part of the discussion of robust methods in [Example 3.1](#), we saw the mean-absolute loss

$$\mathcal{L}_{MA}(\{g_j(y)\}) = \sum_j |g_j(y) - \bar{g}_j|, \quad (11.54)$$

which is widely used in network training, precisely because it is much less sensitive to outliers.

In the case of classification, our primary interest here, we are given a dataset of measurement-class pairs $\mathcal{D} = \{\underline{y}_i, c_i\}$. However note that the dataset does *not* specify the target value \bar{g}_i for the network; the dataset specifies only the correct class label c_i , whereas the network target value depends on *how* you would like the network to undertake classification.

For example, in the case of binary classification ($K = 2$), we could have a single network output, with target value

$$\bar{g}_i = \begin{cases} +1 & c_i = 1 \\ -1 & c_i = 2 \end{cases} \quad (11.55)$$

⁹ In both (11.53) and (11.54) we are just computing a *sum* over j , and omitting the division by a constant (the number of network outputs) to compute a *mean*. The division by a constant just rescales the loss function, but does not actually change the target value \bar{g} , and frees the equations from being cluttered with additional notation about how many outputs a network possesses.

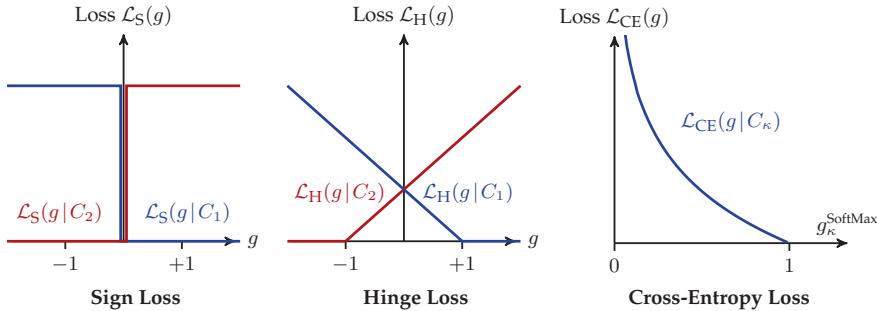


Fig. 11.16. Building on the LOSS FUNCTIONS of Figure 11.15, here three widely-used loss functions for classification. Note that the Sign and Hinge loss, as shown here, apply to binary ($K = 2$) problems only.

such that we could undertake training based on the *Sign loss* of Figure 11.16, essentially what was done with the perceptron:

$$\mathcal{L}_S(g(y)) = \text{sign}(-\bar{g} \cdot g(y)) \quad (11.56)$$

That is, our goal is to have \bar{g} and $g(y)$ both be positive or both be negative, meaning that their product should always be positive. Since network learning seeks to *minimize* the loss function, the loss function in (11.56) is therefore set to equal the *negative* of the product.

Although intuitive, the sign loss of (11.56) is actually quite problematic. The function is discontinuous (not differentiable) and does not assert any margin¹⁰ — the network output could be arbitrarily close to zero and be considered equally acceptable to an output significantly further from zero. An alternative, a continuous loss function which asserts a margin, to actively push the network output *away* from zero, is the simple *Hinge loss*:

$$\mathcal{L}_H(g(y)) = \max(0, 1 - \bar{g} \cdot g(y)) \quad (11.57)$$

So any network output $\bar{g} \cdot g(y) > 1$ is considered equally good, with increasing loss (penalty) as the network output moves toward zero or (even worse) to the wrong classification.

In network-based multi-class classification ($K > 2$) problems a K -output Soft-Max activation, as in (11.52), is almost universally used, with the corresponding target value as in (11.50). It would be possible to develop a generalized hinge loss,¹¹ however far more widely used are loss functions based on tests of distribution, here the discrete-state analogue of the continuous-state methods developed in Section 7.4.

¹⁰ Recall the discussion of margin in the Support Vector Machine discussion in Section 10.2.

¹¹ See the development in Problem 11.5.

That is, given a SoftMax layer as described in (11.50) and (11.52), the network targets and outputs can both be interpreted as describing the probability distribution for K -valued discrete random variables:

$$0 \leq \bar{g}_\kappa \leq 1 \quad \sum_\kappa \bar{g}_\kappa = 1 \quad 0 \leq g_\kappa^{\text{SoftMax}} \leq 1 \quad \sum_\kappa g_\kappa^{\text{SoftMax}} = 1 \quad (11.58)$$

We would like a loss function to measure the similarity of these two distributions. Particularly widely used is the cross-entropy loss, building on the discussion of information theory from page 108 in Chapter 5:

$$\mathcal{L}_{\text{E}}(g(y)) = - \sum_\kappa \bar{g}_\kappa \log_2 g_\kappa^{\text{SoftMax}} \quad (11.59)$$

Although $\{\bar{g}_\kappa\}$ could be non-binary,¹² if we stick to the binary indicator values of (11.50) then only one of the terms in (11.59) is non-zero, and therefore simplifies to

$$\mathcal{L}_{\text{E}}(g(y) | C_\kappa) = - \log_2 g_\kappa^{\text{SoftMax}} \quad (11.60)$$

An alternative, highly-related loss function would be the Kullback-Leibler or KL loss.

Overview of Network Architectures

A great many network architectures have been proposed, and this discussion can only serve as a very basic starting point. Certainly a large number of dedicated textbooks and online tutorials have been developed which can offer much more depth.

Starting with small, fully-connected networks is highly recommended, partly because the programming is simple, but largely because they train very quickly and have none of the GPU¹³ requirements commonly associated with network learning, particularly for very large image processing and computer-vision problems.

Some of the best-known large architectures for image analysis are AlexNet (8 layers), VGGNet (16-19 layers), and ResNet (more than 100 layers), and for each of these architectures implementation code and pre-trained networks (already fully learned) are widely available. The layer count alone is a limited assessment of a network's ability to learn, which is largely determined by the

¹² For example, if the ground-truth labeling for a given input y were to indicate three classes as being equally likely, and not just a single class.

¹³ Graphical Processing Units, which were originally designed for accelerated computer graphics. GPUs are able to perform simple mathematical operations exceptionally fast and in parallel, attributes which proved to be exceptionally well suited for deep network training.

total degrees of freedom (learnable parameters) in the network, which then also dictate the required amount of training data. Indeed, as networks have grown the training data needed for reliable learning has grown as well, and the ability to store and manage such datasets can become a limiting factor in trainable network size.

Large networks are typically designed to process images, thus the number of inputs is very large (equal to the number of pixels in the input image). As a result, the first several layers (roughly corresponding to the feature extractor in Figure 11.14) are convolutional. A convolution essentially implements a local filter, applied in parallel to the whole image, with the network layer characterized by the filter size and the number of filters.

Because the input layer is typically very large, and any intermediate feature representation \underline{x} or output layer g needs to be *far* smaller, some progressive reduction in size, from layer to layer, is required. Such a reduction is accomplished in one of two ways:

1. A convolutional filter *stride*, an integer measure of the pixel spacing between successive or adjacent filter applications, which leads to image downsampling by the stride factor.
2. A *pooling* layer, which groups blocks of pixels by taking a mean or, more commonly, the maximum over the block, downsampling the image by the block size.

At this point, the reader has been exposed to all of the fundamental network components, such that a network architecture like VGG16, consisting of twelve convolutional layers, five pooling layers, three fully-connected layers, all neurons based on a ReLU activation function, and a final SoftMax output layer, are all concepts which we have covered. An understanding of relatively complex architectures and the ability to undertake creative combining or redesign of such networks are therefore within the reader's grasp.

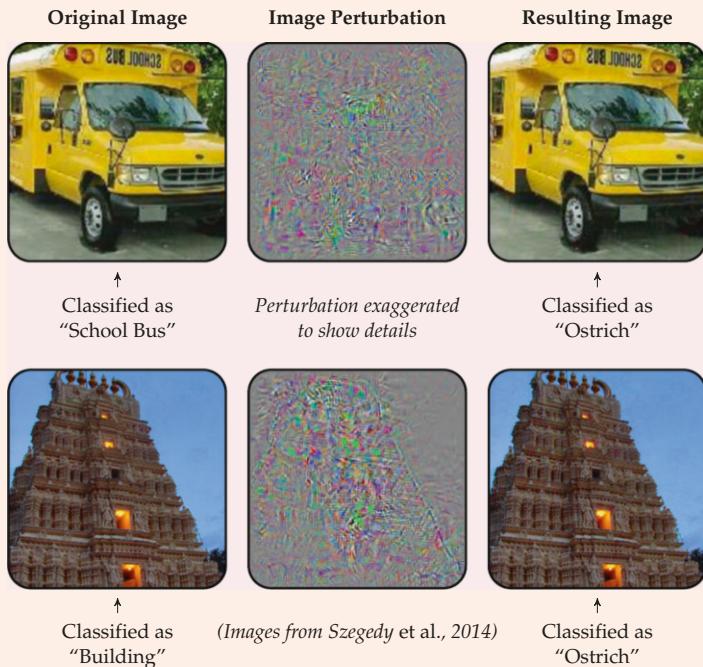
Case Study 11: Interpretability and Ethics of Large Networks

A single linear discriminant (Chapter 10) in a given feature space¹⁴ is trivially interpretable — it has only few degrees of freedom (the normal vector and offset), it is unambiguous what it is doing (dividing the space into two using a hyperplane); nothing is hidden.

¹⁴ "In a given feature space", as opposed to in a near-infinite-dimensional nonlinearly transformed space of a support vector machine, which is also based on a single linear discriminant, but which may be exceptionally difficult to interpret.

Example 11.3: Illusions in Machine Learning

As we saw in [Example 2.2](#), the human visual system is subject to certain optical illusions. We know that almost all pattern classifiers are subject to error, but sometimes the errors can be quite mystifying. Consider an object-recognition neural network, taking as input an image, and producing at its output layer a classification:



Really, *Ostrich*?! To the human eye, the original and resulting images are indistinguishable, and there is really no ambiguity whatsoever what sort of object is being shown. However in the same way that our visual system can be fooled, the tens of millions of nonlinear discriminants in a network can have their calculations manipulated, ever so slightly, such that the resulting classification is terribly wrong.

The result is amusing, until you imagine such a system in an autonomous vehicle, attempting to recognize whether it is a school bus or an ostrich ahead on the highway, and suddenly the issue becomes far more serious, underlining the importance of thorough validation and testing ([Chapter 9](#)). Far greater data augmentation, as in [Example 9.1](#), is one way of improving such network learning.

Further Reading: There are a great many papers on fooling neural networks and, in response, also a great many on more robust methods of training:

"Intriguing properties of neural networks," C. Szegedy *et al.*, ICLR, 2014

"Accessorize to a Crime . . .," M. Sharif *et al.*, Computer & Comm. Security, 2016

"Universal Adversarial Perturbations," S. Moosavi-Dezfooli *et al.*, CVPR, 2017

A nearest-neighbour classifier ([Chapter 6](#)) is very different from a discriminant. We *know* that it has memorized/overfit all of the training data (whether that is effective or desirable or not is a separate matter). However the resulting operation is still relatively simple and interpretable, since it is the immediate reflection of the memorized data.

In contrast, deep neural networks with upwards of 100 million parameters, with thousands to millions of nonlinear discriminants, implicitly implement a mathematical equation that would be nonsensical to write out in full, and completely uninterpretable in any event. Even if the number of discriminants were quite modest (say, roughly 20), the nonlinear activation functions between the discriminants still make the resulting calculation, from input to output, essentially impossible to understand.

Does this matter?

At first glance, not really. Our primary objective in proposing a deep neural network with thousands to millions of neurons is to obtain a classifier with strong performance on difficult, high-level tasks such as object recognition in autonomous driving,¹⁵ and indeed the resulting performance has been spectacular.

Or has it? What does it actually mean, to “*validate*” the performance of a classifier having 100-million degrees of freedom?, as was already examined in [Example 11.3](#). What does it mean to claim a certain probability of error, when the dimensionality of the input space is so high, and the classifier so complex, that it is simply not possible to have enough test data to validate such a claim?

[Case Study 9](#) has made it clear why validation is important, but why does a classifier need to be interpretable? As Artificial Intelligence/Deep Networks become increasingly ubiquitous, it is easy to imagine that they will be used to reach conclusions that impact people’s lives. If a doctor reaches a medical conclusion about a patient, in principle the doctor can later be asked on what basis the conclusion was reached. However if a large neural network makes a decision about which person receives a liver transplant and which person does not, how is that decision later explained and defended: by the person who programmed the network learning algorithm, the person who organized the dataset, or the doctor who submitted the query to the classifier?

If a network is effectively a vast set of nonlinear calculations, and thus in no way explicitly interpretable, then all assessments need to be made indirectly, falling into two categories:

¹⁵ As discussed in [Case Study 9](#).

BLACK BOX TESTS, in which the internal structure of the network is unknown and inaccessible, meaning that all tests need to be made by probing the network with many inputs and observing the outputs.

WHITE BOX TESTS, in which the networks internals (neuron inputs and outputs at all layers) are available to be observed in response to an input. Such network access would be required in order to deduce the classification sensitivity revealed in [Example 11.3](#).

Any network-based system with societal impact of any kind would presumably (or one should insist upon, ethically) be open to white-box scrutiny. However although one can design tests (as in [Example 11.3](#)) to detect particular unusual circumstances of classifier failure, such tests still say almost nothing about unintended consequences, subtle biases in the training, such that liver transplants might preferentially be given to wealthier patients, for example.

Furthermore there can be subtle issues of data privacy. Given confidential training data (say, of people's names or medical histories) we are often concerned in working with or reporting on very small data sets, because it might be possible for someone to reverse-engineer and deduce the identity of an anonymized sample. Typically it might be assumed that a classifier or other statistical inference method could be trained on the basis of private data, but that no meaningful residual of individual data points would remain in the resulting classifier, such that the classifier could then be freely distributed and used. However in a large network with 100 million parameters, it could be possible for a *great* deal of "memorized" information to be stored, in a way that would be completely hidden and implicit, and yet which might possibly be similarly reverse-engineered to give access to confidential data.

Finally, all of the above examples are pertinent even in the absence of any malicious or nefarious intent. In [Example 11.3](#) we saw examples of networks which had not been tampered with at all, and had been understood (and validated!) to have excellent recognition performance, but where the susceptibility to image perturbations were found only later. However it is easy to imagine deliberate biases being subtly encoded into a large network. A computer virus can be found because a given piece of computer code is based on a series of bytes which, once known, is easily searched. However there is no particular signature in the millions of network weight parameters that would reveal a tampered network, making effective validation much more difficult, but all that much more important. Obviously a badly tampered network which leads to poor classification can be very easily validated based on test data, as discussed in [Chapter 9](#), and one can test for even subtle biases, but only if one knows precisely *which* bias (wealthier patients and liver transplants) it is for which one is searching.

Lab 11: Ensemble Classifiers

In this lab we would like to explore decision trees, which we saw (Section 11.3 and particularly in Figure 11.7) to be a fairly promising approach in the combining of multiple classifiers.

In particular, let us focus on a relatively simple example:

- There are only two classes: $K = 2$. Decision trees have no problem whatsoever dealing with larger numbers of classes, however it simplifies our code and book-keeping to have only two classes.
- The decision tree approach does not specify *how* to identify candidate discriminants, and in principle this could be as easy or as hard as we choose to make it. To keep the process conceptually simple, we will find discriminants by selecting a points, at random, from each class, and then find the discriminant as the hyperplane half way between the points.

All other labs have listed the code, explicitly, because it can be insightful to see the steps involved in a method, however in this lab some of the details begin to obscure the overall idea, and in any event the code for all of the labs, including this one, is downloadable from the [textbook web site](#). So our focus in this lab will be a relatively detailed pseudo-code illustration.

The whole decision tree is made up of decision objects,

$$\text{Decision} = (\text{Discriminant}, \text{Decision if Positive}, \text{Decision if Negative}) \quad (11.61)$$

consisting of a discriminant, and the next steps depending on whether the point in question lies on the positive or negative side of the discriminant; we can write the object of (11.61) more compactly as

$$D = (h(), D_+, D_-). \quad (11.62)$$

D_+ and D_- may themselves be further discriminants with decisions, or just a single class if they represent a leaf node in the decision tree.

The learning of the decision tree is based on a recursive function,

```
function  $D = \text{LearnDecisionTree}( \text{ClassC}_1 \text{ Dataset}, \text{ClassC}_2 \text{ Dataset},$ 
 $\text{Remaining Depth} )$ 
```

which, given data points for the two classes and how many layers of recursion remain available on the tree, needs to determine whether

it is a leaf node (at the end of the tree), or to find an appropriate discriminant:

```
function  $D = \text{LearnDecisionTree}(\mathcal{D}_1, \mathcal{D}_2, Depth)$ 
    % If points are from only one class, then we have a leaf node and
    % nothing more to do
    if  $|\mathcal{D}_1| == 0$ ,  $D = C_2$ , return, end
    if  $|\mathcal{D}_2| == 0$ ,  $D = C_1$ , return, end

    % If Depth is zero then we are at the maximum depth
    % No more discriminants are permitted, so just decide larger class
    if ( $Depth == 0$ ) then
         $D = C_2$ 
        if  $|\mathcal{D}_1| > |\mathcal{D}_2|$ ,  $D = C_1$ , end
        return
    end

    % We would like to find a discriminant
    Repeat 100 times:
        Select a point from each class at random
        Find the discriminant as the right-bisector between these points
        Evaluate the entropy loss of the discriminant
        From the 100 trials, keep the best discriminant  $g_{\text{Best}}$ 

    % Recursively call ourselves for both sides of the discriminant
     $D_+ = \text{LearnDecisionTree}(\{\underline{x} \in \mathcal{D}_1 | h_{\text{Best}}(\underline{x}) > 0\}, \{\underline{x} \in \mathcal{D}_2 | h_{\text{Best}}(\underline{x}) > 0\},$ 
    % $Depth - 1$ )
     $D_- = \text{LearnDecisionTree}(\{\underline{x} \in \mathcal{D}_1 | h_{\text{Best}}(\underline{x}) < 0\}, \{\underline{x} \in \mathcal{D}_2 | h_{\text{Best}}(\underline{x}) < 0\},$ 
    % $Depth - 1$ )
     $D = (h_{\text{Best}}, D_+, D_-)$ 
    return
end
```

We then require a companion function, which performs the actual classification based on the learned tree:

```
function  $Class = \text{DecisionTreeClassify}(D, \underline{x})$ 
    % If we are at a leaf node, then just return that class
    if  $\text{numel}(D) == 1$ ,  $Class = D$ , return, end

    % We are not at a leaf, so D contains a discriminant D.h plus decisions
    % D.+ and D.-
    % Evaluate the discriminant at the given point
    % Then return whatever final decision was decided lower down on the
    % decision tree
    if  $D.h(\underline{x}) > 0$ ,
         $Class = \text{DecisionTreeClassify}(D_+, \underline{x})$ 
```

```

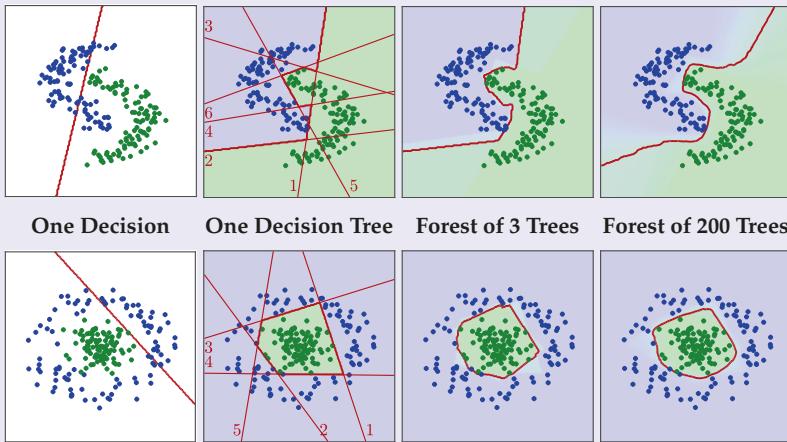
else
    Class = DecisionTreeClassify( D.-, x ) )
end
return
end

```

The above approach to generating a decision tree can trivially be generalized to a random forest, as described on [page 317](#), by generating a sequence of trees and voting. In this case the dataset was not randomly resampled between trees, because the random discriminants introduced sufficient variation from tree to tree.

The decision tree/random forest approach is illustrated for two problems, both non-separable, which would have presented significant challenges to regular discriminant methods, although clearly a nonlinear support vector machine would also be able to successfully learn classifiers.

The rightmost random forest made up of 200 decision trees is clearly something of an exaggeration, since nowhere near that many trees are actually needed to form a classifier. Perhaps five to ten trees, representing a total of approximately 40 binary decisions, would be needed.



Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links: [Ensemble Learning](#), [Bagging](#), [Boosting](#), [AdaBoost](#), [Decision Trees](#), [Random Forest](#)

Wikipedia Links: [Neural Network](#), [Perceptron](#), [Biological Neuron Model](#), [Artificial Neural Network](#), [Deep Learning](#), [Convolutional Neural Network](#)

The whole area of weak classifiers, resampling theory, and classifier ensembles are well developed in many more advanced pattern recognition texts, and the reader is referred to any of [1, 4, 5, 11–13].

There is a nearly endless set of neural network/deep learning related texts, and the field evolves so rapidly that it is not realistic to present a definitive list here. The fundamental theory and mathematics do not so quickly go out of date, so for such background the reader is referred to [2, 7, 9]. There are many texts which are more hands-on, such as [3, 6, 8, 10], however the reader is strongly encouraged to locate up-to-date web resources or online courses.

Sample Problems

Problem 11.1: Short Answer

Give a short definition of each of the following:

- (a) Linear Discriminant
- (b) Weak Learner
- (c) Bootstrap Resampling
- (d) Entropy
- (e) Random Forest
- (f) Activation Function
- (g) Soft-Max Function
- (h) Hinge Loss

Problem 11.2: Short Answer

Offer brief answers to each of the following:

- (a) What is the importance of the idea of having multiple layers of Perceptrons, rather than just a single layer?
- (b) We know that MAP is the unique classifier guaranteed to minimize the probability of classification error. If MAP were easy to apply, then presumably we wouldn't be very interested in discriminants. What makes the use of MAP more difficult than ensembles of discriminants?

- (c) Similar to part (b), we also know that the k NN nonparametric classifier approaches optimality (in terms of minimizing error) as k increases. If k NN were easy to apply, then presumably we wouldn't be very interested in discriminants. What is the attractiveness of ensembles of discriminants over k NN?
- (d) Briefly, what are some strengths and weaknesses of Boosting and Bagging?
- (e) Explain clearly why decision trees require a different objective function for discriminant learning, than discriminants learned for other contexts.
- (f) What are the main strengths and weaknesses of deep neural networks relative to other approaches to classification?

Problem 11.3: Conceptual — Discriminants

Recall the exclusive-or (XOR) problem of [Figure 11.1](#). The problem requires three discriminants to solve: two discriminants as a function of the inputs, and then a third discriminant to combine the first two.

Solve for the weight values for all three discriminants, and show the correctness of your overall classifier by simulating it for all four input possibilities (two values for each of two inputs).

Problem 11.4: Conceptual — Ensemble Classifiers

For a complicated labelled-data classification problem, we more or less have three approaches to consider:

- Nonparametric methods (NN, k NN)
- Problem Transformation (SVM)
- Semi-Parametric or Ensemble methods (Decision Trees, Random Forests, Neural Networks)

Give a number of pros and cons comparing these three categories of approaches.

Problem 11.5: Conceptual — Network Loss Functions

The Hinge Loss function was defined for the $K = 2$ class case in [\(11.57\)](#) and illustrated in [Figure 11.16](#). As was mentioned on [page 330](#), in this question we would like to develop a generalization of the Hinge Loss to $K > 2$.

For $K = 2$, the classification problem is comparatively straightforward, since there is only *one* network output for *two* classes, and that one output can be pushed positive or negative to represent the two classes. The hinge

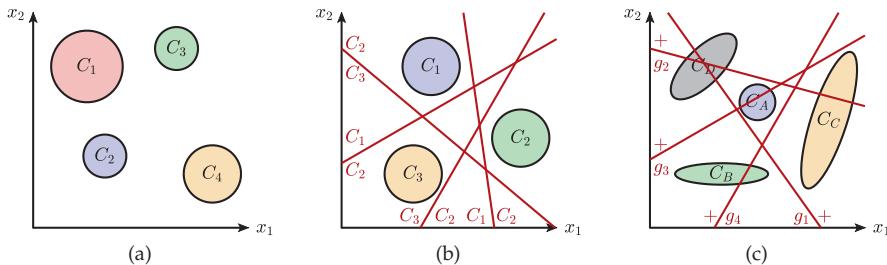


Fig. 11.17. Class and discriminant definitions for [Problem 11.6](#). In (c), the “+” symbol marks the positive side of the discriminant.

loss of (11.57) was designed to push the network output away from zero, to create a margin between the two classes, for more robust classification.

In contrast, for $K > 2$ we normally do not use a single network output, rather we have K outputs, thus one *per class*. We will assume that a SoftMax activation of (11.52) has been applied, so we would like a loss function, like a hinge, to be applied to

$$g_1^{\text{SoftMax}}, \dots, g_K^{\text{SoftMax}} \quad (11.63)$$

Design a hinge loss, similar to that from (11.57), but now asserting a margin with regards to the K network outputs.

Problem 11.6: Conceptual — Combined Discriminants

- (a) For the four classes shown in [Figure 11.17a](#), sketch the classifier resulting from a majority vote applied to the set of all pairwise discriminants.
- (b) Given the four classes shown in [Figure 11.17a](#), we wish to find a classifier combining linear discriminants, but **using as few discriminants as possible**. For each of the following, sketch the resulting discriminants and give an appropriate combining rule:
 - (i) For some sort of scored/weighted/majority (non-sequential) approach
 - (ii) For a decision-tree (sequential) approach
- (c) Given the three classes and four discriminants shown in [Figure 11.17b](#),
 - (i) Suppose we use a majority-vote scheme. Plot the classification boundaries.
 - (ii) Identify the ambiguous regions (where there is no single majority).
 - (iii) How might you change the classifier (discriminants and/or combining rule) to remove all ambiguity?

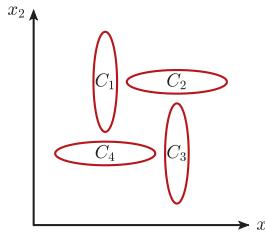


Fig. 11.18. A set of four classes for Problem 11.7.

- (d) Four classes and four discriminants g_1, \dots, g_4 are shown in Figure 11.17c, with the positive side of each discriminant indicated by a “+” sign. Design a combining rule which successfully classifies the four classes.

Problem 11.7: Conceptual — Combined Discriminants

For the four classes of Figure 11.18:

- (a) Develop a majority-voting classifier. Sketch some number of linear discriminants such that a majority vote correctly classifies the classes.
- (b) Develop a decision-tree classifier.

In both cases, sketch the linear discriminants, the combining rule, the resulting classifier, and the classification boundaries.

Problem 11.8: Numeric/Computational — Bagging

We would like to experiment with bootstrap resampling; that is, the process of sampling multiple datasets from one original.

Suppose we have two Gaussian classes in a simple two-dimensional, symmetric arrangement:

$$\underline{x} | C_1 \sim \mathcal{N}(-\alpha, I) \quad \underline{x} | C_2 \sim \mathcal{N}(\alpha, I) \quad (11.64)$$

Assume we have datasets $\mathcal{D}_1, \mathcal{D}_2$, each of N points, from C_1 and C_2 , respectively. Clearly if N were very large, then we would have plenty of training and testing data, and we could use kNN or MAP to learn a near-optimal classifier. Our interest, therefore, is in *small* N .

We will try to keep things fairly simple in this question:

- (a) From each of $\mathcal{D}_1, \mathcal{D}_2$ synthesize S datasets $\mathcal{D}_{1,1}, \dots, \mathcal{D}_{1,S}, \mathcal{D}_{2,1}, \dots, \mathcal{D}_{2,S}$, each consisting of N points. From each pair $\mathcal{D}_{1,s}, \mathcal{D}_{2,s}$, learn a discriminant-based classifier $g_s(\underline{x})$. The S classifiers are then combined via majority vote.

- (b) From part (a) we have a classifier, but no way to validate it, since all of the N points were used in the learning process. So to learn a probability of error, wrap a jackknife around the process of (a), learning the S datasets from $N - 1$ points, testing the combined classifier on the remaining point, and then repeating this process N times.
- (c) From part (b) we have a probability of error $\mathbf{P}(e)$, but no sense of how accurate that error is. That is, with a new dataset, would $\mathbf{P}(e)$ be similar or possibly quite different from before? To answer this question we need an estimate of the variance of the probability of error. To estimate the variance of the probability of error, we can wrap a bootstrap (Figure 11.2, Example 11.1) by sampling \bar{S} datasets, with replacement, and then applying the method of part (b) to each of the \bar{S} datasets.
- (d) From part (c) we then have \bar{S} assessments of the probability of error $\mathbf{P}(e)$, and from those \bar{S} values (Example 11.1) we can find the variability.

Implement, test, and report on the above process. Use modest values for N (say 2, 3, 5, 10), S (2, 3, 4, 5), and \bar{S} (10, 20, 40).

Problem 11.9: Numeric/Computational — Weak Learner

The concept of a weak learner was central to the ability to combine a number of classifiers and obtain a superior result.

Suppose we have q weak classifiers which have a probability \mathbf{P}_W of classifying correctly between C_0 and C_1 . For the purpose of this question, we will suppose that the correct answer is C_1 . For each value of q and \mathbf{P}_W :

- For i from 1 to 100, undertake the following:
 - Simulate each of the classifier outputs c_1, \dots, c_q by setting each to 1 with a probability of \mathbf{P}_W , otherwise to 0.
 - Set v_i as the majority vote of c_1, \dots, c_q , flipping a coin if there is a tie.
- The probability that the voting classifier is correct is $\mathbf{P}_V = \sum_i v_i / 100$
- Plot \mathbf{P}_V as a function of q and \mathbf{P}_W .

Compare your results to the derivation in Section 11.1. Discuss your observations.

Problem 11.10: Real World, Open-Ended — Deep Neural Networks

As is very well known, by now, deep neural networks have dramatically upended the fields of pattern recognition and machine learning. Certainly deep neural networks have met or exceeded the state of the art in performance for almost any classic pattern recognition problem; these results are well documented, but are not of specific interest in this question.

Instead, this question asks you to explore one of the many unexpected or highly creative uses of neural networks, which very roughly fall into two categories:

- (a) *Applications to Previously Unconsidered Problems*: The network architecture might still be fairly traditional, like those outlined in [Section 11.4.2](#), however the breadth or generalizability of such networks opened up target problems that had previously not even been considered or attempted. Give a short survey of a few new problem domains, and what development (GPUs, datasets, ...) made the application possible.
- (b) *Creative Architectures*: Although the early deep neural network results on image classification and object recognition were based on classic convolutional and fully-connected network layers, since then there have been a great many highly innovative developments in terms of network architectures. Look up one of these architectural variations and give an overview of the problem domain which it addresses. Possible architectural terms to search, to get you started: Recurrent Neural Networks, Long Short-Term Memory Networks, Variational Auto-Encoders, Generative Adversarial Networks, Restricted Boltzmann Machines, or Twin Networks.

References

1. C. Bishop, *Pattern Recognition and Machine Learning* (Springer, New York, 2011)
2. C. Bishop, G. Hinton, *Neural Networks For Pattern Recognition* (Oxford University Press, Oxford, 2005)
3. F. Chollet, *Deep Learning with Python* (Manning, Shelter Island, 2021)
4. G. Dougherty, *Pattern Recognition and Classification* (Springer, New York, 2013)
5. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley Interscience, New York, 2009)
6. A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (O'Reilly Media, Sebastopol, 2019)
7. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
8. J. Howard, S. Gugger, *Deep Learning for Coders with fastai and PyTorch* (O'Reilly Media, Sebastopol, 2020)

9. J. Kelleher, *Deep Learning* (MIT Press, Cambridge, 2019)
10. J. Krohn, G. Beyleveld, A. Bassens, *Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence* (Addison-Wesley, Boston, 2019)
11. B. Ripley, *Pattern Recognition and Neural Networks* (Cambridge University Press, Cambridge, 2008)
12. S. Rogers, M. Girolami, *A First Course in Machine Learning* (Chapman and Hall, London, 2016)
13. A. Webb, K. Copsey, *Statistical Pattern Recognition* (Wiley, New York, 2011)



Model-Free Classification

Every problem we have seen so far has assumed some amount of labelled data; that is, pairs of values

$$\{y_i, c_i\} \quad \text{Measurement } y_i \text{ is a member of class } c_i \quad (12.1)$$

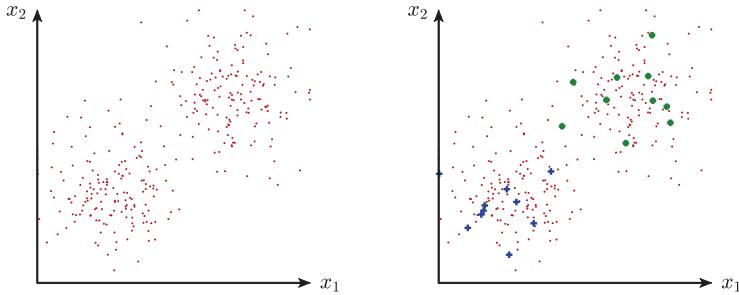
It may seem self-evident that such labelled data are needed; how, otherwise, could it be possible to know anything about the number or type or shape of classes? And yet humans routinely work in unlabelled domains: given a set of trees, animals, or wildflowers, the human mind instinctively begins to develop a taxonomy of what is similar and what is not, seeing the patterns present in an otherwise cluttered sea of data points.

Furthermore it is very easy to motivate the *need* for unsupervised learning: data labelling costs time and money, far more than the time or money to acquire the measurements themselves. That is, there are nearly endless data coming from satellites, weather stations, or billions of cell phones and other smart sensors, but the overwhelming majority of these data are not labelled:

- We do not know what crop is growing at a given pixel on a satellite image;
- We do not know the name of the person who just walked by a security camera;
- We do not know whether the part on the conveyor belt is defective or not.

Given a dataset of unlabelled data, really we have two categories of problems (Figure 12.1):

- **We want to label the data:** We *do* have an idea of what sorts of classes (crop types, people's identities, defects) we are expecting. In this case we need at least some modest amount of information about each class



The Un-Supervised Clustering Problem

The Semi-Supervised Clustering Problem

Fig. 12.1. CLUSTERING: The two fundamental problems involving unlabelled data. *All* of the data may be unknown (left), in which case it is only the inherent pattern of the data distribution to guide us. *Some* of the data may be labelled (right), in which case the labelled data can serve as a guide to interpreting the remaining unlabelled points.

(a prototype or a few labelled data), and we would like to refine/improve our classifier learning based on a much larger set of unlabelled data.

→ This is the semi-supervised learning problem of [Section 12.3](#).

- **We want to explore the clumping present in the data:** We do *not* have an idea of what sorts of classes we are expecting, or even most basic question of how *many* classes there are. We leave aside the question of whether we can associate class names with discovered clusters: this question is usually determined by the particular application. What we would like to understand is what sort of patterns are in the dataset, how are the data clumped, how are clumps shaped, how many different unique behaviours seem to be present?

→ This is the unsupervised learning problem of [Section 12.1](#).

12.1 Unsupervised Learning

A cluster may be defined as a set of measured data points which are similar to each other. Such a definition immediately presupposes the existence of a similarity criterion, and the clusters which result may strongly depend on the choice of similarity. For example, a cluster may be based on points which are closely spaced (a distance criterion) or, instead, a cluster may be defined based on regions in feature space containing a high density of sample points (a density criterion).

Suppose we adopt the latter sample-density definition of a cluster, such that peaks in an estimated density \hat{p} are associated with clusters. Given our N samples in dataset $\mathcal{D} = \{\underline{x}_1, \dots, \underline{x}_N\}$, we can estimate a probability density

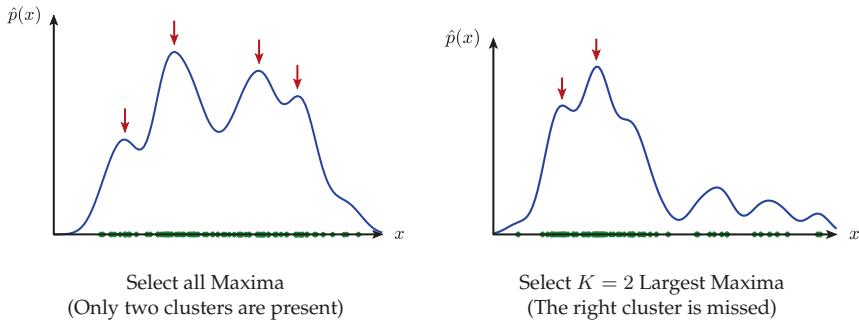


Fig. 12.2. CLUSTERING: An unknown distribution, characterizing a set of given points, will likely have maxima at cluster centres. However in practice maxima can be very sensitive to the chosen kernel and to outlying data points.

function using density estimation. Because the data are not labelled, the probability density here is global, over *all* of the classes, and not class-dependent, as it would have been in estimating $p(\underline{x} | C_\kappa)$ in [Chapter 7](#). As a result the density estimation must be non-parametric, such as the kernel estimate of (7.64),

$$\hat{p}(\underline{x}) = \frac{1}{s^n N} \sum_{i=1}^N \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right), \quad (12.2)$$

or, at least a mixture model,

$$\hat{p}(\underline{x}) = \sum_{\kappa=1}^K \mathbf{P}_\kappa p(\underline{x} | \underline{\theta}_\kappa) \quad (12.3)$$

consisting of the superposition of K parametric models, since a *single* parametric model will almost certainly not represent the combined distribution of K different and separated classes. The mixture model approach is actually highly related to K-Means, to be discussed in [Section 12.1.1](#), and a comparison with mixture models is summarized in [Table 12.1](#).

If we choose to estimate a distribution non-parametrically from (12.2), then we might assume clusters to be reasonably distinct and nearly equally likely, in which case we might expect $\hat{p}(\underline{x})$ to have a peak or local maximum centered on each class. If so, we could let each maximum be a cluster prototype:

$$\underline{z}_\kappa \text{ is the } \kappa\text{th local maximum of } p(\underline{x}). \quad (12.4)$$

Although intuitively appealing, this approach may be poor in practice because outlying or sparsely distributed samples will each be associated with local maxima which do not indicate cluster centres, as shown in [Figure 12.2](#).

In practice we will generally assume that the number of classes, K , is specified ahead of time.¹ We could attempt to improve on (12.4) by selecting only the K largest maxima:

$$\underline{z}_\kappa \text{, is the } \kappa\text{th largest local maximum of } p(\underline{x}), \quad 1 \leq \kappa \leq K, \quad (12.5)$$

As shown in Figure 12.2, even here we may find spurious maxima, and generally the number and location of maxima may be sensitive to the choice of kernel $\phi()$. However the real challenge is that, although straightforward in one dimension (as plotted in Figure 12.2) it is computationally essentially impossible to non-parametrically evaluate $\hat{p}(\underline{x})$ densely-gridded in \underline{x} over $n \gg 1$ dimensions. A computationally feasible approach for non-parametric maxima-finding forms the basis for the mean-shift method of Section 12.1.3.

A second intuitive scheme might be devised by applying the idea of principal components (from Section 5.1) to the total or overall covariance Σ_T from (5.8). Under the assumption that the separation *between* classes (whether two or more) will generally be larger than the size *within* a class, the directions (eigenvectors) associated with the largest eigenvalues of Σ_T should, under that assumption, point along those directions in which classes are well separated. These eigenvectors could (if few in number) possibly offer a lower-dimensional space in which to estimate the distribution, along the lines of (12.5), under the principle that it becomes progressively easier and more reliable to estimate a distribution as the number of dimensions is reduced. However the finding of maxima remains unreliable, the number of eigenvectors to keep is unclear, and it is easy to construct examples (Problem 12.3) in which cluster spacing assumptions do not hold.

A final intuitive approach directly selects cluster prototypes from the data points. Given a point-to-point distance function $d(\underline{x}_1, \underline{x}_2)$ from Section 4.1, and some cluster distance threshold T , then considering one data point \underline{x}_i at a time, the clustering rule is simple:

If $d(\underline{x}_i, \underline{z}_\kappa) < T$ for some κ , then consider \underline{x}_i to be associated with cluster κ , otherwise assign a new prototype $\underline{z}_{\text{New}} = \underline{x}_i$.

The virtue of this procedure lies in its simplicity, however its drawback lies in the arbitrariness of the threshold T , which could easily be too small (too many clusters) or too large (too few clusters), and the apparent sensitivity to the order of the data points in \mathcal{D} , since the very first point will necessarily be chosen as a prototype, but may not actually be representative of any cluster.

¹ Or, if not, then we would probably run some clustering algorithm for a variety of values of K , and then visualize or otherwise assess the results.

The intent of this section has been to encourage the reader to reflect upon what makes unsupervised (unlabelled) problems difficult, and the challenges in formulating a robust algorithm in the absence of much knowledge of the problem.

12.1.1 K-Means Clustering

The ideas of the preceding section are simple and intuitive, but are not how we should be attempting to formulate the problem. Given the relatively formal approach to defining optimization objectives in [Chapter 10](#) in order to find the optimal discriminant, similarly here we should be formulating an objective, and not guessing at an algorithm.

What does it really mean for a dataset \mathcal{D} to be optimally represented by a set of K prototypes $\underline{z}_1, \dots, \underline{z}_K$? Presumably the data points should, on average, be close to a prototype, so that our criterion is to minimize the average squared distance from each point to its closest prototype:

$$\mathcal{L}(\{\underline{z}_\kappa\}) = \sum_{\underline{x} \in \mathcal{D}} \min_{\kappa \in \{1, \dots, K\}} d^2(\underline{x}, \underline{z}_\kappa) \quad (12.6)$$

This objective function was parameterized in terms of the prototypes only, and for (somehow) previously-known K . The obvious generalization of [\(12.6\)](#) would be to allow K to be learned along with the prototypes,

$$\mathcal{L}(\{\underline{z}_\kappa\}, K) = \sum_{\underline{x} \in \mathcal{D}} \min_{\kappa \in \{1, \dots, K\}} d^2(\underline{x}, \underline{z}_\kappa) \quad 1 \leq \kappa \leq K \quad (12.7)$$

or for the distance function to be somehow dependent on class shape, as would be the case for the Mahalanobis distance, so that [\(12.6\)](#) could be generalized to learn both the prototypes and class covariances:

$$\mathcal{L}(\{\underline{z}_\kappa, S_\kappa\}) = \sum_{\underline{x} \in \mathcal{D}} \min_{\kappa \in \{1, \dots, K\}} (\underline{x} - \underline{z}_\kappa)^T S_\kappa^{-1} (\underline{x} - \underline{z}_\kappa) \quad 1 \leq \kappa \leq K \quad (12.8)$$

Both of the objectives in [\(12.7\)](#) and [\(12.8\)](#) have problems, in that the objectives can be minimized with meaningless solutions:

- In [\(12.7\)](#), objective $\mathcal{L}(\{\underline{z}_\kappa\}, K)$ can be minimized to zero by badly overfitting,² by allowing K to increase to N , thus having one prototype for *each* data point. This reveals nothing about the structure of the data, and just creates a “cluster” (of one point) at each data point.

² Recall that overfitting has been discussed in some detail in [Chapters 3 and 9](#).

The solution to such overfitting needs to be a penalty term on K , similar to that in (3.10), some expectation of the degree of improvement expected in (12.6) corresponding to an increase in K :

$$\mathcal{L}(\{\underline{z}_\kappa\}, K) = \sum_{\underline{x} \in \mathcal{D}} \min_{\kappa \in \{1, \dots, K\}} d^2(\underline{x}, \underline{z}_\kappa) + \gamma K \quad (12.9)$$

This formulation assumes that all clusters are similar in size, and a suitable value of γ will furthermore need to be based on prior knowledge of cluster size, since γ needs to be chosen to distinguish between correctly splitting two separate clusters (a desirable increase in K), as opposed to incorrectly splitting a single cluster (overfitting, an undesirable increase in K).

- In (12.8), objective $\mathcal{L}(\{\underline{z}_\kappa, S_\kappa\})$ can be minimized to zero, even for $K = 1$, by having $S_1 \rightarrow I \cdot \infty$, since that leads to $S_1^{-1} \rightarrow \mathbf{0}$.

The obvious limitation in (12.8) is that the values for S_κ are not constrained by or tied to the data points. What is needed is to have a common interdependence between prototype, covariance, and the data:

$$\begin{aligned} \mathcal{L}(\{\underline{z}_\kappa\}) &= \sum_{\kappa} \sum_{\underline{x} \in \mathcal{D}_\kappa} d_\kappa^2(\underline{x}, \underline{z}_\kappa) \quad d_\kappa^2(\underline{x}, \underline{z}_\kappa) = (\underline{x} - \underline{z}_\kappa)^T S_\kappa^{-1} (\underline{x} - \underline{z}_\kappa) \\ \mathcal{D}_\kappa &= \{\underline{x} \in \mathcal{D} \text{ such that } d_\kappa^2(\underline{x}, \underline{z}_\kappa) \leq d_j^2(\underline{x}, \underline{z}_j), j = 1, \dots, K\} \quad S_\kappa = \text{Cov}(\mathcal{D}_\kappa) \end{aligned} \quad (12.10)$$

In principle this seems circular: S_κ depends on \mathcal{D}_κ , \mathcal{D}_κ depends on \underline{z}_κ , and \underline{z}_κ depends on S_κ . This is, however, the whole point: we *want* these three quantities to be interrelated. The circularity also illustrates the fundamental difference between an *objective* and an *algorithm*: an objective is simply a statement of *fact* or *goal*, whereas an algorithm is a statement of *method*, a systematic and sequential way of accomplishing a given objective. So (12.10) simply states an objective; there may be a variety of algorithms to solve this problem approximately or exactly.

Before proposing an algorithm, it is essential to understand what is known and what is not known:

- Is K , the number of clusters, known or uncertain? If uncertain, then some form of hypothesis test need to be undertaken, as explained in significant detail in [Appendix D](#), to test whether single clusters should be split into two, or whether pairs of clusters should be combined into one.
- What is known about the minimum number of points per class, which could significantly impact how concerned we are about overfitting? Or about the minimum separation between classes or the minimum extent of any class, any of which would impact the class splitting/merging hypotheses of [Appendix D](#).

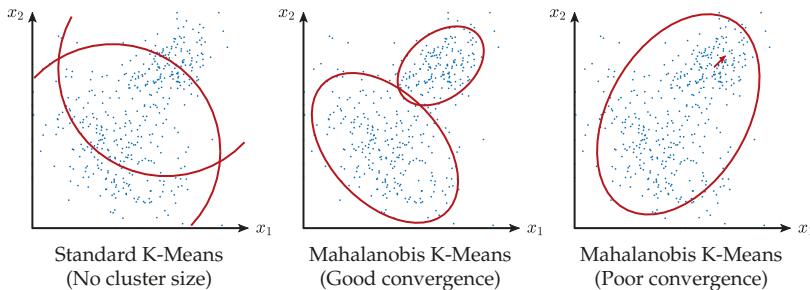


Fig. 12.3. K-MEANS: Standard K-means (left) characterizes each cluster based on a prototype only, and so has no data-driven notion of class size. Generalizing the clustering problem to include a dependence on class statistics via a Mahalanobis distance seems like an excellent idea (middle), because it can accommodate non-spherical class shapes. However such additional flexibility can easily lead to poor convergence (right), since the absence of class labels means that the K-Means iteration lacks certain restraints. Here, for example, the right panel shows Mahalanobis K-means to have learned one large cluster to capture nearly all of the data points, and a second very small cluster (small arrow) of essentially zero size.

- What is known about class shape? The full Mahalanobis distance would seem like a good idea, because of its flexibility, however it is essential to understand that greater model flexibility can very easily lead to overfitting and poor convergence, as illustrated in Figure 12.3. Instead, in most cases the scaled-Euclidean distance of (6.6) is well advised, as discussed in Example 6.1 and Figure 6.5, because of the arbitrariness of the units of measurements or features. This scaling would not be part of learning, meaning that the distance metric is selected separately, ahead of time, *before* the clustering process, such that we would be using (12.6) and not (12.10).

From the discussion in Section 6.2, for a *single* class of data points, the prototype z_κ which minimizes that single class squared-error is just the class mean. Furthermore, as derived in Appendix D, this same conclusion applies to any of the Euclidean, rescaled, rotated, or Mahalanobis distance functions. As a result, since a given distance function plus a set of prototypes implies a partitioning of the data into clusters, and each cluster (as just discussed) is best represented by its sample mean as prototype, we arrive at the standard K-means algorithm:

1. Partition the data set, associating each data point with its closest prototype.
2. Update the prototypes as the sample mean over data points in each cluster.
3. If using a class-dependent distance, update the sample covariance of each cluster.
4. Repeat steps 1 through 3 until the prototypes are no longer changing.

The above steps are illustrated in Figure 12.4 and written out in greater detail in Algorithm 12.1.

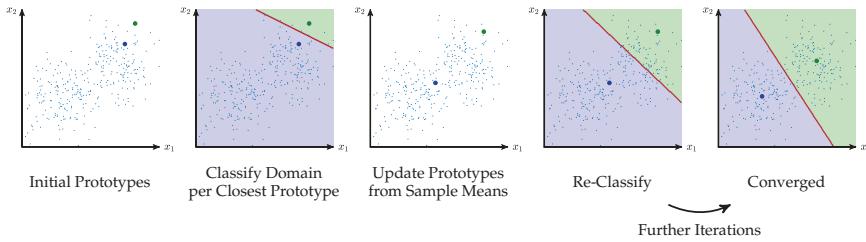


Fig. 12.4. K-MEANS: Given an initial set of prototypes (left), the basic idea behind K-Means is to iteratively classify all of the given data points and then to update the prototypes, as described in [Algorithm 12.1](#). There can be a variety of stopping criteria, however typically the prototypes stop changing in relatively few iterations. The method illustrated here is based on a Euclidean distance.

Algorithm: K-Means Clustering

```

Goals: Cluster  $N$  data points  $\{\underline{x}_i\}$  into  $K$  clusters based on distance  $d()$ 
Function Prototypes  $\{\underline{z}_\kappa\} = \text{K\_Means}(\{\underline{x}_i\}, K, d())$ 
  Choose  $K$  prototypes  $\underline{z}_1, \dots, \underline{z}_K$  at random from  $\underline{x}_1, \dots, \underline{x}_N$ 
  Done = 0
  while Done == 0 do
    for  $i = 1 : N$  do
       $c_i = \text{Dist\_Class}(\underline{x}_i, K, \{\underline{z}_\kappa\}, d())$            Classify using Dist_Class from
      Algorithm 6.1
    end for
     $s_1 = \dots = s_K = 0$   $\bar{p}_1 = \dots = \bar{p}_K = \underline{0}$ 
    for  $i = 1 : N$  do
       $s_{c_i} = s_{c_i} + 1$            Count samples in  $c_i$ th class
       $\bar{p}_{c_i} = \bar{p}_{c_i} + \underline{x}_i$            Sum up points in  $c_i$ th class
    end for
    Done = 1
    for  $\kappa = 1 : K$  do
       $\bar{p}_\kappa = \bar{p}_\kappa / s_\kappa$            Normalize to compute mean
      if  $\bar{p}_\kappa \neq \underline{z}_\kappa$  then
        Done = 0
         $\underline{z}_\kappa = \bar{p}_\kappa$ 
      end if
    end for
  end while

```

Algorithm 12.1. K-Means is essentially an iterative distance-based classifier as in [Chapter 6](#). Given a set of class prototypes and a distance function, each data point can be classified to the closest prototype. Then, given the classified data points, we can take their sample mean to update the class prototypes, as illustrated in [Figure 12.4](#). The algorithm as written here is based on a Euclidean distance, but is easily generalized to include sample covariance calculation to allow for non-Euclidean distances.

Example 12.1: Vector Quantization

K-Means clustering is very closely related to the problem of *Vector Quantization* from signal processing. The vector quantization problem is to find the optimal set of K vectors $\{\underline{z}_\kappa\}$ to approximate a given set of N vectors $\{\underline{x}_i\}$. That is, we would like to find the K vectors $\{\underline{z}_\kappa\}$ to minimize the average squared-error between \underline{x}_i and its nearest approximant:

$$\text{MSE}(\{\underline{x}_i\} | \{\underline{z}_\kappa\}) = \sum_i \min_{\kappa \in \{1, \dots, K\}} \sum_s (x_{i,s} - z_{\kappa,s})^2 \quad (12.11)$$

This criterion is identical to that of K-Means in (12.6), except that (12.11) is written explicitly in terms of a squared (Euclidean) distance, and that the intent or purpose of vector quantization is somewhat different from that of K-Means. In particular, K-Means is solving a clustering problem, whereby we believe that $\{\underline{x}_i\}$ are grouped into compact classes. Vector quantization, in contrast, is an approximation tool; there is no expectation of clustering in $\{\underline{x}_i\}$, rather the question is one of approximation, to minimize the average squared error.

For example, consider a 500×500 colour image, as shown. Each pixel is associated with three 8-bit values, the amounts of Red, Green, and Blue representing the colour and brightness of the pixel. Therefore each pixel (i, j) is represented by a vector $\underline{x}_{i,j}$, requiring 24 bits to encode.



Painting Credit: Anya Fieguth

Given K vectors $\{\underline{z}_\kappa\}$, vector quantization has each pixel $\underline{x}_{i,j}$ represented by its closest prototype:

$$\underline{x}_{i,j} \simeq \underline{z}_{\kappa_{i,j}} \quad 1 \leq \kappa_{i,j} \leq K \quad (12.12)$$

That is, $\kappa_{i,j}$ specifies which prototype is closest; given that one of K values can be chosen, encoding the value of $\kappa_{i,j}$ requires $\log_2 K$ bits. We now have a tradeoff between the quality of the reconstruction (larger K) and the degree of compression (smaller K):



$K = 2$ (1 bit/pixel)



$K = 4$ (2 bits/pixel)



$K = 16$ (4 bits/pixel)

The individual means (discrete colours) can be seen very clearly in the $K = 2$ and $K = 4$ cases.

In practice, image compression is much more sophisticated (for example, taking advantage of the fact that most of the time adjacent pixels are similar in value, whereas the approach illustrated here has coded each pixel independently). However in more general cases, where the vectors $\{\underline{x}_i\}$ are not pixels in an image, vector quantization/K-means can be an effective approach to approximation.

Gaussian mixture models	K-means clustering
K parameters to describe the distribution of a given set of points	K parameters to describe the distribution of a given set of points
Resulting model describes one class	Resulting model describes all classes
Typically low-dimensional ($n = 1, 2$)	Typically high-dimensional ($n \gg 1$)
Nearly always learning means and variances	Typically learning only class means
Weighted combination of classes	Classes are not weighted
A given data point contributes to multiple mixtures	A given data point contributes to one cluster

Table 12.1. A comparison of mixture models and clustering.

It is instructive to consider the similarity of the unsupervised K-Means clustering discussed here, and the Gaussian mixture models of Section 7.2, summarized in Table 12.1. In principle, K-Means and Gaussian Mixture Models are quite similar, in that both have K parameters describing the behaviour of a given set of data points. In practice, their use cases are quite different: a Gaussian Mixture Model is learning a parameterization for a statistical distribution for one class, whereas K-Means is inferring a partitioning of an entire data set into K disjoint clusters.

Many variations have been proposed, based on different choices of the distance function, such as the Minkowski class of Figure 6.4, or more flexible notions of class membership, such as the Fuzzy c -means algorithm (Problem 12.9), or rules for automatically increasing and decreasing K , based on certain tests of the cluster statistics.

The K-Means approach is attractive because the basic algorithm of Algorithm 12.1/Figure 12.4 is easy to understand, straightforward to implement, and fast to converge. In contrast, the underlying optimization problem of (12.6) is actually exceptionally difficult to solve optimally, because of the near-infinite number of possible partitions of the unlabelled data into clusters.

The parametric nature of K-Means leads to a limited flexibility with regards to class shape: for a Euclidean or scaled-Euclidean definition of distance, the K-Means approach assumes all clusters to be compact. Although it is possible to generalize the parametric model to allow for a greater variety of class shapes, such a generalization is *much* more problematic in unlabelled clustering than it was in earlier chapters which explored parametric models:

- Labelled-data problems (that we have seen in Chapters 4 through 11) constrain any learning problem because the data association is *known*. We want to avoid overfitting the data (as discussed in Chapter 3), however aside from the overfitting risk, we are free to choose any parametric or non-parametric model, whether simple or complex, and learn a class representation.

- In contrast, the unlabelled-data problem of this chapter has *unknown* data association, meaning that we do not know which data points will be in the same class as other data points. As a result, the risk of overfitting is *far* greater, since a more flexible class model (e.g., Mahalanobis) can coincidentally fit all manners of point groupings (tiny, huge, elongated) which are not actual clusters, but which can be made to (over)fit the model.

It is for this reason that K-Means is nearly always used with only prototypes, and not a Mahalanobis distance. Clearly this approach implies fairly restrictive limitations with regards to the clustering problem:

- **Number of Clusters:** Standard K-Means requires that the value of K , the number of clusters, be known. This issue was discussed at some length, with K incorporated into the objective in (12.9), or statistical rules for deciding whether clusters should be combined or not derived in [Appendix D](#). If the problem is of modest size, it may be reasonable to run K-Means for a range of values of K , and then to visually inspect the plot of $\mathcal{L}(\{\underline{z}_k\})$ from (12.7) as a function of K .
- **Cluster Boundaries:** Based on a Euclidean or scaled-Euclidean distance function, standard K-Means necessarily leads to straight (hyperplane) boundaries between clusters, which means that linearly non-separable clusters cannot properly be separated, which will be addressed under Kernel K-Means in [Section 12.1.2](#).
- **Hard Clustering:** Each data point is forced to be in only a single cluster, leading to a discontinuous optimization problem, since the optimization objective \mathcal{L} will change abruptly (discontinuously) as a classification boundary moves across a data point. The Mean-Shift method ([Section 12.1.3](#)) uses a continuous kernel to avoid this problem, or the Fuzzy c-Means variation ([Problem 12.9](#)) on K-Means, which allows for continuous class membership.
- **Sensitivity to Initialization:** Standard K-Means says little regarding how the prototypes are to be initialized at the beginning of the algorithm; typically the K prototypes are chosen, at random, from K of the given data points, however the resulting K cluster prototypes can be highly dependent on the initialization.³ In many cases it is then recommended to run K-Means repeatedly, with different initial prototypes, and to select

³ Many nonlinear iterative methods can display significant initialization dependence, because the overall objective function has many local minima, as discussed in [Appendix C](#). A classic example is the superficially straightforward problem of root finding, where the initialization dependence can be fractal, as illustrated in Chapter 7 of [6].

those prototypes which persistently/repeatedly appear in the converged results, essentially like an ensemble approach from [Chapter 11](#).

An attractive alternative is to use a *non-parametric* method, which does not have any prototype parameters to initialize, and therefore having no initialization dependence. Two non-parametric methods will be discussed: Mean-Shift in [Section 12.1.3](#), and Hierarchical Clustering in [Section 12.1.4](#).

- **Sensitivity to Outliers:** As was illustrated in [Figure 6.12](#) and discussed in [Appendix B.5](#), the mean is somewhat sensitive to outliers, whereas the median is not. It would be straightforward to update [Algorithm 12.1](#) to be based on a median⁴ as opposed to a mean.

12.1.2 Kernel K-Means Clustering

Since K-Means is normally based on a Euclidean or scaled-Euclidean distance function, the resulting decision boundaries are necessarily hyperplanes, meaning that clusters must be linearly separable in order to be successfully clustered.

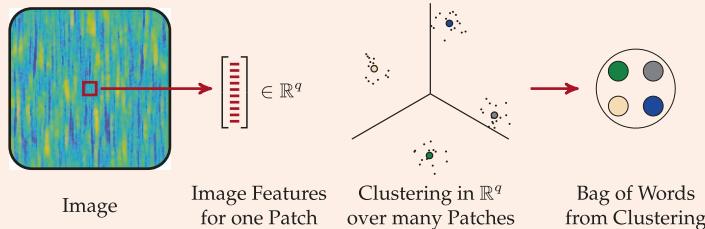
We have seen precisely the same limitation in the context of linear discriminants, in [Chapter 10](#), which *also* assert hyperplane boundaries between classes. This text formulated two responses to this limitation:

- Select ensembles of linear discriminants and combine, as discussed in [Chapter 11](#). However the ensemble methods of [Chapter 11](#) were applied to *labelled* data, which constrain the problem; here in the *unlabelled* case the increase in parameterization associated with an ensemble of discriminants would lead to overfitting problems, as was discussed on [page 357](#). That being said, it *is* possible to use neural networks (which are essentially ensembles of discriminants), to work with unlabelled data, as discussed in [Section 12.2](#)
- Nonlinearly warp the entire space to transform classes, linearly non-separable in the original feature space, into linearly-separable classes in a very high-dimensional transformed space, as was discussed in [Section 10.3](#). One might very well ask whether the same overfitting concern, just discussed, would apply here; after all, a high-dimensional transformation involves a great number of degrees of freedom. The key issue is that the *same* transformation applies to all of the points, from *all* clusters, the transformation is not cluster-dependent. As a result, the unknown data association of the unlabelled data is not an issue.

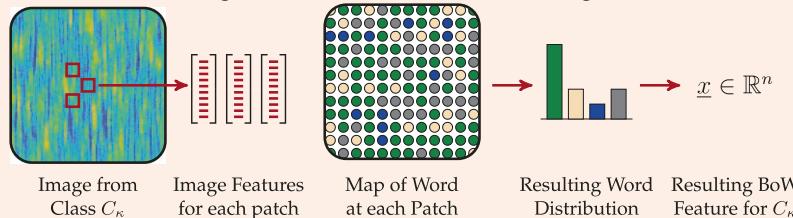
⁴ For example see [K-medians Clustering](#) or [K-Medoids](#).

Example 12.2: Bag-of-Words and Visual Representation

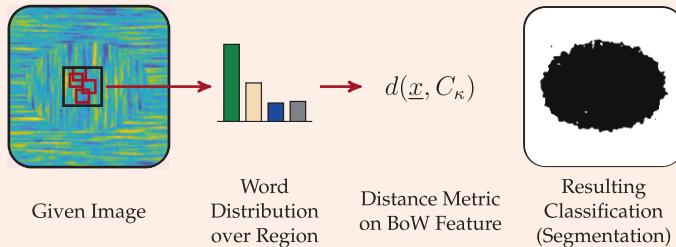
The **Bag-of-Words** concept of [Section 5.2](#) can be applied to non-text problems, however we will then have to define what a “*word*” means. For example, suppose we wish to classify image texture. For each small image patch, we could collect very simple features (pixel differences etc.) and perform clustering in that space:



Each cluster represents a concentration in the image-primitive space. Each prototype (coloured circle) then becomes one of our “*words*” which we can use to describe a texture. However these words are not the textures themselves. Each word describes a very simple, local behaviour, whereas a texture is characterized by the spatial (non-local) pattern of behaviours. So we look at the *distribution* (histogram) of words over an entire image:



So it is the distribution of words which is our actual pattern-recognition feature \underline{x} . Now, given a new image to classify (either classifying the whole image as a single texture or, as shown here, to classify each region to allow the image to be segmented into its texture classes), we calculate a distribution, and compare it to the learned distribution of each class:



This is a very powerful idea that can be applied in many different contexts.

Further Reading: L. Liu, L. Wang, L. Zhao, P. Fieguth, “Random projections and Single BoW for fast and Robust texture segmentation,” *Information Sciences*, 2016

In conclusion, it is therefore plausible to consider applying a nonlinear kernel to transform a given clustering problem. At first glance the Kernel K-Means method can appear very complicated. In actual fact it is fairly intuitive, and follows as the consequence of three pieces, each of which we have already seen:

1. **The Transformation:** Recall from [Section 10.3](#) that a feature vector \underline{x} is nonlinearly transformed into high-dimensional $\bar{\underline{x}} = \varphi(\underline{x})$. If the transformation were applied to all data points, then the whole dataset would be transformed: $\bar{\mathcal{D}} = \varphi(\mathcal{D})$. Keep in mind that the whole point is *not* to explicitly transform the problem, rather to implicitly solve the problem in the transformed domain, but without ever finding $\bar{\mathcal{D}}$.
2. **The Computation:** In the transformed space the clusters are assumed to be linearly separable, therefore the Euclidean distance of basic K-Means will be used, and no class-dependent statistics need to be learned. In K-Means, from [page 353](#), the two basic operations are of partitioning the data set

$$\bar{\mathcal{D}}_\kappa = \{\bar{\underline{x}} \in \bar{\mathcal{D}} \text{ such that } d_E(\bar{\underline{x}}, \bar{\underline{z}}_\kappa) \leq d_E(\bar{\underline{x}}, \bar{\underline{z}}_j)\} \quad 1 \leq j \leq K \quad (12.13)$$

and updating the prototypes from the sample means

$$\bar{\underline{z}}_\kappa = \text{Sample Mean}(\bar{\mathcal{D}}_\kappa) = \frac{1}{|\bar{\mathcal{D}}_\kappa|} \sum_{\bar{\underline{x}} \in \bar{\mathcal{D}}_\kappa} \bar{\underline{x}} \quad (12.14)$$

In principle these two steps are straightforward, except for the huge caveat that we do not wish to represent *anything* in the transformed domain: *none* of $\bar{\underline{x}}$, $\bar{\mathcal{D}}$, $\bar{\mathcal{D}}_\kappa$ or even the prototypes $\bar{\underline{z}}_\kappa$ can be calculated or stored.

3. **The Kernel Trick:** Here is the crucial idea ... we do not *actually* need to find the prototypes: most fundamentally our job is to divide \mathcal{D} into clusters, *not* to produce a set of prototypes.

We begin by moving the datasets of [\(12.13\)](#) back into the original domain:

$$\mathcal{D}_\kappa = \{\underline{x} \in \mathcal{D} \text{ such that } d_E(\varphi(\underline{x}), \bar{\underline{z}}_\kappa) \leq d_E(\varphi(\underline{x}), \bar{\underline{z}}_j)\} \quad 1 \leq j \leq K \quad (12.15)$$

Next, we need to express the transformed prototype $\bar{\underline{z}}$ in terms of the original domain:

$$\bar{\underline{z}}_\kappa = \frac{1}{|\mathcal{D}_\kappa|} \sum_{\underline{a} \in \mathcal{D}_\kappa} \varphi(\underline{a}) \quad (12.16)$$

Finally, we need to express the Euclidean distance between a point and a prototype, again based on the original domain:

$$d_E(\bar{x}, \bar{z}_\kappa) = d_E(\varphi(\underline{x}), \bar{z}_\kappa) = d_E \left(\varphi(\underline{x}), \frac{1}{|\mathcal{D}_\kappa|} \sum_{\underline{a} \in \mathcal{D}_\kappa} \varphi(\underline{a}) \right) \quad (12.17)$$

The kernel Φ_φ expresses dot products of transformed vectors,

$$\Phi_\varphi(\underline{x}_i, \underline{x}_j) = \varphi(\underline{x}_i) \bullet \varphi(\underline{x}_j) = \bar{x}_i \bullet \bar{x}_j, \quad (12.18)$$

so the key is to express d_E in (12.17) in terms of dot products:

$$\begin{aligned} d_E^2(\varphi(\underline{x}), \bar{z}_\kappa) &= (\varphi(\underline{x}) - \bar{z}_\kappa)^T (\varphi(\underline{x}) - \bar{z}_\kappa) = \varphi(\underline{x}) \bullet \varphi(\underline{x}) \\ &\quad - 2\varphi(\underline{x}) \bullet \bar{z}_\kappa + \bar{z}_\kappa \bullet \bar{z}_\kappa \end{aligned} \quad (12.19)$$

which we can solve entirely in the original domain if we substitute in for the prototypes and replace dot products with kernel entries:

$$d_E^2(\varphi(\underline{x}), \bar{z}_\kappa) = \Phi_\varphi(\underline{x}, \underline{x}) - \frac{2}{|\mathcal{D}_\kappa|} \sum_{\underline{a} \in \mathcal{D}_\kappa} \Phi_\varphi(\underline{x}, \underline{a}) + \frac{1}{|\mathcal{D}_\kappa|^2} \sum_{\underline{a} \in \mathcal{D}_\kappa} \sum_{\underline{b} \in \mathcal{D}_\kappa} \Phi_\varphi(\underline{a}, \underline{b}) \quad (12.20)$$

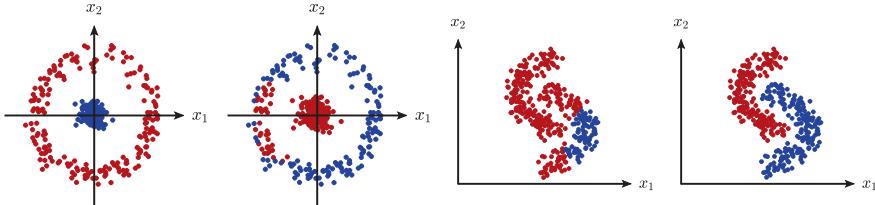
So the prototype is now entirely implicit, it is never calculated. Not only that, the prototype *does not exist* in the original space, because the transformation φ is not invertible. That is, *in principle* \bar{z} exists in the transformed space (even though we do not compute it), however

$$\text{There does } \textit{not} \text{ exist any } \underline{z} \text{ such that } \bar{z} = \varphi(\underline{z}) \quad (12.21)$$

There simply is no point in the original domain that corresponds to the transformed prototype. Really the only way to understand \bar{z} is from (12.16); that is, to understand the transformed prototype \bar{z}_κ as being made up of *all* of the points in \mathcal{D}_κ .

Even more abstract, not only is the prototype implicit, as just discussed, but even the *transformation* $\varphi()$ is implicit! That is, what is explicitly specified is the kernel Φ_φ . The kernel is, of course, associated with a particular transformation family. However, as derived in Appendix D, it turns out that there are actually infinitely-many transformations associated with the *same* kernel Φ , meaning that we should not be trying to think, in concrete terms, about the definition of the individual transformations $\varphi_j()$.

Since the prototype is not (and cannot be) calculated, the only thing that changes from one iteration to the next is which data point \underline{x} is in which cluster, so it is the data partitioning $\{\mathcal{D}_\kappa^{(t)}\}$ which we need to keep track of as a function of iteration t , leading to the Kernel-Kmeans algorithm:



The RBF kernel solves this problem very reliably (left), however the results can still be initialization dependent (right).

These clusters are more closely spaced, so most of the time (left) convergence is poor. The correct answer (right) is a fixed point of the algorithm, but can be found only rarely.

Fig. 12.5. KERNEL K-MEANS: Allows for nonlinear clustering, and the separation (left) is very impressive. However closely-spaced clusters (right) tend to mis-converge, and the results can be very sensitive to the choice of kernel. The resulting K-means prototypes cannot be shown, since they exist only in the transformed (high-dimensional) space, and even there are only implicit, not explicitly calculated.

- Initialize $\{\mathcal{D}_\kappa^{(0)}\}$ with one data point, at random, for each of K clusters
- Iterate $t = 1, 2, \dots$
 - Use $\{\mathcal{D}_\kappa^{(t-1)}\}$ to compute the Euclidean distances, in the transformed space, from each point to each cluster, based on (12.20)
 - Use the Euclidean distances, just computed, to determine the updated partitioning $\{\mathcal{D}_\kappa^{(t)}\}$ based on (12.15)
- Continue iterating until the partitioning has converged: $\{\mathcal{D}_\kappa^{(t-1)}\} = \{\mathcal{D}_\kappa^{(t)}\}$

Figure 12.5 shows results on two datasets. The leftmost panel, which converges very reliably, is highly motivating, in that the kernel-based K-Means clearly is able to nonlinearly separate clusters. The right example is far less reliable; the correct solution (rightmost panel) is a stable fixed point,⁵ meaning that kernel K-Means *can* converge to the right answer, but in actual fact rarely does so.

Technically the kernel K-Means method is parametric, in that some prototype does exist in the transformed high-dimensional space. In practice, it is nearly non-parametric, since the distance from a point to a class primarily depends on the middle term in (12.20),

$$-\frac{2}{|\mathcal{D}_\kappa|} \sum_{\underline{a} \in \mathcal{D}_\kappa} \Phi_\varphi(\underline{x}, \underline{a}), \quad (12.22)$$

⁵ A *stable fixed point* of an iterative algorithm [6] is a configuration which does not change with further iterations, and to which the algorithm will converge if initialized close by. However a successful convergence may require very proximate initialization (that is, with the clusters mostly correctly separated), normally an unrealistic assumption.

essentially a weighted average kernel, the average value of ϕ evaluated between \underline{x} and all members of cluster \mathcal{D}_κ . If the kernel is very local the weighted average is only over nearby pixels, easily creating local clumping (right example of Figure 12.5), whereas if the kernel is very broad the weighted average is close to uniform over all pixels, leading to regular K-Means.

Kernel K-Means is a very elegant idea, however it is computationally slow, can be poor to converge, and very sensitive to the choice of kernel. In practice, for difficult clustering cases, we would prefer the explicitly non-parametric approaches of Mean-Shift or Hierarchical-Clustering.

12.1.3 Mean-Shift Clustering

The Mean-Shift Clustering algorithm is actually very much like K-Means, in that a mean is shifted based on local data points until it converges, but with specific changes from K-Means to address the following three limitations:

- The need to specify K , the number of clusters;
- The outlier sensitivity, since the means can be pulled by distant outliers;
- Initialization sensitivity.

The concept underlying Mean-Shift is to find the peaks (local maxima) of a non-parametric density estimation

$$\hat{p}(\underline{x}) = \frac{1}{s^n N} \sum_{i=1}^N \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right), \quad (12.23)$$

from (12.2), originally developed in Section 7.3. If a realistic evaluation of \hat{p} requires each feature direction x_j to be sampled at, say, 100 values, then $\hat{p}(\underline{x})$ will need to be evaluated over a grid of 100^n points, likely to be computationally infeasible.

Instead, the Mean-Shift algorithm finds the peaks of $\hat{p}()$ iteratively. At any feature point \underline{x} , the kernel-weighted sample mean of the points around \underline{x} is

$$\hat{\mu}(\underline{x}) = \frac{\sum_{\underline{x}' \in \mathcal{D}} \underline{x}' \cdot \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right)}{\sum_{\underline{x}' \in \mathcal{D}} \phi\left(\frac{\underline{x} - \underline{x}_i}{s}\right)} \quad (12.24)$$

Since $\phi()$ is normally local in extent, in practice the sum is taken only over those data points in the neighbourhood of \underline{x} , and not over the entire dataset \mathcal{D} .

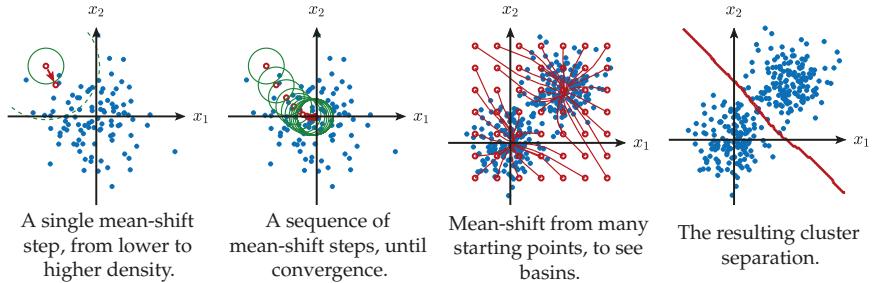


Fig. 12.6. MEAN SHIFT: By starting at a point and evaluating a kernel-weighted mean, an iteration (left) moves towards a higher density of points, converging at a peak of density in the middle of some cluster. By applying Mean-Shift to a grid of points, right, we can see the basins of convergence, each associated with one cluster.

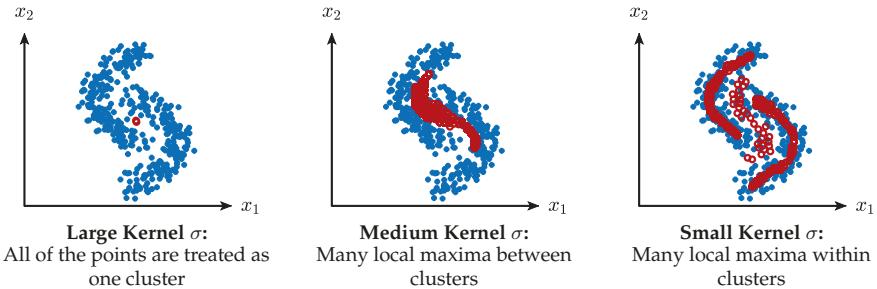


Fig. 12.7. MEAN SHIFT: Building on Figure 12.6, extended/non-compact clusters are problematic for mean-shift, since the method is premised on associating clusters with peaks in the probability distribution, whereas the clusters depicted here have more of an extended ridge as the maximum, rather than a point.

The difference $\underline{x} - \hat{\mu}(\underline{x})$ is referred to as the *mean shift*. By starting from some point $\underline{x}^{(0)}$ and iterating

$$\underline{x}^{(t+1)} = \hat{\mu}(\underline{x}^{(t)}), \quad (12.25)$$

we perform gradient ascent on $\hat{p}(\underline{x})$, converging on a peak (mode) of the distribution:

$$\underline{z} = \lim_{t \rightarrow \infty} \underline{x}^{(t)} \quad \rightarrow \quad \nabla \hat{p}(\underline{z}) = 0 \quad (12.26)$$

The iteration is run, normally in parallel, starting from all $\underline{x} \in \mathcal{D}$, and the set of converged points represent the modes and corresponding proposed cluster centres $\{\underline{z}_\kappa\}$.

By starting the algorithm at *each* data point (which eliminates initialization sensitivity), based on a local kernel $\phi()$ (which eliminates outlier sensitivity), and finding the set of points of convergence (which eliminates the need to specify K) we have addressed three of the limitations of K-Means.

Given the K converged centres $\{\underline{z}_\kappa\}$ (where, to be clear, K was *found* as the number of unique modes, and not asserted ahead of time), the clustering domain associate with cluster C_κ is found as the so-called *basin of attraction* \mathcal{B}_κ for mode \underline{z}_κ , that part of feature space \mathcal{X} that would converge to that mode under mean-shift:

$$\mathcal{B}_\kappa = \left\{ \underline{x} \text{ such that if } \underline{x}^{(0)} = \underline{x} \text{ then } \lim_{t \rightarrow \infty} \underline{x}^{(t)} = \underline{z}_\kappa \right\} \quad (12.27)$$

as illustrated in [Figure 12.6](#).

The algorithm does require that we specify a kernel $\phi()$, and particularly its scaling s , which can be difficult to select, particularly in high-dimensional spaces which are difficult to visualize. The mean-shift method is premised on associating clusters with peaks in the (kernel-estimated) probability distribution, and there are cases where the maxima may not be representative of cluster behaviour, as was already anticipated in [Figure 12.2](#). An example is shown in [Figure 12.7](#), where no single kernel scaling s is appropriate to separate the clusters, and indeed the elongated arc-like shape is inconsistent with the premise of each cluster having a single peak in the distribution.

Although outlier sensitivity is removed, in that a given mean-shift converged cluster centre is unaffected by distant outlying points, a *new* outlier sensitivity is *introduced*, in that distant, isolated points will, almost certainly, be associated with local maxima in $\hat{p}(\underline{x})$, and thus become their *own* cluster centres.

The Mean-Shift algorithm operates in the feature space \mathcal{X} , and not in a nonlinearly-transformed space of Kernel K-Means, so we do not have benefit of nonlinear transformations to accommodate a wider range of cluster shapes.

12.1.4 Hierarchical Clustering

Each clustering method discussed to this point has made certain assumptions about cluster shape, whether parametric via a prototype (K-Means, Kernel K-Means) or non-parametrically, but still assuming each cluster to be characterized by a single peak in an estimated probability density (Mean-Shift). Clearly it would be desirable for clustering to support a wider variety of cluster shapes, but without the overfitting difficulties of complex parameterizations.

We encountered a similar issue in distance-based classification, in [Chapter 6](#), where non-parametric classification schemes (NN, k NN) were developed to relax assumptions and to accommodate a much greater variety of class shapes. In [Chapter 6](#), this was accomplished by defining a non-parametric point-to-class distance function $d(\underline{x}, C)$. We will undertake something very similar here in the context of clustering.

Given a dataset \mathcal{D} which has been partitioned into a set of clusters

$$\mathcal{D} = \bigcup_{\kappa} \mathcal{D}_{\kappa} \quad 1 \leq \kappa \leq Q \quad (12.28)$$

where we use value Q quite deliberately, as opposed to K , since the number, Q , of partitions may very well not equal the expected number of clusters.

Suppose, in fact, that $Q \gg K$, meaning that the problem is *over-partitioned*, meaning that there will be several $\{\mathcal{D}_{\kappa}\}$ which divide up what is meant to be a single cluster. Therefore our task is to combine certain pieces of $\{\mathcal{D}_{\kappa}\}$ together, specifically those which are *meant* to be together in a single cluster.

So which of $\{\mathcal{D}_{\kappa}\}$ are meant to be together? Simply stated, hierarchical clustering proposes that the two elements in $\{\mathcal{D}_{\kappa}\}$ *most* likely to lie in the same cluster are those having the minimum distance between them. That is, generalizing from the point-to-class distance function $d(\underline{x}, C)$, we now need a cluster-to-cluster distance function $d(\mathcal{D}_a, \mathcal{D}_b)$. Simple examples would build on the ideas of [Chapter 6](#), such as the separation between cluster sample means,

$$d_{\text{Mean}}(\mathcal{D}_a, \mathcal{D}_b) = d_S(\text{Mean}(\mathcal{D}_a), \text{Mean}(\mathcal{D}_b)) \quad (12.29)$$

the separation between the nearest neighbours, the closest points in the two clusters,

$$d_{\text{NN}}(\mathcal{D}_a, \mathcal{D}_b) = \min_{\underline{x}_a \in \mathcal{D}_a, \underline{x}_b \in \mathcal{D}_b} d_S(\underline{x}_a, \underline{x}_b) \quad (12.30)$$

or the furthest neighbours,

$$d_{\text{FN}}(\mathcal{D}_a, \mathcal{D}_b) = \max_{\underline{x}_a \in \mathcal{D}_a, \underline{x}_b \in \mathcal{D}_b} d_S(\underline{x}_a, \underline{x}_b) \quad (12.31)$$

Clearly a wide variety of metrics is possible, such as a variant on k NN. Since hierarchical clustering begins each cluster as a single point, we need to account for the possibility that k will exceed the number of points in a cluster:

$$k_a = \min(|\mathcal{D}_a|, k) \quad k_b = \min(|\mathcal{D}_b|, k) \quad (12.32)$$

So we will measure to the $(k_a \leq k)$ th and $(k_b \leq k)$ th points in the clusters:

$$d_{k\text{NN}}(\mathcal{D}_a, \mathcal{D}_b) = d_S \left(\begin{array}{l} k_a \text{th closest point in } \mathcal{D}_a \text{ to } \mathcal{D}_b, \\ k_b \text{th closest point in } \mathcal{D}_b \text{ to } \mathcal{D}_a \end{array} \right) \quad (12.33)$$

All three distance functions were written in terms of the scaled Euclidean distance d_S , as discussed in the clustering context in [Section 12.1.1](#), and earlier in [Example 6.1](#) and [Figure 6.5](#).

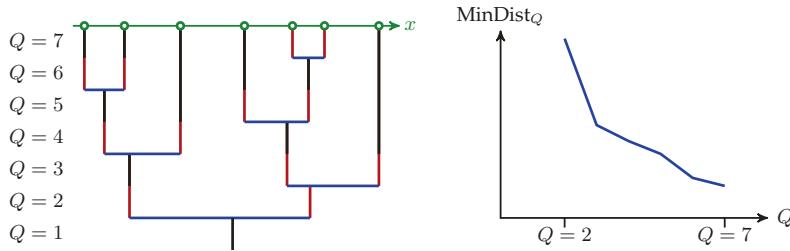


Fig. 12.8. A DENDROGRAM illustrating the principle of hierarchical clustering. The method is initialized at the top, where each data point (green) becomes a single cluster. At each iteration, proceeding from top to bottom, the two closest clusters (red) are combined, and the corresponding minimum distance (blue) at each iteration is plotted, right. The process is illustrated here based on the mean prototype.

The steps, then, for hierarchical clustering are conceptually fairly straightforward. Given a cluster–cluster distance function $d()$ and a dataset \mathcal{D} consisting of N points:

1. Initialize a cluster at each given data point:

$$Q = N \quad \mathcal{D}_i = x_i \quad (12.34)$$

2. Find the two closest⁶ clusters:

$$(\hat{a}, \hat{b}) = \arg_{(a,b)} \min d(\mathcal{D}_a, \mathcal{D}_b) \quad \text{MinDist}_Q = d(\mathcal{D}_{\hat{a}}, \mathcal{D}_{\hat{b}}) \quad (12.35)$$

3. Merge these two clusters, which reduces Q by one:

$$\mathcal{D}_{\text{New}} = \mathcal{D}_{\hat{a}} \cup \mathcal{D}_{\hat{b}} \quad \text{Delete}(\mathcal{D}_{\hat{a}}) \quad \text{Delete}(\mathcal{D}_{\hat{b}}) \quad Q = Q - 1 \quad (12.36)$$

4. If multiple clusters remain present ($Q > 1$) then continue to Step 2

The process is illustrated for a very simple one-dimensional example in [Figure 12.8](#). The sequence of grouping steps, from top to bottom, illustrates the pair of closest clusters at every stage of the method. The resulting diagram, known as a dendrogram, offers a visual interpretation of the clumping in the domain, an approach often used in plotting taxonomies (of plants or animals, for example). Of course, in a large problem with a vast number of data points in many dimensions, such a dendrogram would *not* be useful.

In contrast, the minimum cluster separation MinDist_Q of (12.35) is a relatively simple one-dimensional, monotonically-decreasing⁷ function regardless of

⁶ Keep in mind that even using *furthest neighbour* we are still looking for the *closest* clusters. The furthest-neighbour prototype tells us only how distance is measured; once we have a definition of distance (whether based on furthest-neighbour or any other prototype) we are always looking for the two closest clusters.

⁷ A decreasing function of Q , but an *increasing* function of hierarchical clustering iteration, since Q gets smaller with every iteration.

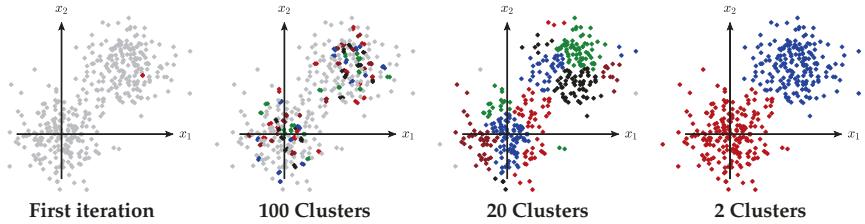


Fig. 12.9. HIERARCHICAL CLUSTERING: An illustration of the progression of hierarchical clustering, here based on a Euclidean distance and mean prototype. At each iteration the *closest* clusters are grouped, which means that at the very first iteration (left) the two closest points (red dot) are grouped. The mean prototype used here encourages local clumps, as we can see in the cases of $Q = 20$ or $Q = 2$ clusters, right. All points in clusters of size one (not yet grouped) are shown in grey.

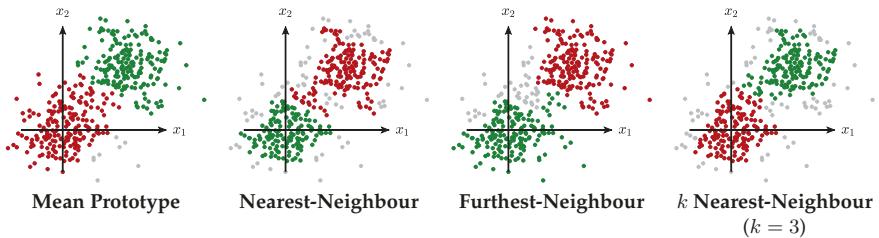


Fig. 12.10. HIERARCHICAL CLUSTERING: Four examples of clustering, each based on a different prototype, as shown. The light grey dots are those in separate small clusters, not yet joined with one of the main clusters. Since the data are unlabelled, the choice of colour is arbitrary (that is, the clustering algorithm has no way of knowing which cluster to call “red” and which “green”). Since the clusters are compact, circular, and fairly dense, all metrics perform similarly well.

the dimensionality of the feature space or size of the dataset. Although a detailed analysis of MinDist_Q is difficult, in principle to find K (the “true” number of clusters) we are looking for a transition between *inter*-cluster distance and *intra*-cluster distance. That is, when $Q > K$ we are (ideally) combining pieces which are part of the same cluster (*intra*), whereas when $Q < K$ we are (again, ideally) combining pieces which are part of separate clusters (*inter*). The behaviour of MinDist_Q therefore closely parallels the underfitting/overfitting Figure 3.3, and discriminating between intra-cluster and inter-cluster distances is thus closely related to the Akaike information criterion of Figure 3.6.

Clustering results for a much larger number of data points are shown in Figures 12.9 and 12.10, where Figure 12.9 illustrates the iterative progression of a single run of hierarchical clustering, whereas Figure 12.10 shows the final two clusters which result, but for various choices of prototype. The behaviour of the various prototypes is consistent with our understanding from Figure 6.12

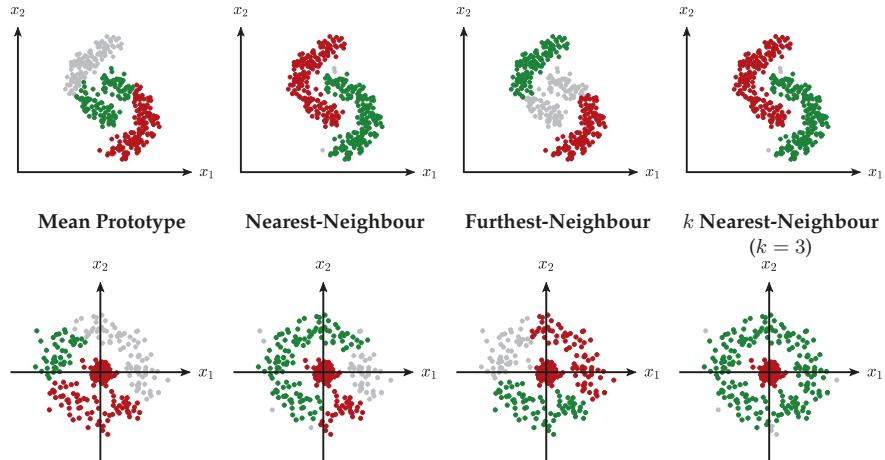


Fig. 12.11. HIERARCHICAL CLUSTERING: Here the clustering problem is somewhat more challenging than in [Figure 12.10](#): the clusters are elongated (non-compact) and are not linearly separable. The mean and furthest-neighbour prototype both assume compactness and are unable to perform effectively. The nearest-neighbour can easily be distracted to follow a path of noisy data to connect two clusters (as here, in the bottom row, but could equally well have taken place in the top example). The k nearest-neighbour prototype, coupled with the non-parametric nature of hierarchical clustering, is remarkably capable and robust.

and [Table 6.1](#). Furthest-neighbour fairly strongly asserts compactness, and the mean prototype prefers compactness but is less sensitive to outliers than furthest or nearest neighbour. Nearest neighbour completely ignores global cluster shape, and is interested only in inter-point spacing, giving it great flexibility, but also a tendency to connect clusters that (inconveniently) happen to have a few points scattered between them.

Two more difficult examples are demonstrated in [Figure 12.11](#). In neither case are the clusters compact, so it is only the nearest-neighbour family of prototypes that is applicable. The clusters are also not widely separated, meaning that nearest-neighbour might accidentally connect the clusters, as can be seen in the lower example of [Figure 12.11](#). The k -nearest-neighbour prototype performs remarkably well, and is able to handle non-compact clusters with irregular shapes in the presence of noisy outliers.

Overall the hierarchical clustering strategy is non-parametric, deterministic, produces a clustering result for all possible numbers of clusters (so that the number of clusters, K , does not need to be specified in advance), and allows any distance function and prototype definition to be asserted.

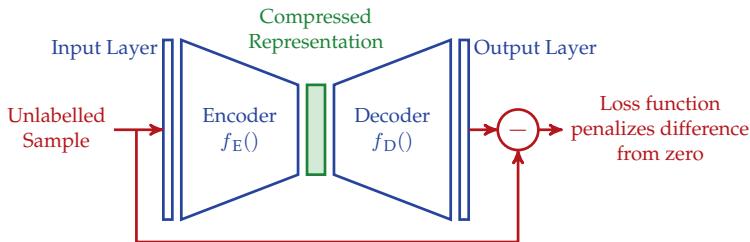


Fig. 12.12. AUTO-ENCODER: Even though the input samples are unlabelled, we can perform *self-supervision* by constructing a network in which we wish the output layer (right) to equal the input layer (left). The number of neurons in the compressed layer (green) controls the degrees of freedom in the compressed representation. The idea, then, is that the network is challenged to reproduce its m -element input at its m -element output, but being forced to pass through a $k < m$ element bottleneck. The actual operations within the bottleneck will be illustrated further in [Figure 12.13](#).

12.2 Network-Based Clustering

Neural networks were defined and developed in [Section 11.4](#). Their strength lies in combining a very large number of linear discriminants, each followed by a nonlinear activation function, leading to an overall network which is capable of essentially arbitrary nonlinear feature-space transformation and classification.

In the context of clustering, we have explored nonlinear feature transformation, using Kernel K-Means in [Section 12.1.2](#), however that method does require the specification of, and can be fairly sensitive to, the choice of an appropriate kernel, which requires us to have some understanding and insight into the clustering feature space, possibly very difficult in high-dimensional problems.

We have also explored hierarchical clustering, in [Section 12.1.4](#), and which proved to be a remarkably capable approach, however the distance metric and prototype do need to be selected, again, as with kernel k-means, requiring some insight into the feature space.

We would be motivated to consider network methods, however at first glance they would appear to be inapplicable: network learning is inherently *supervised*, and so would seem to be incompatible with the unlabelled clustering problem. There is a clever alternative, however, known as *self-supervision*.

In particular, consider the so-called auto-encoder or encoder-decoder network shown in [Figure 12.12](#), possessing an hourglass shape, such that the input and output layers have the same size, however the information in the network is constrained to pass through a bottleneck in the middle:

$$\begin{array}{ccccc} \mathcal{Y}_{\text{Input}} \in \mathbb{R}^m & \xrightarrow{\text{Encoder}} & \text{Compressed} \\ & f_E() & \text{Representation} \\ & & \underline{x} \in \mathbb{R}^k & \xrightarrow{\text{Decoder}} & \mathcal{Y}_{\text{Output}} \in \mathbb{R}^m \\ & & f_D() & & \end{array} \quad (12.37)$$

Given an unlabelled dataset of measurements $\mathcal{D} = \{y_i\}$, the network is trained to make its output the same as its input:

$$\mathcal{L}(\mathcal{D}) = \sum_i \|\mathcal{Y}_i - f_D(f_E(\mathcal{Y}_i))\| \quad (12.38)$$

We don't actually particularly care about the decoder $f_D()$; what we really want is the encoder $f_E()$, which transforms an input to its compressed representation. The decoder is mainly there to give us a network which can be self-supervised. That is, including a decoder leads to a network which we can train in a supervised way, because we know the desired value for the decoder's output layer (which is, of course, to be equal to the encoder's input).

So far this does not quite look like unlabelled clustering. What such a network actually learns depends on a few details; let us consider three variants:

1. Suppose the network is linear; that is, the encoder and decoder are made up of linear discriminant neurons, but *without* any nonlinear activation function. A sequence of layers of linear functions is, in aggregate, still a linear function,⁸ therefore we can write the encoder $f_E()$ and decoder $f_D()$ functions in matrix-vector form:

$$\underline{x} = F_E \mathcal{Y}_{\text{Input}} + \underline{b}_E \quad \mathcal{Y}_{\text{Output}} = F_D \underline{x} + \underline{b}_D \quad (12.39)$$

Ignoring the bias/offset terms \underline{b} , since they are unrelated to the input, and using a mean-squared-error form for the loss \mathcal{L} , means that (12.37) is doing

$$\text{Find } F_E, F_D \text{ to minimize } \text{MSE}(\mathcal{Y}_{\text{Input}} - F_D F_E \mathcal{Y}_{\text{Input}}) \quad (12.40)$$

which is the same as (5.10). That is, we have created a self-supervised network structure to compute the principal components of \mathcal{D} . In other words, if optimally trained, the resulting encoder and decoder networks will contain the eigenvectors corresponding to the largest k eigenvalues of \mathcal{D} .

2. The whole strength of neural networks is, however, to be *non-linear*, since it is nonlinearity which allows networks to be such powerful inference engines.

⁸ See discussion in Appendix D.

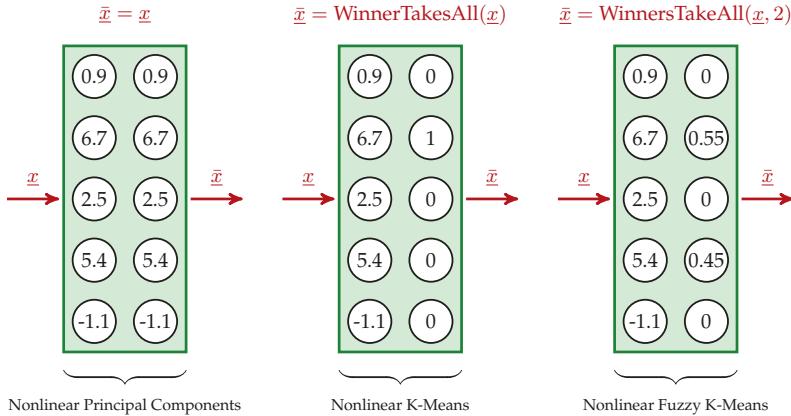


Fig. 12.13. AUTO-ENCODER VARIATIONS: Asserting different behaviours for the “Compressed Representation” layer in the Auto-Encoder of Figure 12.12 can give rise to different unsupervised operations. If the compressed layer is left as-is (left), we produce nonlinear principal components. In contrast, if only the largest of the k values in \underline{x} leads to a non-zero output in $\underline{\underline{x}}$, this forces the network to choose between one of k prototypes, essentially a nonlinear version of K-Means. However there is no rule that only one output neuron can be enabled; by allowing, say, two neurons to be activated (right) we allow each input vector to be represented by a mix of two prototypes, essentially a nonlinear fuzzy K-Means.

Given our understanding, just discussed, of the *linear* version of the auto-encoder in Figure 12.12, it should be relatively simple to imagine, at least conceptually, the behaviour of the generalized *non-linear* auto-encoder. In particular, the nonlinear auto-encoder is still trying to best represent \mathcal{D} in only k degrees of freedom, however rather than being limited to linear functions, it is now attempting to learn the best k nonlinear functions

$$f_E(y) = \begin{bmatrix} f_1(y) \\ f_2(y) \\ \vdots \\ f_k(y) \end{bmatrix} \quad (12.41)$$

in the encoder, with a partnered inversion function $f_D()$ in the decoder, to perform generalized nonlinear dimensionality reduction:

$$\text{Find } f_E(y), f_D(x) \text{ to minimize MSE}\left(y_{\text{Input}} - f_D(f_E(y_{\text{Input}}))\right) \quad (12.42)$$

This is a remarkable result: assuming the auto-encoder network can be successfully trained, we have a general-purpose strategy for inferring optimal nonlinear principal components.

3. The nonlinear auto-encoder is *still* not actually clustering. We need to think more carefully about the difference between principal components and clustering:

- Principal components seeks to represent an input y of m elements via a single vector \underline{x} consisting of k elements.
- Clustering seeks to represent an input y of m elements via *one* of k vectors \underline{z}_κ , each consisting of m elements.

In clustering, the bottleneck is the requirement to choose only *one* of k vectors, however each of those vectors (the prototypes) has m elements. In other words, our compressed vector \underline{x} is *not* the prototype, rather it is a vector of k elements which *selects* among the prototypes. So we need to create some sort of competition among the elements of \underline{x} , such that only the neuron with the largest input can turn on, and the rest are forced off. As is illustrated in [Figure 12.13](#), it is easiest to think about the compressed representation being split into two layers, an input \underline{x} and an output $\bar{\underline{x}}$, with some operation between them:

$$\bar{\underline{x}} = \text{CompetitiveOperation}(\underline{x}) \quad (12.43)$$

such that we can realize a network-based approach to nonlinear K-Means as

$$\bar{\underline{x}} = \text{CompetitiveOperation}(\underline{x}) = \begin{bmatrix} \delta(x_1, \max(\{x_j\})) \\ \vdots \\ \delta(x_k, \max(\{x_j\})) \end{bmatrix} \quad (12.44)$$

Three examples are given in [Figure 12.13](#), however once the basic metaphor is understood, it is easy to imagine other creative variations to give rise to other compressed representations, all self-supervised, based on the underlying encoder-decoder principle.

12.3 Semi-Supervised Learning

Our remaining problem category is how we can improve the classification process by augmenting a limited amount of labelled data with a significantly greater amount of unlabelled data. The rationale or motivation is clear: unlabelled data are much cheaper and easier to acquire than labelled data.

[Figure 12.14](#) gives an overview of the idea: if we can develop a mechanism by which (typically many) unlabelled points can reliably be associated with (typically few) labelled ones, then we can obtain a much more comprehensive classifier.

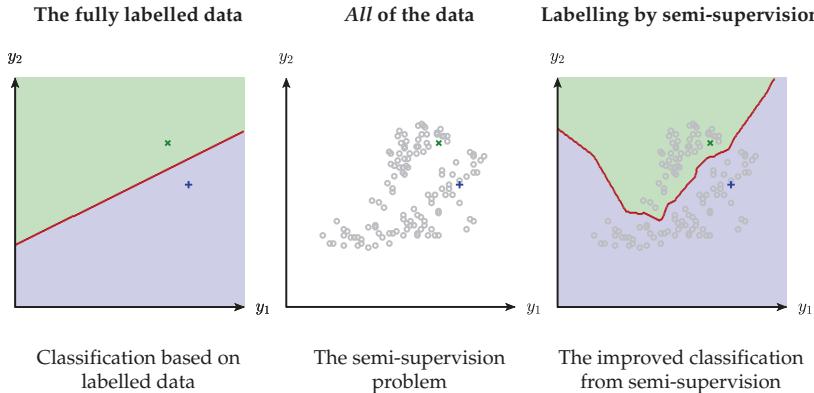


Fig. 12.14. SEMI-SUPERVISION: A semi-supervised problem has some amount of labelled data, plus some additional amount of *unlabelled* data. In the extreme case sketched here, we have only a single labelled data point (left), from which we can infer only a very primitive classifier. However, given additional unlabelled data (middle), if we are able to associate the unlabelled data with the labelled points, then a far more detailed classifier emerges (right).

The methods used for semi-supervised learning are typically related to those for un-supervised learning, except that the semi-supervised problem is typically more forgiving, depending on the ratio of unlabelled to labelled points:

- For a low ratio (many labelled points), the problem is essentially a classification problem, since for any unlabelled point there are labelled points nearby to act as prototypes.
- For a high ratio (few labelled points), the problem is essentially an unlabelled clustering problem. Most unlabelled data will be some distance away from a labelled point, and therefore need to infer clusters based on the spatial patterns in the data.

The labelled data which are present serve to regularize/constrain the labelling process. That is, in clustering unlabelled data, ideally all of the unlabelled points in a class are grouped together, which can be challenging (as we saw in Figure 12.11) if a class is non-compact, geometrically unusual, or close to other classes. In contrast, for semi-supervised learning we only need to cluster the unlabelled data to their nearest labelled point; that is, essentially we have a small, local clustering problem around each labelled data point.

As with any of the approaches to unlabelled clustering in this chapter, semi-supervised learning can only proceed with some sort of assumption regarding the behaviour of points or classes. For example, the assumption that points

from a given sample are closely spaced relative to the spacing between classes, or that all points from a given class tend to be compactly clumped.

Two obvious approaches to semi-supervised learning are based on the clustering methods of this chapter:

K-MEANS VARIANT: Much faster than K-Means, since it is not iterative: just treat every labelled point as a prototype and cluster (classify) all of the unlabelled data to their closest prototype, where closeness is based on an assumed distance $d(\underline{y}, \underline{z})$. Obviously the density of labelled points needs to be sufficiently high such that each prototype needs to represent only a compact local part of a cluster. The problem of [Figure 12.14](#) would not, for example, be well solved this way.

HIERARCHICAL CLUSTERING VARIANT: An inter-cluster distance metric is needed, as before ([page 366](#)). The hierarchical clustering method proceeds as usual, however as soon as a labelled point is incorporated its class label is assigned to that cluster; clusters from different classes are not permitted to merge.

More sophisticated variants of semi-supervised learning have been proposed. In particular, in [Section 10.2](#) we developed the support vector machine in the case of supervised learning, with a goal of finding that optimal decision boundary which maximized the margin (separation) between the two classes in labelled dataset \mathcal{D}_L , as we had formulated in [\(10.32\)](#):

$$\text{Minimize } \mathcal{L}_{\text{Margin}}(\underline{\theta}) = \mathcal{L}\left(\begin{bmatrix} \underline{w} \\ w_o \end{bmatrix}\right) = |\underline{w}| \quad \text{such that} \quad \bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) \geq 1 \quad \text{for all } i \in \mathcal{D}_L \quad (12.45)$$

For semi-supervised data, we can generalize this formulation quite intuitively:

1. All of the labelled data must be on the correct side of the boundary;
2. We wish to maximize the margin between the boundary and *any* data point.

That is, given the labelled \mathcal{D}_L and unlabelled \mathcal{D}_U data, the distance boundary must first correctly separate all data in \mathcal{D}_L , and then never stray close to *any* data point in $\mathcal{D}_L \cup \mathcal{D}_U$, thereby hopefully correctly separating the classes:

$$\text{Minimize } \mathcal{L}_{\text{Margin}}(\theta) = \mathcal{L}\left(\begin{bmatrix} \underline{w} \\ w_o \end{bmatrix}\right) = |\underline{w}| \quad \text{such that}$$

$$\underbrace{\bar{c}_i \cdot h(\underline{x}_i | \underline{w}, w_o) \geq 1 \quad \text{for all } i \in \mathcal{D}_L}_{\text{All labelled data must have adequate margin on correct side of boundary}} \quad \text{and} \quad \underbrace{|h(\underline{x}_i | \underline{w}, w_o)| \geq 1 \quad \text{for all } i \in \mathcal{D}_U}_{\text{All unlabelled data must have adequate margin, on either side of boundary}}$$

(12.46)

The objective in (12.46) was written in hard-margin form, but can be generalized to the soft-margin form of (10.35).

The optimization problem of (12.46) is actually rather difficult to solve, and so would normally be used only if already available as part of a numerical software package.

Case Study 12: Ancient Text Analysis: Who Wrote What?

How do you know what book, story, or text has been written by whom? Whether certain works attributed to Shakespeare, parts of the Bible, or novels written under a pseudonym, there is significant literary interest in knowing whether a claimed authorship is actually true. For recent books the true author frequently emerges, however for books written centuries ago, where the historical record is much less clear, the question of authorship may simply not be adequately documented.

Lest you think this is a niche matter, there are, indeed, entire books written on this very subject ...

W. Leahy (editor), *My Shakespeare: The Authorship Controversy*, Edward Everett Root, 2017

We have had some discussion of pattern recognition applied to written text, in particular the extraction of features from text in Chapter 5 under feature aggregation, and the related example from this chapter in Example 12.2.

If we would like to imagine this as a pattern recognition problem, we begin with two questions:

1. Are there any labelled data you can trust? Is this a clustering or a classification problem?
2. What are the features (Chapter 5)? What information do we extract from the text?

Clearly the classification (supervised) problem is more straightforward, from a pattern-recognition perspective, however it requires training data. That is, supervised learning is possible only if literary historians can definitively declare the authorship of a certain text or, ideally, of *multiple* texts, for more

robust learning. In contrast, the clustering (unsupervised) is typically more difficult to perform well (as we saw throughout this chapter), however it is *very* convenient for a collaborating historian, since *nothing* needs to be assumed about authorship: the expectation is that all of the works from a given author should spontaneously emerge as a single cluster.

Both approaches have been taken in assessing text authorship. The approach in

P. Plecháč, "Relative contributions of Shakespeare and Fletcher in Henry VIII: . . .," *Digital Scholarship in the Humanities*, 2021

is supervised, whereas

A. Arefin, R. Vimieiro, C. Riveros, H. Craig, P. Moscato, "An Information Theoretic Clustering Approach for Unveiling Authorship Affinities in Shakespearean Era Plays and Poems", *Plos One*, 2014

uses graph-based and *k*NN-based clustering, and

D. Kernot, T. Bossomaier, R. Bradbury, "Using Shakespeare's Sotto Voce to Determine True Identity From Text," *Frontiers in Psychology*, 2018

uses hierarchical clustering, as in [Section 12.1.4](#).

The latter question, the features to extract from text, is perhaps more difficult. What is it that actually stays invariant, for a given author, from one text to another? The choice of effective feature may vary from one type of literature to another (poetry vs. biographies), from one language to another (different languages have very different grammatical structures, for example) or whether the work has been translated.

Common choices have been word frequency (essentially the **Bag-of-words approach**), or its generalization to *n*-gram frequencies, the frequency with *n*-word sequences are found in text, some of which may be unique to an author. Other examples could be word lengths, punctuation, or tense.

The Plecháč paper, for example, is based on word frequencies and, interestingly, rhythmic patterns, which syllables in a sequence are **emphasized** or not, giving rise to a binary pattern, such as

"To be or not to be, that is the question" → 010100 10010

The Arefin et al. paper similarly used word frequencies, whereas the Kernot *et al.* paper extracted more abstract features (vocabulary size, use of pronouns).

The text analysis problem is an excellent illustration of the way in which pattern recognition can end up being a helpful research and analysis tool in nearly any domain.

And, if you're interested, the Plecháč paper is fairly certain that Shakespeare did *not*, in fact, author certain parts of *Henry VIII*, but there could always be further validation with additional features.

Lab 12: Clustering

Example 12.1 looked at using K-Means to infer the set of K colours to optimally represent a grid of pixels $\underline{x}_{i,j}$, each a vector of Red-Green-Blue values. What **Example 12.1** did was to purely cluster the colours of all of the pixels in Red-Green-Blue space, without any regard to position.

We can generalize **Example 12.1** very easily to include spatial dependence by including spatial location as part of the feature vector,

$$\underline{x}_{i,j} = \begin{bmatrix} \text{Red} \\ \text{Green} \\ \text{Blue} \\ i \\ j \end{bmatrix} \quad \left. \begin{array}{l} \text{Colour Features} \\ \text{Spatial Features} \end{array} \right\} \quad (12.47)$$

and then also reflected in the pixel-similarity criterion,

$$d^2(\underline{x}, \underline{x}') = (\underline{x} - \underline{x}')^T \begin{bmatrix} \beta_C^2 & & & \\ & \beta_C^2 & & \\ & & \beta_C^2 & \\ & & & \beta_S^2 \\ & & & & \beta_S^2 \end{bmatrix} (\underline{x} - \underline{x}') \quad (12.48)$$

where constants β_C, β_S weight the colour and spatial components, respectively. We begin by setting up the problem:

load Painting % image is in variable 'im'

```
Sx = size(im,2); Sy = size(im,1);
Kx = 15; Ky = 15; K = Kx*Ky;
```

% define spatial locations

```
xrange = 1:Sx; yrangle = 1:Sy;
[x,y] = meshgrid(xrange,yrangle);
```

% add spatial locations to image


Painting Credit: Anya Fieguth

% create data points, Matlab expects one point per row

```
pts = double(reshape(ims, size(ims,1)*size(ims,2), 5 ));
```

% allow algorithm to run for 200 iterations

```
opts = statset( 'MaxIter', 200 );
```

Clearly β_C, β_S in (12.48) need to reflect the units in which the colours and spatial locations are represented. Since some K-Means

implementations (strangely) do not support a scaled distance metric, we will pre-scale our feature vector

$$\underline{\bar{x}}_{i,j} = \begin{bmatrix} \beta_C \cdot \text{Red} \\ \beta_C \cdot \text{Green} \\ \beta_C \cdot \text{Blue} \\ \beta_S \cdot i \\ \beta_S \cdot j \end{bmatrix} \quad (12.49)$$

such that the Euclidean distance can be applied:

$$d^2(\underline{x}, \underline{x}') = d_E^2(\underline{\bar{x}}, \underline{\bar{x}'}) \quad (12.50)$$

Since it is only the *relative* values of β_C and β_S that matter, we will set $\beta_C = 1$ and vary β_S :

`% increase beta_S, the pixel spacing weight relative to colour
pts(:,4:5) = pts(:,4:5)*4;`

So we have all of the pixels, with their colours and spatial locations, listed row-by-row in variable *pts*. In principle we could just call K-Means, and let it randomly choose the starting prototypes, however for clarity we will do that explicitly:

`% random starting points
zinit = pts(randperm(size(pts,1),K),:);
[i,z] = kmeans(pts,[],'start', zinit, 'options', opts);`

Variable *z* contains the prototypes, and *i* is the resulting class for each given unlabelled data point (i.e., each pixel). We would like to visualize the result, somehow,

`image(kmeans_image(im, i, z, 1));`

by creating function *kmeans_image* which converts the K-Means result back into an image which can be plotted and visualized. The function copies the selected prototype into each pixel location, and then optionally draws region borders in red:

```
function im_k = kmeans_image( im, km_i, km_m, showbdy )
    i = reshape(i, size(im,1), size(im,2) );

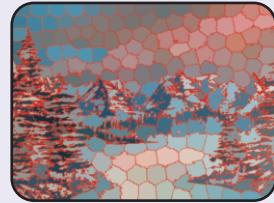
    % insert K-means prototype at every pixel location
    im_k = im;
    for x=1:size(im,2),
        for y=1:size(im,1),
            im_k(y,x,1:3) = z(i(y,x),1:3);
        end
    end
```

```

if (nargin>3),
    % colour boundaries between K-means patches as red
    for x=1:size(im,2),
        q = find(abs(i(2:end,x)-i(1:end-1,x))>0);
        im_K(q,x,1) = 255; im_K(q,x,2) = 0; im_K(q,x,3) = 0;
    end
    for y=1:size(im,1),
        q = find(abs(i(y,2:end)-i(y,1:end-1))>0);
        im_K(y,q,1) = 255; im_K(y,q,2) = 0; im_K(y,q,3) = 0;
    end
end
end

```

The resulting image is shown at right. The results are quite reasonable: in areas with limited colour variations (top, bottom) it is the spatial constraint which limits the region size, creating roughly evenly-sized patches, whereas in areas with significant colour variations (left, right, centre) the K-Means patch boundaries are much more irregular in shape, since they are largely driven by colour.



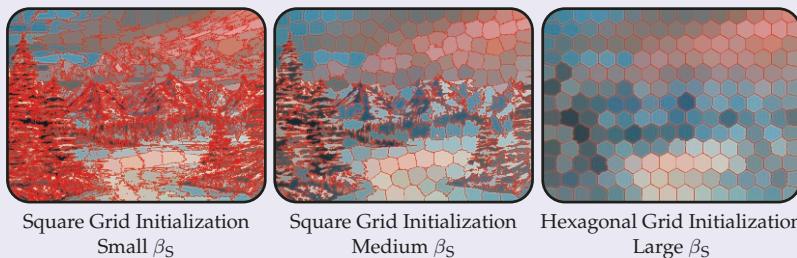
A random starting point will require more K-Means iterations to converge, since we might have some regions with a higher concentration of starting points, and another region with a lower concentration, which would take time (iterations) to even out. We could deliberately initialize the prototypes as evenly spaced, either on a square or hexagonal grid:

```

% set initial prototypes to a square grid
j=1; zinit = zeros(2*Kx*Ky,5);
for x=round(Sx/(2*Kx):(Sx/Kx):Sx),
    for y=round(Sy/(2*Ky):(Sy/Ky):Sy),
        zinit(j,:) = pts(y+(x-1)*size(im,1),:);
        j = j + 1;
    end
end
% created (j-1) prototypes; prune back zinit to keep only these
zinit = zinit(1:(j-1),:);

% show results
[ i, z ] = kmeans(pts,[],'start', zinit , 'options', opts);
image( kmeans_image( im, i, z, 1 ) );

```



The three panels illustrate the behaviour of the code for different values of β_S , to visualize the impact of the tradeoff between space and colour in the distance metric, showing a nice transition between mostly ignoring space (left) to mostly ignoring colour (right). Essentially what we have implemented here is known as *SuperPixel* image segmentation, a standard technique in image processing.

Further Reading

The [references](#) may be found at the end of each chapter. Also note that the [textbook further reading page](#) maintains updated references and links.

Wikipedia Links — Supervision: [Supervised Learning](#), [Semi-supervised Learning](#), [Reinforcement Learning](#), [Unsupervised Learning](#)

Wikipedia Links — Clustering Methods: [Cluster Analysis](#), [Hierarchical Clustering](#), [K-Means Clustering](#), [K-Medians Clustering](#), [K-Medoids](#), [Fuzzy Clustering](#), [Mixture Model](#), [Vector Quantization](#)

Two texts dedicated to the subject of unsupervised clustering are [3] and [2]. The reader may wish to consider survey papers on unsupervised clustering [1, 5], kernel methods [4], and semi-supervised clustering [7].

Sample Problems

Problem 12.1: Short Answer

Give a short definition of each of the following:

- (a) Unsupervised Learning
- (b) Semi-Supervised Learning

- (c) Dendrogram
- (d) Kernel
- (e) Class prototype
- (f) Outlier

Problem 12.2: Short Answer

Offer brief answers to each of the following:

- (a) What are a few pros and cons regarding nonparametric and parametric approaches to clustering?
- (b) For a given set of data points, for each of the following algorithms, are the same clusters produced each time the method is run? That is, is the method deterministic or not?
 - K-Means
 - Kernel K-Means
 - Mean-Shift
 - Hierarchical Clustering
 - Neural-Network based Clustering

What are some advantages and disadvantages for a clustering algorithm to produce the same results each time it is run, as opposed to not?

- (c) What role does a kernel play in Kernel K-Means? What aspects of the problem might influence our choice of kernel?

Problem 12.3: Conceptual — Clustering Assumptions

On page 350 it was suggested that feature directions could be selected from eigenvectors, based on the total covariance over all data points, since the directions of greatest spread (the eigenvectors associated with the largest eigenvalues) should presumably be those directions in which separate clusters are spaced. It was then claimed, however, that one could construct examples in which this cluster spacing assumption would not hold.

Develop two or three examples, in two- or three-dimensional space, whereby one eigenvector (in 2D) or two eigenvectors (in 3D) fail to meaningfully capture the directions in which the clusters are separated.

Problem 12.4: Conceptual — Clustering

The beginning of Section 12.1 suggested that the notion of “cluster” might depend on the choice of a similarity criterion, such that a cluster could be based on points which are closely spaced (a distance criterion) or, instead, based on regions in feature space containing a high density of sample points (a density criterion).

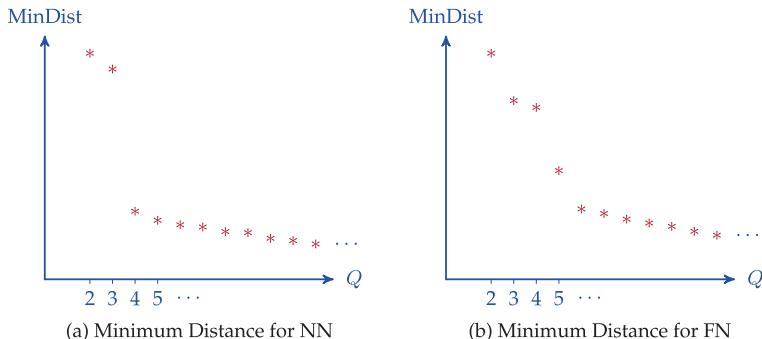


Fig. 12.15. Two examples of minimum-distance functions are given, as defined in (12.35). The goal of Problem 12.6 is to find clusters/data points which match the given distance behaviour.

Superficially, the distance and density criteria might seem like the same thing.

- (a) Thinking about these criteria more carefully, how are the distance and density criteria different?
 - (b) Give one or two examples of data points which would be very differently clustered based on distance and density criteria.

Problem 12.5: Conceptual — Hierarchical Clustering

Suppose we wish to use hierarchical clustering and Euclidean distance, but are deciding which prototype to use. For each of the following cases describing prototypes, sketch a group of *three* clusters which satisfies the description:

- (a) NN, FN, and Mean all produce the same result.
 - (b) NN works well but FN doesn't.
 - (c) FN works well but NN doesn't.
 - (d) Neither NN nor FN work very well.

Problem 12.6: Conceptual — Minimum Distance Interpretation

Recall from (12.35) that MinDist_Q describes the minimum distance when aggregation has reduced a clustering problem to Q clusters.

Given the two plots in Figure 12.15,

- (a) Sketch a set of points consistent with the given $\text{MinDist}_Q^{(NN)}$
 (b) Sketch a set of points consistent with the given $\text{MinDist}_Q^{(FN)}$

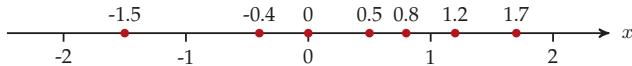


Fig. 12.16. The one-dimensional set of points for Problem 12.7.

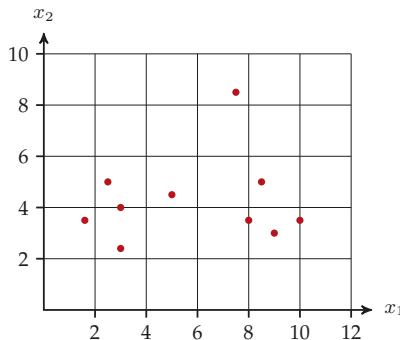


Fig. 12.17. A set of points in two dimensions for Problem 12.8.

Problem 12.7: Conceptual — Dendograms

Consider the one dimensional problem and points sketched in Figure 12.16.

Using the Euclidean distance, draw the dendrogram that hierarchical clustering would produce for these seven points for each of the following three prototype choices:

- (a) Mean
- (b) Furthest neighbour
- (c) 2-Nearest neighbour

Problem 12.8: Conceptual — Clustering

Given the ten data points plotted in Figure 12.17,

- (a) Using two random points as starting prototypes, apply K-means for $K = 2$.
- (b) Using three random points as starting prototypes, apply K-means for $K = 3$.
- (c) Apply NN hierarchical clustering. What two clusters does it find?
- (d) Apply FN hierarchical clustering. What two clusters does it find?

Problem 12.9: Numeric/Computational — Clustering

One of the limitations of K-Means is that a given data point is forced to be associated with a single prototype, even though some other prototype might lie at a similar (only slightly larger) distance. It is therefore perhaps more realistic to consider each data point $\{\underline{x}_i\}$ as having a *degree* of association with *each* prototype z_k , rather than the binary 0/1 association of

K-Means. Such an approach is variously known as Fuzzy Clustering, Soft K-Means, or Fuzzy c-Means.

The basic approach is conceptually the same as K-Means:

- Select a set of K prototypes
- Then, iteratively ...
 1. Cluster the data points given the prototypes
 2. Re-learn the prototypes given the clustering
- ... until converged or a maximal iteration count is reached.

So the pieces which need to be updated are the two numbered steps.

Let $w_\kappa(i)$ be the degree of association between datum \underline{x}_i and prototype \underline{z}_κ , clearly dependent on the distance $d(\underline{x}_i, \underline{z}_\kappa)$ between them and the distances $d(\underline{x}_i, \underline{z}_r)$ to other prototypes:

$$w_\kappa(i) = \frac{1}{\sum_{r=1}^K \left(\frac{d(\underline{x}_i, \underline{z}_\kappa)}{d(\underline{x}_i, \underline{z}_r)} \right)^{\frac{2}{\rho-1}}} \quad (12.51)$$

Here parameter ρ controls the rate at which the weight decays with distance. As $\rho \rightarrow 1$ the weight decays infinitely quickly, meaning that only the closest prototype matters, and the algorithm becomes the same as the usual K-Means. As ρ is increased the weight dependence becomes flatter and the clusters more fuzzy. A common choice is $\rho = 2$.

Given the weights, the updated prototypes are then calculated as a weighted and normalized sum over all data points:

$$\underline{z}_\kappa = \frac{\sum_i \underline{x}_i w_\kappa(i)^\rho}{\sum_i w_\kappa(i)^\rho} \quad (12.52)$$

Implement this Fuzzy K-Means method and compare its behaviour to that of regular K-Means, perhaps for the clustering arrangement of [Problem 12.10](#).

Problem 12.10: Numeric/Computational — Clustering

Let us suppose that we have K unit-variance clusters in two dimensions, equally spaced on a ring of radius r :

$$C_\kappa \sim \mathcal{N} \left(\begin{bmatrix} r \cos(\kappa 2\pi/K) \\ r \sin(\kappa 2\pi/K) \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \quad 1 \leq \kappa \leq K \quad (12.53)$$

That is, the clusters are parameterized in terms of K and r . Develop code to produce 200 data points per cluster.

- (a) *K-Means:* Apply K-Means to your data. How will we assess the results? Each cluster may not have an obvious K-Means prototype associated with it?

K-Means will produce a label for each data point. For each cluster, of the 200 data points from that cluster, see which label occurs most frequently, and consider that label “correct,” which will then allow you to report the fraction of points correctly labelled.

Plot the fraction of correctly labelled points as a function of r and K . What do you observe?

- (b) *Hierarchical Clustering:* Apply Hierarchical Clustering with a Euclidean-NN metric. Iterate the clustering method until K clusters remain — what do you observe?

In all likelihood, at the point where K clusters remain, some of the actual clusters will have been combined, since isolated outlying points will be treated as separate clusters. We then have two possibilities:

1. We could continue to use the Euclidean-NN metric, however to have K large clusters separated we might expect to have to stop the iteration with $Q > K$ clusters remaining.
2. A metric other than Euclidean-NN could be applied. Try an alternative and see to what extent this changes the results.

For both of the above cases, plot MinDist_Q , as a function of K and r . As a function of the chosen distance metric, how widely do the clusters need to be separated in order for the MinDist behaviour to reveal the correct choice of K ?

Problem 12.11: Real World, Open-Ended — Semi-Supervision

There are many quite compelling semi-supervised problems. For example, consider

- Fake email (spam) detection
- Fake news detection
- Website ad detection and blocking

In all three cases the user wishes a program (email reader/web browser) to automatically distinguish between meaningful content (researched news, personal emails) from undesirable content (spam, ads, fraudulent news).

In principle, these seem like classification/hypothesis testing problems. What is it that makes these semi-supervised?

1. The user does not wish to tag too many emails/articles (therefore limited supervision).
2. The developers of fraudulent content will constantly work to overcome any classification system. Therefore any fixed classifier slowly becomes less capable over time, meaning that older labelled data slowly becomes less relevant, again reducing the amount of supervised data available for training.

Select one of these topics (or a related one, having similar attributes) and provide a brief review of the problem and the current methods being used to try to address it.

Since very few data are labelled, what approaches are taken? It is easy to imagine approaches based on semi-supervision (as in this chapter), or based on data augmentation (from [Chapter 9](#)), or just learning a classifier ([Chapters 6 through 11](#)) directly from the limited data?

References

1. X. Liu, F. Zhang et al., Self-supervised Learning: Generative or Contrastive, *IEEE Transactions on Knowledge and Data Engineering*, 2021
2. C. Aggarwal, C. Reddy, *Data Clustering: Algorithms and Applications* (Chapman and Hall, London, 2013)
3. C. Bouveyron, G. Celeux, T. Murphy, A. Raftery, *Model-Based Clustering and Classification for Data Science* (Cambridge University Press, Cambridge, 2019)
4. C. Campbell, Kernel methods: a survey of current techniques. *Neurocomputing* **48**, 1–4 (2002)
5. A. Fahad, N. Alshatri, Z. Tari, et al., A survey of clustering algorithms for big data: taxonomy and empirical analysis. *IEEE Trans. Emerg. Topics Comput.* **2**, 267–279 (2014)
6. P. Fieguth, *An Introduction to Complex Systems* (Springer, Berlin, 2021)
7. J. van Engelen, H. Hoos, A survey on semi-supervised learning. *Mach. Learn.* **109**, 373–440 (2020)



Conclusions and Directions

It is hoped that the text has provided the reader with a broad arc of the pattern recognition problem:

- **Learning and Inference:** How do we learn and then assess that learning — [Chapters 3, 7, and 9](#)
- **Problem definition:** What problem are we trying to solve — [Chapters 4 and 5](#)
- **Fundamentals of Classification:** The basic tools at our disposal — [Chapters 6, 8, and 10](#)
- **Classification Generalization:** The creative use of basic tools — [Chapters 11 and 12](#)

The text has tried to emphasize insight and depth of understanding throughout, and there really are a great many details to learn, as in almost any field, in order to progress from being a *user* of tools to being a *creator* of new tools. However if one had to select a few key concepts that were central to becoming a sophisticated pattern recognition practitioner, one might start with

1. Wrappers ([Chapter 3](#)): Fundamental to layering one method (classifier, inference, validation) on top of another
2. Classifier generalization ([Chapter 11](#)): The ability to treat a given classifier as a tool which can be built upon
3. Testing and Validation ([Chapter 9](#)): The knowledge of what it means to objectively test a proposed idea

That is, as long as we have systematic and legitimate methods of validation (i.e., not accidentally overfitting or mixing training and testing data), we can be quite creative in adapting the conceptual elements encountered in this text. A few examples:

- One could use a deep neural network (Section 11.4), but only keeping the first part of the network, which does feature extraction, then followed by a nonlinear SVM to perform classification.
- One could perform hierarchical clustering (Section 12.1.4), but with a different distance metric, perhaps a variation on (12.33), to introduce some understanding of the statistics of the outliers in the dataset.
- One could perform clustering, using K-Means, along the lines that we already saw in Example 12.1 and Lab 12, in which we would like to cluster image pixels only on the basis of their colour, and not brightness. This could be accomplished by transforming the image into a different colour space (essentially feature extraction), or by applying the angular (magnitude-invariant) distance function d_A of Figure 6.3.

One admitted limitation is that the discussion of this text has focused exclusively¹ on statistical pattern recognition, characterized by the familiar metaphor which we have seen repeatedly:

$$\underbrace{y \in \mathbb{R}^m, y \sim p(y)}_{\text{Measurement}} \xrightarrow{F \text{ or } f()} \underbrace{x \in \mathbb{R}^n, x \sim p(x)}_{\text{Feature}} \xrightarrow{h() \text{ or } g()} \underbrace{c \in \mathcal{C}}_{\text{Class}} \quad (13.1)$$

for linear/non-linear feature extraction $F / f()$, and linear-discriminant/general classification $h() / g()$.

That the text focused on this structure is not so terribly limiting, in that most pattern recognition problems *do* fall into it, including essentially any classification problem starting with a fixed number, m , of measurements. Most of deep neural networks also fall into the form of (13.1), with the first part of the network performing nonlinear feature extraction $f()$, and the second part of the network performing classification $g()$.

However there *are* domains of pattern recognition outside of the domain of statistical pattern recognition (13.1); in particular, any problem in which a measurement or instance is not easily represented as a fixed-length vector $y \in \mathbb{R}^m$.

¹ Perhaps with the rare exception of bag-of-words methods from Chapter 5, Example 12.2, and Case Study 12.

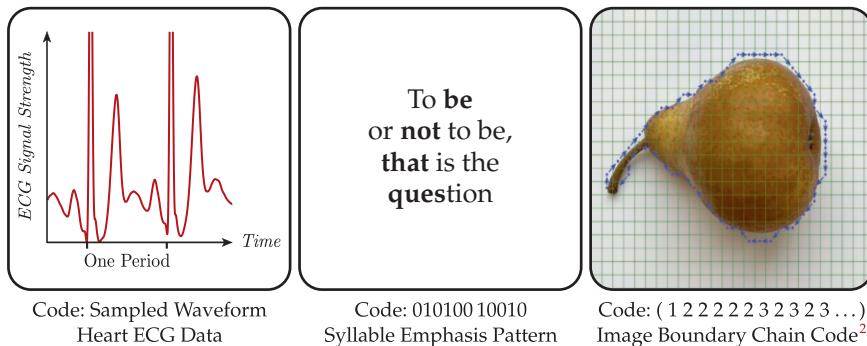


Fig. 13.1. STRUCTURAL PATTERN RECOGNITION: Each of the three problems listed has a relatively simple definition (sampled waveform, sequence of syllables, sequence of directions), however all are variable length, since a change in pulse rate (left), sentence length (middle), or shape complexity (right) will change the length of the corresponding code. We therefore require either a pattern recognition approach which can accept variable-length inputs, or the code must be projected/embedded into a constant-length vector.

Structural Pattern Recognition

There are a great many phenomena which do not obviously or easily allow themselves to be represented as a fixed-length vector:

- Variable length strings, such as any piece of writing, from a short poem to a PhD thesis.
- Images of different sizes, or images containing varying numbers of objects, or varying numbers of features.
- Almost any video in which people, animals, or objects enter or leave the scene over time.
- Graphs or trees, describing inter-relationships, whether of people and objects in a video or an organizational structure.
- Almost any category of varying detail or complexity, such as music (some pieces are short, some are long) or a patient's medical history in a hospital (where symptoms and medications will vary significantly from one person to another).

² For any reader who is interested, starting at the stem of the pear, the complete chain code used in the figure is (1 2 2 2 2 2 3 2 3 2 2 3 3 3 3 4 4 4 5 4 5 5 5 6 5 6 7 6 7 7 8 7 8 8 8 7 8 8 7 6 5 6). Digit "1" points upwards, with each successive value 45° further clockwise.

We then have two basic alternatives:

1. *Vector Embedding*, whereby creative approaches need to be developed to meaningfully convert a problem into a fixed-length real vector, as had been mentioned in (5.45):

$$\begin{array}{c} \text{Irregular, Unordered} \\ \text{Variable-Length, Variable-Context} \\ \text{Data} \end{array} \xrightarrow{\text{Vector Embedding}} \underline{x} \in \mathbb{R}^n \quad (13.2)$$

2. *Non-Embedded Approaches*, the domain of Structural Pattern Recognition, developing classifiers which work directly on non-ordered/non-real/non-fixed-length data. The one such approach which we have seen were the Decision Trees/Random Forests of Section 11.3.

Three examples of structural pattern recognition problems are shown in Figure 13.1. Clearly we could propose *ad-hoc* ways of projecting or embedding each of the codes of Figure 13.1 into a vector of fixed length. For example, for the ECG heartbeat data we could sample each period of the waveform with samples spaced in time by 1% of the period, guaranteeing that each period is represented by a vector of 101 samples. Given that the embedding is, essentially, a kind of feature extraction (Chapter 5), whether the proposed embedding is effective or not ultimately depends on whether the resulting fixed-length feature vector is effective at addressing the classification problem at hand.

The process of embedding a variable-sized problem into a fixed-length vector is a popular approach, since once we have a fixed-length feature vector, all of the distance/statistical/discriminant/ensemble/neural methods of this text immediately apply. However there has been much research undertaken on directly representing more abstract structures, such as strings, trees, or graphs, typically building up complex patterns from simpler elements. In any event, classification still boils down to some sort of similarity comparison, such that a given sample pattern is classified to its most similar class, however clearly the notion of distance metric then needs to be generalized significantly from those which were developed for fixed-length vectors in Chapter 6.

Syntactic Pattern Recognition

An exceptionally important special case of structural pattern recognition is that of *Natural Language* methods, central to any pattern recognition system that is meant to communicate with people in some interactive way, or which is meant to analyze human communications. Such systems need to have some capacity to work with words, grammar, or language.

Many problems fall into this broad category, including translation, grammar correction, or open-ended question answering, all of which would have seemed like a fantasy not so many years ago, but which are now routinely taken for granted given the ubiquity of online translation via *Google Translate*, grammar correction in *Microsoft Word*, and “conversations” with *Alexa* or *Siri*.

Early work in this field focused on formal grammars, coming out of formal language theory in linguistics. A grammar describes a set of rules, describing how a root object S (sentence) can be decomposed into other objects (noun or verb, say) and finally into a string of elements (words) which can be decomposed no further. A very simple set of grammar rules

$$S \rightarrow aS \quad S \rightarrow Sb \quad S \rightarrow c \quad (13.3)$$

can give rise to strings like

$$S \rightarrow \{c, ac, cb, acb, aac, aacb, \dots\} \quad (13.4)$$

and the overall language of this particular grammar is the infinite set

$$\{a^n cb^m \mid m \geq 0, n \geq 0\} \quad (13.5)$$

Grammars are conceptually appealing, in that relatively simple rules can give rise to fairly complex behaviour, and different categories of grammars have been defined (context sensitive, context free, regular) which constrain the permitted rules.

In principle the applicability of such grammars is quite broad, in that terminal symbols (abc) could represent letters, words, or people. Even more general, a grammar does not need to be limited to creating one-dimensional strings, such that grammar rules could be created that attach symbols in multiple dimensions or even in combining graph structures.

One significant challenge is that grammars are quite difficult to infer, and many grammar learning strategies end up learning grammars with far too many rules, in the limit learning (overfitting) one rule per instance,

$$S \rightarrow a \quad S \rightarrow b \quad \dots \quad S \rightarrow z \quad (13.6)$$

a result which is not at all helpful since there is no actual grammatical structure, just a list of all possible outcomes.

Most natural language processing has moved into the domain of deep neural networks of [Section 11.4](#). The network, as a universal black box, is able to undertake (in principle) any kind of learning, so the challenge is primarily how to transform a natural language input into a fixed-length vector to present to the network input, what is known as [Word Embedding](#), such as the currently popular *Word2vec* or Bag-of-Words, which we saw in [Chapter 5](#) under [feature aggregation](#) and in [Case Study 12](#).

Appendices

A

Algebra Review

All features or measurements associated with a pattern are just a collection of numbers, which we will organize as a column vector,

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} \quad \underline{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} \quad (\text{A.1})$$

Although we may talk about these vectors as living in a high-dimensional space, $\underline{x} \in \mathbb{R}^n$ or $\underline{y} \in \mathbb{R}^m$, really there is nothing terribly complicated about having stacked a set of numbers together. Having a single vector \underline{x} is essentially just a compact shorthand for referring to all of the individual elements in the vector.

Given that we have features and measurements represented as vectors, the most basic mathematical transformations on vectors are accomplished by matrices,

$$\underline{x} = A\underline{y} \quad (\text{A.2})$$

which requires us to understand a certain degree of matrix algebra. Matrices are, like vectors, really nothing more than a collection of numbers, a way of grouping together longer mathematical expressions into a simpler representation, such that (A.2) is much simpler than a long sequence of equations like

$$x_1 = \sum_j A_{1j}y_j \quad \dots \quad x_n = \sum_j A_{nj}y_j. \quad (\text{A.3})$$

The representations comparing (A.2) and (A.3) make sense only if we understand the principle of matrix–vector and matrix–matrix multiplication, in

which we take dot products of rows on the left with columns on the right, thus a matrix-vector product

$$\begin{bmatrix} 4 & 2 \\ 2 & -3 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 4 \cdot 1 + 2 \cdot -1 \\ 2 \cdot 1 + -3 \cdot -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix} \quad (\text{A.4})$$

or a matrix-matrix product

$$\begin{bmatrix} 3 & 1 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ -2 & 4 \end{bmatrix} = \begin{bmatrix} 3 \cdot -1 + 1 \cdot -2 & 3 \cdot 1 + 1 \cdot 4 \\ 0 \cdot -1 + -2 \cdot -2 & 0 \cdot 1 + -2 \cdot 4 \end{bmatrix} = \begin{bmatrix} -5 & 7 \\ 4 & -8 \end{bmatrix} \quad (\text{A.5})$$

In the general case, we find a matrix-vector product as

$$\underline{x} = A \cdot \underline{y} \quad \longrightarrow \quad x_i = \sum_r A_{i,r} \cdot y_r \quad (\text{A.6})$$

or the matrix-matrix product as

$$C = A \cdot B \quad \longrightarrow \quad C_{i,j} = \sum_r A_{i,r} \cdot B_{r,j} \quad (\text{A.7})$$

For the summations in (A.6) and (A.7) to be valid, the index r must sum over the same number of elements in A and \underline{y} in (A.6) or A and B in (A.7), meaning that in any product, the number of columns in the left element (A) must equal the number of rows in the right element (\underline{y} or B). Therefore in general matrix and vector dimensions must obey

$$\begin{array}{c} k \\ m \end{array} \cdot \begin{array}{c} m \\ n \end{array} = \begin{array}{c} k \\ n \end{array} \quad (\text{A.8})$$

for matrix dimension parameters k, m, n . As a result, if we can multiply $A \cdot B$ we may *not* be able to multiply $B \cdot A$; furthermore even if both products are valid, in general

$$A \cdot B \neq B \cdot A. \quad (\text{A.9})$$

Figure A.1 illustrates the most common, basic forms for a matrix A :

- A could be an $n \times n$ square matrix;
- A could be an $m \times n$ rectangular matrix;
- A could be dense, meaning that most matrix entries are non-zero;
- A could be sparse, in that only few entries are non-zero;
- A could be diagonal, such that all matrix elements are zero, except the elements A_{ii} along the diagonal;

Square	Rectangular	Diagonal	Identity	Symmetric
$\begin{bmatrix} 4 & 4 & 1 \\ 2 & 1 & 7 \\ 0 & 2 & 4 \end{bmatrix}$	$\begin{bmatrix} 3 & 1 & 1 \\ 4 & 2 & 5 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 8 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 2 & 1 & 3 \\ 1 & 5 & 7 \\ 3 & 7 & 9 \end{bmatrix}$

Fig. A.1. Five very basic matrix types, in cartoon form (top) and simple examples (bottom).

$$\begin{array}{lll} \frac{\partial}{\partial \underline{x}} \underline{x} = I & \frac{\partial}{\partial \underline{x}} (\underline{a}^T \underline{x}) = \underline{a}^T & \text{Constant vector } \underline{a} \\ \frac{\partial}{\partial \underline{x}} (\underline{x}^T \underline{x}) = 2\underline{x}^T & \frac{\partial}{\partial \underline{x}} (\underline{x}^T A \underline{x}) = \underline{x}^T (A + A^T) & \text{Constant matrix } A \\ \frac{\partial}{\partial X} \det(X^T S X) = 2 \det(X^T S X) (X^T S X)^{-1} X^T S & & \text{Symmetric matrix } S \end{array}$$

Table A.1. A few vector and matrix derivatives, in so-called *Numerator Layout* form. These derivatives will be of use in maximizing or minimizing algebraic expressions.

where $A_{i,j}$ refers¹ to the scalar element in row i and column j of matrix A .

Matrix addition and subtraction are simpler than multiplication, and proceed element by element:

$$C = \alpha A + \beta B \quad \longrightarrow \quad C_{i,j} = \alpha A_{i,j} + \beta B_{i,j} \quad (\text{A.10})$$

It is possible to compute derivatives of expressions involving vectors and matrices, with the derivative taken with respect to a scalar, a vector, or a matrix. Since vectors and matrices are just collections of scalars, derivatives involving vectors and matrices are, similarly, really just collections of individual scalar derivatives:

$$\frac{\partial f(x)}{\partial x} = \frac{\partial}{\partial x} \begin{bmatrix} f_1(x) \\ \vdots \\ f_q(x) \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x} \\ \vdots \\ \frac{\partial f_q(x)}{\partial x} \end{bmatrix} \quad (\text{A.11})$$

The primary notational ambiguity appears in taking the derivative of a vector function with respect to a vector,

$$Q = \frac{\partial f(x)}{\partial \underline{x}} \quad \leftarrow \quad \begin{array}{l} \text{Column vector } f \\ \text{Column vector } \underline{x} \end{array} \quad (\text{A.12})$$

¹ The mathematical notation used in this text is summarized on page xix.

The derivative Q must be a matrix, since both \underline{f} and \underline{x} are vectors, each indexing separately over a sequence of elements. The issue, however, is that \underline{f} and \underline{x} are both *column* vectors, whereas Q is indexed over rows and columns. That is, we need to decide (somewhat arbitrarily) whether the rows of Q index the rows of \underline{f} or the rows of \underline{x} :

$$\underbrace{Q_{\textcolor{brown}{i},j} = \frac{\partial f_i(\underline{x})}{\partial x_j}}_{\text{Numerator Layout}} \quad \underbrace{Q_{j,\textcolor{brown}{i}} = \frac{\partial f_i(\underline{x})}{\partial x_j}}_{\text{Denominator Layout}} \quad (\text{A.13})$$

In this text, in the few instances where the vector-matrix derivatives of [Table A.1](#) are needed, the Numerator Layout form will be used, such that the rows of Q index the rows of \underline{f} .

Given a matrix A , one of the most fundamental operations on A finding its *transpose*, essentially a mirror image,

$$\text{Transpose}(A) = A^T \quad \longrightarrow \quad A_{i,j}^T = A_{j,i} \quad (\text{A.14})$$

such that a matrix is said to be *symmetric* if $A^T = A$. Matrix transposition obeys

$$(A^T)^T = A \quad (A + B)^T = A^T + B^T \quad (A \cdot B)^T = B^T \cdot A^T \quad (\text{A.15})$$

A vector is just a thin (narrow) matrix, so the same transpose rules apply: a column vector \underline{x} can be transposed to a row vector \underline{x}^T , and vice versa. Building on [\(A.8\)](#), it is very important to understand that the location of the transpose matters a great deal,

$$\begin{array}{c} \text{---} \cdot \boxed{|} = \square \\ | \cdot \text{---} = \boxed{\square} \end{array} \quad (\text{A.16})$$

since both of these forms are used repeatedly in the text.

Next, a square matrix A is said to have an inverse A^{-1} if

$$A \cdot A^{-1} = A^{-1} \cdot A = I \quad (\text{A.17})$$

Similar to [\(A.15\)](#), matrix inversion inverse obeys

$$(A^{-1})^{-1} = A \quad (A^T)^{-1} = (A^{-1})^T \quad (A \cdot B)^{-1} = B^{-1} \cdot A^{-1} \quad (\text{A.18})$$

The inverse can be found as

$$[a]^{-1} = \frac{1}{a} \quad \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad (\text{A.19})$$

and progressively more complicated for larger matrices. The matrix inverse of A is essentially a reversed transformation, such that

$$\text{If } \underline{x} \xrightarrow{A} \underline{y} \quad \text{then} \quad \underline{y} \xrightarrow{A^{-1}} \underline{x}. \quad (\text{A.20})$$

For many square matrices there is no inverse, and a matrix which cannot be inverted is said to be *singular*. The singularity of a matrix A can be assessed by its *determinant* $\det(A)$:

$$\begin{aligned} A \text{ singular} &\iff \det(A) = 0 \\ A \text{ invertible} &\iff \det(A) \neq 0 \end{aligned} \quad (\text{A.21})$$

where

$$\begin{aligned} \det([a]) &= a & \det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) &= ad - bc \\ \det\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\right) &= a(ei - fh) - b(di - fg) + c(dh - eg) \end{aligned} \quad (\text{A.22})$$

Certainly the analytical form of the determinant can be defined for larger matrices, but it is notationally cluttered and offers little insight. Intuitively, $\det(A)$ sort of measures the overall degree of amplification associated with A as a linear transformation. It therefore stands to reason, from (A.20), that for A^{-1} to represent the inverse transformation of A it must undo the amplification of A , from which it becomes clear that

$$\det(A^{-1}) = \frac{1}{\det(A)} \quad (\text{A.23})$$

If $\det(A) = 0$ then the effect of transformation A is to multiply by zero in some direction; therefore in the same way that we cannot divide by zero, similarly we cannot invert a matrix having a determinant of zero. In general, matrix determinants obey

$$\det(AA) = \det(A)^2 \quad \det(AB) = \det(A)\det(B) \quad \det(A^T) = \det(A) \quad (\text{A.24})$$

$$\det(\alpha A) = \alpha^n \det(A) \quad \text{for } n \times n \text{ matrix } A \quad (\text{A.25})$$

The preceding discussion on matrix invertibility and determinants focused entirely on square matrices, since rectangular (non-square) matrices cannot be invertible² and do not have a determinant.

² There is the related notion of a *pseudo-inverse*, which does define a sort of inverse for rectangular matrices.

Next, the *eigendecomposition* of a square³ matrix Σ is one of the most powerful tools in linear algebra. The eigendecomposition essentially reveals those key vectors, the eigenvectors, which do not change direction when transformed (multiplied) by Σ . In particular, \underline{v} is an eigenvector of Σ if

$$\Sigma \underline{v} = \lambda \underline{v}, \quad (\text{A.26})$$

such that \underline{v} has not changed direction, only length; the amplification factor (the change in length) is captured by the associated *eigenvalue* λ . Formally, the eigenvalues of Σ are found as the roots of the *characteristic polynomial* of Σ :

$$\det(\Sigma - \lambda I) = 0. \quad (\text{A.27})$$

In general, for an $n \times n$ matrix Σ there will be n eigenvalues $\lambda_1, \dots, \lambda_n$, which relate to Σ as

$$\det(\Sigma) = \prod_{j=1}^n \lambda_j \quad \text{trace}(\Sigma) = \sum_{j=1}^n \Sigma_{jj} = \sum_{j=1}^n \lambda_j, \quad (\text{A.28})$$

from which it is clear that any singular matrix Σ , meaning that $\det(\Sigma) = 0$, must be associated with at least one eigenvalue equal to zero.

The other insight from (A.28) is that, intuitively, $\det(\Sigma)$ measures the squared size of the hyper-volume of the hyper-ellipse associated with covariance Σ :

$$\text{Volume of hyper-ellipse} \propto \prod_j (\text{Length of semi-axis})_j = \prod_j \sqrt{\lambda_j} = \sqrt{\det(\Sigma)} \quad (\text{A.29})$$

For a given eigenvalue λ_j , the associated eigenvector \underline{v}_j is found from (A.26) by solving the linear system

$$(\Sigma - I\lambda_j) \underline{v}_j = 0 \quad (\text{A.30})$$

although, as with matrix inversion, in practice we will rely on numerical software to compute eigendecompositions for us.

It is important to understand that the *ordering* of eigenvectors is irrelevant. That is, there is nothing special or unique about which eigenvector is chosen as \underline{v}_1 , which as \underline{v}_2 etc. What is important is that a given eigenvector must always be correctly associated with its partner eigenvalue.

We will only ever be computing eigendecompositions of *covariance matrices*, which must be real symmetric matrices, for which the eigenvalues and

³ There is the related concept of the singular value decomposition (SVD), which is essentially the generalization of eigendecompositions to rectangular matrices, a powerful concept but beyond the scope of this text.

eigenvectors will always be real, and the eigenvectors all orthogonal to one another.

Since it is only the direction of the eigenvector which matters, and not the sign or length, we will normally assume eigenvectors to be normalized to unit length, that is⁴

$$\sum_j v_j^2 = 1. \quad (\text{A.31})$$

It is frequently convenient to collect the eigenvalues/eigenvectors into matrices⁵ as

$$\begin{aligned} \Sigma \underline{v}_j = \lambda_j \underline{v}_j, \quad 1 \leq j \leq n \quad \rightarrow \quad V = \begin{bmatrix} | & | \\ \underline{v}_1 & \cdots & \underline{v}_n \\ | & | \end{bmatrix} \quad \Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \\ \rightarrow \quad \Sigma V = V \Lambda \end{aligned} \quad (\text{A.32})$$

$$(\text{A.33})$$

Since Σ is symmetric its eigenvectors must be orthogonal, thus V will always be invertible, so that from the derivation in Appendix D, (A.33) becomes

$$\Sigma = V \Lambda V^{-1} \quad (\text{A.34})$$

from which it follows that

$$\Sigma^2 = (V \Lambda V^{-1})(V \Lambda V^{-1}) = V \Lambda I \Lambda V^{-1} = V \Lambda^2 V^{-1} \quad (\text{A.35})$$

or, more generally,

$$\Sigma^p = V \Lambda^p V^{-1}. \quad (\text{A.36})$$

The power Λ^p is easily computed because it Λ is a diagonal matrix:

$$\Lambda^p = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}^p = \begin{bmatrix} \lambda_1^p & & \\ & \ddots & \\ & & \lambda_n^p \end{bmatrix} \quad (\text{A.37})$$

and thus Λ^p is just a diagonal matrix of *scalar* powers.

Although not of importance to us in this text, the matrix power of (A.37) allows us to compute unusual things like fractional or complex powers of a matrix,

⁴ Note that we need to be careful with indexing: \underline{v}_j means the j^{th} vector in a sequence, whereas v_j means the j^{th} element of vector \underline{v} .

⁵ Note that blank areas in matrices, as in Λ in (A.32), are understood to be zero. That is, Λ is a diagonal matrix.

$$\Sigma^{2.5} = V \Lambda^{2.5} V^{-1} \quad \Sigma^{1-2i} = V \Lambda^{1-2i} V^{-1} \quad (\text{A.38})$$

or, even more strangely, based on power series, a matrix exponent e^Σ

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} \longrightarrow e^\Sigma = \sum_{k=0}^{\infty} \frac{\Sigma^k}{k!} = V \left(\sum_{k=0}^{\infty} \frac{\Lambda^k}{k!} \right) V^{-1} = V \begin{bmatrix} e^{\lambda_1} & & \\ & \ddots & \\ & & e^{\lambda_n} \end{bmatrix} V^{-1} \quad (\text{A.39})$$

Reversing the direction of (A.34) we have

$$V^{-1} \Sigma V = \Lambda \quad (\text{A.40})$$

the so-called process of diagonalization, transforming matrix Σ to diagonal matrix Λ .

If all of the eigenvalues $\lambda_j \neq 0$, then Λ and consequently Σ are invertible, such that

$$\Sigma^{-1} = V \Lambda^{-1} V^{-1} \quad (\text{A.41})$$

Given that Σ is a covariance matrix (and therefore symmetric) and with the eigenvectors normalized in length, as in (A.31), then (from Appendix D)

$$V^T \cdot V = I \longrightarrow V^T = V^{-1} \quad (\text{A.42})$$

That is, remarkably, the inverse of V is simply its transpose, a special property of what are known as *orthogonal* matrices, in which case

$$\Sigma = V \Lambda V^T \quad \Sigma^{-1} = V \Lambda^{-1} V^T. \quad (\text{A.43})$$

Given the eigendecomposition $\Sigma \underline{v}_j = \lambda_j \underline{v}_j$ of a matrix Σ , we say that

- Σ is positive definite if $\lambda_j > 0$ for all j
- Σ is positive semi-definite if $\lambda_j \geq 0$ for all j

All **covariance** matrices, widely used in pattern recognition, must be positive semi-definite. In most cases covariances will be positive definite, in which case the matrix is invertible.

Further Reading

Wikipedia Links: [Linear Algebra](#), [Matrix](#), [Eigenvalues and Eigenvectors](#), [Matrix Calculus](#)

There are many books on linear algebra, and the interested student will find a wide selection at Library of Congress call number QA251 at any library.

The text by Deisenroth *et al.* [2] offers an overview of mathematics specifically for the machine-learning context. Four further suggestions would include the books by Poole [5], Lay *et al.* [3], Damiano *et al.* [1], or the relatively advanced text by Meyer [4].

Sample Problems

Problem A.1: Short Answer

Give a short definition of each of the following:

- (a) Singular matrix
- (b) Eigenvalue
- (c) Eigenvector
- (d) Matrix diagonalization

Problem A.2: Conceptual — Matrix Multiplication

With reference to (A.8),

- (a) Given an example of matrices A and B such that we can compute AB but cannot compute BA
- (b) Given an example of matrices A and B such that AB and BA both exist, but $AB \neq BA$

Problem A.3: Conceptual — Eigenvalues

- (a) Given a diagonalizable matrix A , show that the trace of A is equal to the sum of its eigenvalues:

$$\text{tr}(A) = \sum_{i=1}^n \lambda_i \quad (\text{A.44})$$

Recall that $\text{tr}(BC) = \text{tr}(CB)$ as long as both products BC, CB are defined. You will definitely want to express A in diagonalized form.

- (b) Given a diagonalizable matrix A , show that the determinant of A is equal to the product of its eigenvalues:

$$\det(A) = \prod_{i=1}^n \lambda_i \quad (\text{A.45})$$

Recall that $\det(BC) = \det(B)\det(C)$ for two $n \times n$ matrices B, C .

Problem A.4: Conceptual — Eigendecompositions

Most of the eigendecompositions in this text are taken of **covariance matrices**, which are always real, symmetric matrices.

Show that for real, symmetric matrix A , if two eigenvalues λ_i, λ_j are different, then their corresponding eigenvectors must be orthogonal. That is, show that

$$A\underline{v}_i = \lambda_i \underline{v}_i \quad A\underline{v}_j = \lambda_j \underline{v}_j \quad \lambda_i \neq \lambda_j \quad \rightarrow \quad \underline{v}_i^T \underline{v}_j = 0 \quad (\text{A.46})$$

It can be hard to know how to even get started. Keep in mind that A is symmetric, so the eigendecomposition of A must be the same as the eigendecomposition of A^T .

Problem A.5: Conceptual — Orthogonal Matrices

A $n \times n$ matrix A is said to be orthogonal if $A^T A = AA^T = I$.

- (a) Show that the determinant $\det(A) = \pm 1$.
- (b) Show that the product of two orthogonal matrices is orthogonal.
- (c) The matrix to rotate a two-dimensional coordinate by angle θ is given as

$$A = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (\text{A.47})$$

Show that any such matrix is orthogonal with $\det(A) = 1$.

(The result can be inverted and generalized to show that any $n \times n$ orthogonal matrix A with $\det(A) = 1$ corresponds to a rotation in \mathbb{R}^n .)

References

1. D. Damiano, J. Little, *A Course in Linear Algebra* (Dover, Mineola, 2011)
2. M. Deisenroth, A. Faisal, C. Ong, *Mathematics for Machine Learning* (Cambridge University Press, Cambridge, 2020)
3. D. Lay, S. Lay, J. McDonald, *Linear Algebra and Its Applications* (Pearson, New York, 2014)
4. C. Meyer, *Matrix Analysis and Applied Linear Algebra* (SIAM, New York, 2000)
5. D. Poole, *Linear Algebra: A Modern Introduction* (Brooks Cole, Long Beach, 2014)

Random Variables and Random Vectors

B.1 Random Variables

In pattern recognition we are typically representing patterns using random vectors; that is, as a vector of random variables. To be sure, a random variable x being “random” does *not* mean that nothing is known about x ; to the contrary, we may very well have detailed information (so-called prior knowledge) about x , such that some values of x are known to be more likely than others.

In pattern recognition, randomness appears primarily because of within-class variability. Except for exceptionally narrow definitions of class, nearly all pattern recognition classes will contain members with some degree of variation from one another: *tigers* have different weights, *adult humans* have different heights, and *ripe bananas* have different lengths. Although the height of a particular person is not “random,” in the sense of there being genetic factors etc. that will have influenced their growth, given thousands of samples in a pattern recognition class, it will be easiest to use the methods of random variables and random vectors in describing the variability which we observe.

The most complete description of a random variable x is given by its probability distribution. If x describes *discrete* events, such as a type of mammal, then the probability distribution is just a list of possible mammals and their respective probabilities. For a somewhat simpler example, if y is the roll of a die, then

$$\text{Die Probability } \mathbf{P}(y) = \begin{cases} 1/6 & y \in \{1, \dots, 6\} \\ 0 & \text{Otherwise} \end{cases} \quad (\text{B.1})$$

Since every random variable must take on *some* value, the probabilities of all possible outcomes must always sum to one.

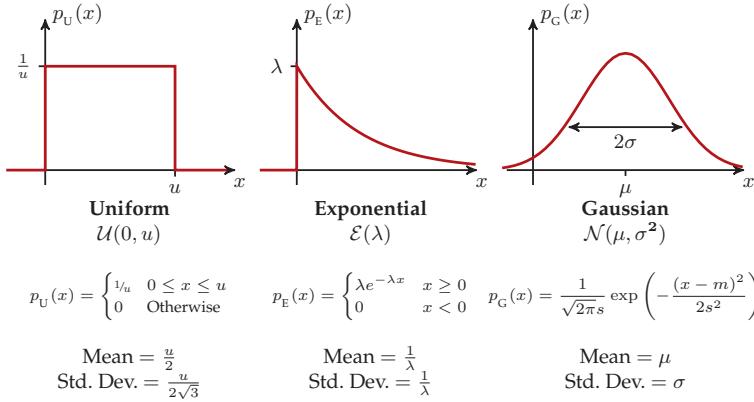


Fig. B.1. PROBABILITY DISTRIBUTION FUNCTIONS: The figure shows mathematical definitions and representative illustrations for three common distributions. In all cases x is a random variable, whereas u, λ, μ, σ are parameters controlling the width (u, λ, s) or location (m) of the distribution.

In contrast, if x describes a *continuous* distribution, such as adult human height, we can no longer talk about the *probability* of x , since the probability of any *exact* event, for example that an adult has a height of *exactly* 171.2483342 cm, is zero. Instead, we now need to talk about the probability that x lies in some *range*, such as

$$\text{Probability that } 170 \text{ cm} < x < 175 \text{ cm} \quad (\text{B.2})$$

Even though the *probability* of some specific value of x is zero, clearly some human heights are more likely than others, which would be described via a probability *distribution* or *density* $p(x)$, such that

$$\text{Probability that } x \text{ lies between } \alpha \text{ and } \beta = \int_{\alpha}^{\beta} p(x) dx \quad (\text{B.3})$$

That is, probability is the *integral* of probability *density* $p()$. A related concept is the cumulative distribution F_x ,

$$F_x(\alpha) = \text{Probability that } x < \alpha = \int_{-\infty}^{\alpha} p(x) dx \quad (\text{B.4})$$

As before, since x must take on some value, it is now the *integral* of the probability distribution which must equal one:

$$\int_{-\infty}^{+\infty} p(x) dx = 1. \quad (\text{B.5})$$

Three of the most common probability distribution functions are illustrated in [Figure B.1](#).

B.2 Expectations

A probability distribution is often an inconvenient description of a random variable, normally for one of two reasons:

1. The equation of the probability distribution for x is much too complicated, or
2. We don't actually know the probability distribution of x .

We will frequently prefer to evaluate some function of a random variable, known as an expectation:

$$\mathbb{E}[f(x)] = \int_{-\infty}^{+\infty} f(x)p(x) dx \quad (\text{B.6})$$

for some specified function $f()$. That is, $\mathbb{E}[f(x)]$ is essentially the average value of $f(x)$, over all possible values of random variable x . For those times where we manipulate expectations mathematically, from (B.6) it is helpful to keep in mind that the rules for expectations are the same as the rules for integration:

$$\mathbb{E}[\alpha f(x)] = \int_{-\infty}^{+\infty} \alpha f(x)p(x) dx = \alpha \int_{-\infty}^{+\infty} f(x)p(x) dx = \alpha \mathbb{E}[f(x)] \quad (\text{B.7})$$

$$\begin{aligned} \mathbb{E}[f(x) + g(x)] &= \int_{-\infty}^{+\infty} (f(x) + g(x))p(x) dx \\ &= \int_{-\infty}^{+\infty} f(x)p(x) dx + \int_{-\infty}^{+\infty} g(x)p(x) dx = \mathbb{E}[f(x)] + \mathbb{E}[g(x)] \end{aligned} \quad (\text{B.8})$$

The expectation allows us to work with notions of random variable behaviour which are much simpler than a probability distribution:

- The *mean* or average is a simple measure of central tendency,

$$\text{Mean}(x) = \mu_x = \mathbb{E}[x] = \int x \cdot p(x) dx \quad (\text{B.9})$$

- The *variance* is a measurement of variability or spread, calculated as the expected squared deviation of random variable x from its mean:

$$\text{Variance}(x) = \sigma_x^2 = \mathbb{E}[(x - \mu_x)^2] = \int (x - \mu_x)^2 \cdot p(x) dx \quad (\text{B.10})$$

Because the variance is in squared units, it is frequently much simpler to think about and plot/visualize the spread of the distribution in terms of the *standard deviation* σ_x , the square-root of the variance, since σ_x is measured in the same units as x and μ_x .

Thus, per our **notation**, it will be much more compact to refer to $x \sim (\mu, \sigma^2)$ to mean that random variable x has mean μ and variance σ^2 . The rules for manipulating means and variances are very simple; given random variable $x \sim (\mu_x, \sigma_x^2)$ then for any constant α ,

$$\text{Mean}(\alpha x) = \alpha \mu_x \quad \text{Variance}(\alpha x) = \alpha^2 \sigma_x^2 \quad (\text{B.11})$$

The mean and variance are shown for a few distributions in [Figure B.1](#).

Other notions of distribution, also computed as expectations, are possible, such as the higher-order standardized¹ moments

$$\text{Skewness}(x) = \mathbb{E} \left[\left(\frac{x - \mu_x}{\sigma_x} \right)^3 \right] \quad \text{Kurtosis}(x) = \mathbb{E} \left[\left(\frac{x - \mu_x}{\sigma_x} \right)^4 \right] \quad (\text{B.12})$$

where skewness assesses asymmetry in the distribution, whether the distribution has a long extension or tail on only one side, for example, and kurtosis assesses the heaviness of the tails of the distribution, how much of the probability lies in the tails as opposed to the central bulge. For any Normal random variable $x \sim \mathcal{N}(\mu_x, \sigma_x^2)$, the first four standardized moments are

$$\begin{aligned} \mathbb{E} \left[\left(\frac{x - \mu_x}{\sigma_x} \right)^1 \right] &= 0 & \mathbb{E} \left[\left(\frac{x - \mu_x}{\sigma_x} \right)^2 \right] &= 1 & \mathbb{E} \left[\left(\frac{x - \mu_x}{\sigma_x} \right)^3 \right] &= 0 \\ \mathbb{E} \left[\left(\frac{x - \mu_x}{\sigma_x} \right)^4 \right] &= 3 \end{aligned} \quad (\text{B.13})$$

which can be of some use in assessing whether a distribution is Normal or not, as in [Appendix D](#). However in general pattern recognition will rarely employ skewness or kurtosis or other expectations of higher powers, since most class shapes are not helpfully characterized by these measures.

B.3 Conditional Statistics

Frequently we have some information about a random variable, leading to a *conditional statistic* $(x|y)$, read as x given y , meaning that random vector x is now constrained by the information present in y . The information in y can take essentially any form: a measurement of x , a constraint on x , or information on another random variable that may or may not be related to x . For example, if x is a measurement of someone's heart-rate in beats-per-minute:

¹ A basic statistical moment has the form $\mathbb{E}[x^\eta]$, the central or mean-removed moment has the form $\mathbb{E}[(x - \mu_x)^\eta]$, and the standarized moment is mean-removed and normalized by the standard deviation.

(x)	Random variable describing a heart-rate
$(x \mid x \geq 100 \text{ bpm})$	x , given an inequality constraint
$(x \mid y = 100 \text{ bpm} \pm 10 \text{ bpm})$	x , given a noisy measurement
$(x \mid 50 \text{ year-old female})$	x , given a contextual constraint
$(x \mid \text{End of 5 km run})$	x , given a contextual constraint
$(x \mid \text{Google stock price})$	x , given irrelevant information

Conditional statistics are tremendously important in pattern recognition, since we are constantly assessing some unknown (e.g., the class associated with a pattern) conditioned on knowing certain information (e.g., a measurement). For example,

$$\begin{aligned} p(\text{feature} \mid \text{class}) &\text{ is the distribution of a feature for a given class} \\ \mathbf{P}(\text{class} \mid \text{feature}) &\text{ is the probability of a class given a feature value} \end{aligned} \quad (\text{B.14})$$

The conditional statistic $(x \mid y)$ is still a random variable, and therefore we can talk about its conditional probability distribution $p(x \mid y)$ and its resulting conditional expectation

$$\underbrace{\mathbb{E}[f(x \mid y)]}_{\text{The same thing, just different notation.}} = \mathbb{E}[f(x) \mid y] = \int_{-\infty}^{+\infty} f(x) p(x \mid y) dx. \quad (\text{B.15})$$

You should observe that we are integrating only over x , *not* over y , since y is just a given piece of information.

From the definition of conditional expectation in (B.15) follows the conditional mean $\mu_{x \mid y}$, conditional variance $\sigma_{x \mid y}^2$ etc. If the information in y is completely irrelevant to or unrelated to x , then the conditioning has no effect:

$$p(x \mid y) = p(x) \quad \mathbb{E}[f(x \mid y)] = \mathbb{E}[f(x)] \quad \mu_{x \mid y} = \mu_x \quad \sigma_{x \mid y}^2 = \sigma_x^2 \quad \text{etc.} \quad (\text{B.16})$$

The most fundamental equation governing conditional statistics is Bayes' Rule, which allows us to reverse the order of conditioning:

$$p(x \mid y) p(y) = p(y \mid x) p(x) \quad (\text{B.17})$$

It is important to understand that Bayes' Rule applies to probability densities, as in (B.17), probabilities, and combinations of probabilities and probability densities. Thus if x, y are continuous random variables and C, D are discrete, then in addition to (B.17) we have

$$p(x \mid C) \mathbf{P}(C) = \mathbf{P}(C \mid x) p(x) \quad \mathbf{P}(D \mid C) \mathbf{P}(C) = \mathbf{P}(C \mid D) \mathbf{P}(D) \quad (\text{B.18})$$

B.4 Random Vectors and Covariances

In many cases we don't just have one random element, but several. If n random variables are stacked into a vector,

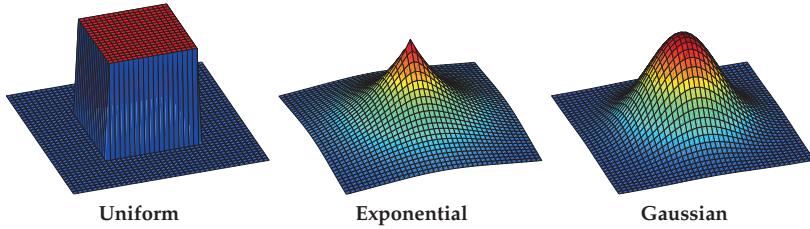


Fig. B.2. PROBABILITY DISTRIBUTION FUNCTIONS: The one-dimensional distributions defined in Figure B.1 can all be generalized to multi-dimensional distributions corresponding to random vectors. In all three cases shown here, the two-dimensional function shows the probability distribution (height) as a function of two random variables.

$$\underline{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad (\text{B.19})$$

then we refer to \underline{x} as a *random vector*. In principle we can generalize the probability densities of Figure B.1 to the random vector case, such that

$$\begin{aligned} & \mathbf{P}(x_1 < a_1, x_2 < a_2, \dots, x_n < a_n) \\ &= \int_{-\infty}^{a_1} \int_{-\infty}^{a_2} \cdots \int_{-\infty}^{a_n} p(x_1, x_2, \dots, x_n) dx_1 dx_2 \dots dx_n \end{aligned} \quad (\text{B.20})$$

as sketched for the two-dimensional case in Figure B.2. Any probability distribution over multiple random variables is referred to as a *joint* distribution. Any joint distribution can be written in chained conditional form, as

$$p(x, y, z) = p(x|y, z)p(y, z) = p(x|y, z)p(y|z)p(z) \quad (\text{B.21})$$

from which we can derive Bayes' Rule from (B.17):

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x) \quad (\text{B.22})$$

We can also remove variables from a joint distribution by integrating them out, to form a *marginal* distribution:

$$\underbrace{p(x)}_{\text{Marginal}} = \int_{-\infty}^{\infty} \underbrace{p(x, y)}_{\text{Joint}} dy \quad (\text{B.23})$$

Given a joint distribution, we can compute expectations or conditional expectations as before:

$$\begin{aligned} \mathbb{E}[f(x, y)] &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y)p(x, y) dx dy \\ \mathbb{E}[f(x, y)|z] &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y)p(x, y|z) dx dy \end{aligned} \quad (\text{B.24})$$

Since we have *multiple* random variables in a random vector, the most fundamental question is how these variables relate. If

$$p(x, y) = p(x) \cdot p(y) \quad (\text{B.25})$$

then x and y are *independent*. More generally, all of the elements of a random vector \underline{x} are independent if

$$p(\underline{x}) = p(x_1) \cdot p(x_2) \cdot \dots \cdot p(x_n) = \prod_{i=1}^n p(x_i) \quad (\text{B.26})$$

Although the plots in [Figure B.2](#) offer a nice intuitive sense of the dependence of the probability distribution for two variables, in practice the probability distribution equations for random vectors will be *even* more difficult to work with than for individual random variables, and so we will rarely be manipulating such probability distributions.

Instead, as before, we prefer the simpler concepts of mean and variance, but now generalized to a random vector. The vector mean is straightforward, being the means of the individual random variables:

$$\text{Mean}(\underline{x}) = \text{Mean} \left(\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \right) = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_n \end{bmatrix} = \boldsymbol{\mu} \quad (\text{B.27})$$

The variance becomes more complex, since we don't just have the variability of individual random variables, but also their interdependence in terms of the variability of pairwise products. Thus the co-variance is an $n \times n$ matrix, describing the interaction of all n variables with one another:

$$\begin{aligned} \text{Cov}(\underline{x}) &= \mathbb{E}[(\underline{x} - \boldsymbol{\mu})(\underline{x} - \boldsymbol{\mu})^T] \\ &= \begin{bmatrix} \mathbb{E}[(x_1 - \mu_1)^2] & \mathbb{E}[(x_1 - \mu_1)(x_2 - \mu_2)] & \cdots \\ \mathbb{E}[(x_2 - \mu_2)(x_1 - \mu_1)] & \mathbb{E}[(x_2 - \mu_2)^2] & \cdots \\ \mathbb{E}[(x_3 - \mu_3)(x_1 - \mu_1)] & \mathbb{E}[(x_3 - \mu_3)(x_2 - \mu_2)] & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \quad (\text{B.28}) \end{aligned}$$

Each of the pairwise terms is a *correlation*,

$$\text{Correlation}(x_i, x_j) = \mathbb{E}[(x_i - \mu_i)(x_j - \mu_j)] \equiv C_{ij} \quad (\text{B.29})$$

where the correlation of a variable with itself is equivalent to a variance,

$$\text{Correlation}(x_i, x_i) = \mathbb{E}[(x_i - \mu_i)^2] = C_{ii} \equiv \text{Variance}(x_i) \equiv \sigma_i^2 \quad (\text{B.30})$$

which allows us to write [\(B.28\)](#) somewhat more compactly as

$$\text{Cov}(\underline{x}) = \mathbb{E}[(\underline{x} - \mu)(\underline{x} - \mu)^T] = \begin{bmatrix} \sigma_1^2 & C_{12} & C_{13} & \cdots \\ C_{21} & \sigma_2^2 & C_{23} & \cdots \\ C_{31} & C_{32} & \sigma_3^2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \Sigma_x \quad (\text{B.31})$$

Since the interaction of x_i with x_j must, by definition, be the same as the interaction of x_j with x_i , therefore

$$C_{ij} \equiv C_{ji} \quad \rightarrow \quad \text{Cov}(\underline{x}) = \text{Cov}(\underline{x})^T \quad (\text{B.32})$$

That is, *every* covariance $\Sigma_x = \text{Cov}(\underline{x})$ must necessarily be a symmetric² matrix. Furthermore, from (B.31) we understand the diagonal elements of a covariance to represent the variance σ_i^2 of each of the individual random variables x_i in random vector \underline{x} , and the off-diagonal elements are the pairwise correlations C_{ij} , as defined in (B.30).

Frequently more interpretable than a correlation is a normalized version of the correlation, known as a correlation coefficient,

$$\text{CorrelationCoefficient}(x_i, x_j) = \frac{\mathbb{E}[(x_i - \mu_i)(x_j - \mu_j)]}{\sigma_i \sigma_j} \equiv \rho_{ij}. \quad (\text{B.34})$$

The correlation coefficient measures the strength of the linear relationship between two random variables:

Given	... then x_i and x_j are
$\rho_{i,j} = -1$	On a straight line with negative slope
$-1 < \rho_{i,j} < 0$	Negatively related
$\rho_{i,j} = 0$	Uncorrelated
$0 < \rho_{i,j} < 1$	Positively related
$\rho_{i,j} = 1$	On a straight line with positive slope

Two variables x_i, x_j are *uncorrelated* if their correlation is zero:

$$x_i, x_j \text{ Uncorrelated} \quad \rightarrow \quad C_{ij} = 0 \Leftrightarrow \rho_{ij} = 0 \Leftrightarrow \mathbb{E}[x_i x_j] = \mathbb{E}[x_i] \mathbb{E}[x_j] \quad (\text{B.35})$$

If *all* of the elements of \underline{x} are uncorrelated with one another, then the covariance $\text{Cov}(\underline{x})$ in (B.28) must necessarily be diagonal, which greatly simplifies matrix inversion, eigendecompositions etc.

² We can also define a *cross-covariance* matrix

$$\Sigma_{xy} = \text{Cov}(\underline{x}, y) = \mathbb{E}[(\underline{x} - \mu_x)(y - \mu_y)^T] \quad (\text{B.33})$$

which reflects the pairwise correlations between \underline{x} and y . Such cross-covariances are, in general, *not* symmetric, invertible, or even square.

In general, covariance matrices are **positive-definite** and thus invertible. A covariance Σ will be singular (non-invertible) if *any* of the following hold:

- The determinant is zero: $\det(\Sigma) = 0$
- At least one eigenvalue of Σ is zero
- At least one of the variances (diagonal elements of Σ) is zero
- Two of the variables in \underline{x} are perfectly correlated or anti-correlated ($\rho_{i,j} = \pm 1$)
- The variance of some linear function of \underline{x} is zero; that is,

$$\text{There exists an } f \neq 0 \text{ such that } \text{var}(f^T \underline{x}) = f^T \Sigma f = 0 \quad (\text{B.36})$$

The equivalence of the above is derived in [Appendix D](#). Singular covariances are most commonly encountered with sample statistics, discussed at the end of [Appendix B.6](#).

Conveniently, given any linear transformation, expressed via matrix T and deterministic offset vector $\underline{\alpha}$, the mean and the covariance of a random vector \underline{x} transform as

$$\text{Mean}(\underline{\alpha} + T\underline{x}) = \underline{\alpha} + T\mu_x \quad \text{Cov}(\underline{\alpha} + T\underline{x}) = T\Sigma_x T^T \quad (\text{B.37})$$

which is derived in [Appendix D](#). Given a vector of random variables, the mean of a sum is the sum of the means:

$$\text{Mean}\left(\sum_i x_i\right) = \sum_i \text{Mean}(x_i) \quad (\text{B.38})$$

The computation of the variance is a bit more difficult. If the random variables $\{x_i\}$ are all independent, then it is true that

$$\text{Variance}\left(\sum_i x_i\right) = \sum_i \text{Variance}(x_i) \quad (\text{B.39})$$

However if the variables are related in some way, then [\(B.39\)](#) does not hold; we can derive the more general case from [\(B.37\)](#). In particular,

$$\sum_i x_i = [1 \ 1 \ \cdots \ 1] \underline{x} \rightarrow \text{Variance}\left(\sum_i x_i\right) = [1 \ 1 \ \cdots \ 1] \Sigma \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = \sum_{i,j} \Sigma_{i,j} \quad (\text{B.40})$$

That is, in general, the variance of the sum of elements in a random vector is the sum over all elements in the covariance.

The transformation of probability densities is, in general, fairly difficult, precisely why we focus primarily on transformations of means and covariances.

However it is useful to understand that, given two independent random variables $x \sim p_x()$ and $y \sim p_y()$ then, as derived in [Appendix D](#) the distribution of the sum is the convolution of the distributions:

$$z = x + y \quad x, y \text{ independent} \quad \rightarrow \quad p_z(z) = p_x(z) * p_y(z) \quad (\text{B.41})$$

Finally, it is important to note that independence is a much stronger statement than uncorrelatedness. A covariance measures only the correlations, the strength of the *linear* relationships between all pairs of random variables in a vector, however the *actual* relationship between a pair of variables may not be linear. The stronger test, whether there is *any* relationship between two random variables, is that of *independence*:

- x and y are *uncorrelated* if there is no *linear* relationship between them;
- x and y are *independent* if there is no relationship of *any* kind between them.

Independent variables are therefore also uncorrelated; however uncorrelated variables may or may not be independent.

B.5 Outliers and Heavy-Tail Distributions

An *outlier* is a data point which is relatively far, by some distance measure, from the majority of points which characterize a class. An outlier may be caused by

- **Class Variability:** a data point reflecting the occasional very high variability within the objects in the class. For example suppose we have a set of classes describing types of trees. We take measurements from one particular oak tree, which belongs to the C_{Oak} class, but which, due to microclimate or soil or other growing conditions, grows in a way that is exceptionally unlike other trees, and therefore appears as a highly anomalous outlier among the measured points.
- **Measurement Noise:** a data point reflecting the occasional severe measurement error. That is, the measured object may be a typical member of its class, however due to measurement error or instrument failure it ends up appearing as an anomalous outlier.

There is no particular threshold at which a point is declared to be an outlier, although three standard-deviations is sometimes used as a threshold, as shown in [Figure B.3](#). Our interest is not so much the process of deciding which points are outliers and which are not, as opposed to what attributes make an algorithm sensitive to or robust to the presence of outliers.

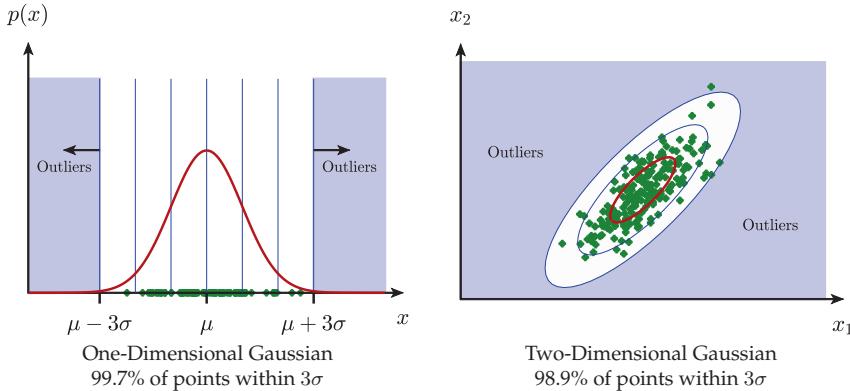


Fig. B.3. OUTLIERS: An outlier is a data point which is not representative of most of the points in a class. There is no particular threshold where outliers begin, however three standard-deviations is frequently used, as shown here in one and two dimensions. The data points (green) are sampled from a given distribution (red), for which three standard-deviation offsets/contours are shown. The examples shown here are both Gaussian, however any distribution or any class shape or any set of data points will have some sense of typical or usual, as opposed to rare far-flung outliers.

Outliers are modelled via high-variance or heavy-tailed distributions. The “tails” of a distribution are the relatively unlikely near-zero parts of the distribution far from the mean. A Gaussian distribution has “thin” tails, because asymptotically the distribution converges *very* quickly to zero:

$$p(x) \propto e^{-x^2} \quad \text{for } |x - \mu| \gg \sigma \quad (\text{B.42})$$

In contrast, there are many other distributions, most famously the power-law distributions of [Example B.1](#), which converge much more slowly, have relatively more likely tails, and therefore relatively more likely distant outliers.

A simple example of a distribution having heavier tails would be that of a Gaussian mixture model, combining likely “regular” (low variance) and unlikely “outlier” (high variance) portions:

$$p(x) = \underbrace{0.9 \cdot \mathcal{N}(0, 1)}_{90\% \text{ Regular distribution}} + \underbrace{0.1 \cdot \mathcal{N}(0, \nu^2)}_{10\% \text{ High variance outliers}} \quad (\text{B.43})$$

Given a dataset from such a distribution, the sample variance (from [Section B.6](#)) would be relatively modest,

$$\text{Sample Variance} \quad \hat{\sigma}^2 = 0.9 \cdot 1 + 0.1 \cdot \nu^2 = 0.9 + \frac{\nu^2}{10} \quad (\text{B.44})$$

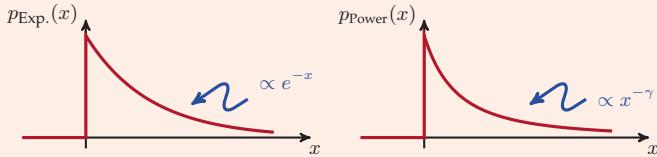
which rather obscures the fact that there are outliers present with a variance ν^2 times as large as the sample variance.

Example B.1: Power Laws and Infinite Variance

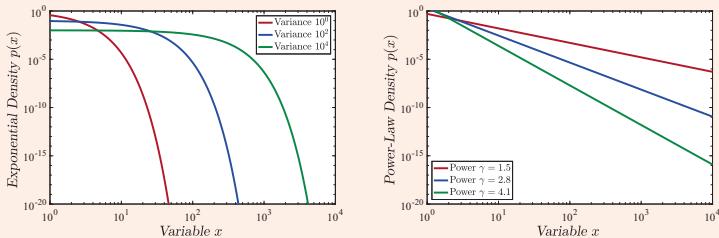
A power law is a distribution which falls asymptotically as a power of random variance x , rather than exponentially:

$$p_{\text{Exp.}}(x) \propto \exp(-\lambda x) \quad p_{\text{Power}}(x) \propto x^{-\gamma} \quad (\text{B.45})$$

where the distributions, when plotted, look like



Superficially these two distributions *appear* to be relatively similar, however a log-log plot allows for a closer look at the tails, where the differences become much more striking:



In particular, an exponential (or also Gaussian) distribution converges very steeply to zero (left), whereas a power-law distribution (right) is a straight line in a log-log plot, and so has a far higher likelihood of extreme outliers. What is remarkable is that a power law can have not just very large, but indeed *infinite* variance. If we define

$$p_{\text{Power Law}}(x) = \begin{cases} \beta x^{-\gamma} & x \geq 1 \\ 0 & x < 1 \end{cases} \quad (\text{B.46})$$

where γ is the power-law exponent, and β is a normalization constant, chosen such that $\int p(x) dx = 1$, then the distribution, mean, and variance can be found to be

$\gamma \leq 1$	Non-normalizable PDF		
$1 < \gamma \leq 2$	Normalizable PDF	Infinite mean	Infinite variance
$2 < \gamma \leq 3$	Normalizable PDF	Finite mean	Infinite variance
$3 < \gamma$	Normalizable PDF	Finite mean	Finite variance

Far from being just theoretical or exotic, power law distributions actually occur in a great many everyday contexts: economics (individual wealth, corporate profits), the natural world (earthquakes, meteor impacts, avalanches), human society (city sizes, war casualties), and human interaction (journal citations, movie popularity), among many others. Most of these contexts have a power-law exponent in the range $1 < \gamma < 3$, thus very much in the domain of (theoretically) infinite variances.

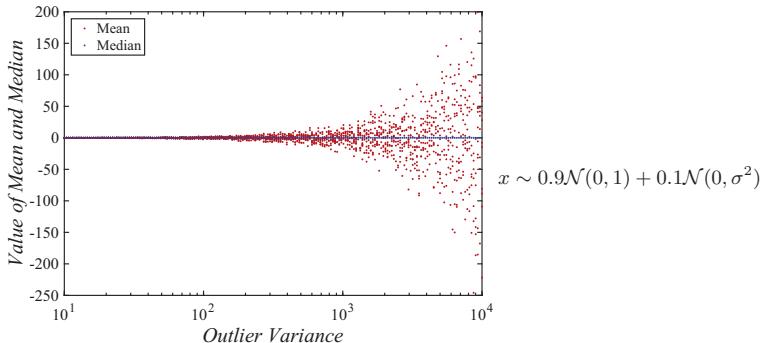


Fig. B.4. OUTLIER ROBUSTNESS: The plot shows the mean (red) and median (blue) as a function of outlier variance σ^2 (horizontal axis), for the given distribution (right).

In general, the robustness of an algorithm or computation to outliers will depend on how far-flung values do or do not influence the computation. For example, given a dataset of three data points,

$$\mathcal{D} = \{0, 1, x\} \quad \text{for some value } x > 1, \quad (\text{B.47})$$

then obviously

$$\text{Mean}(\mathcal{D}) = \frac{1+x}{3} \quad \text{Median}(\mathcal{D}) = 1 \quad (\text{B.48})$$

That is, the mean will be a function of x since, by definition, the mean is sensitive to *all* of the data points, and as x grows without bound, so does the mean. In contrast, the median of \mathcal{D} will be 1, *regardless* how large x becomes. This behaviour is shown graphically in Figure B.6, which shows the sensitivity of the mean to increasing variance, whereas the median is highly robust, even in the presence of large variances (Figure B.4).

In summary:

- **Robust to Outliers:** Any operation or criterion which is explicitly *not* a function of all of the data points — Median, Percentiles, Local kernel functions, Mean-Shift clustering (Section 12.1.3), Truncated/tapered optimization loss functions (Example 3.1)
- **Sensitive to Outliers:** Any operation which is a function of all of the data points — Mean, k -Nearest Neighbours
- **Highly Sensitive to Outliers:** Any operation which is sensitive to individual points or which assigns increasing weight to distant values — Variance, Minimum, Maximum, Nearest-Neighbour, Quadratic optimization loss functions

B.6 Sample Statistics

All of the preceding discussion is premised on somehow knowing the statistics of a random variable or vector. In practice, we need to take given sample data and *deduce* the statistics, then referred to as *sample* statistics.

In general, given N independent samples x_1, \dots, x_N of a random variable x , and given some function $f(x)$ of interest, we can estimate the sample expectation of $f(x)$ as

$$\mathbb{E}[f(x)] = \int f(x)p(x) dx \simeq \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (\text{B.49})$$

Particular common choices of $f()$ give rise to the sample mean or variance:

$$\begin{aligned} f(x) = x &\rightarrow \text{Sample Mean}(\{x_i\}) = \frac{1}{N} \sum_i x_i = \hat{\mu} \\ f(x) = (x - \mu)^2 &\rightarrow \text{Sample Variance}(\{x_i\}) = \frac{1}{N-1} \sum_i (x_i - \hat{\mu})^2 = \hat{\sigma}^2 \end{aligned} \quad (\text{B.50})$$

The slightly more complex equations in the random vector case appear as

$$\begin{aligned} \text{Sample Mean}(\{\underline{x}_i\}) &= \frac{1}{N} \sum_i \underline{x}_i = \hat{\underline{\mu}} \\ \text{Sample Covariance}(\{\underline{x}_i\}) &= \frac{1}{N-1} \sum_i (\underline{x}_i - \hat{\underline{\mu}})(\underline{x}_i - \hat{\underline{\mu}})^T = \hat{\Sigma} \end{aligned} \quad (\text{B.51})$$

Given a random variable x with true mean μ and standard deviation σ , the standard deviation of the *error* between the true and sample means is given by

$$\text{Standard Deviation}(\hat{\mu} - \mu) = \frac{\sigma}{\sqrt{N}} \quad (\text{B.52})$$

Similarly if the true variance is σ^2 , then the standard deviation of the *error* between the true and sample variances is given by

$$\text{Standard Deviation}(\hat{\sigma}^2 - \sigma^2) = \sigma^2 \sqrt{\frac{2}{N-1}} \quad (\text{B.53})$$

Therefore to improve the accuracy of the sample mean or the sample variance requires either a reduction in sample variability (σ) or an increase in the number of samples (N).

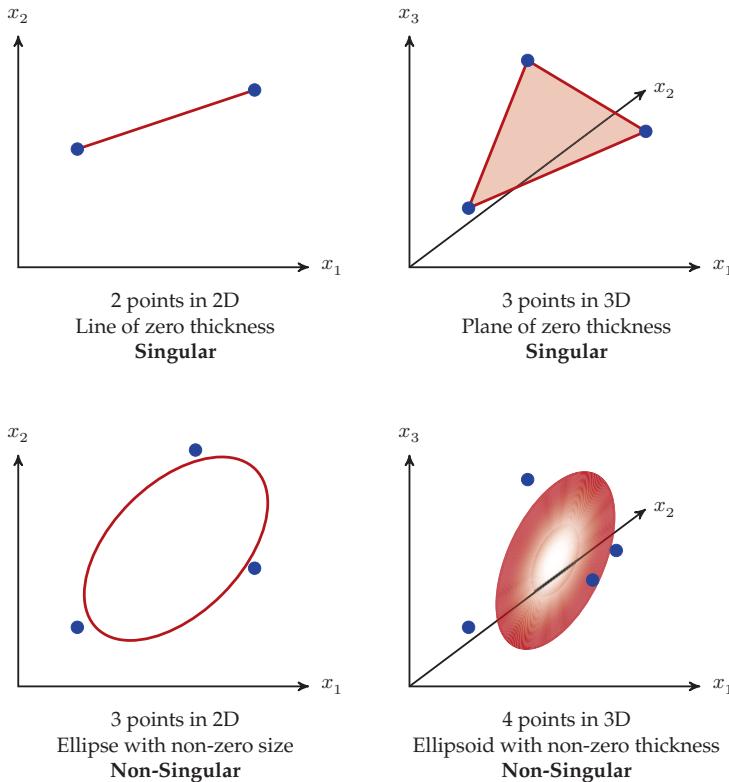


Fig. B.5. In n dimensions you need at least $n + 1$ points to “fill” the space (bottom). For n points in n dimensions (top) the points fit on a line/plane/hyperplane of zero thickness, so the resulting sample covariance is singular. The covariance singularity comes from the fact that there is some direction, normal to the line/plane/hyperplane, in which there is no variability, so the variance in that direction is zero.

Finally, it is important to understand some of the mechanics around the formation of the sample covariance. In particular, in terms of degrees of freedom (Chapter 3) the sample mean in n dimensions absorbs n degrees (i.e., one data point, since each data point in n dimensions specifies n values). The sample covariance then needs at least one data point *per dimension* to have even the most basic notion of class “size”. In other words, in n dimensions you need at least $n + 1$ sample points to specify the covariance; any fewer and the covariance will be singular, as illustrated in Figure B.5.

Even with *more* than n data points, if they all lie on the same line/plane/hyperplane, then the resulting sample covariance will *still* be singular. However, if the data points are subject to random measurement noise, it is exceptionally unlikely for $n + 1$ or more n -dimensional data points to lead to a singular

sample covariance. In practice, we would, of course, prefer to have significantly *more* data points, per (B.53), to meaningfully estimate a covariance, so perhaps at least $10n$ data points, rather than just $n + 1$.

Further Reading

As in the preceding appendix on linear algebra, there are a great many books written on all aspects of probability and statistics. In general, you would begin your search at Library of Congress call number QA273.

Very accessible starting points for probability and statistics include the texts by Brase & Brase [1], Kiemele [5], Haigh [4], and Urdan [7]. More comprehensive texts would include those by DeGroot *et al.* [2] and Walpole *et al.* [8].

In terms of the topic of outliers, heavy-tail distributions, and power laws, by far the best-known and very readable popular text on the subject is that by Taleb [6]. Because heavy-tail distributions are frequently stemming from or associated with complex systems, the reader may wish to consult my textbook on complex systems [3], where particularly Chapter 9 focuses on heavy-tail and power-law phenomena.

Sample Problems

Problem B.1: Short Answer

Give a short definition of each of the following:

- (a) Random variable
- (b) Probability distribution
- (c) Expectation
- (d) Bayes' Rule
- (e) Independence
- (f) Covariance
- (g) Outlier
- (h) Sample statistic

Problem B.2: Conceptual — Correlation

Two random variables being uncorrelated is not the same thing as being independent.

Therefore, identify two random variables (probably by sketching their distributions) that are uncorrelated but are, in fact, *not* independent.

Problem B.3: Conceptual — Joint Distributions

The relationship between joint and marginal statistics was shown in (B.23).

Describe the joint statistics of two random variables, x, y , such that the marginal distributions of x, y are both Gaussian, however the joint distribution is *not* Gaussian.

Note that the solution does not depend on Gaussianity in any way. The question could equally well have been stated with x, y marginally uniform or exponential, but not jointly uniform or exponential.

Problem B.4: Conceptual — Multivariate Gaussian

Suppose that we have two jointly-Gaussian random variables, x, y , where

$$\begin{bmatrix} x \\ y \end{bmatrix} = \mathcal{N} \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix} \right) \quad (\text{B.54})$$

- (a) What is the correlation coefficient $\rho_{x,y}$
- (b) Write out the equation for the joint distribution $p(x, y)$
- (c) Determine the marginal distribution $p(x)$
- (d) Find the general form for $p(x|y)$, and then find $p(x|y=0)$.

Problem B.5: Analytical — Means and Variances

Find the mean and variance for random variable x for each of the following density functions:

- (a) $p_x(x) = \begin{cases} a \exp(-ax) & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$
- (b) $p_x(x) = \begin{cases} \frac{1}{2}(x-1) & 1 \leq x \leq 3 \\ 0 & \text{otherwise} \end{cases}$
- (c) $p_x(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$

Problem B.6: Analytical — Expectations

In general, for random variable x and transformation $f()$, it is *not* true that

$$\mathbb{E}[f(x)] = f(\mathbb{E}[x]) \quad (\text{B.55})$$

- (a) Prove that (B.55) is true for any linear f .
- (b) Show that (B.55) can be false if f is nonlinear. For example, try $f(x) = 1/x$.

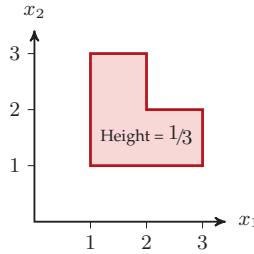


Fig. B.6. The two-dimensional distribution for Problem B.8.

Problem B.7: Analytical — Linear Transformations

Let x and y be random variables with means μ_x, μ_y and variances σ_x^2, σ_y^2 . Let C_{xy} be the correlation between X and Y . Find the mean and variance for each of

- (a) $z = x + a$
- (b) $z = ax$
- (c) $z = ax + by$

for constants a, b .

Problem B.8: Analytical — Distributions

Let \underline{x} be a two-dimensional random vector, where $p(\underline{x})$ is given by Figure B.6:

$$p(\underline{x}) = \begin{cases} \frac{1}{3} & \text{if } \underline{x} \text{ is in the shaded region} \\ 0 & \text{if } \underline{x} \text{ is outside} \end{cases} \quad (\text{B.56})$$

- (a) Argue persuasively the sign of the correlation coefficient between x_1 and x_2 .
- (b) Sketch or write down equations for $p(x_1), p(x_2), p(x_1 | x_2), p(x_2 | x_1)$.

References

1. C. Brase, C. Brase, *Understanding Basic Statistics* (Cengage Learning, Boston, 2012)
2. M. DeGroot, M. Schervish, *Probability and Statistics* (Pearson, London, 2018)
3. P. Fieguth, *An Introduction to Complex Systems* (Springer, New York, 2021)
4. J. Haigh, *Probability: A Very Short Introduction* (Oxford University Press, Oxford, 2012)

5. M. Kiemele, *Basic Statistics* (Air Academy, Colorado Springs, 1997)
6. N. Taleb, *The Black Swan: The Impact of the Highly Improbable* (Random House, New York, 2010)
7. T. Urdan, *Statistics in Plain English* (Routledge, Milton Park, 2010)
8. R. Walpole, R. Myers, S. Myers, K. Ye, *Probability & Statistics for Engineers & Scientists*. (Pearson, London, 2016)

Introduction to Optimization

C.1 Basic Principles

Optimization centres around the minimization or maximization of a given objective or loss function $\mathcal{L}(\theta_1, \dots, \theta_q)$ which depends on one or more variables $\theta_1, \dots, \theta_q$. Since maximizing $\mathcal{L}(\theta)$ is equivalent to minimizing $-\mathcal{L}(\theta)$, *any* maximization problem can be converted into a minimization, and vice-versa. Therefore we will *only* discuss minimization, with the understanding that all of the discussion in this appendix applies equally well to function maximization.

As illustrated in Figure C.1, the most critical distinction in optimization is to clearly understand the difference between the minimum *value* of a function, and the *minimizing* parameter value:

$$\underbrace{\text{Minimum value } \mathcal{L}_{\min} = \min_{\theta} \mathcal{L}(\theta)}_{\text{What is the smallest value of } \mathcal{L}(\theta)?} \quad \underbrace{\text{Minimizing value } \theta_{\min} = \arg_{\theta} \min \mathcal{L}(\theta)}_{\text{Which value of } \theta \text{ minimizes } \mathcal{L}(\theta)?}$$
(C.1)

These two are *very* different: the first is measured along the dependent (vertical) axis, the second along the independent (horizontal) axis. The slightly unusual notation

$$\arg_{\theta} \min \mathcal{L}(\theta) \tag{C.2}$$

should be read as *the argument (value) of θ which minimizes $\mathcal{L}(\theta)$* .

In the pattern recognition context of this book, for example, it is common for us to wish to minimize the probability of classification error $P(e|\underline{\theta})$, where the classifier is parameterized in terms of a vector $\underline{\theta}$ of parameters. In this context, the notation of (C.1) is understood as

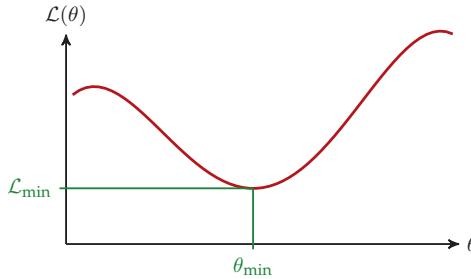


Fig. C.1. OPTIMIZATION: We would like to find the global minimum of some given function $\mathcal{L}(\theta)$. Both the *location* θ_{\min} of the minimum and the *value* \mathcal{L}_{\min} at the minimum are of interest to us.

$$\min_{\underline{\theta}} \mathbf{P}(e|\underline{\theta}) = \text{What is the smallest possible probability of error?} \quad (\text{C.3})$$

$$\arg_{\underline{\theta}} \min \mathbf{P}(e|\underline{\theta}) = \begin{array}{l} \text{What is the value of classifier parameter } \underline{\theta} \\ \text{which leads to the smallest probability of error?} \end{array} \quad (\text{C.4})$$

That is, the first statement tells us about performance, the second statement tells us how to achieve it.

With this context set, the rest of this appendix offers a very brief overview of concepts in optimization, with far more complete works listed in [Further Reading](#).

C.2 One-Dimensional Optimization

A one-dimensional optimization problem is sketched in [Figure C.1](#): a single function $\mathcal{L}(\theta)$ is dependent on a single variable θ . How easy or hard the problem is to solve, and what sort of strategies we may use to approach it, depend on certain attributes of $\mathcal{L}()$ and θ .

First, in terms of the independent variable θ over which we are optimizing, there are two basic possibilities:

CONTINUOUS: That is, θ can take on any real value,

$$\text{Possibly bounded } \theta_{\min} \leq \theta \leq \theta_{\max} \quad \text{or} \quad \text{Unbounded } -\infty < \theta < \infty \quad (\text{C.5})$$

The location or angle of a classification boundary would involve one or more continuous values, for example.

DISCRETE: That is, θ can take on one of a discrete set of values,

$$\theta \in \{\theta_1, \theta_2, \theta_3, \dots\} \quad (\text{C.6})$$

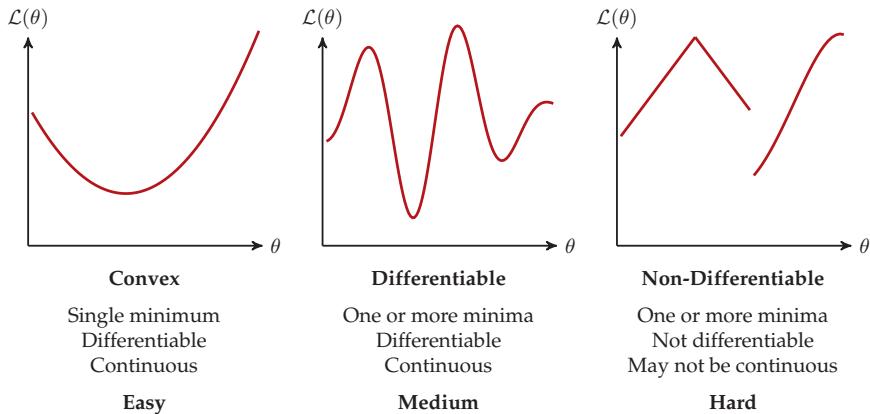


Fig. C.2. OBJECTIVE FUNCTIONS: The objective or loss function is the function which we are trying to optimize (minimize). Three basic categorizations (convex, continuous, discontinuous) are shown.

Discrete parameters appear frequently as the *order* of a polynomial, or the *number* of straight lines in a classification boundary, or the *number* of clusters making up a class.

Second, in terms of the dependent function $\mathcal{L}()$ which is being optimized, we may have three categories as illustrated in Figure C.2:

CONVEX: A function is convex if the line segment connecting *any* two points on the function does not cross the function (as illustrated in Figure C.3), which guarantees that the function has exactly one minimum, and this minimum is then relatively easy to locate. In many cases if a given function is convex then it is quadratic (parabolic), however there are also non-quadratic functions which are convex (Problem C.2).

DIFFERENTIABLE: A function being differentiable means that it is continuous and that its slope can be computed at every point, which allows gradient-descent strategies to be used in optimizing. The function can have multiple minima, so it may be difficult to find the global minimum.

NON-DIFFERENTIABLE: If a function is not differentiable then its slope may not exist at some points, and indeed the function may even be discontinuous. Minimizing such functions can be exceptionally difficult.

The resulting optimization problem will then have its minimum in one of four places, as shown in Figure C.4, and how to approach/solve the problem will then depend on which of those four contexts is being faced. The difficulty of the problem proceeds from unbounded-convex, as the easiest,

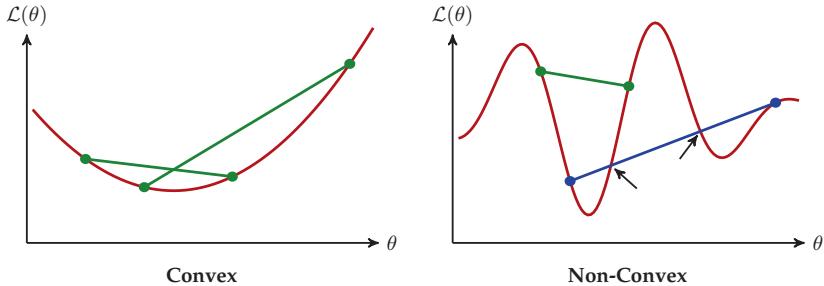


Fig. C.3. CONVEXITY: A function is convex if the line segments connecting all possible pairs of points on the function do not cross the function, a criterion which is satisfied for convex functions (left), but quite clearly not for a function having multiple minima (right). Note that non-convex functions can have *some* line segments satisfying the criterion (green), however we need this to be true for *all* possible line segments, and the blue segment clearly does fail, since the function (red) crosses the line segment twice (arrows).

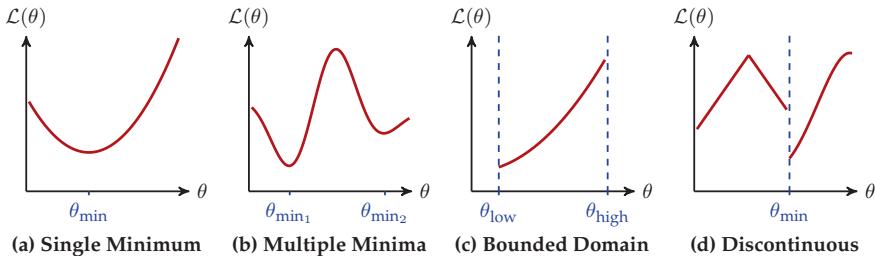


Fig. C.4. MINIMIZATION: Depending on the problem, the global minimum can be (a) at the single local minimum, (b) at one of multiple local minima, (c) at an endpoint when θ is bounded, or (d) at a point of discontinuity in $L()$.

then bounded-convex, then differentiable, and finally non-differentiable as the most challenging. The actual algorithmic approach can depend greatly on the details of θ and $L()$, and is largely outside of the scope of this appendix, with more details to be found under [Further Reading](#). Since a local minimum of $L(\theta)$ at $\theta = \bar{\theta}$ implies that the derivative $f'(\bar{\theta}) = 0$, $\bar{\theta}$ must be a root of $f'()$. Therefore there is significant overlap/commonality between optimization and root-finding methods.

Very broadly, many one-dimensional optimization strategies will be a variant of one of the following three categories, illustrated in [Figure C.5](#):

BISECTION METHODS: For convex problems, if we are able to bound the single minimum as

$$\theta_{\text{low}} < \theta_{\text{min}} < \theta_{\text{high}} \quad (\text{C.7})$$

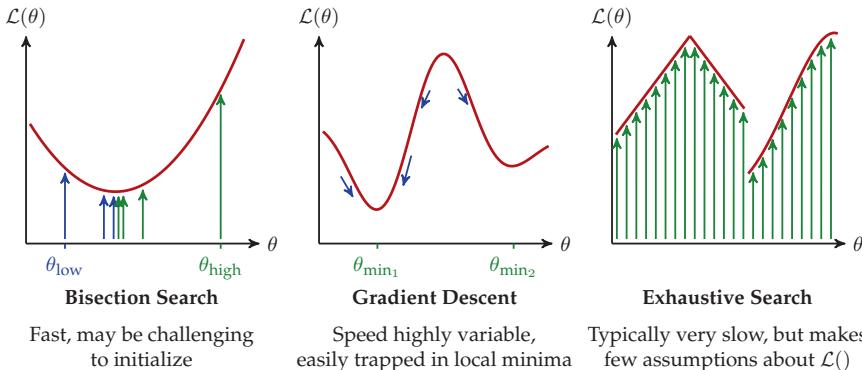


Fig. C.5. OPTIMIZATION STRATEGIES: A great many approaches have been formulated for solving one-dimensional optimization problems, however broadly they fall into one of the three categories illustrated here.

then it can be possible to repeatedly subdivide/bisect the domain by testing the slope of $\mathcal{L}()$ at the mid-point $(\theta_{\text{low}} + \theta_{\text{high}})/2$, and then determining whether the minimum is above or below the mid-point.

GRADIENT DESCENT: For any continuous function $\mathcal{L}(\theta)$, the function decreases in value if θ is moved in the direction of $-\mathcal{L}'(\theta)$, that is, in the direction opposite to the slope. A basic gradient descent thus iterates

$$\theta_{i+1} = \theta_i - \alpha \mathcal{L}'(\theta_i) \quad (\text{C.8})$$

for some amplification factor α . There are many challenges, including how to decide where to start the iteration θ_0 , how to set α , and the high likelihood of becoming trapped in undesired local minima.

EXHAUSTIVE SEARCH: In principle, a function $\mathcal{L}(\theta)$ can be evaluated for a great many closely-spaced points:

$$\theta_{\text{low}}, \theta_{\text{low}} + \delta, \theta_{\text{low}} + 2\delta, \dots, \theta_{\text{high}} - \delta, \theta_{\text{high}} \quad (\text{C.9})$$

Whether this is feasible or not depends on the size of the domain $\theta_{\text{high}} - \theta_{\text{low}}$ and the fineness δ with which the domain of θ needs to be explored to reliably find an accurate minimum. Exhaustive search can be perfectly reasonable for discrete-valued θ , but becomes impractical for the multi-dimensional problems of [Appendix C.3](#).

C.3 Multi-Dimensional Optimization

In most cases it is not a single parameter θ over which we are optimizing, as explored in [Appendix C.2](#), rather some *number* $q > 1$ of parameters collected

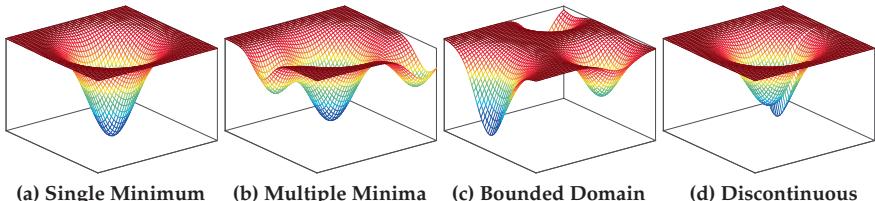


Fig. C.6. MINIMIZATION: The multi-dimensional generalization of Figure C.4 — the global minimum can be (a) at the single local minimum, (b) at one of multiple local minima, (c) at an endpoint when θ is bounded, or (d) at a point of discontinuity in $\mathcal{L}(\cdot)$.

in vector $\underline{\theta}$. In the case of large neural network classifiers, we could be talking about millions to *billions* of parameters in $\underline{\theta}$.

Figure C.6 shows the multi-dimensional generalization of Figure C.4, illustrating that it is the same four contexts, as in one dimension, in which the global minimum may be found. It should be noted, in passing, that the objective in Figure C.6a has only a single minimum, but is *not* in fact convex.

One might then think, from the similarity of Figures C.4 and C.6, that multi-dimensional optimization methods might be essentially the same as those shown in Figure C.5 for one dimension, however that is not the case, for two key reasons:

1. **The search space is *much* larger:** Unless the parameters are discrete and allow for very few possible values (e.g., the parameters θ_j are binary), the combinatorics of multi-dimensional problems make exhaustive search effectively impossible.
2. **A single point does not divide the space:** In one dimension a single point separates the search space into “left” and “right”, allowing strategies like a bisection search to be proposed. In multiple dimensions the space is *not* separated by a point, so bisection approaches do not generalize.

As a result, one of the basic strategies is to take a q -dimensional problem and address the optimization one dimension at a time, essentially casting the multi-dimensional problem back into the one-dimensional context of Appendix C.2, a strategy known as coordinate descent. Starting with some initial point $\underline{\theta}^0$, we would begin by optimizing over the first dimension,

$$\hat{\theta}_1 = \arg_{\vartheta} \min \mathcal{L}(\vartheta, \theta_2^0, \dots, \theta_q^0) \quad \rightarrow \quad \underline{\theta}^1 = \begin{bmatrix} \hat{\theta}_1 \\ \theta_2^0 \\ \vdots \end{bmatrix} \quad \begin{array}{l} \leftarrow \text{Insert new value} \\ \leftarrow \text{Keep previous values} \\ \vdots \end{array} \quad (C.10)$$

where ϑ is a dummy variable over which we are optimizing, returning the optimal value $\hat{\underline{\theta}}_1$, which is then inserted into $\underline{\theta}^0$ to give us $\underline{\theta}^1$. We then repeat (C.10) over θ_2 through θ_q . A single pass through all q variables does *not* in any way guarantee convergence to the global optimum, so normally the process of (C.10) needs to be applied repeatedly.

The gradient descent strategy of Figure C.5 *does* generalize to any number of dimensions, and is in fact very widely used, including most of neural network learning (as described in Section 11.4.1). The basic iteration from (C.8) needs to be generalized to the vector derivative,¹ which is straightforward to compute:

$$\underline{\theta}_{i+1} = \underline{\theta}_i - \alpha \frac{d\mathcal{L}}{d\underline{\theta}} \quad \frac{d\mathcal{L}}{d\underline{\theta}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \theta_q} \end{bmatrix} \quad (\text{C.11})$$

Finally, it is possible to consider a multi-dimensional generalization of the one-dimensional exhaustive search, essentially a local grid search. Given a current estimate $\underline{\theta}$, we can imagine exhaustively testing $\mathcal{L}()$ at all of the permutations

$$\{\theta_1 - \delta, \theta_1, \theta_1 + \delta\} \text{ and } \{\theta_2 - \delta, \theta_2, \theta_2 + \delta\} \text{ and } \dots \text{ and } \{\theta_q - \delta, \theta_q, \theta_q + \delta\}, \quad (\text{C.12})$$

thus for a total of 3^q points to evaluate, since (C.12) is proposing to test three points along each of the q dimensions. Clearly, a grid can be set up to test more points per dimension, if q is sufficiently small to allow this. Having found the best point $\hat{\underline{\theta}}$ among the 3^q tested, (C.12) then needs to be repeated, centered about $\hat{\underline{\theta}}$, typically then for a smaller value of δ , such that the grid slowly converges on a minimum.

In practice, none of these methods is guaranteed to find the global optimum, and significant parameter tuning or boundary checks need to be undertaken, all of which can make optimization a relatively complex business. The intent of this appendix is not to teach these details, rather to expose the reader to ideas of convexity, multiple minima, and bounded versus unbounded domains, in order to be able to effectively use built-in optimization routines in packages such as Matlab, Octave, or Python.

¹ Refer to Appendix A and Table A.1 for a discussion of vector derivatives. Although not important for the discussion here, technically (C.11) is written in so-called *Denominator Layout* form, since it makes (C.11) much more intuitive and readable.

C.4 Multi-Objective Optimization

In some contexts, we may have more than one objective which we wish to optimize. It is relatively common, for example, to hear people say something like

What is the cheapest and fastest route from here to Toronto?

which, strictly speaking, doesn't make sense. There are *two* objectives here — cost and time — the fastest route may be more expensive, and the cheapest route may be slower. In our pattern recognition context² we could imagine a variety of objectives which we might wish to minimize: computational complexity, storage memory requirements, and the probability of classification error.

In any event, given a multi-objective minimization problem of the form

$$\text{Find } \underline{\theta} \text{ to minimize } \mathcal{L}_1(\underline{\theta}) \text{ and also to minimize } \mathcal{L}_2(\underline{\theta}) \quad (\text{C.13})$$

we have two basic strategies:

1. Form a Single Objective:

Somewhat the multiple objectives need to be combined into a single objective to be optimized, such that the two objectives $\mathcal{L}_1()$ and $\mathcal{L}_2()$ of (C.13) are combined in some way:

$$\mathcal{L}_{\text{combined}}(\underline{\theta}) = \alpha_1 \mathcal{L}_1(\underline{\theta}) + \alpha_2 \mathcal{L}_2(\underline{\theta}) \quad (\text{C.14})$$

In (C.14) the objectives are combined in a linearly weighted fashion, however the combination could just as well be nonlinear. The key issue is that the combination of (C.14) is assumed/asserted, it cannot be derived from (C.13). There is no optimal or inherently correct way of combining multiple objectives.

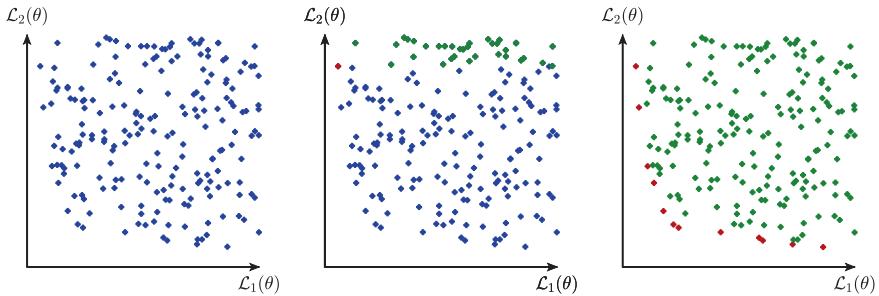
2. Evaluate over Both Objectives:

Each possible value of $\underline{\theta}$ leads to a pair $(\mathcal{L}_1(\underline{\theta}), \mathcal{L}_2(\underline{\theta}))$. If we have two values $\underline{\theta}_1$ and $\underline{\theta}_2$ such that

$$\mathcal{L}_1(\underline{\theta}_1) < \mathcal{L}_1(\underline{\theta}_2) \quad \text{and} \quad \mathcal{L}_2(\underline{\theta}_1) < \mathcal{L}_2(\underline{\theta}_2) \quad (\text{C.15})$$

then we can declare that solution $\underline{\theta}_2$ is inferior to $\underline{\theta}_1$, since *every* objective is larger at $\underline{\theta}_2$ than at $\underline{\theta}_1$, as illustrated in Figure C.7. The so-called *Pareto*

² The one pattern recognition context in which we face multi-objective optimization quite explicitly is the ROC plot of Chapter 9, for example Figure 9.2.



Two objectives $\mathcal{L}_1()$ and $\mathcal{L}_2()$ evaluated over all θ .

For the one red test, *all* of the green tests are inferior, since both $\mathcal{L}_1()$ and $\mathcal{L}_2()$ are worse.

The red tests are therefore the only ones that matter; all other points are inferior.

Fig. C.7. MULTIOBJECTIVE OPTIMIZATION: Given *two* objectives to minimize, there is rarely a single choice of θ which minimizes both. Instead, we can form the *Pareto Front*, the set of points (red, right) for which there is no solution which is superior in every objective.

Front is then those points which are not inferior to any other point. These points (shown in red in the right panel of Figure C.7) are the only ones that matter; it is then the task of the person undertaking the optimization to examine the Pareto points and to decide, by some rationale, which one is the optimum.

Further Reading

Wikipedia Links — Optimization Concepts: [Mathematical Optimization](#), [Multi-objective Optimization](#), [Pareto Frontier](#),

Wikipedia Links — Optimization Methods: [Gradient Descent](#), [Stochastic Gradient Descent](#), [Coordinate Descent](#), [Conjugate Gradients](#),

There are a great many books on optimization, and the interested reader will find a wide selection at Library of Congress call numbers QA164 and QA402.5 at any library.

Three initial suggestions to get you started would include the books by Boyd & Vandenberghe [1], Kochenderfer & Wheeler *et al.* [3], and the accessible text by Fischetti [2].

Sample Problems

Problem C.1: Short Answer

Give a short definition of each of the following:

- (a) Convex problem
- (b) Function with multiple minima
- (c) Gradient descent
- (d) Multi-objective optimization
- (e) Pareto front

Problem C.2: Conceptual — Convexity

A convex problem does not have to be quadratic. For a one-dimensional optimization problem over θ , identify a function $\mathcal{L}()$ which is convex but not quadratic.

Problem C.3: Conceptual — Optimization

From [Figure C.4](#), for a one-dimensional function $\mathcal{L}()$ over a bounded domain which has multiple minima, for what values of θ do you need to evaluate $\mathcal{L}(\theta)$ in order to find a global minimum?

How does your answer change if the objective function $\mathcal{L}()$ is a function of two-dimensional $\underline{\theta} \in \mathbb{R}^2$?

References

1. S. Boyd, L. Vandenberghe, *Convex Optimization* (Cambridge University Press, Cambridge, 2004)
2. M. Fischetti, *Introduction to Mathematical Optimization*. (Independent, New York, 2019)
3. M. Kochenderfer, T. Wheeler, *Introduction to Mathematical Optimization* (MIT Press, Cambridge, 2019)

D

Mathematical Derivations

There is a tension, in any writing, between *clarity* and *completeness*. Ideally a text should be accessible, engaging, and uncluttered, but at the same time also complete, so that the reader is not left frustrated with gaps in logic or analysis.

This appendix contains those mathematical derivations which are too detailed or too long to be desirable in the flow of the main body of the text. However these derivations are not unusually technical or advanced: all of the mathematics in this appendix should be accessible to any reader of the main text. Each derivation identifies the point in the text where the derivation is referred to, and the reader should be referring to that point in the text in order to have the appropriate context on which the derivation builds.

Total Covariance Over a Set of Classes:

(Page 87)

Given K classes

$$C_\kappa \sim (\mu_\kappa, \Sigma_\kappa) \quad \kappa = 1, \dots, K \quad (\text{D.1})$$

we would like to know the total covariance Σ_T of this system,

$$\Sigma_T = \mathbb{E}[(\underline{x} - \underline{\mu})(\dots)^T] \quad \text{where} \quad \underline{\mu} = \frac{1}{K} \sum_{\kappa} \mu_{\kappa} \quad (\text{D.2})$$

the overall covariance which would result if we sampled points at random from all K classes, assuming all classes to be equally likely.

Let \underline{x}_κ represent a random vector sampled from class C_κ , then

$$\Sigma_T = \frac{1}{K} \sum_{\kappa=1}^K \mathbb{E}[(\underline{x}_\kappa - \mu)(\dots)^T] \quad (\text{D.3})$$

$$= \frac{1}{K} \sum_{\kappa=1}^K \mathbb{E}[(\underline{x}_\kappa - \mu_\kappa + \mu_\kappa - \mu)(\dots)^T] \quad (\text{D.4})$$

$$= \frac{1}{K} \sum_{\kappa=1}^K \left\{ \underbrace{\mathbb{E}[(\underline{x}_\kappa - \mu_\kappa)(\dots)^T]}_{\text{Regular Covariance}} + \underbrace{\mathbb{E}[(\mu_\kappa - \mu)(\dots)^T]}_{\text{Covariance over Means}} \right.$$

$$\left. + 2 \cdot \mathbb{E}[(\underline{x}_\kappa - \mu_\kappa)(\mu_\kappa - \mu)^T] \right\} \quad (\text{D.5})$$

Deterministic

$$= \left(\frac{1}{K} \sum_{\kappa=1}^K \Sigma_\kappa \right) + \text{cov}(\mu_1, \dots, \mu_K) \\ + \left(\frac{1}{K} \sum_{\kappa=1}^K 2 \cdot \underbrace{\mathbb{E}[(\underline{x}_\kappa - \mu_\kappa)]}_{\text{Zero}} \cdot (\mu_\kappa - \mu)^T \right) \quad (\text{D.6})$$

$$= \text{cov}(\mu_1, \dots, \mu_K) + \frac{1}{K} \sum_{\kappa=1}^K \Sigma_\kappa \quad (\text{D.7})$$

as in (5.8). It is straightforward to adjust the derivation if the class probabilities are not equal.

Principal Components Analysis (PCA):

(Page 88)

Suppose we are given a random vector y , such that

$$y \in \mathbb{R}^m \quad y \sim (\underline{0}, \Sigma) \quad (\text{D.8})$$

Let us begin by finding the single principal component that best allows us to reconstruct y . That is, we seek a unit-length feature

$$f \in \mathbb{R}^m \quad |f| = f^T f = 1 \quad (\text{D.9})$$

which best allows us to represent y by its component along f :

$$\hat{y} = \text{The representation of } y = (f^T y) f \quad (\text{D.10})$$

We wish to minimize the squared error

$$(y - \hat{y})^T (y - \hat{y}) = (y - (f^T y) f)^T (y - (f^T y) f) \quad (\text{D.11})$$

$$= \underline{y}^T \underline{y} - (\underline{f}^T \underline{y}) \underline{f}^T \underline{y} - \underline{y}^T (\underline{f}^T \underline{y}) \underline{f} + (\underline{f}^T \underline{y}) \underbrace{\underline{f}^T \underline{f}}_{\text{Equal to 1}} (\underline{f}^T \underline{y}) \quad (\text{D.12})$$

where, because $(\underline{f}^T \underline{y})$ is scalar, we can freely move surrounding terms back and forth:

$$\underline{f}^T (\underline{f}^T \underline{y}) = (\underline{f}^T \underline{y}) \underline{f}^T \quad (\text{D.13})$$

Ignoring the first term of (D.12), which is unaffected by \underline{f} , and observing that each of these terms is a scalar, and therefore equal to its transpose, allows us to minimize the squared error by minimizing

$$-(\underline{f}^T \underline{y})(\underline{f}^T \underline{y}) - (\underline{y}^T \underline{f})(\underline{f}^T \underline{y}) + (\underline{f}^T \underline{y})(\underline{f}^T \underline{y}) = -(f^T \underline{y})(f^T \underline{y}) \quad (\text{D.14})$$

and therefore we wish to *maximize* $(f^T \underline{y})^2$. Since \underline{y} is a random vector, really it is the expectation of this expression which we wish to maximize, thus

$$\text{find } \underline{f} \text{ to maximize } \mathbb{E}[(f^T \underline{y})^2] \quad \text{such that } f^T \underline{f} = 1 \quad (\text{D.15})$$

Therefore

$$\mathbb{E}[(f^T \underline{y})^2] = \text{var}(f^T \underline{y}) = f^T \Sigma f \quad (\text{D.16})$$

Maximizing this term, subject to $|f| = 1$ is equivalent to

$$\text{Find } \underline{f} \text{ to maximize } \frac{f^T \Sigma f}{f^T f} \quad (\text{D.17})$$

From Table A.1, we can differentiate

$$\left(\frac{\partial}{\partial f} \right) \frac{f^T \Sigma f}{f^T f} = \frac{2f^T \Sigma}{f^T f} - \frac{f^T \Sigma f}{(f^T f)^2} 2f^T = 0 \quad (\text{D.18})$$

Multiplying through by $(f^T f)$, transposing, and collecting terms, we recognize that we have an eigendecomposition,

$$\Sigma f = \frac{f^T \Sigma f}{(f^T f)^2} f \quad \rightarrow \quad \Sigma f = \lambda f \quad (\text{D.19})$$

where the eigenvalue λ is precisely the quantity we wish to maximize. Therefore the optimal representation, the principal component, is found as that normalized (unit-length) eigenvector \underline{v}_1 corresponding to the largest eigenvalue λ_1 of Σ .

Having found the first principal component, now let \underline{y}' be the original random vector with the first principal component removed:

$$\underline{y}' = \underline{y} - \hat{\underline{y}} = \underline{y} - (\underline{f}^T \underline{y}) \underline{f} = \underline{y} - (\underline{v}_1^T \underline{y}) \underline{v}_1 = (I - \underline{v}_1 \underline{v}_1^T) \underline{y} \quad (\text{D.20})$$

Therefore

$$\text{cov}(\underline{y}') = \Sigma' = \text{cov}((I - \underline{v}_1 \underline{v}_1^T) \underline{y}) = \Sigma - 2\Sigma \underline{v}_1 \underline{v}_1^T + \underline{v}_1 \underline{v}_1^T \Sigma \underline{v}_1 \underline{v}_1^T \quad (\text{D.21})$$

Since $\Sigma \underline{v}_1 = \lambda_1 \underline{v}_1$ and $\underline{v}_1^T \underline{v}_1 = 1$, therefore

$$\Sigma' = \Sigma - 2\lambda_1 \underline{v}_1 \underline{v}_1^T + \underline{v}_1 \underline{v}_1^T \lambda_1 \underline{v}_1 \underline{v}_1^T = \Sigma - \lambda_1 \underline{v}_1 \underline{v}_1^T \quad (\text{D.22})$$

Following precisely the same derivation as above, we now wish to find the principal component best representing \underline{y}' , which we know will be the eigenvector corresponding to the largest eigenvalue in Σ' . However

$$\Sigma' \underline{v}_j = (\Sigma - \lambda_1 \underline{v}_1 \underline{v}_1^T) \underline{v}_j = \Sigma \underline{v}_j - \lambda_1 \underline{v}_1 \underline{v}_1^T \underline{v}_j = \begin{cases} 0 & j = 1 \\ \lambda_j \underline{v}_j & j \neq 1 \end{cases} \quad (\text{D.23})$$

where the $j = 1$ result follows from the eigenvectors being unit-length, and the $j \neq 1$ result follows from covariance eigenvectors being orthogonal.

From (D.23), we conclude that the eigendecomposition of Σ and that of Σ' are the *same*, except with the first (largest) eigenvalue removed. Therefore the principal component to be found from Σ' is precisely the eigenvector corresponding to the *second*-largest eigenvalue in Σ .

Continuing the above argument recursively, we conclude that the first n principal components in the optimal mean-square representation of a random vector \underline{y} is found as the n eigenvectors corresponding to the largest n eigenvalues of Σ .

Covariance for Two Classes:

(Page 92)

We are given a $K = 2$ class problem, with the two classes having means μ_1, μ_2 . Then (5.19) needed an expression for

$$\text{cov}(\mu_1, \mu_2) \quad (\text{D.24})$$

From (B.51) we know the form of the sample mean and covariance:

$$\text{mean}(\mu_1, \mu_2) = \frac{\mu_1 + \mu_2}{2} \quad \text{cov}(\underline{x}_1, \dots, \underline{x}_N) = \frac{\sum_{i=1}^N (\underline{x}_i - \text{mean}(\{\underline{x}_j\})) (\dots)^T}{N-1} \quad (\text{D.25})$$

Therefore

$$\text{cov}(\mu_1, \mu_2) = \left(\mu_1 - \frac{\mu_1 + \mu_2}{2} \right) (\dots)^T + \left(\mu_2 - \frac{\mu_1 + \mu_2}{2} \right) (\dots)^T \quad (\text{D.26})$$

$$= \left(\frac{\mu_1 - \mu_2}{2} \right) (\dots)^T + \left(\frac{\mu_2 - \mu_1}{2} \right) (\dots)^T \quad (\text{D.27})$$

$$= 2 \cdot \frac{1}{4} (\mu_1 - \mu_2) (\mu_1 - \mu_2)^T \quad (\text{D.28})$$

Fisher's Linear Discriminant for Multiple Features: (Page 93)

From (5.23) we wish to

$$\text{Find } F \in \mathbb{R}^{m \times n} \text{ to maximize } J(F) = \frac{|F^T \Sigma_T F|}{|F^T \Sigma_W F|} \quad (\text{D.29})$$

From Table A.1,

$$\begin{aligned} \frac{\partial J(F)}{\partial F} &= \frac{\partial}{\partial F} \frac{|F^T \Sigma_T F|}{|F^T \Sigma_W F|} \\ &= 2 \frac{|F^T \Sigma_T F| (F^T \Sigma_T F)^{-1} F^T \Sigma_T}{|F^T \Sigma_W F|} \\ &\quad - \frac{|F^T \Sigma_T F|}{|F^T \Sigma_W F|^2} 2 |F^T \Sigma_W F| (F^T \Sigma_W F)^{-1} F^T \Sigma_W \end{aligned} \quad (\text{D.30})$$

which we set equal to zero to maximize:

$$2 \frac{|F^T \Sigma_T F| (F^T \Sigma_T F)^{-1} F^T \Sigma_T}{|F^T \Sigma_W F|} = \frac{|F^T \Sigma_T F|}{|F^T \Sigma_W F|^2} 2 |F^T \Sigma_W F| (F^T \Sigma_W F)^{-1} F^T \Sigma_W \quad (\text{D.31})$$

Multiplying/dividing through the determinants to remove them, we are left with

$$(F^T \Sigma_T F)^{-1} F^T \Sigma_T = (F^T \Sigma_W F)^{-1} F^T \Sigma_W \quad (\text{D.32})$$

therefore, taking the transpose and rearranging,

$$(\Sigma_W^{-1} \Sigma_T) F = F \underbrace{(F^T \Sigma_W F)^{-1} (F^T \Sigma_T F)}_{\text{Is this diagonal?}} \quad (\text{D.33})$$

which looks strikingly like the eigendecomposition we had earlier, in (5.18), for the case of optimizing a *single* feature. However for this to be an eigendecomposition, the latter term on the right would need to be a diagonal matrix, so we need to validate whether this is true.

Let us suppose, for a moment, that indeed F is the solution to the generalized eigendecomposition

$$(\Sigma_W^{-1} \Sigma_T) F = F \Lambda \quad (\text{D.34})$$

for diagonal Λ . Then

$$\Sigma_T F = \Sigma_W F \Lambda \rightarrow F^T \Sigma_T F = F^T \Sigma_W F \Lambda \rightarrow (F^T \Sigma_W F)^{-1} (F^T \Sigma_T F) = \Lambda \quad (\text{D.35})$$

That is, choosing F as the solution to the generalized eigendecomposition *does* in fact satisfy (D.33). This does not prove that F is the *only* possible solution, however it is indeed *a* solution to (D.33). Our original criterion was

$$J(F) = \frac{|F^T \Sigma_T F|}{|F^T \Sigma_W F|} = |(F^T \Sigma_T F)(F^T \Sigma_W F)^{-1}| = |\Lambda| = \prod_{j=1}^n \lambda_j \quad (\text{D.36})$$

That is, we are able to reinterpret our criterion $J(F)$ as being the product of eigenvalues in diagonal Λ . Therefore to maximize J we need to choose F to contain the n eigenvectors corresponding to the n largest eigenvalues of $(\Sigma_W^{-1} \Sigma_T)$.

The Mean as Optimal Prototype:

(Page 125)

Suppose we are given N points $\{\underline{x}_i\}$ representing a class. The optimal prototype \underline{z} should be that point which is closest (least inconsistent), on average, to all of the points in the class. That is, minimizing the mean-squared error

$$\hat{\underline{z}}_E = \underbrace{\arg_{\underline{z}} \min \sum_i d_E^2(\underline{z}, \underline{x}_i)}_{\text{Closest, in squared-Euclidean distance}} \quad \text{or} \quad \hat{\underline{z}}_M = \underbrace{\arg_{\underline{z}} \min \sum_i d_M^2(\underline{z}, \underline{x}_i)}_{\text{Closest, in squared-Mahalanobis distance}} \quad (\text{D.37})$$

where we would like to derive the optimal prototype for both the Euclidean (left) and Mahalanobis (right) distances. Chapter 6 did specify that the Mahalanobis distance was assumed to be measuring to the mean of a class, however this derivation will briefly relax that requirement while we find the optimal prototype $\hat{\underline{z}}_M$.

To optimize $\hat{\underline{z}}_E, \hat{\underline{z}}_M$, we need to substitute in for $d()$ and differentiate with respect to \underline{z} :

$$\frac{\partial}{\partial \underline{z}} \sum_i (\underline{z} - \underline{x}_i)^T (\underline{z} - \underline{x}_i) \Big|_{\underline{z}=\hat{\underline{z}}_E} = 0 \quad \frac{\partial}{\partial \underline{z}} \sum_i (\underline{z} - \underline{x}_i)^T \Sigma^{-1} (\underline{z} - \underline{x}_i) \Big|_{\underline{z}=\hat{\underline{z}}_M} = 0 \quad (\text{D.38})$$

for class covariance Σ . The derivatives can be found from [Table A.1](#), in which case

$$\sum_i 2(\underline{z} - \underline{x}_i)^T \Big|_{\underline{z}=\hat{\underline{z}}_E} = 0 \quad \sum_i (\underline{z} - \underline{x}_i)^T (\Sigma^{-1} + \Sigma^{-T}) \Big|_{\underline{z}=\hat{\underline{z}}_M} = 0 \quad (\text{D.39})$$

Since Σ is a covariance it is symmetric, thus

$$(\Sigma^{-1} + \Sigma^{-T}) = (\Sigma^{-1} + \Sigma^{-1}) = 2\Sigma^{-1} \quad (\text{D.40})$$

So in [\(D.39\)](#) we can discard the constant factors¹ of 2 or $2\Sigma^{-1}$ and transpose,

$$\sum_i (\underline{z} - \underline{x}_i) \Big|_{\underline{z}=\hat{\underline{z}}_E} = 0 \quad \sum_i (\underline{z} - \underline{x}_i) \Big|_{\underline{z}=\hat{\underline{z}}_M} = 0 \quad (\text{D.41})$$

Separating each sum into two pieces, we have

$$\sum_i \hat{z}_E = \sum_i \underline{x}_i \quad \sum_i \hat{z}_M = \sum_i \underline{x}_i \quad (\text{D.42})$$

such that both derivations give the same result, which is that the sample mean of [Appendix B.6](#) is the optimum prototype:

$$\hat{z}_E = \hat{z}_M = \frac{1}{N} \sum_{i=1}^N \underline{x}_i \quad (\text{D.43})$$

Maximum Likelihood Estimation of a Variance:

(Page 156)

Suppose we are given a set of Gaussian-distributed independent $x_i \sim \mathcal{N}(\mu, \sigma^2)$. We wish to derive the maximum likelihood estimate of unknown σ^2 . Since it can be misleading to take derivatives with respect to a squared quantity σ^2 , it will be cleaner to let the unknown variance be $v = \sigma^2$. Although [Section 7.2](#) derived $\hat{\mu}_{ML}$ on its own, technically this was possible only because the solution for $\hat{\mu}_{ML}$ was not a function of v . Strictly speaking, *all* of the parameters need to be estimated jointly, thus

$$\underline{\theta} = \begin{bmatrix} \mu \\ v \end{bmatrix} \quad \hat{\theta}_{ML} = \arg \underline{\theta} \max \prod_{i=1}^N p(x_i | \underline{\theta}) \quad (\text{D.44})$$

from which it follows that

$$\frac{\partial}{\partial \underline{\theta}} \prod_{i=1}^N p(x_i | \underline{\theta}) = 0 \quad \rightarrow \quad \frac{\partial}{\partial \mu} \prod_{i=1}^N p(x_i | \underline{\theta}) = 0 \quad \frac{\partial}{\partial v} \prod_{i=1}^N p(x_i | \underline{\theta}) = 0 \quad (\text{D.45})$$

¹ To be precise, to discard a constant matrix that matrix must be on the left or right end of the expression and must be invertible. Because we aren't actually dividing through, rather we are left- or right-multiplying by the matrix inverse. Here, Σ^{-1} is on the right side of the expression and is known to be invertible.

The first of the last two terms, to estimate the mean, we have already solved, so it is only the last term to solve here. Since the statistics are Gaussian, we are motivated to take a logarithm:

$$\frac{\partial}{\partial v} \log \left\{ \prod_{i=1}^N \frac{1}{\sqrt{2\pi v}} \exp \left(-\frac{(x_i - \mu)^2}{2v} \right) \right\} \Big|_{\begin{subarray}{l} \mu = \hat{\mu}_{\text{ML}} \\ v = \hat{v}_{\text{ML}} \end{subarray}} = 0 \quad (\text{D.46})$$

Applying the logarithm to the product and the exponential leaves us with

$$\frac{\partial}{\partial v} \sum_{i=1}^N \left\{ -\log \sqrt{2\pi} - \log \sqrt{v} - \left(\frac{(x_i - \mu)^2}{2v} \right) \right\} = 0 \quad (\text{D.47})$$

Taking the derivative, we find

$$\sum_{i=1}^N \left\{ 0 - \frac{1}{\sqrt{v}} \cdot \frac{1}{2} \frac{1}{\sqrt{v}} + \frac{(x_i - \mu)^2}{2v^2} \right\} = 0 \quad \rightarrow \quad \frac{N}{v} = \sum_{i=1}^N \frac{(x_i - \mu)^2}{v^2} = 0 \quad (\text{D.48})$$

from which the maximum likelihood estimate is found as

$$\hat{v}_{\text{ML}} = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{\mu}_{\text{ML}})^2 \quad (\text{D.49})$$

Note that this estimator is biased, so in many circumstances the *unbiased* (but no longer ML) estimator is preferred,

$$\hat{v} = \frac{1}{N-1} \sum_{i=1}^N (x_i - \hat{\mu}_{\text{ML}})^2 \quad (\text{D.50})$$

for example as listed in the sample statistics in (B.50).

Bias and Variance of Probability Estimates: (Page 158)

Given N independent observations, of which M are observed to come from event Q , then (7.30) found

$$\vartheta \text{ is the probability of event } Q \quad \rightarrow \quad \hat{\vartheta}_{\text{ML}} = \frac{M}{N} \quad (\text{D.51})$$

We would like to determine the bias and error variance of this estimator. Recalling the binomial distribution governing random variable M ,

$$\begin{aligned}\mathbf{P}(M \text{ of } N \text{ samples from } Q) &= \binom{N}{M} \vartheta^M (1 - \vartheta)^{N-M} \\ &= \frac{N!}{M!(N-M)!} \vartheta^M (1 - \vartheta)^{N-M}\end{aligned}\quad (\text{D.52})$$

it follows that

$$\mathbb{E} \left[\hat{\vartheta}_{\text{ML}} \right] = \mathbb{E} \left[\frac{M}{N} \right] = \sum_{i=0}^N \frac{i}{N} \cdot \mathbf{P}(M \text{ has value } i) \quad (\text{D.53})$$

When $i = 0$ the summand is zero, so the sum can index from 1:

$$\mathbb{E} \left[\hat{\vartheta}_{\text{ML}} \right] = \sum_{i=1}^N \frac{i}{N} \frac{N!}{i!(N-i)!} \vartheta^i (1 - \vartheta)^{N-i} \quad (\text{D.54})$$

We cannot solve such a sum, in general. Our goal is to reshape the equation into something that we know, like the sum over a whole binomial distribution, as in (D.52):

$$\mathbb{E} \left[\hat{\vartheta}_{\text{ML}} \right] = \sum_{i=1}^N \frac{i}{N} \frac{N}{i} \frac{(N-1)!}{(i-1)!(N-i)!} \vartheta \cdot \vartheta^{i-1} (1 - \vartheta)^{N-i} \quad (\text{D.55})$$

$$= \vartheta \cdot \sum_{i=0}^{N-1} \frac{(N-1)!}{i!(N-1-i)!} \vartheta^i (1 - \vartheta)^{N-1-i} \quad (\text{D.56})$$

$$= \vartheta \cdot \underbrace{\sum_{i=0}^{N-1} \binom{N-1}{i} \vartheta^i (1 - \vartheta)^{N-1-i}}_{\text{Sum over whole distribution must sum to 1}} = \vartheta \quad (\text{D.57})$$

Therefore $\mathbb{E}[\hat{\vartheta}_{\text{ML}}] = \vartheta$ and we conclude that the estimator $\hat{\vartheta}_{\text{ML}}$ is unbiased. The error variance of $\hat{\vartheta}_{\text{ML}}$ is slightly more difficult. We begin with the definition of the error variance:

$$\text{var}(\tilde{\vartheta}_{\text{ML}}) = \mathbb{E} \left[(\hat{\vartheta}_{\text{ML}} - \mathbb{E}[\hat{\vartheta}_{\text{ML}}])^2 \right] = \mathbb{E} \left[\left(\frac{M}{N} - \vartheta \right)^2 \right] = \mathbb{E} \left[\frac{M^2}{N^2} \right] - 2\vartheta \mathbb{E} \left[\frac{M}{N} \right] + \vartheta^2 \quad (\text{D.58})$$

where, in the last expression, it is only the first (green) term which we do not know, since the second (orange) we just derived as part of calculating the bias. We begin the same way as in (D.53):

$$\mathbb{E} \left[\frac{M^2}{N^2} \right] = \sum_{i=0}^N \frac{i^2}{N^2} \frac{N!}{i!(N-i)!} \vartheta^i (1-\vartheta)^{N-i} \quad (\text{D.59})$$

$$= \sum_{i=1}^N \frac{i^2}{N^2} \frac{N}{i} \frac{(N-1)!}{(i-1)!(N-i)!} \vartheta \cdot \vartheta^{i-1} (1-\vartheta)^{N-i} \quad (\text{D.60})$$

$$= \vartheta \cdot \sum_{i=0}^{N-1} \frac{i+1}{N} \frac{(N-1)!}{i!(N-1-i)!} \vartheta^i (1-\vartheta)^{N-1-i} \quad (\text{D.61})$$

At this point we have $(i+1)$ in the numerator, but $i!$ in the denominator, so we cannot pull out a common term to cancel. Instead, the $(i+1)$ term needs to be split, so that we have two expressions, each of which we can solve:

$$\sum_{i=0}^{N-1} \frac{i+1}{N} \frac{(N-1)!}{i!(N-1-i)!} \vartheta^i (1-\vartheta)^{N-1-i} \quad (\text{D.62})$$

$$\begin{aligned} &= \left(\underbrace{\sum_{i=0}^{N-1} \frac{i}{N} \frac{(N-1)!}{i!(N-1-i)!} \vartheta^i (1-\vartheta)^{N-1-i}}_{\text{\textit{i portion of } } (i+1)} + \underbrace{\sum_{i=0}^{N-1} \frac{1}{N} \frac{(N-1)!}{i!(N-1-i)!} \vartheta^i (1-\vartheta)^{N-1-i}}_{\text{\textit{Remaining portion of } } (i+1)} \right) \\ &= \left(\underbrace{\frac{N-1}{N} \cdot \sum_{i=1}^{N-1} \frac{i}{N-1} \frac{(N-1)!\vartheta^i (1-\vartheta)^{N-1-i}}{i!(N-1-i)!}}_{\text{Same as (D.54)}} + \underbrace{\frac{1}{N} \cdot \sum_{i=0}^{N-1} \frac{(N-1)!\vartheta^i (1-\vartheta)^{N-1-i}}{i!(N-1-i)!}}_{\text{Sum over regular binomial } = 1} \right) \\ &= \frac{N-1}{N} \cdot \vartheta + \frac{1}{N} \cdot 1 \end{aligned} \quad (\text{D.63})$$

Inserting this result and the result from the bias derivation into (D.54), we have

$$\text{var}(\tilde{\vartheta}_{\text{ML}}) = \mathbb{E} \left[\frac{M^2}{N^2} \right] - 2\vartheta \mathbb{E} \left[\frac{M}{N} \right] + \vartheta^2 = \vartheta^2 \frac{N-1}{N} + \frac{\vartheta}{N} - 2\vartheta \cdot \vartheta + \vartheta^2 \quad (\text{D.64})$$

$$= \frac{\vartheta(1-\vartheta)}{N} \quad (\text{D.65})$$

This result is actually fairly intuitive:

If an event Q is impossible $\rightarrow \vartheta = 0 \rightarrow M = 0 \rightarrow \hat{\vartheta} = 0 \rightarrow \tilde{\vartheta} = 0$
If an event Q is guaranteed $\rightarrow \vartheta = 1 \rightarrow M = N \rightarrow \hat{\vartheta} = 1 \rightarrow \tilde{\vartheta} = 0$

That is, the error variance must be *zero* at the endpoints, as derived, and is maximized for an intermediate probability of $\vartheta = 0.5$.

In many cases we are estimating the probabilities of a large number of events, such as the individual bars in a histogram as part of estimating a probability distribution. In such cases, each individual event has a probability $\vartheta \ll 1$, such that $(1 - \vartheta) \simeq 1$ and

$$\sigma_{\tilde{\vartheta}} = \left\{ \text{var}(\tilde{\vartheta}_{\text{ML}}) \right\}^{1/2} \simeq \left(\frac{\vartheta}{N} \right)^{1/2} \simeq \frac{\sqrt{M}}{N} \quad (\text{D.66})$$

Since frequently what really matters is the *fractional* error, the size of the standard-deviation relative to the mean, this reduces very elegantly to

$$\frac{\sigma_{\tilde{\vartheta}}}{\vartheta} \simeq \frac{\sqrt{M}/N}{M/N} = \frac{1}{\sqrt{M}} \quad (\text{D.67})$$

Asymptotic Behaviour of kNN Probability Distributions: (Page 174)

From (7.73) we have the kNN probability distribution estimation

$$\hat{p}_{\text{kNN}}(x) = \frac{M}{|R(x)| \cdot N} \quad (\text{D.68})$$

where $R(x)$ is the smallest region, centered on x , enclosing at least M data points $\{x_i\}$, $1 \leq i \leq N$. For notational simplicity, assume the data points to be sorted, such that

$$x_1 \leq x_2 \leq \dots \leq x_N \quad (\text{D.69})$$

Asymptotically, for any $x > x_N$, the M th closest point will be x_{N+1-M} , therefore the size of the region $R(x)$, centered on x , reaching the closest M points, must be

$$|R(x)| = 2 \cdot (x - x_{N+1-M}) \quad (\text{D.70})$$

Therefore for $x > x_N$ the estimated distribution in (D.68) is

$$\hat{p}_{\text{kNN}}(x) = \frac{M}{2(x - x_{N+1-M})N} > \frac{M}{2xN} \quad (\text{D.71})$$

therefore

$$\int_{-\infty}^{\infty} \hat{p}_{\text{kNN}}(x) dx \geq \int_{x_N}^{\infty} \hat{p}_{\text{kNN}}(x) dx \geq \int_{x_N}^{\infty} \frac{M}{2xN} dx = \frac{M}{2N} \ln(x) \Big|_{x_N}^{\infty} = \infty \quad (\text{D.72})$$

The result in (D.72) applies to one-dimensional problems. In two or more dimensions the issue is a little more subtle, since a set of points cannot be sorted, as in (D.69), because there is no natural ordering. We *can* sort the points

by their magnitude (distance from the origin), so let us assume the points to be sorted as

$$|\underline{x}_1| \leq |\underline{x}_2| \leq \dots \leq |\underline{x}_N| \quad (\text{D.73})$$

Let us consider the problem in two dimensions and assume the distance metric to be Euclidean. Then for $|\underline{x}| > |\underline{x}_N|$, the region centered on \underline{x} will be a circle of radius at most $|\underline{x}| + |\underline{x}_N|$, in which case (D.70) becomes

$$|R(\underline{x})| \leq \pi(|\underline{x}| + |\underline{x}_N|)^2 \quad (\text{D.74})$$

and the estimated distribution from (D.71) becomes

$$\hat{p}_{\text{kNN}}(\underline{x}) = \frac{M}{R(\underline{x})N} \geq \frac{M}{N\pi(|\underline{x}| + |\underline{x}_N|)^2} \quad (\text{D.75})$$

So \hat{p} asymptotically goes as $1/x^2$, which seems normalizable, however this integral is now in *two* dimensions, not one. Converting to polar coordinates, we find

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \hat{p}_{\text{kNN}}(\underline{x}) dx_1 dx_2 \geq \int_{|\underline{x}_N|}^{\infty} \int_0^{2\pi} \hat{p}_{\text{kNN}}(r)r dr d\theta \quad (\text{D.76})$$

$$\geq \int_{|\underline{x}_N|}^{\infty} 2\pi \frac{M}{N\pi(|\underline{x}| + |\underline{x}_N|)^2} r dr \quad (\text{D.77})$$

$$\geq \int_{|\underline{x}_N|}^{\infty} \frac{2M}{4r^2} r dr = \int_{|\underline{x}_N|}^{\infty} \frac{M}{2r} dr = \infty \quad (\text{D.78})$$

So, yet again, not normalizable. This conclusion generalizes to n dimensions: the region size $|R(\underline{x})|$ will grow asymptotically as $|\underline{x}|^n$, which means that \hat{p} decays asymptotically as $1/|\underline{x}|^n$, but which integrates to ∞ over n -dimensional space.

Goodness of Fit to a Multi-Variate Normal Distribution: (Page 178)

Section 7.4 examined the question of whether a given set of data points comes from a particular distribution or family of distributions. Because the Normal/-Gaussian distribution is a relatively common occurrence, it is particularly relevant to be able to test whether a set of points is Normally distributed.

To test for Normality, we need to have some idea of what properties a set of normally-distributed data points must obey. A *moment* of a random variable, such as

$$\mathbb{E}[x^\eta] \quad \text{or} \quad \mathbb{E}\left[\left(\frac{x - \mu_x}{\sigma_x}\right)^\eta\right] \quad (\text{D.79})$$

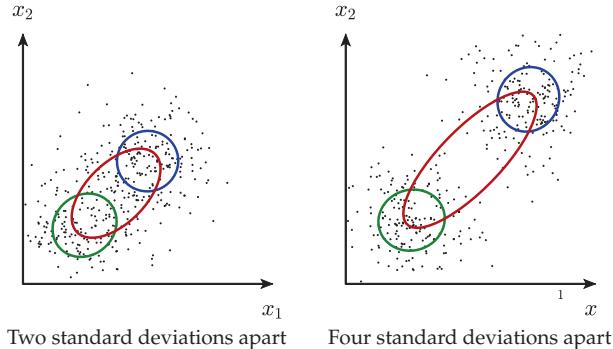


Fig. D.1. MULTIVARIATE NORMALITY: Suppose we know that our classes are Normal. Given a cluster of points (dots), are these two separate clusters (blue, green) or one large one (red)? If the separate clusters are far apart (right), it should be easy to design a statistical test to reveal that a single Gaussian fit (red) is a poor match, however when the clusters are closer together (left) this can be much less obvious.

as described in (B.12), is a scalar which captures some aspect of the overall distribution $p(\underline{x})$, such as its mean, variance, or degree of asymmetry. All such moments are known, analytically, in the Normal case, as listed in (B.13). Finding the sample moments, per (B.49), and comparing to the expected values for the Normal distribution, forms the basis of many of the statistical tests described in Section 7.4. However,

- The tests of Section 7.4 apply only to one-dimensional (scalar) x ;
- Nearly all pattern recognition problems involve very *high* dimensional (vector) \underline{x} .

The issue is of particular significance in unlabelled clustering, in Chapter 12, because the absence of labels means that we do not know whether a given set of points represents one larger cluster or two smaller ones, as illustrated in Figure D.1.

Given multivariate (multidimensional) data $\{\underline{x}_i\}$, we can define

$$\text{Multivariate Skewness: } \beta_3 = \frac{1}{N^2} \sum_i \sum_j \left((\underline{x}_i - \hat{\mu})^T \hat{\Sigma}^{-1} (\underline{x}_j - \hat{\mu}) \right)^3 \quad (\text{D.80})$$

$$\text{Multivariate Kurtosis: } \beta_4 = \frac{1}{N} \sum_i \left((\underline{x}_i - \hat{\mu})^T \hat{\Sigma}^{-1} (\underline{x}_i - \hat{\mu}) \right)^4 \quad (\text{D.81})$$

where $\hat{\mu}$ and $\hat{\Sigma}$ are the sample mean and sample covariance, as discussed in Section B.6. If \underline{x} is Normal, then these moments are distributed as

$$\beta'_3 = \frac{N}{6} \beta_3 \sim \chi^2_{n(n+1)(n+2)/6} \quad \beta'_4 = \frac{\beta_4 - (n(n+2)(N-1)/(N+1))}{\sqrt{8n(n+2)/N}} \sim \mathcal{N}(0, 1) \quad (\text{D.82})$$

In most of this text we are classifying a *single* feature point, such as the hypothesis testing of [Section 9.2](#),

$$\text{Measured Features } \underline{x} \xrightarrow{\text{Classify}} \underbrace{H_0}_{\text{Hypothesis } H_0: \text{Default}} \text{ versus } \underbrace{H_1}_{\text{Hypothesis } H_1: \text{Anomalous}} \quad (\text{D.83})$$

whereas for a test of *distribution*, as with the generalized hypothesis testing problem of [Case Study 4](#), we now want to classify the collective behaviour of the distribution implied by an *ensemble* of points:

$$\text{Distribution Attributes } \{\underline{x}_i\} \xrightarrow{\text{Classify}} \underbrace{H_0}_{\text{Hypothesis } H_0: \text{Normal}} \text{ versus } \underbrace{H_1}_{\text{Hypothesis } H_1: \text{Non-Normal}} \quad (\text{D.84})$$

In the same way that we cannot really describe what a defective part looks like in [Case Study 4](#), whereas we *do* know what a correct part looks like, similarly here we cannot say what sort of distribution “non”-Normal is (could be almost anything), from [\(D.82\)](#) we *do* know how normally-distributed variables behave. That is, we know $p(\beta'_3 | H_0)$, $p(\beta'_4 | H_0)$. Therefore our test for non-normality can be expressed as

$$\text{Classify } \{\underline{x}_i\} \text{ as non-Normal if } \beta'_3(\{\underline{x}_i\}) > \tau_3 \text{ or } |\beta'_4(\{\underline{x}_i\})| > \tau_4 \quad (\text{D.85})$$

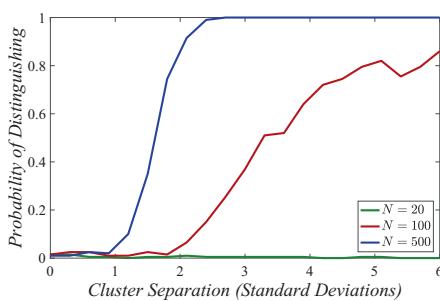
where the classification thresholds are based on confidence parameter² $\alpha = \mathbf{P}(e | H_0)$, by integrating over the tails of H_0 in the decision region of H_1 :

$$\int_{\tau_3}^{\infty} \chi_{n(n+1)(n+2)/6}^2(u) du = \alpha \quad 2 \int_{\tau_4}^{\infty} \mathcal{N}(u; 0, 1) du = 2Q(\tau_4) = \alpha \quad (\text{D.86})$$

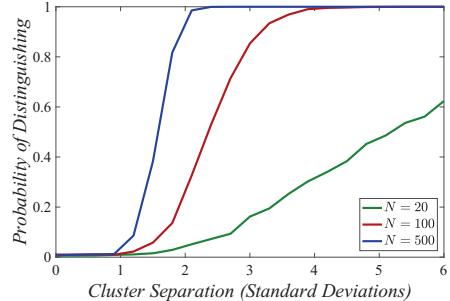
where the factor of 2 in the latter test stems from the fact that the distribution for β'_4 is two-sided, and we need to capture both tails.

The test of [\(D.85\)](#) gives rise to the left panel in [Figure D.2](#). The results are a little underwhelming — two clusters with 20 data points each appear to *never* be distinguished, and even at 100 points, well-separated clusters are distinguished only half of the time.

² A minor detail, perhaps ... the thresholds of [\(D.86\)](#) do not necessarily precisely match $\alpha = \mathbf{P}(e | H_0)$, because there are *two* tests listed in [\(D.85\)](#), and we would need to know *how* or *whether* these tests are correlated. This question could be answered computationally, by simulation.



Based on the Gaussian moments
of (D.80) and (D.81)



Based on the number of points
within the unit standard-deviation contour

Fig. D.2. TESTS OF MULTIVARIATE NORMALITY: We would like to know how distinguishable two clusters are, as a function of how far they are separated. Two sets of tests are shown, based on moments (left) or a distance threshold (right). The confidence parameter $\alpha = 0.01$, meaning that only 1% of the time should a Gaussian cluster accidentally be misclassified as non-Gaussian. Clearly a weaker confidence (larger α) would more easily detect non-normality, but would also more frequently mis-classify normally-distributed clusters.

For the purpose of clustering, knowing whether a set of points represents one compact cluster or two, a detailed test of Normality is not necessarily relevant, because we typically do not expect clusters to actually be Gaussian-distributed, rather we would like to distinguish between hypotheses of one compact Normal-like cluster as opposed to two compact Normal-like clusters. Motivated by the red contour in Figure D.1, a much simpler multi-variance test of Normality would be to ask what fraction ϑ of points are within a unit distance of the mean, a topic already explored in Example 4.5 and plotted as a function of dimension on page 71. That is,

$$\vartheta = \mathbf{P}((\underline{x} - \hat{\mu})^T \hat{\Sigma}^{-1} (\underline{x} - \hat{\mu}) < 1) \quad (\text{D.87})$$

For N Normal data points $\{\underline{x}_i\}$, the *number* M of data points inside one standard deviation is just binomial,

$$\mathbf{P}(M \text{ of } N \text{ samples inside of the unit-contour}) = \binom{N}{M} \vartheta^M (1 - \vartheta)^{N-M} \quad (\text{D.88})$$

which, for N sufficiently large, can be approximated as $M \sim \mathcal{N}(N\vartheta, N\vartheta(1 - \vartheta))$, and can then be normalized to

$$\text{Fraction of points inside the unit-distance contour} \sim \mathcal{N}(\vartheta, \vartheta(1 - \vartheta)/N) \quad (\text{D.89})$$

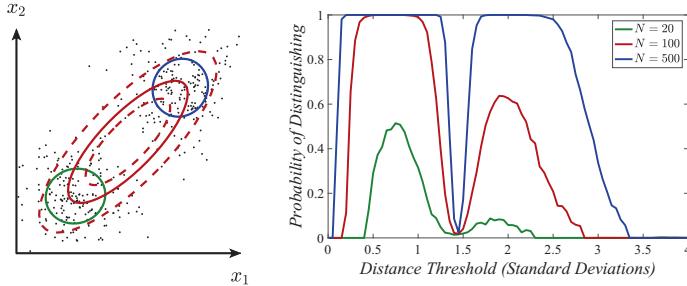


Fig. D.3. TEST OF MULTIVARIATE NORMALITY: The distance-test of Figure D.2 was based on a distance contour at $\zeta = 1$, however other contours (dashed ellipses, left) might be more (or less) discriminating in sensing non-normality. Indeed, for the two clusters shown, a smaller contour of distance 0.7 standard deviations is significantly more sensitive. Oddly at 1.4 standard deviations the test appears to fail: at that larger contour the two clusters have the *same* fraction of points within the contour as a Gaussian having the same mean and covariance, and therefore the distinguishability is zero.

The resulting hypothesis test is analogous to the earlier test based on moments in (D.85):

$$\text{Classify } \{\underline{x}_i\} \text{ as non-Normal if } \left| \frac{M/N - \vartheta}{\sqrt{\vartheta(1-\vartheta)/N}} \right| > \tau \quad 2Q(\tau) = \frac{\alpha}{2} \quad (\text{D.90})$$

with a factor of two before Q because tails in both directions are being integrated, just as in (D.86). Just as a reminder, we do need to specify the value of ϑ in (D.90). For the two-dimensional context of Figure D.1, from Example 4.5 we know that $\vartheta = 39\%$.

The very simple test of (D.90) gives rise to the results in the right panel in Figure D.2.

Finally, there is nothing magical, so to speak, about counting the number of points inside a distance of 1.0 from the mean. It is entirely plausible that some other distance contour could be more discriminating. That is, we could generalize (D.87) as

$$\vartheta(\zeta) = \mathbf{P}\left((\underline{x} - \hat{\mu})^T \hat{\Sigma}^{-1} (\underline{x} - \hat{\mu}) < \zeta^2\right) \quad (\text{D.91})$$

The cluster separability as a function of ζ is shown in Figure D.3. The previous distance contour $\zeta = 1$ is indeed not the most discriminating, for the particular cluster separation in the test undertaken here. In practice, not knowing how far apart two clusters might be, it would probably be prudent to test at *several* difference choices of ζ .

Accuracy of NN and k NN Classifiers:

(Page 205)

Since MAP maximizes the posterior probability $\mathbf{P}(C_\kappa | \underline{x})$, it minimizes the probability of the *not* chosen classes

$$\mathbf{P}(e | \underline{x}) = \sum_{j \neq \kappa} \mathbf{P}(C_j | \underline{x}) = 1 - \mathbf{P}(C_\kappa | \underline{x}) \quad (\text{D.92})$$

therefore we know that the MAP classifier must minimize the probability of error. It turns out that we can make very similar claims for the nearest neighbour (NN) and k NN classifiers.

From the derivation of the k NN distribution estimation³ in Section 7.3.3,

$$\hat{p}_{\text{kNN}}(\underline{x} | C_\kappa) = \frac{k}{|R_\kappa(\underline{x})| \cdot N_\kappa} \quad (\text{D.93})$$

therefore

$$\frac{1}{|R_\kappa(\underline{x})|} = \frac{N_\kappa \cdot \hat{p}_{\text{kNN}}(\underline{x} | C_\kappa)}{k} \quad (\text{D.94})$$

Under the assumption that the overall frequency of data points mirrors the class prior probabilities,⁴ then

$$P(C_\kappa) = N_\kappa / \sum_j N_j \quad (\text{D.95})$$

thus

$$\frac{1}{|R_\kappa(\underline{x})|} = \frac{\mathbf{P}(C_\kappa) \cdot \hat{p}_{\text{kNN}}(\underline{x} | C_\kappa)}{k \sum_j N_j} \quad (\text{D.96})$$

In the limit of many data points ($N \rightarrow \infty$), k large enough to have an accurate estimation of probability ($k \rightarrow \infty$) but small enough to be sensitive to local variations ($k/N \rightarrow 0$), then the k NN distribution converges to the underlying density; applying Bayes' rule then leads to

$$\frac{1}{|R_\kappa(\underline{x})|} = \mathbf{P}(C_\kappa | \underline{x}) \cdot \underbrace{\frac{p(\underline{x})}{k \sum_j N_j}}_{\text{Not a function of } \kappa} \quad (\text{D.97})$$

³ Note that symbols k and M are interchangeable: k is the number of closest points considered in the k NN classifier or distribution estimator, M is the number of points in a histogram bin. In the context of this derivation they are the same. Since we are deriving a result related to k NN, only letter k will be used here.

⁴ The frequency of data points might *not* mirror the class prior probabilities if the data had been deliberately resampled or rebalanced, for example, as discussed in Section 9.1.

where, dropping the latter term, we find

$$\frac{1}{|R_\kappa(\underline{x})|} \propto \mathbf{P}(C_\kappa | \underline{x}) \quad (\text{D.98})$$

from which it follows that choosing κ to minimize $|R_\kappa|$, which means choosing the class κ whose k th point is closest, is equivalent to maximizing $\mathbf{P}(C_\kappa | \underline{x})$ which is, of course, the MAP rule. That is, asymptotically k NN converges to MAP.

For nearest neighbour (NN), suppose we consider a region \mathcal{R} containing points from classes C_1 and C_2 . To keep the derivation conceptually simple, let \mathcal{R} be chosen sufficiently small so that $p(\underline{x} | C_\kappa)$ is constant over \mathcal{R} . Defining the class probabilities

$$\mathbf{P}_\kappa = \frac{p(\underline{x} | C_\kappa) \cdot \mathbf{P}(C_\kappa)}{p(\underline{x} | C_1) \cdot \mathbf{P}(C_1) + p(\underline{x} | C_2) \cdot \mathbf{P}(C_2)} \quad (\text{D.99})$$

allows us to talk about the relative frequency of points in \mathcal{R} for the two classes.

The MAP rule is simple, classifying all points in \mathcal{R} based on the larger of \mathbf{P}_1 and \mathbf{P}_2 , meaning that

$$\mathbf{P}_{\text{MAP}}(e | \mathcal{R}) = \min(\mathbf{P}_1, \mathbf{P}_2) \quad (\text{D.100})$$

Similarly the NN rule is simple. In the limit as the number of data points $N \rightarrow \infty$ statistical fluctuations disappear⁵ and a fraction \mathbf{P}_1 of \mathcal{R} will be closest to points from C_1 , and a fraction \mathbf{P}_2 of \mathcal{R} will be closest to points from C_2 :

$$\mathbf{P}_{\text{NN}}(e | \mathcal{R}) = \mathbf{P}(\text{Say } C_1 | C_2) \cdot \mathbf{P}(C_2) + \mathbf{P}(\text{Say } C_2 | C_1) \cdot \mathbf{P}(C_1) \quad (\text{D.101})$$

$$= \mathbf{P}_1 \cdot \mathbf{P}_2 + \mathbf{P}_2 \cdot \mathbf{P}_1 = 2\mathbf{P}_1\mathbf{P}_2 \quad (\text{D.102})$$

So the ratio of the NN error to the MAP error is

$$\frac{\mathbf{P}_{\text{NN}}(e | \mathcal{R})}{\mathbf{P}_{\text{MAP}}(e | \mathcal{R})} = \frac{2\mathbf{P}_1\mathbf{P}_2}{\min(\mathbf{P}_1, \mathbf{P}_2)} = \begin{cases} 2\mathbf{P}_2 & \mathbf{P}_1 < \mathbf{P}_2 \\ 2\mathbf{P}_1 & \mathbf{P}_1 > \mathbf{P}_2 \end{cases} \quad (\text{D.103})$$

That is, in the limit of large numbers of points, the two-class NN error is twice the MAP error. This conclusion also applies, but not derived here, to the case of $K > 2$ classes.

It is important to keep in mind that the above results apply asymptotically, in the limit of large N , large k , small k/N . In many practical cases, involving limited data or classes having noisy outliers, the behaviour can be quite different from the asymptotic case.

⁵ The statement sounds vague, but is essentially stating (D.65). That is, for binomial behaviours, like sampling points from two classes, the variance of the estimated probability goes to zero as the number of points grows.

Memoryless Arrivals and Counting Processes: (Page 262)

It is fairly common to wish to measure *events* over time: people passing through a door, lightning flashes in a storm, clicks in a Geiger counter, or cars passing along a road. If the events (people, cars etc.) are not connected to one another (not travelling as a group, say), then their arrival times have no connection to one another, in which case we say that the process is *memoryless*. In other words, having waited for a while for the next arrival does *not* make it more likely that one is coming soon, since the appearance of the next arrival is unrelated to when the previous one came.

Indeed, suppose that the time to the next arrival is given by distribution $p(t)$. If the arrivals are memoryless, then the arrival statistics must be unaffected by a delay:

$$p(t \mid \text{no arrival up to time } \tau) = p(t - \tau) \quad (\text{D.104})$$

But $p(t \mid \text{no arrival up to time } \tau)$ must also just be the tail of the original distribution, with the absence of arrivals up to time τ removed and renormalized, thus

$$p(t \mid \text{no arrival up to time } \tau) = p(t) / \int_{\tau}^{\infty} p(s) ds \quad (\text{D.105})$$

Equating (D.104) and (D.105) we have

$$p(t) = p(t - \tau) \int_{\tau}^{\infty} p(s) ds \quad (\text{D.106})$$

Differentiating (D.106) with respect to τ we obtain

$$0 = p(t - \tau) \cdot -p(\tau) + \dot{p}(t - \tau) \cdot -1 \cdot \int_{\tau}^{\infty} p(s) ds \quad (\text{D.107})$$

The result in (D.107) must hold true for all values of τ , so let's make things as simple as possible and select $\tau = 0$:

$$0 = -p(t) \cdot p(0) - \dot{p}(t) \underbrace{\int_0^{\infty} p(s) ds}_{\text{equals one, since integrating a distribution}} \quad (\text{D.108})$$

and so we arrive at

$$\dot{p}(t) = -p(t) \cdot p(0) \quad (\text{D.109})$$

This is a simple, first-order, linear differential equation, whose solution is the exponential

$$p(t) = \begin{cases} 0 & t < 0 \\ p(0) \cdot \exp(-p(0)t) & t \geq 0 \end{cases} \quad (\text{D.110})$$

We should validate that the distribution is normalized:

$$\int_{-\infty}^{\infty} p(t) dt = 1 \rightarrow \int_0^{\infty} p(0) \cdot \exp(-p(0)t) dt = 1 \quad (\text{D.111})$$

To clean up the notation, we will let the distribution be described by single parameter $\lambda = p(0)$.

The remarkable conclusion, then, of the preceding development is that *any* set of independent (memoryless) arrivals *must* have inter-arrival times which are exponentially distributed. Having established the fundamental role of the exponential distribution, the next question is how we would like to represent a sequence of arrivals. We have two basic options:

1. Based on **Time**: Measuring the *time* between arrivals, or the time $t_{\lambda}(r)$ until the r th arrival. This is a *continuous* (real) value.
2. Based on **Counts**: Measuring the *number* of arrivals over some time period, the counting process $R_{\lambda}(t)$ measuring the number of arrivals up to time t . This is a *discrete* (integer) value.

Both $t_{\lambda}(r)$ and $R_{\lambda}(t)$ are fully characterized by the arrival rate λ .

We begin with the statistics of $t_{\lambda}(1)$. Random variable $t_{\lambda}(1)$, the time of the first arrival, is the same as the time of the *next* arrival, which we know from (D.110):

$$p(t_{\lambda}(1)) = \lambda \exp(-\lambda t_{\lambda}(1)) \mu(t) \quad (\text{D.112})$$

where $\mu(t)$ is the unit step function

$$\mu(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases} \quad (\text{D.113})$$

The second arrival time, $t_{\lambda}(2)$, is the sum of the first two inter-arrival times. As derived later in this Appendix, its corresponding distribution is therefore the convolution of $p(t)$ with itself:

$$p(t_{\lambda}(2)) = \int_{-\infty}^{\infty} \lambda e^{-\lambda \tau} \mu(\tau) \cdot \lambda e^{-\lambda(t_{\lambda}(2)-\tau)} \mu(t_{\lambda}(2) - \tau) d\tau \quad (\text{D.114})$$

The two step functions allow the integral to be truncated from zero to $t_{\lambda}(2)$:

$$\begin{aligned} p(t_{\lambda}(2)) &= \int_0^{t_{\lambda}(2)} \lambda e^{-\lambda \tau} \cdot \lambda e^{-\lambda(t_{\lambda}(2)-\tau)} d\tau = \int_0^{t_{\lambda}(2)} \lambda^2 e^{-\lambda(t_{\lambda}(2))} d\tau \\ &= \lambda^2 t_{\lambda}(2) \cdot e^{-\lambda(t_{\lambda}(2))} \end{aligned} \quad (\text{D.115})$$

Further convolutions would give rise to the distribution of the r th arrival:

$$p(t_\lambda(r)) = \lambda^r (t_\lambda(r))^{r-1} e^{-\lambda(t_\lambda(r))} / (r-1)! \quad t_\lambda(r) \geq 0 \quad (\text{D.116})$$

Turning the problem around, instead of asking *when* the next arrival comes, we ask *how many* arrivals have come. We start with the probability of *no* arrivals:

$$\mathbf{P}(R_\lambda(t) = 0) = \mathbf{P}(\text{There was no arrival until time } t) \quad (\text{D.117})$$

$$= \mathbf{P}(\text{The first arrival is after time } t) \quad (\text{D.118})$$

$$= \int_t^\infty \lambda e^{-\lambda\tau} d\tau = e^{-\lambda t} \quad (\text{D.119})$$

Next,

$$\mathbf{P}(R_\lambda(t) = 1) = \mathbf{P}(\text{There was exactly one arrival until time } t) \quad (\text{D.120})$$

$$= \int_0^t d\tau p(\text{Arrival at time } \tau) \mathbf{P}(\text{Next arrival after time } t) \quad (\text{D.121})$$

$$= \int_0^t d\tau \lambda e^{-\lambda\tau} \mu(\tau) \int_{t-\tau}^\infty d\sigma \lambda e^{-\lambda\sigma} \mu(\sigma) \quad (\text{D.122})$$

$$= \int_0^t d\tau \lambda e^{-\lambda\tau} \cdot \lambda \frac{-1}{\lambda} (0 - e^{-\lambda(t-\tau)}) \quad (\text{D.123})$$

$$= \lambda t e^{-\lambda t} \quad (\text{D.124})$$

Continuing the above derivation, the more general solution would be found as

$$\mathbf{P}(R_\lambda(t) = r) = (\lambda t)^r e^{-\lambda t} / r! \quad (\text{D.125})$$

which is a Poisson distribution.

Distance from a point to a Hyperplane:

(Page 270)

From Pythagorus/orthogonality, the shortest Euclidean distance from a point \underline{x}' to a hyperplane $g(\underline{x}) = 0$ must be parallel to the hyperplane normal vector \underline{w} . Therefore, as shown in Figure D.4, the line $\underline{x}' + \alpha \underline{w}$ must intersect the hyperplane at the point of shortest distance:

$$h(\underline{x}' + \alpha \underline{w}) = 0 \quad (\text{D.126})$$

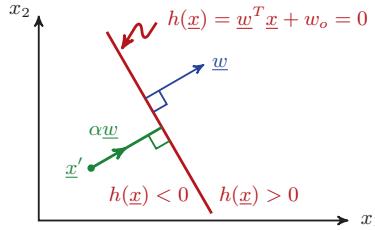


Fig. D.4. HYPERPLANE GEOMETRY: The basic geometry of a linear discriminant, from Figure 10.3.

Therefore, expanding the form of function $h()$,

$$\underline{w}^T(\underline{x}' + \alpha \underline{w}) + w_o = 0 \quad (\text{D.127})$$

from which we can solve for α as

$$\underline{w}^T \alpha \underline{w} = -\underline{w}^T \underline{x}' - w_o = -h(\underline{x}') \rightarrow \alpha = -\frac{h(\underline{x}')}{|\underline{w}|^2} \quad (\text{D.128})$$

Therefore the distance from any point \underline{x}' to a hyperplane is

$$d_E(\underline{x}', \{h(\underline{x}) = 0\}) = |\alpha| \cdot |\underline{w}| = \frac{|h(\underline{x}')|}{|\underline{w}|} \quad (\text{D.129})$$

From [Section 6.1](#), it is not at all clear that a Euclidean distance is necessarily the appropriate distance metric. Suppose that we believe that

$$\underline{r} = Q\underline{x} \quad (\text{D.130})$$

is a normalized/rescaled space such that the Euclidean distance applies on \underline{r} :

$$\text{Distance}_Q^2(\underline{x}, \underline{x}') = d_E^2(\underline{r}, \underline{r}') = (\underline{r} - \underline{r}')^T(\underline{r} - \underline{r}') = (\underline{x} - \underline{x}')^T Q^T Q (\underline{x} - \underline{x}') \quad (\text{D.131})$$

In the transformed space, the transformed linear discriminant $\bar{h}()$ is

$$\bar{h}(\underline{r}) = h(Q^{-1}\underline{r}) = \underline{w}^T(Q^{-1}\underline{r}) + w_o = (Q^{-T}\underline{w})^T \underline{r} + w_o, \quad (\text{D.132})$$

that is, having a normal vector of $Q^{-T}\underline{w}$. Therefore the closest distance in the transformed space is

$$d_Q(\underline{x}', \{h(\underline{x}) = 0\}) = d_E(Q\underline{x}', \{\bar{h}(\underline{r}') = 0\}) = \frac{|\bar{h}(Q\underline{x}')|}{|Q^{-T}\underline{w}|} = \frac{|h(\underline{x}')|}{|Q^{-T}\underline{w}|} \quad (\text{D.133})$$

Analytical Solution to Squared-Linear Problems:

(Page 275)

Suppose you wish to learn a vector of unknowns $\underline{\theta}$ on the basis of a set of measurements y_i , each measurement a linear function of $\underline{\theta}$:

$$y_i = q_i^T \underline{\theta} \quad (\text{D.134})$$

We propose to learn $\underline{\theta}$ on the basis of minimizing the mean-squared inconsistency with the measurements, that is

$$\text{Optimal Estimate } \hat{\underline{\theta}} = \arg_{\underline{\theta}} \min \sum_i (y_i - q_i^T \underline{\theta})^2 \quad (\text{D.135})$$

Where (D.134) was written for an individual scalar measurement, we can stack the measurements into a vector to rewrite (D.134) as

$$\underline{y} = Q^T \underline{\theta} \quad (\text{D.136})$$

in which case the mean-squared inconsistency of (D.135) becomes

$$\hat{\underline{\theta}} = \arg_{\underline{\theta}} \min (\underline{y} - Q^T \underline{\theta})^T (\underline{y} - Q^T \underline{\theta}) \quad (\text{D.137})$$

We can find the minimum by taking derivatives and setting to zero. From Table A.1,

$$\begin{aligned} \frac{\partial}{\partial \underline{\theta}} (\underline{y} - Q^T \underline{\theta})^T (\underline{y} - Q^T \underline{\theta}) &= \frac{\partial}{\partial \underline{\theta}} \left\{ \underline{y}^T \underline{y} - 2\underline{y}^T Q^T \underline{\theta} + \underline{\theta}^T Q Q^T \underline{\theta} \right\} \\ &= 0 - 2\underline{y}^T Q^T + 2\underline{\theta}^T Q Q^T \end{aligned}$$

Therefore, setting the derivative to zero and solving for $\hat{\underline{\theta}}$:

$$-2\underline{y}^T Q^T + 2\hat{\underline{\theta}}^T Q Q^T = 0 \xrightarrow{\text{Transpose}} Q Q^T \hat{\underline{\theta}} = Q \underline{y} \rightarrow \hat{\underline{\theta}} = (Q Q^T)^{-1} Q \underline{y}$$

That is, we have a clean, closed-form expression

$$\text{Optimal Estimate } \hat{\underline{\theta}} = (Q Q^T)^{-1} Q \underline{y} \quad (\text{D.138})$$

for the unique, optimal estimate given a mean-squared penalty on the error. It is important to emphasize that the solution in (D.138) is quite broad, and applies to *any* problem of the form in (D.134), which includes linear regression, polynomial regression, and hyperplane discriminants.

The result in (D.138) is guaranteed to produce a solution as long as $(Q Q^T)$ is invertible. If $\underline{\theta}$ contains $(n+1)$ parameters, then normally having at least $(n+1)$ measurements in (D.134) guarantees the invertibility of $(Q Q^T)$. Technically the invertibility of $(Q Q^T)$ requires Q to be full row rank (equivalently Q^T to be full column rank), which means that we need at least $(n+1)$ *different* measurements — not the *values* of the measurements in \underline{y} , rather the *definitions* of the measurements in Q . The requirement is *linear independence*: no row of Q must be able to be written as a linear function of the other rows of Q .

Maximum Likelihood and Balanced Accuracy: (Page 241)

Suppose we are given class distributions

$$p(\underline{x} | C_1) \quad \dots \quad p(\underline{x} | C_K) \quad (\text{D.139})$$

We wish to find the classifier $g()$ maximizing the balanced accuracy, thus minimizing the summed errors

$$\sum_{\kappa=1}^K \mathbf{P}_g(e | C_\kappa) \quad (\text{D.140})$$

We will implicitly specify the classifier in terms of its decision regions:

$$\text{Decision Region for Class } C_\kappa = \mathcal{R}_\kappa = \{\underline{x} | g(\underline{x}) = C_\kappa\} \quad (\text{D.141})$$

Then the summed errors can be evaluated as

$$S = \sum_{\kappa=1}^K \mathbf{P}_g(e | C_\kappa) = \sum_{\kappa=1}^K \sum_{j \neq \kappa} \int_{\mathcal{R}_j} p(\underline{x} | C_\kappa) d\underline{x} = \sum_{j=1}^K \int_{\mathcal{R}_j} \underbrace{\sum_{\kappa \neq j} p(\underline{x} | C_\kappa)}_{\text{Minimize Integrand}} d\underline{x} \quad (\text{D.142})$$

S is minimized by minimizing the integrand in (D.142) for each value of \underline{x} , which is accomplished by *omitting* the largest $p(\underline{x} | C_\kappa)$ from the sum, which is accomplished by *selecting* the largest $p(\underline{x} | C_\kappa)$ in classification, which is the Maximum Likelihood classifier.

Kernels and Transformations: (Page 361)

In Chapter 10 (Support Vector Machines) and Chapter 12 (Kernel K-Means), we used a nonlinear kernel Φ_φ ,

$$\Phi_\varphi(\underline{x}, \underline{x}') = \underline{x} \bullet \underline{x}' = \varphi(\underline{x}) \bullet \varphi(\underline{x}') = \sum_j \varphi_j(\underline{x}) \varphi_j(\underline{x}') \quad (\text{D.143})$$

to describe a transformation $\varphi()$ from a feature \underline{x} to some other high-dimensional \bar{x} .

It is conceptually attractive to suppose that we select transformations $\{\varphi_j()\}$ from which the kernel Φ is found. In actual fact it is much simpler to select the kernel directly, but with a notion that this identifies the associated transformation, however in actual fact the transformations $\{\varphi_j()\}$ are *not* uniquely defined by kernel Φ .

Since working with nonlinear transformations is challenging, we will demonstrate this in the linear case. We begin with a kernel Φ_A based on a transformation matrix A :

$$\Phi_A(\underline{x}, \underline{x}') = (A^T \underline{x})^T (A^T \underline{x}') = \underline{x}^T A A^T \underline{x}' = \underline{x}^T B \underline{x}' \quad (\text{D.144})$$

That is, if \underline{a}_j represents the j th column of matrix A , then each linear transformation is given by

$$\varphi_j(x) = \underline{a}^T \underline{x} = \underline{a} \bullet \underline{x} \quad (\text{D.145})$$

Note that A can be any matrix, and is not assumed to be square.

In general, we would find the eigendecomposition of B as

$$B = V \Lambda V^{-1}, \quad (\text{D.146})$$

however since B is real and symmetric the eigenvectors in V lead to an orthogonal matrix, thus

$$B = V \Lambda V^T \quad \text{since} \quad V^T = V^{-1} \quad (\text{D.147})$$

We can use the eigendecomposition to find a transformation A from B :

$$B = V \Lambda V^T \longrightarrow A = V \Lambda^{\frac{1}{2}} \longrightarrow A A^T = B \quad (\text{D.148})$$

So far so good. However it turns out that A is not unique. That is, given matrix B ,

$$\text{if } A A^T = B \quad \text{then} \quad A Q Q^T A^T = B \quad \text{for any matrix such that} \quad Q Q^T = I \quad (\text{D.149})$$

That is, the kernel associated with the transformations in A is the *same* kernel associated with the transformations in (AQ) , where Q is a matrix square-root of I , of which there are *infinitely* many! For example, as a very simple illustration in the 2×2 case,

$$Q = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \longrightarrow Q Q^T = I \quad \text{for any } \theta \quad (\text{D.150})$$

To be fair, Q does not completely change the transformations, rather (as a rotation matrix) more of a re-ordering or re-mixing of the transformations. Nevertheless, whereas the chosen kernel Φ_φ is explicit and well defined, the associated transformations $\varphi_j()$ are, at best, *implicit*.

Layers of Linear Functions:

(Page 371)

What is the net result of a network of operations, with one output from one layer acting as the input to the next? Suppose that $f_j()$ is the function operating at the j th layer, then

$$\underline{x}_1 \longrightarrow \underline{x}_2 = f_1(\underline{x}_1) \longrightarrow \underline{x}_3 = f_2(\underline{x}_2) \longrightarrow \dots \longrightarrow \underline{x}_n = f_{n-1}(\underline{x}_{n-1}) \quad (\text{D.151})$$

If the functions are nonlinear, then it is not possible to reach any particular conclusion, in general, regarding the composited function

$$x_n = f_{n-1}(f_{n-2}(\dots f_1(x_1))) \quad (\text{D.152})$$

although such iterated functions are essentially like dynamic systems, in the sense of \underline{x}_t being a state evolving over time t , which have been widely studied.

However in the special case that the functions are linear, the situation is very different. First, each linear function $f_j()$ can be represented by a matrix multiplication F_j :

$$\underline{x}_1 \longrightarrow \underline{x}_2 = F_1 \underline{x}_1 \longrightarrow \underline{x}_3 = F_2 \underline{x}_2 \longrightarrow \dots \longrightarrow \underline{x}_n = F_{n-1} \underline{x}_{n-1} \quad (\text{D.153})$$

which we can rewrite as

$$x_n = F_{n-1} \cdot \dots \cdot F_2 \cdot F_1 \cdot x_1 = \bar{F} \underline{x}_1 \quad (\text{D.154})$$

That is, *regardless* how many successive layers of functions there are, and regardless whether the dimensionality of \underline{x} grows, shrinks, or stays the same from layer to layer, the net result of many layers of linear operations is *still* linear. In other words, there is *no* difference in the range of behaviours that *one* linear operation can represent, as opposed to *many*.

The argument is slightly more nuanced if we constrain the *type* of linear function. For example, the $\{F_j\}$ could all be constrained to be *local* linear functions, whereas the layered behaviour F could be *nonlocal*. Or the $\{F_j\}$ could all be constrained to be sparse, and yet it would be possible for the layered behaviour F to be dense.

Eigendecompositions of Covariances:

(Page 403)

Given a covariance Σ , from (A.32) its eigendecomposition can be written as

$$\Sigma \underline{v}_i = \lambda_i \underline{v}_i, \quad 1 \leq i \leq n \quad \longrightarrow \quad V = \begin{bmatrix} | & | \\ \underline{v}_1 & \cdots & \underline{v}_n \\ | & | \end{bmatrix} \quad \Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \quad (\text{D.155})$$

$$\longrightarrow \quad \Sigma V = V \Lambda \quad (\text{D.156})$$

Suppose we select any two vectors $\underline{a}, \underline{b} \in \mathbb{R}^n$. Then $\underline{a}^T \Sigma \underline{b}$ is a scalar, and the transpose of a scalar is just the scalar itself. Therefore

$$\underline{a}^T \Sigma \underline{b} = (\underline{a}^T \Sigma \underline{b})^T = \underline{b}^T \Sigma^T \underline{a} \quad (\text{D.157})$$

Since Σ is a covariance it is symmetric, therefore (D.157) becomes

$$\underline{a}^T \Sigma \underline{b} = \underline{b}^T \Sigma \underline{a} \quad (\text{D.158})$$

Now suppose that $\underline{a}, \underline{b}$ are eigenvectors $\underline{v}_i, \underline{v}_j$. Applying the eigendecomposition of Σ :

$$\underline{v}_i^T \Sigma \underline{v}_j = \underline{v}_j^T \Sigma \underline{v}_i \rightarrow \underline{v}_i^T \underline{v}_j \cdot \lambda_j = \underline{v}_j^T \underline{v}_i \cdot \lambda_i \rightarrow \underline{v}_i^T \underline{v}_j \cdot (\lambda_i - \lambda_j) = 0 \quad (\text{D.159})$$

So if \underline{v}_i and \underline{v}_j were chosen to correspond to different eigenvalues $\lambda_i \neq \lambda_j$, then from (D.159) it follows that $\underline{v}_i^T \underline{v}_j = 0$. That is, the eigenvectors corresponding to different eigenvalues must be orthogonal.

As long as all of the eigenvalues are positive (Σ is positive-definite) then V must be invertible, so that (D.156) becomes

$$\Sigma V = V \Lambda \rightarrow \Sigma V V^{-1} = V \Lambda V^{-1} \rightarrow \Sigma = V \Lambda V^{-1} \quad (\text{D.160})$$

From (A.18) we know that if A, B, C are all invertible, then

$$(ABC)^{-1} = C^{-1} B^{-1} A^{-1}, \quad (\text{D.161})$$

therefore (D.160) becomes

$$\Sigma = V \Lambda V^{-1} \rightarrow \Sigma^{-1} = (V \Lambda V^{-1})^{-1} = (V^{-1})^{-1} \Lambda^{-1} V^{-1} \rightarrow \Sigma^{-1} = V \Lambda^{-1} V^{-1} \quad (\text{D.162})$$

The Algebra of Singular Covariance Matrices:

(Page 414)

The bulleted list on page 414 identified a number of criteria by which a covariance would be singular:

- *The determinant is zero:* $\det(\Sigma) = 0$:

By definition, from (A.21) *any* matrix is singular if its determinant is zero.

- *At least one eigenvalue of Σ is zero:*

From (A.28),

$$\det(\Sigma) = \prod_{i=1}^n \lambda_i \rightarrow \det(\Sigma) = 0 \text{ if and only if } \lambda_i = 0 \text{ for some } i \quad (\text{D.163})$$

Therefore a zero eigenvalue implies a zero determinant which implies singularity.

- *The variance of some linear function of \underline{x} is zero:*

Interpreted mathematically, the claim is stating that

$$\text{There exists an } \underline{f} \neq \underline{0} \text{ such that } \text{var}(\underline{f}^T \underline{x}) = \underline{f}^T \Sigma \underline{f} = 0 \quad (\text{D.164})$$

If we take the eigendecomposition of $\Sigma = V \Lambda V^T$, as in (A.43), then

$$0 = \underline{f}^T \Sigma \underline{f} = \underline{f}^T V \Lambda V^T \underline{f} = \underline{g}^T \Lambda \underline{g} = \sum_{i=1}^n g_i^2 \cdot \lambda_i \quad (\text{D.165})$$

But $\underline{f} \neq \underline{0}$ and orthogonal matrix V is a rotation matrix, therefore $\underline{g} = V^T \underline{f} \neq \underline{0}$. Since covariance eigenvalues are non-negative, the only way we can have

$$\sum_{i=1}^n g_i^2 \cdot \lambda_i = 0 \quad \text{but} \quad \sum_{i=1}^n g_i^2 \neq 0 \quad (\text{D.166})$$

is for at least one eigenvalue to equal zero. The remaining two conditions follow from this just-derived general case.

- *At least one of the variances (diagonal elements of Σ) is zero:*

Suppose $\Sigma_{i,i} = 0$, then if

$$\underline{f} = \underline{e}_i, \text{ the } i\text{th unit vector, that is } f_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \quad (\text{D.167})$$

then, by definition, $\text{var}(\underline{f}^T \underline{x}) = 0$, as in (D.164).

- *Two of the variables in \underline{x} are perfectly correlated or anti-correlated ($\rho_{i,j} = \pm 1$):*

The two-by-two covariance matrix corresponding to x_i and x_j is

$$\begin{bmatrix} x_i \\ x_j \end{bmatrix} \sim \begin{bmatrix} \sigma_i^2 & \rho_{i,j} \cdot \sigma_i \sigma_j \\ \rho_{i,j} \cdot \sigma_i \sigma_j & \sigma_j^2 \end{bmatrix} \quad (\text{D.168})$$

However for $\rho_{i,j} = \pm 1$ we can easily show that this matrix has a zero eigenvalue, since

$$\underbrace{\begin{bmatrix} \sigma_i^2 & \sigma_i \sigma_j \\ \sigma_i \sigma_j & \sigma_j^2 \end{bmatrix}}_{\rho_{i,j}=1} \cdot \underbrace{\begin{bmatrix} -\sigma_i \sigma_j \\ \sigma_i^2 \end{bmatrix}}_{\text{Eigenvector}} = \underline{0} \quad \underbrace{\begin{bmatrix} \sigma_i^2 & -\sigma_i \sigma_j \\ -\sigma_i \sigma_j & \sigma_j^2 \end{bmatrix}}_{\rho_{i,j}=-1} \cdot \underbrace{\begin{bmatrix} \sigma_i \sigma_j \\ \sigma_i^2 \end{bmatrix}}_{\text{Eigenvector}} = \underline{0} \quad (\text{D.169})$$

meaning that

$$\text{var}(-\rho_{i,j}\sigma_i\sigma_jx_i + \sigma_i^2x_j) = 0 \quad (\text{D.170})$$

So we can define f from (D.164) as

$$f_k = \begin{cases} -\rho_{i,j} \cdot \sigma_i\sigma_j & k = i \\ \sigma_i^2 & k = j \\ 0 & \text{Otherwise} \end{cases} \rightarrow \text{such that } \text{var}(f^T \underline{x}) = 0. \quad (\text{D.171})$$

Linear Transformations of Random Vectors:

(Page 415)

Suppose we are given a random vector \underline{x} . Then for any linear transformation $T\underline{x} + \underline{\alpha}$ we can first derive the mean

$$\text{Mean}(T\underline{x} + \underline{\alpha}) = \mathbb{E}[T\underline{x} + \underline{\alpha}] = \mathbb{E}[T\underline{x}] + \mathbb{E}[\underline{\alpha}] = T \cdot \mathbb{E}[\underline{x}] + \underline{\alpha} \quad (\text{D.172})$$

where the mathematics of expectations follow from (B.6); that is, that $\mathbb{E}[\underline{\alpha}] = \underline{\alpha}$ and $\mathbb{E}[T\underline{x}] = T\mathbb{E}[\underline{x}]$, since both $\underline{\alpha}$ and T are just constant (not random). Therefore

$$\text{Mean}(T\underline{x} + \underline{\alpha}) = T \cdot \text{Mean}(\underline{x}) + \underline{\alpha} \quad (\text{D.173})$$

That is, applying a linear transformation to a random vector applies the same linear transformation to the mean.

Next, from Appendix B.4, recall the definition for a covariance as

$$\text{Cov}(\underline{r}) = \mathbb{E}[(\underline{r} - \mathbb{E}[\underline{r}])(\dots)^T] \quad (\text{D.174})$$

Therefore the covariance of $(T\underline{x} + \underline{\alpha})$ transforms as

$$\begin{aligned} \text{Cov}(T\underline{x} + \underline{\alpha}) &= \mathbb{E}[(T\underline{x} + \underline{\alpha} - \mathbb{E}[T\underline{x} + \underline{\alpha}])(\dots)^T] = \mathbb{E}[(T\underline{x} - T\mu)(\dots)^T] \\ &\quad \text{for } \mathbb{E}[\underline{x}] = \mu \end{aligned} \quad (\text{D.175})$$

From (A.9) recall that matrix multiplication typically does not commute, so we need to be very careful about keeping terms in order. That is, if T is on the left *inside* the expectation, it must be pulled out to the left *outside* of the expectation:

$$\mathbb{E}[T\underline{x}] = T \cdot \mathbb{E}[\underline{x}] \quad \text{but} \quad \mathbb{E}[T\underline{x}] \neq \mathbb{E}[\underline{x}] \cdot T \quad (\text{D.176})$$

Therefore we continue with (D.175) as

$$\begin{aligned}\mathbb{E}[(T\underline{x} - T\mu)(T\underline{x} - T\mu)^T] &= \mathbb{E}\left[\left(T(\underline{x} - \mu)\right)\left(T(\underline{x} - \mu)\right)^T\right] \\ &= \mathbb{E}\left[T \cdot (\underline{x} - \mu)(\underline{x} - \mu)^T \cdot T^T\right] \\ &= T \cdot \mathbb{E}\left[(\underline{x} - \mu)(\underline{x} - \mu)^T\right] \cdot T^T = T \cdot \text{Cov}(\underline{x}) \cdot T^T\end{aligned}\quad (\text{D.177})$$

That is, a shift $\underline{\alpha}$ has no effect on the covariance, which makes sense, since shifting a class does not change its shape. The transformation T applied to a random vector applies the same linear transformation symmetrically before and after the covariance.

Addition of Two Random Variables:

(Page 416)

We rarely want to be working with probability distributions, in detail, since they are, in general, very difficult to work with. This is precisely why pattern recognition is built around means, variances, prototypes, discriminants etc.

Let us consider, for example, one of the simplest possible steps: the addition of two, independent random variables:

$$z = x + y \quad x \sim p_x(x), \quad y \sim p_y(y) \quad x, y \text{ independent} \quad (\text{D.178})$$

We would like to derive $p_z()$, the distribution for random variable z . Supposing, first, that we *condition* on the value of x :

$$p_{z|x}(z|x) = p_{y+x|x}(z|x) = p_{y|x}(z-x|x) = p_y(z-x)$$

Represent in y
Shift PDF
Independence

(D.179)

We can find the distribution for z as the marginal from the joint distribution z, x :

$$p_z(z) = \int p_{z,x}(z, x) dx = \int p_{z|x}(z|x) p_x(x) dx \quad (\text{D.180})$$

Substituting in the result from (D.179),

$$p_z(z) = \int p_y(z-x) p_x(x) dx \quad (\text{D.181})$$

we have the desired result, which is that summing x and y leads to a random variable z , whose distribution is the convolution of the distributions of x and y .

Index

- Activation function, 272, 321, 325
AdaBoost, 309
Akaike information criterion, 34, 38, 368
Angular distance, 57, 119
Auto-encoder network, 370
- Backpropagation, 324
Bag of words, 101, 359, 376, 393
Bagging, 305, 317, 342
Balanced accuracy, 241, 460
Base rate fallacy, 216
Bayes risk, 212
Bayes rule, 153, 216, 411
Bayesian methods, 153, 193, 198
Bell curve, 64
Bias, 155, 174, 444
Binomial distribution, 157, 445
Bisection search, 430
Boosting, 307
Bootstrap resampling, 305, 306, 343
- Class, xx, 11, 12, 57
Covariance, 69, 86, 123
Mean, 69, 125, 442
Prototype, 13, 59, 124
Class imbalance, 41, 237
Classification, 14, 117, 193, 267
Classification error, 43, 204, 239
Classifier ensemble, 299
AdaBoost, 309
Bagging, 305, 317, 342
- Boosting, 307
Decision tree, 312, 313
Deep neural network, 325
Neural network, 319
Random forest, 317
Scored / weighted, 301
Stacking, 310
Voting, 301
- Classifier validation, 16, 231, 249, 250
Akaike information criterion, 34, 38, 368
Confusion matrix, 41, 143, 239, 259
Cross validation, 249, 250
Jackknife, 251
Leave One Out, 252
Monte Carlo, 251
Performance criteria, 242
Recursively nested, 253
- Classifiers
Decision tree, 287
k-Nearest neighbour, 46, 136, 138
Linear discriminant, 267, 269, 271
Maximum a posteriori, 198
Maximum likelihood, 194, 460
Mean-Euclidean, 133, 202
Mean-Mahalanobis, 132, 141, 203
Nearest neighbour, 30, 135
Neyman-Pearson, 213, 215
Perceptron, 272, 321
Support vector machine, 282
- Cluster synthesis, 63, 73, 76

- Clustering, 19, 347
 Hierarchical, 365
 K-Means, 351
 Kernel K-Means, 358
 Mean-shift, 363
 Neural network, 370
 Compressed sensing, 98
 Conditional statistic, 410
 Confusion matrix, 41, 143, 239, 259
 Convolution, 171, 332, 416, 456
 Correlation, 413
 Correlation coefficient, 414
 Counting process, 455
 Covariance, 66, 413
 Eigendecomposition, 68, 69, 73, 87, 462
 Estimation error, 174
 Invertibility, 414, 420, 463
 Sample, 420
 Total, 88, 437
 Cumulative distribution, 408
 Data augmentation, 232, 233, 256, 333
 Data processing theorem, 10, 83
 Data visualization, 26
 Datasets
 Iris, 23, 104, 141, 257
 Sunshine, 180
 Two arced clusters, 47, 291
 Decision tree, 287, 312, 313
 Deep neural network, 100, 325
 Degeneracy, 133
 Degrees of freedom, 30, 62, 421
 Dendogram, 367
 Determinant, 93, 401
 Dimensionality reduction, 10, 83, 88, 98, 438
 Discriminant, 267
 Fisher, 91, 441
 Linear, 269, 271
 Quadratic, 267
 Distance function, xx, 56, 57, 117
 Angular, 57, 119
 Euclidean, 57, 118, 119
 Mahalanobis, 132, 141, 203
 Manhattan, 57, 119
 Distribution
 Binomial, 157, 303, 445
 Exponential, 190, 408, 456
 Heavy tailed, 190, 417
 Mixture model, 158
 Normal, 64, 201, 408
 Power law, 418
 Uniform, 190, 408
 Distribution estimation
 Histogram, 165
 k-Nearest neighbour, 172
 Kernel, 168
 Maximum likelihood, 154
 Domain adaptation, 232, 326
 Eigendecomposition, 402
 Of covariance, 68, 69, 73, 87, 462
 Eigenvalue, xxii, 69, 87, 402
 Eigenvector, xxii, 69, 87, 402
 Embedding, 101, 391, 392
 Ensemble methods, 299
 AdaBoost, 309
 Bagging, 305, 317, 342
 Boosting, 307
 Decision tree, 312, 313
 Deep neural network, 325
 Neural network, 319
 Random forest, 317
 Scoring, 301
 Stacking, 310
 Voting, 301
 Entropy, 108, 314, 330
 Error, 43, 204, 239
 Estimation, *see* Learning
 Ethics in pattern recognition, 332
 Euclidean distance, 57, 118, 119
 Examples
 Alien atmospheres, 20
 Autonomous Vehicles, 255
 Biometrics, 19
 Counting problems, 262
 Counting processes, 455
 Defect detection, 74
 Digit recognition, 17
 Digital communications, 288
 Dolphin vocalizations, 21
 Face detection, 41
 Face recognition, 58
 Heart monitor, 20
 Illusions, 8, 333
 Image searching, 103
 Image segmentation, 17
 Iris classification, 23, 141

- Medical tests, 213
Neurons, 321
Object Recognition, 179
Optical character recognition, 7
Reverse image searching, 103
Shakespeare, 376
Text analysis, 376
Texture classification, 17, 359
Unfair coin, 75
Expectation, 66, 409
Expectation maximization, 184
- Feature
 Bag of words, 101
 Definition, 9
 Fisher's, 90, 91, 441
 Image processing, 95
 Nonlinear, 99, 280
 Notation, xx
 Random, 98
 Selection, 96
 Similarity, 55
 Wrapper, 97
- Feature dissimilarity, 56
- Feature extraction, 326
- Feature selection, 96
- Fisher's discriminant, 90, 91, 93, 441
- Nearest neighbour
 Clustering, 366, 368
 Prototype, 126, 366
- Gaussian distribution, 64, 408
- Gradient descent, 431, 433
- Hierarchical clustering, 365
- Histogram, 165, 178
- Hyperellipse, 62, 63, 68, 130
- Hyperplane, 133, 270, 457
- Hypothesis testing, 75, 240, 450
- Imbalanced data, 41, 237
- Independence, 303, 413
- Information theory, 108, 314, 331
- Interpretability, 98, 316, 332
- Inverse problem, 51
- Jackknife, 251, 306
- K-Means clustering, 351
- k-Nearest neighbour, 172
- Classifier, 46, 136, 138
Clustering, 366
Prototype, 126, 366
- Kernel
 Classification, 283
 Clustering, 358
 Estimation, 168
 Polynomial, 285
 Radial basis function, 284, 293
- Kernel K-Means, 358
- Kernel trick, 285, 360
- Learning, 15, 29, 39, 459
 Maximum a posteriori, 153
 Maximum likelihood, 152
 Nonparametric, 164
 Parametric, 152
 Self-supervised, 370
 Semi-supervised, 18, 348, 373
 Unsupervised, 19, 348
- Leave one out, 251, 252
- Linear Discriminant, *see* Discriminant, Linear
- Linear regression, 32, 35, 36, 252, 277
- Linear separability, 47, 271
- Loss function, 100, 271, 328, 427, 429
 Cross-entropy loss, 330
 Hinge loss, 330, 340
 Mean-absolute loss, 329
 Mean-squared loss, 329
 Sign loss, 330
- Mahalanobis distance, 132, 134, 203
- Manhattan distance, 57, 119
- Manifold, 85, 110
- MAP, *see* Maximum a posteriori
- Matrix
 Algebra, 397
 Covariance, 87, 413
 Determinant, 401
 Eigendecomposition, 68, 87, 402
 Inverse, 400
 Orthogonal, 404, 406
 Positive definite, 404
 Positive semi-definite, 404
 Selection, 96
 Toeplitz, 113
- Maximum a posteriori
 Classification, 198

- Estimation, 153, 193
- Maximum likelihood
 - Bias, 155, 444
 - Classification, 194
 - Estimation, 152, 154, 193
- Mean, 154, 409
 - Prototype, 125, 366, 442
- Mean-Shift clustering, 363
- Mean-squared error, 88, 275, 459
- Measurement, xx, 9, 83
- Memorylessness, 455
- Mixture model, 158, 356
- ML, *see* Maximum likelihood, 443
- Model
 - Nonparametric, 59, 60, 164
 - Parametric, 59, 152
- Monte Carlo methods, 251, 264
- MSE, *see* Mean-squared error
- Natural language methods, 376, 392
- Nearest neighbour
 - Classifier, 30, 135
 - Clustering, 366, 368
 - Prototype, 126, 366
- Neural network, 319
 - Activation, 325
 - Auto-encoder, 370
 - Clustering, 370
 - Convolutional, 326, 332
 - Deep, 325
 - Perceptron, 272, 321
- Neyman-Pearson, 213, 215, 248, 261
- No free lunch, 31
- Nonparametric model, 59, 60, 164
- Normal distribution, 64, 201, 408
- Occam's Razor, 32
- Optical character recognition, 7
- Optical illusions, 8, 333
- Optimization, 427
 - Bisection search, 430
 - Convex, 429, 430
 - Coordinate descent, 432
 - Exhaustive, 431
 - Gradient descent, 431, 433
 - Grid search, 433
 - Multi-dimensional, 431
 - Multi-objective, 434
 - One dimensional, 428
- Pareto, 434
- Quadratic, 429
- Optimization objective, 271, 427, 429
- Order statistic, 177, 190
- Outlier, 35, 129, 229, 416
- Overfitting, 33, 34, 49, 62, 315
- Parameter estimation, 12, 152, 268
- Parametric model, 59, 152
- Pareto front, 434, 435
- Pattern, 5–9, 55
- Pattern recognition contexts
 - Clustering, 19, 347
 - Semi-supervised, 18, 373
 - Supervised, 18
 - Unsupervised, 19, 348
- PCA, *see* Principal components analysis
- Perceptron, 272, 300, 321, 330
- Performance criteria, 231, 242
- Polynomial fitting, 32, 277
- Positive definite, 404
- Power law, 418
- Principal components, 88, 371
- Principal components analysis, 88, 438
- Principle of Parsimony, 32, 62
- Probability distribution, 59, 193, 407
- Prototype, 13, 59, 124, 129
 - Furthest neighbour, 126, 366
 - k-Nearest neighbour, 126, 366
 - Mean, 125, 366, 442
 - Nearest neighbour, 126, 366
- Pruning, 136, 279
- Q-Q plot, 177
- Radial basis functions, 284, 293
- Random features, 98
- Random forest, 317
- Random variable, 407
- Random vector, 411, 465
- Receiver operating characteristic, 244–247
 - Discrete measurements, 262
- Rectified linear unit, 325
- Resampling, 237, 305
- RMSE, *see* Root-mean-squared error
- Robustness, 35, 129, 229, 419
- ROC, *see* Receiver operating characteristic
- Root-mean-square error, 32
- Root-mean-squared error, 252, 254

- Sample mean, 154, 420
Sample statistics, 420
Selection matrix, 96
Self-supervised learning, 370
Semi-supervised learning, 18, 348, 373, 386
Separability, 271
Sigmoid, 325
Sigmoid function, 323
Similarity, 55, 56
Soft max, 328
Stacking, 310
Support vector, 278
Support vector machine, 282, 375
Kernel trick, 285
SVM, *see* Support vector machine
Syntactic pattern recognition, 11, 392
Synthesis, 73
- Testing, *see* Classifier validation
Transfer learning, 232, 326, 327
- Unit standard deviation contour, 68
Universal approximator, 324
Unsupervised Learning, 348
Unsupervised learning, 19, 347, 348
- Validation, *see* Classifier validation, 39
Variance, 409
Vector embedding, 101, 391, 392
Vector quantization, 355
- Weak learner, 302, 343
Wrapper methods, 42, 43, 97
- XOR problem, 300, 319, 340