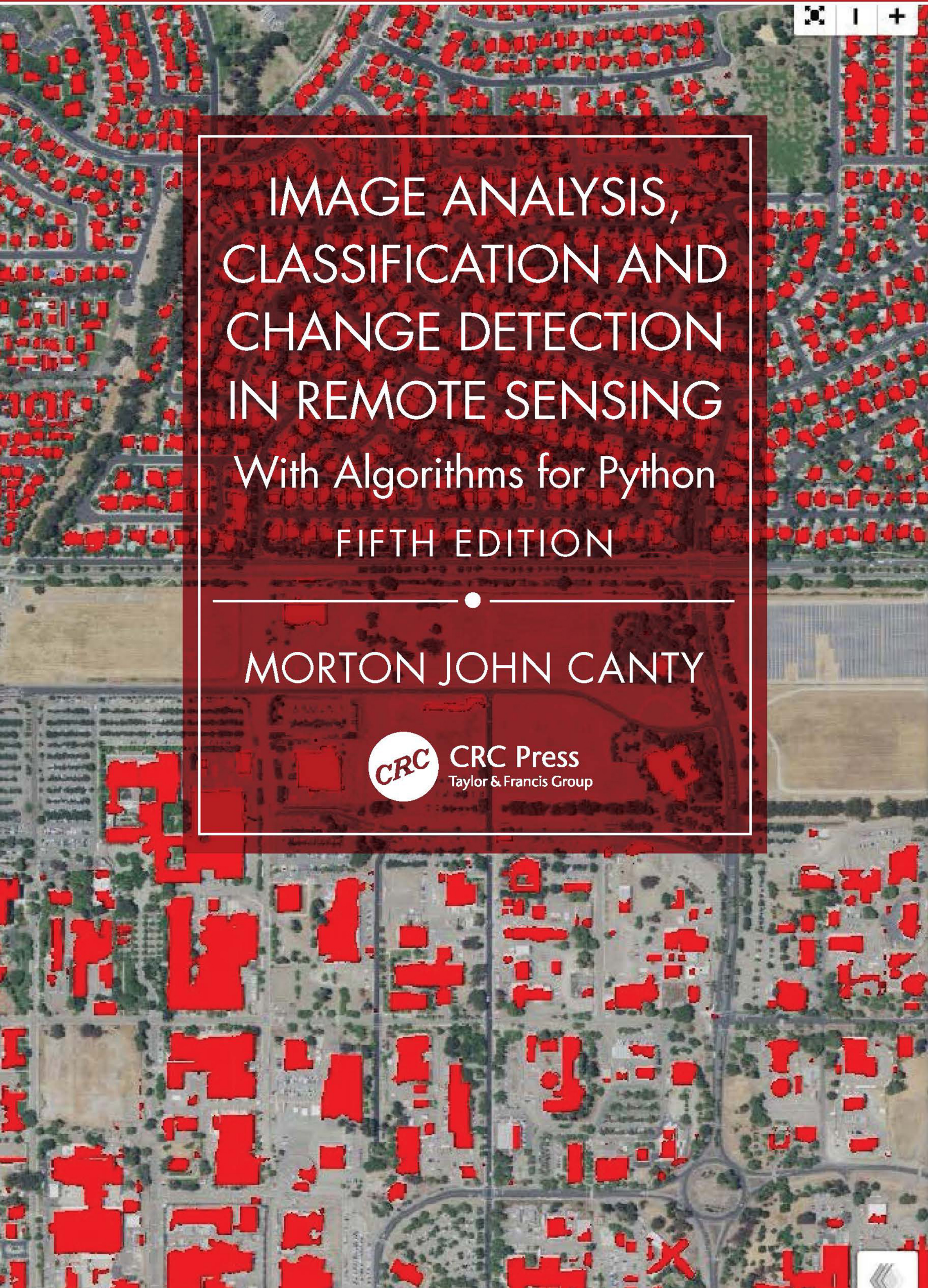


IMAGE ANALYSIS, CLASSIFICATION AND CHANGE DETECTION IN REMOTE SENSING

With Algorithms for Python

FIFTH EDITION

MORTON JOHN CANTY

CRC Press
Taylor & Francis Group

Image Analysis, Classification and Change Detection in Remote Sensing

The fifth edition of this core textbook in advanced remote sensing continues to maintain its emphasis on statistically motivated, data-driven techniques for remote sensing image analysis. The theoretical substance remains essentially the same, with new material on convolutional neural networks, transfer learning, image segmentation, random forests, and an extended implementation of sequential change detection with radar satellites. The tools which apply the algorithms to real remote sensing data are brought thoroughly up to date. As these software tools have evolved substantially with time, the fifth edition replaces the now obsolete Python 2 with Python 3 and takes advantage of the high-level packages that are based on it, such as Colab, TensorFlow/KERAS, Scikit-Learn, and the Google Earth Engine Python API.

New in the Fifth Edition:

- Thoroughly revised to include the updates needed in all chapters because of the necessary changes to the software.
- Replaces Python 2 with Python 3 tools and updates all associated subroutines, Jupyter notebooks and Python scripts.
- Presents easy, platform-independent software installation methods with Docker containers.
- Each chapter concludes with exercises complementing or extending the material in the text.
- Utilizes freely accessible imagery via the Google Earth Engine and provides many examples of cloud programming (Google Earth Engine API).
- Examines deep learning examples including TensorFlow and a sound introduction to neural networks.

This new text is essential for all upper-level undergraduate and graduate students pursuing degrees in Geography, Geology, Geophysics, Environmental Sciences and Engineering, Urban Planning, and the many subdisciplines that include advanced courses in remote sensing. It is also a great resource for researchers and scientists interested in learning techniques and technologies for collecting, analyzing, managing, processing, and visualizing geospatial datasets.

Image Analysis, Classification and Change Detection in Remote Sensing

With Algorithms for Python

Fifth Edition

Morton John Canty

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Designed cover image: Mort Canty, Figure 7.18: Segmentation of manmade structures with a convolutional neural network.

Fifth edition published 2025
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

First edition published by CRC Press 2007
Second edition published by CRC Press 2010
Third edition published by CRC Press 2014
Fourth edition published by CRC Press 2019

ISBN: 978-1-032-82168-9 (hbk)
ISBN: 978-1-032-82170-2 (pbk)
ISBN: 978-1-003-50328-6 (ebk)

DOI: 10.1201/9781003503286

Typeset in CMR10 font
by KnowledgeWorks Global Ltd.

Access the Support Material: www.routledge.com/9781032821689

Publisher's Note: This book has been prepared from camera-ready copy provided by the author.

Contents

Preface to the First Edition

This textbook had its beginnings as a set of notes to accompany seminars and lectures conducted at the Geographical Institute of Bonn University and at its associated Center for Remote Sensing of Land Cover. Lecture notes typically continue to be refined and polished over the years until the question inevitably poses itself: "Why not have them published?" The answer of course is "By all means, if they contribute something new and useful."

So what is "new and useful" here? This is a book about remote sensing image analysis with a distinctly mathematical-algorithmic-computer-oriented flavor, intended for graduate-level teaching and with, to borrow from the remote sensing jargon, a rather restricted FOV. It does not attempt to match the wider *fields of view* of existing texts on the subject, such as Schowengerdt (1997), Richards (2012), Jensen (2005) and others. However, the topics that are covered are dealt with in considerable depth, and I believe that this coverage fills an important gap. Many aspects of the analysis of remote sensing data are quite technical and tend to be intimidating to students with moderate mathematical backgrounds. At the same time, one often witnesses a desire on the part of students to apply advanced methods and to modify them to fit their particular research problems. Fulfilling the latter wish, in particular, requires more than superficial understanding of the material.

The focus of the book is on pixel-oriented analysis of visual/infrared Earth observation satellite imagery. Among the topics that get the most attention are the discrete wavelet transform, image fusion, supervised classification with neural networks, clustering algorithms and statistical change detection methods. The first two chapters introduce the mathematical and statistical tools necessary in order to follow later developments. Chapters 3 and 4 deal with spatial/spectral transformations, convolutions and filtering of multispectral image arrays. Chapter 5 treats image enhancement and some of the preprocessing steps that precede classification and change detection. Chapters 6 and 7 are concerned, respectively, with supervised and unsupervised land cover classification. The last chapter is about change detection with heavy emphasis on the use of canonical correlation analysis. Each of the 8 chapters concludes with exercises, some of which are small programming projects, intended to illustrate or justify the foregoing development. Solutions to the exercises are included in a separate booklet. Appendix A provides some additional mathematical/statistical background and Appendix B develops two efficient training algorithms for neural networks. Finally, Appendix C describes the installation and use of the many computer programs introduced in the course of the book.

I've made considerable effort to maintain a consistent, clear mathematical style throughout. Although the developments in the text are admittedly uncompromising, there is nothing that, given a little perseverance, cannot be followed by a reader who has grasped the elementary matrix algebra and statistical concepts explained in the first two chapters. If the student has ambitions to write his or her own image analysis programs, then he or she must be prepared to "get the maths right" beforehand. There are, heaven knows, enough pitfalls to worry about thereafter.

All of the illustrations and applications in the text are programmed in RSI's ENVI/IDL. The software is available for download at the publisher's website:

http://www.crcpress.com/e_products/downloads/default.asp

Given the plethora of image analysis and geographic information system (GIS) software systems on the market or available under open source license, one might think that the choice of computer environment would have been difficult. It wasn't. IDL is an extremely powerful, array- and graphics-oriented, universal programming language with a versatile interface (ENVI) for importing and analyzing remote sensing data—a peerless combination for my purposes. Extending the ENVI interface in IDL in order to implement new methods and algorithms of arbitrary sophistication is both easy and fun.

So, apart from some exposure to elementary calculus (and the aforesaid perseverance), the only other prerequisites for the book are a little familiarity with the ENVI environment and the basic knowledge of IDL imparted by such excellent introductions as Fanning (2000) or Gumley (2002). For everyday problems with IDL at any level from "newbie" on upward, help and solace are available at the newsgroup `comp.lang.idl-pvwave` frequented by some of the friendliest and most competent gurus on the net.

I would like to express my thanks to Rudolf Avenhaus and Allan Nielsen for their many comments and suggestions for improvement of the manuscript and to CRC Press for competent assistance in its preparation. Part of the software documented in the text was developed within the Global Monitoring for Security and Stability (GMOSS) network of excellence funded by the European Commission.

Morton Canty

Preface to the Second Edition

Shortly after the manuscript for the first edition of this book went to the publisher, ENVI 4.3 appeared along with, among other new features, a support vector machine classifier. Although my decision not to include the SVM in the original text was a conscious one (I balked at the thought of writing my own IDL implementation), this event did point to a rather glaring omission in a book purporting to be partly about land use/land cover classification. So, almost immediately, I began to dream of a Revised Second Edition and to pester CRC Press for a contract. This was happily forthcoming and the present edition now has a fairly long section on supervised classification with support vector machines.

The SVM is just one example of so-called kernel methods for nonlinear data analysis, and I decided to make kernelization one of the themes of the revised text. The treatment begins with a dual formulation for ridge regression in Chapter 2 and continues through kernel principal components analysis in Chapters 3 and 4, support vector machines in Chapter 6, kernel K-means clustering in Chapter 8 and nonlinear change detection in Chapter 9. Other new topics include entropy and mutual information (Chapter 1), adaptive boosting (Chapter 7) and image segmentation (Chapter 8). In order to accommodate the extended material on supervised classification, discussion is now spread over the two Chapters 6 and 7. The exercises at the end of each chapter have been extended and re-worked and, as for the first edition, a solutions manual is provided.

I have written several additional IDL extensions to ENVI to accompany the new themes, which are available, together with updated versions of previous programs, for download on the Internet. In order to accelerate some of the more computationally intensive routines for users with access to CUDA (parallel processing on NVIDIA graphics processors), code is included which can make use of the IDL bindings to CUDA provided by Tech-X Corporation in their GPULib product:

`http://gpulib.txcorp.com`

Notwithstanding the revisions, the present edition remains a monograph on pixel-oriented analysis of intermediate-resolution remote sensing imagery with emphasis on the development and programming of statistically motivated, data-driven algorithms. Important topics such as object-based feature analysis (for high-resolution imagery), or the physics of the radiation/surface interaction (for example, in connection with hyperspectral sensing) are only touched upon briefly, and the huge field of radar remote sensing is left out completely. Nevertheless, I hope that the in-depth focus on the topics covered will continue to be of use both to practitioners as well as to teachers.

I would like to express my appreciation to Peter Reinartz and the German Aerospace Center for permission to use the traffic scene images in Chapter 9 and to NASA's Land Processes Distributed Active Archive Center for free and uncomplicated access to archived ASTER imagery. Thanks also go to Peter Messmer and Michael Galloy, Tech-X Corp., for their prompt responses to my many cries for help with GPULib. I am especially grateful to my colleagues Harry Vereecken and Allan Nielsen, the former for generously providing me with the environment and resources needed to complete this book, the latter for the continuing inspiration of our friendship and long-time collaboration.

Morton Canty

Preface to the Third Edition

A main incentive for me to write a third edition of this book stemmed from my increasing enthusiasm for the Python programming language. I began to see the advantage of illustrating the many image processing algorithms covered in earlier editions of the text not only in the powerful and convenient, but not inexpensive, ENVI/IDL world, but also on a widely available open source platform. Python, together with the NumPy and Scipy packages, can hold its own with any commercial array processing software system. Furthermore, the Geospatial Data Abstraction Library (GDAL) and its Python wrappers allow for great versatility and convenience in reading, writing and manipulating different image formats. This was enough to get me going on a revised textbook, one which I hope will have appeal beyond the ENVI/IDL community.

Another incentive for a new edition was hinted at in the preface to the previous edition, namely the lack of any discussion of the vast and increasingly important field of radar remote sensing. Obviously this would be a topic for (at least) a whole new book, so I have included material only on a very special aspect of particular interest to me, namely multivariate statistical classification and change detection algorithms applied to polarimetric synthetic aperture radar (polSAR) data. Up until recently, not many researchers or practitioners have had access to this kind of data. However with the advent of several spaceborne polarimetric SAR instruments such as the Japanese ALOS, the Canadian Radarsat-2, the German TerraSAR-X, and the Italian COSMO-SkyMed missions, the situation has greatly improved. Chapters 5, 7, and 9 now include treatments of speckle filtering, image co-registration, supervised classification, and multivariate change detection with multi-look polSAR data.

The software associated with the present edition includes, along with the ENVI/IDL extensions, Python scripts for all of the main processing, classification and change detection algorithms. In addition, many examples discussed in the text are illustrated with Python scripts as well as in IDL. The Appendices C and D separately document the installation and use of the ENVI/IDL and Python code. For readers who wish to use the Eclipse/Pydev development environment (something which I highly recommend), the Python scripts are provided in the form of a Pydev project.

What is missing in the Python world, of course, is the slick GUI provided by ENVI. I have made no attempt to mimic an ENVI graphical environment in Python, and the scripts provided content themselves with reading imagery from, and writing results to, the file system. A rudimentary command line script for RGB displays of multispectral band combinations in different histogram enhancement modes is included.

For an excellent introduction to scientific computing in Python see Langtangen (2009). The book by Westra (2013) provides valuable tips on geospatial development in Python, including GDAL programming. The definitive reference on IDL is now certainly Galloy (2011), an absolute must for anyone who uses the language professionally.

With version 5.0, a new ENVI graphics environment and associated API has appeared which has a very different look and feel to the old "ENVI Classic" environment, as it is officially referred to. Fortunately the classic environment is still available and, for reasons of compatibility with previous versions, the IDL programming examples in the text use the classic interface and its associated syntax. Most of the ENVI/IDL extensions as documented in Appendix C are provided both for the new as well as for the classic GUI/API.

I would like to express my appreciation to the German Aerospace Center for permission to use images from the TerraSAR-X platform and to Henning Skriver, DTU Space Denmark, for allowing me to use his EMISAR polarimetric data. My special thanks go to Allan Nielsen and Frank Thonfeld for acquainting me with SAR imagery analysis and to Rudolf Avenhaus for his many helpful suggestions in matters statistical.

Morton Canty

Preface to the Fourth Edition

The fourth revision marks the completion of a transition, begun in the preceding edition, from ENVI/IDL to the Python language for implementing the algorithms discussed in the text. It was with some hesitation that I abandoned the comfort and convenience of the powerful ENVI/IDL environment and ventured into the raw world of open source. But it has become apparent that open source software is the future for scientific computing in general and for geospatial analysis in particular. The popularity of R, JavaScript, or Python in the remote sensing community, the potential of machine learning software such as TensorFlow for object recognition, the Python and JavaScript APIs to the wonderful Google Earth Engine, the many open source mapping platforms and servers like Mapbox, OpenLayers, Leaflet, or the OpenStreetMap project, the elegance of Jupyter notebooks for interactive and collaborative development, the power of container technology like Docker for painless distribution of scientific software, all of the advantages of these languages, tools and platforms are freely available and under continual development by a gigantic community of software engineers, both commercial and voluntary.

So I have jumped off the fence and onto the open source bandwagon in order to ensure that the computer code used in the present version of the book will be not only in line with the current trend, but also accessible to anyone, student or scientist, with a computer and an Internet connection. Each of the nine chapters of the text is now accompanied by its own Jupyter notebook illustrating all, or almost all, of the concepts and algorithms presented in that chapter. The Python scripts are uniformly command-line oriented so as to be able to be started easily from within a notebook input cell. All of the software is packaged into a single Docker container which, when run on the user's machine, serves the Jupyter notebooks to his or her favourite web browser. The necessary packages and modules, including the Google Earth Engine and TensorFlow APIs, are already built into the container so that there is no need to install anything at all, apart from the Docker engine. This is of course great for the reader, and for me it means no longer worrying about 32-bit vs. 64-bit Windows vs. Linux vs. MacOS, or who has what pre-installed version of which Python package. The container is pulled from DockerHub automatically when run for the first time, and the source software can be cloned/forked from GitHub. The details are all given in an appendix.

Had I approached this revision just a couple of years ago, I would have had some misgivings about retaining the long and rigorous descriptions of neural network training algorithms in Chapter 6 and Appendix B. Neural network land cover classifiers had until recently gone somewhat out of fashion, giving way to random forests, support vector machines and the like. However, given the present artificial intelligence craze, the mathematical detail in the text should help to provide a solid background for anyone interested in understanding and exploiting deep learning techniques.

Like the earlier editions, this is not a text on programming or on the intricacies of the various packages, tools, and APIs referred to in the text. As a solid introduction to scientific computing with Python, I would still recommend Langtangen (2009) and, for TensorFlow, the book by Géron (2017). I expect that I'm not alone in hoping for a good textbook on the Google Earth Engine API. Fortunately the on-line documentation is excellent.

Apart from taking advantage of many of these exciting advances in open source computing, the revised text continues to concentrate on an in-depth treatment of pixel-oriented, data-driven, statistical methods for remote sensing image processing and interpretation. The choice of topics and algorithms is by no means all-encompassing and reflects strongly the author's personal interests and experience. Those topics chosen, however, are presented in depth and from first principles. Chapters 1 and 2 on linear algebra and statistics continue to be pretty much essential for an understanding of the rest of the material. Especially new in the present edition is the discussion of an elegant sequential change detection method for polarimetric synthetic aperture radar imagery developed by Knut Conradsen and his colleagues at the Danish Technical University. It has been a pleasure for me to be involved in its implementation, both in "conventional" Python and for the Google Earth Engine Python and JavaScript APIs.

I would like to thank my editor Irma Shagla Britton at CRC Press for waking me up to the idea of a fourth edition, and to give a big thank you to the friendly, competent and infinitely patient GEE development team.

Morton Canty

Preface to the Fifth Edition

In 2019, in the preface to the fourth edition of this book, I proudly proclaimed that I had dropped the commercial ENVI/IDL software environment entirely and committed to open source and to the Python universe. As a reward for my conversion to the true faith, Python 2.7 was officially "sunsetted" on January 1, 2020 in favour of the rather incompatible Python 3. The Python 2 software exclusively used in the fourth edition was packaged into a self-contained Docker image, but by then the Earth Engine authentication procedure had changed and the situation was anything but satisfactory. So when my publisher contacted me in February 2024 for a fifth revision I happily agreed, my first thought being: *now you can upgrade the damn software.* Which I have done and believe me, it doesn't just involve adding parentheses to the `print` statements.

Another incentive for me for a fifth edition was the unbelievable evolution of neural networks in so many aspects of data analysis in the last few years. The classic feed forward network was always given prominence in the discussion of supervised image classification in the earlier editions, but to move with the times I've taken the opportunity to extend the treatment to include convolutional networks for supervised transfer learning and segmentation of remote sensing image data. The examples are programmed in the latest TensorFlow/Keras environment, compatible with the GPU runtime available on Google's Colab. The new neural network transformer architectures, LLMs and generative AI in general are not (yet) relevant to the material discussed here, but I have also included an experimental LLM/RAG application for informally querying the book's content.

Other novelties in the present edition include a treatment of binary classification trees and random forests, an improved and extended implementation of the sequential SAR change detection algorithm including the Loewner order definition of change direction, and a new adaptive temporal speckle filtering application. The accompanying JupyterLab notebooks now provide interactive widget interfaces to the GEE archives for bi-temporal change detection with visual/infrared imagery, multi-temporal sequential SAR change detection, as well as high resolution building segmentation with a fully convolutional neural network.

Although quite old and not especially good, in the present edition I have kept the same ground truth data that were[*] used in all of the previous editions

[*]A propos moving or not moving with the times, when I went to school the word *data* was the plural of *datum*. In this textbook, at least, it still is.

to demonstrate the various supervised classification approaches. This, if only because I collected the training data myself, on my bicycle, on the same sunny day in May, 2007 that the ASTER satellite passed overhead and acquired the multispectral scene. Unfortunately, at 15m ground resolution, you can see neither me nor my bicycle.

As for the rest, everything I said in the prefaces to the earlier editions still holds. So I don't want to repeat myself, save once again to thank my editor Irma Britton and CRC Press for so faithfully providing me with a platform over the last 18 years.

Morton Canty

Author Biography

Morton John Canty, now semi-retired, was a senior research scientist in the Institute for Bio- and Geosciences at the Jülich Research Center in Germany.

He received his PhD in Nuclear Physics in 1969 at the University of Manitoba, Canada and, after post-doctoral positions in Bonn, Groningen and Marburg, began work in Jülich in 1979. There, his principal interests have been the development of statistical and game-theoretical models for the verification of international treaties and the use of remote sensing data for monitoring global treaty compliance. He has served on numerous advisory bodies to the German Federal Government and to the International Atomic Energy Agency in Vienna and was a coordinator within the European Network of Excellence on Global Monitoring for Security and Stability, funded by the European Commission. Morton Canty is the author of three monographs in the German language: on the subject of non-linear dynamics (*Chaos und Systeme*, Vieweg, 1995), neural networks for classification of remote sensing data (*Fernerkundung mit neuronalen Netzen*, Expert, 1999), and algorithmic game theory (*Konfliktlösungen mit Mathematica*, Springer 2000). The latter text has appeared in a revised English version (*Resolving Conflicts with Mathematica*, Academic Press, 2003). He is co-author of a monograph on mathematical methods for treaty verification (*Compliance Quantified*, Cambridge University Press, 1996). He has published many papers on the subjects of experimental nuclear physics, nuclear safeguards, applied game theory, and remote sensing and has lectured on nonlinear dynamical growth models and remote sensing digital image analysis at Universities in Bonn, Berlin, Freiberg/Saxony, and Rome.

1

Images, Arrays, and Matrices

There are many Earth observation satellite-based sensors, both active and passive, currently in orbit or planned for the near future. In representative of these, we briefly describe the multispectral ASTER system (Abrams et al., 1999) and the TerraSAR-X synthetic aperture radar satellite (Pitz and Miller, 2010). See Jensen (2018), Richards (2012), and Mather and Koch (2010) for overviews of major remote sensing satellite platforms.

The Advanced Spaceborne Thermal Emission and Reflectance Radiometer (ASTER) instrument was launched in December 1999 on the Terra spacecraft. It is being used to obtain detailed maps of land surface temperature, reflectance and elevation and consists of sensors to measure reflected solar radiance and thermal emission in three spectral intervals:

- VNIR: Visible and near-infrared bands 1, 2, 3N, and 3B, in the spectral region between 0.52 and 0.86 μm (four arrays of charge-coupled detectors (CCDs) in pushbroom scanning mode).
- SWIR: Short wavelength infrared bands 4 to 9 in the region between 1.60 and 2.43 μm (six cooled PtSi-Si Schottky barrier arrays, pushbroom scanning).
- TIR: Thermal infrared bands 10 to 14 covering a spectral range from 8.13 to 11.65 μm (cooled HgCdTe detector arrays, whiskbroom scanning).

The altitude of the spacecraft is 705 km. The across- and in-track *ground sample distances* (GSDs), i.e., the detector widths projected through the system optics onto the Earth's surface, are 15 m (VNIR), 30 m (SWIR), and 90 m (TIR).* The telescope associated with the 3B sensors is back-looking at an angle of 27.6^o to provide, together with the 3N sensors, along-track stereo image pairs. In addition, the VNIR camera can be rotated from straight down (nadir) to $\pm\ 24^o$ across-track. The SWIR and TIR instrument mirrors can be pointed to $\pm\ 8.5^o$ across-track. Like most platforms in this ground resolution category, the orbit is near polar, sun-synchronous. Quantization levels are 8 bits for VNIR and SWIR and 12 bits for TIR. The sensor systems have an

*At the time of writing, both the VNIR and TIR systems are still producing good data. The SWIR sensor was declared to be unusable in 2008.

DOI: 10.1201/9781003503286-1

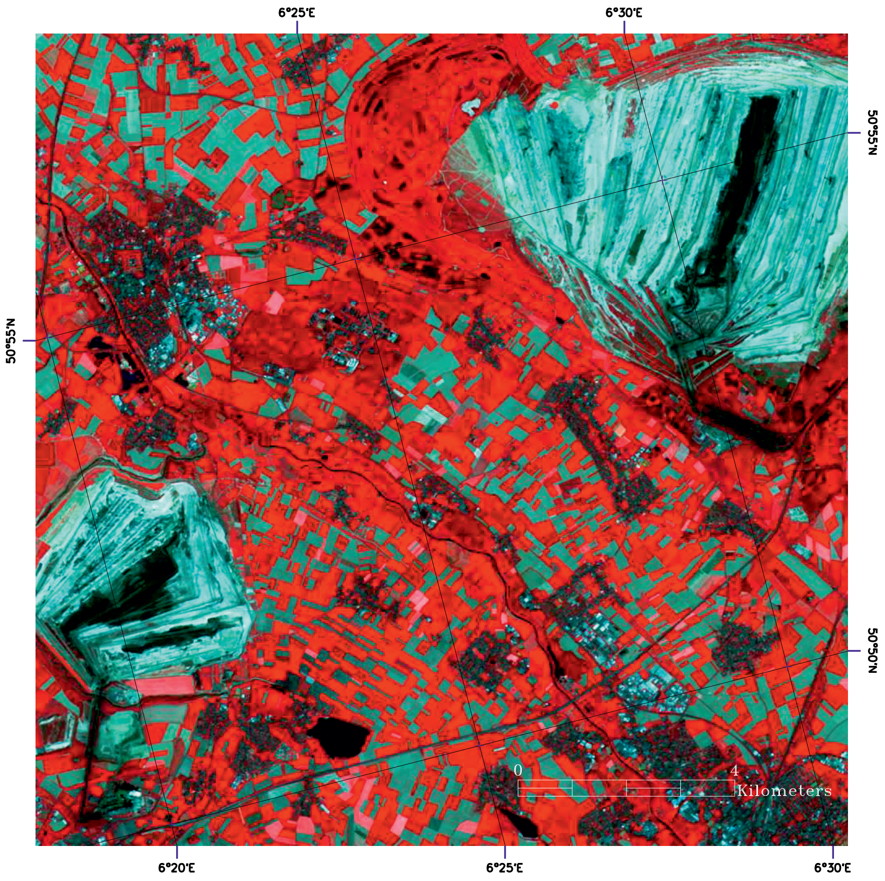


FIGURE 1.1
ASTER color composite image (1000 × 1000 pixels) of VNIR bands 1 (blue), 2 (green), and 3N (red) over the towns of Jülich and Düren in Germany, acquired on May 1, 2007. The bright areas are open-cast lignite coal mines.

average duty cycle of 8% per orbit (about 650 scenes per day, each 60×60 km^2 in area) with revisit times between 4 and 16 days.

Figure 1.1 shows a spatial/spectral subset of an ASTER scene. The image is a UTM (Universal Transverse Mercator) projection oriented along the satellite path (rotated approximately 16.4^o from the north) and orthorectified using a digital terrain model generated from the stereo bands.

Unlike passive multi- and hyperspectral imaging sensors, which measure reflected solar energy or the Earth's thermal radiation, synthetic aperture radar (SAR) airborne and satellite platforms supply their own microwave radiation source, allowing observations which are independent of time of day

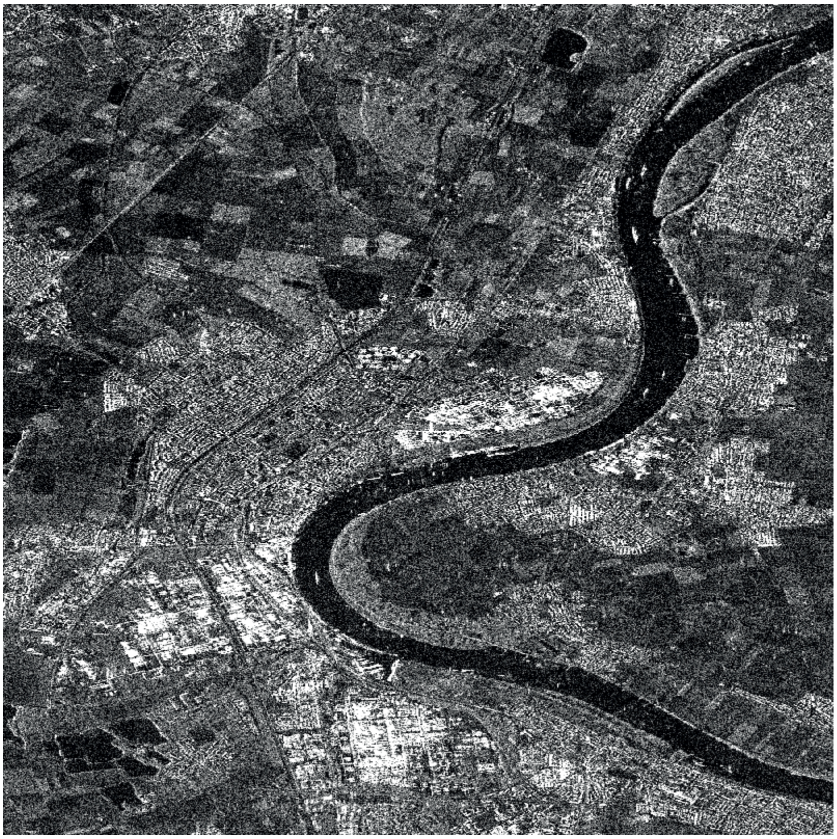


FIGURE 1.2
A 5000 × 5000-pixel spatial subset of the HH polarimetric band of a TerraSAR-X quad polarimetric image acquired over the Rhine River, Germany, in so-called Stripmap mode. The data are slant-range, single-look, complex. The gray-scale values correspond to the magnitudes of the complex pixel values.

or cloud cover. The radar antenna on the TerraSAR-X satellite, launched in June, 2007, emits and receives X-band radar (9.65 GHz) in both horizontal and vertical polarizations to provide surface imaging with a geometric resolution from about 18 m (scanSAR mode, 10 km×150 km swath) down to 1 m (high-resolution Spotlight mode, 10 km×5 km swath). It flies in a sun-synchronous, near-polar orbit at an altitude of 514 km with a revisit time for points on the equator of 11 days. Figure 1.2 shows a TerraSAR-X HH polarimetric band (horizontally polarized radiation emitted and detected) acquired over the Rhine River, Germany, in April, 2010. The data are at the single-look, slant-range complex (SLC) processing level, and are not map-projected.

In the following two sections of this chapter, the properties of multispectral and SAR images in general are described in more detail. In the third section, we go into some of the mathematical properties of the numerical arrays which represent those images, and conclude in Section 1.4 with simple methods for finding extrema in image properties.

1.1 Multispectral satellite images

A multispectral, optical/infrared image such as that shown in Figure 1.1 may be represented as a three-dimensional array of gray-scale values* or pixel intensities

$$g_k(i,j), \quad 1 \le i \le c,\ 1 \le j \le r,\ 1 \le k \le N,$$

where c is the number of pixel columns (also called *samples*) and r is the number of pixel rows (or *lines*). The index k denotes the spectral band, of which there are N in all. For data at an early processing stage, a pixel may be stored as a *digital number* (DN), often in a single byte so that $0 \le g_k \le 255$. This is the case for the ASTER VNIR and SWIR bands at processing level L1A (unprocessed reconstructed instrument data), whereas the L1A TIR data are quantized to 12 bits (as unsigned integers) and thus stored as digital numbers from 0 to $2^{12} - 1 = 4095$. Processed image data may of course be stored in byte, integer, or floating point format and can have negative or even complex values.

The gray-scale values in the various bands encode measurements of the radiance $L_{\Delta\lambda}(x,y)$ in wavelength interval $\Delta\lambda$ due to sunlight reflected from some point (x,y) on the Earth's surface, or due to thermal emission from that surface, and focused by the instrument's optical system along the array of sensors. Ignoring all absorption and scattering effects of the intervening atmosphere, the at-sensor radiance available for measurement from reflected sunlight from a horizontal, *Lambertian* surface, i.e., a surface which scatters reflected radiation uniformly in all directions, is given by

$$L_{\Delta\lambda}(x,y) = E_{\Delta\lambda} \cdot \cos\theta_z \cdot R_{\Delta\lambda}(x,y)/\pi. \tag{1.1}$$

The units are $[\mathrm{W}/(\mathrm{m}^2 \cdot \mathrm{sr} \cdot \mu\mathrm{m})]$, $E_{\Delta\lambda}$ is the average spectral solar irradiance in the spectral band $\Delta\lambda$, θ_z is the solar zenith angle, $R_{\Delta\lambda}(x,y)$ is the surface reflectance at coordinates (x,y), a number between 0 and 1, and π accounts for the upper hemisphere of solid angle. The conversion between DN and at-sensor radiance is determined by the sensor calibration as measured (and

*Which we will usually denote by the letter g throughout the text.

maintained) by the satellite image provider. For example, for ASTER VNIR and SWIR L1A data,

$$L_{\Delta\lambda}(x,y) = A \cdot DN/G + D.$$

The quantities A (linear coefficient), G (gain), and D (offset) are tabulated for each of the detectors in the arrays and included with each acquisition. Atmospheric scattering and absorption models may be used to deduce at-surface radiance, surface temperature, and emissivity or surface reflectance from the observed radiance at the sensor. Reflectance and emissivity are directly related to the physical properties of the surface being imaged. See Schowengerdt (2006) for a thorough discussion of atmospheric effects and their correction.

Various conventions are used for storing the image array $g_k(i,j)$ in computer memory or other storage media. In *band interleaved by pixel* (BIP) format, for example, a two-channel, 3×3 pixel image would be stored as

$$\begin{matrix} g_1(1,1) & g_2(1,1) & g_1(2,1) & g_2(2,1) & g_1(3,1) & g_2(3,1) \\ g_1(1,2) & g_2(1,2) & g_1(2,2) & g_2(2,2) & g_1(3,2) & g_2(3,2) \\ g_1(1,3) & g_2(1,3) & g_1(2,3) & g_2(2,3) & g_1(3,3) & g_2(3,3), \end{matrix}$$

whereas in *band interleaved by line* (BIL), it would be stored as

$$\begin{matrix} g_1(1,1) & g_1(2,1) & g_1(3,1) & g_2(1,1) & g_2(2,1) & g_2(3,1) \\ g_1(1,2) & g_1(2,2) & g_1(3,2) & g_2(1,2) & g_2(2,2) & g_2(3,2) \\ g_1(1,3) & g_1(2,3) & g_1(3,3) & g_2(1,3) & g_2(2,3) & g_2(3,3), \end{matrix}$$

and in *band sequential* (BSQ) format as

$$\begin{matrix} g_1(1,1) & g_1(2,1) & g_1(3,1) \\ g_1(1,2) & g_1(2,2) & g_1(3,2) \\ g_1(1,3) & g_1(2,3) & g_1(3,3) \\ g_2(1,1) & g_2(2,1) & g_2(3,1) \\ g_2(1,2) & g_2(2,2) & g_2(3,2) \\ g_2(1,3) & g_2(2,3) & g_2(3,3). \end{matrix}$$

In the computer language Python, augmented with the numerical package `numpy`, so-called *row-major indexing* is used for arrays and the elements in an array are numbered from zero. This means that if a gray-scale image g is assigned to a Python array variable `g`, then the intensity value $g(i,j)$ is addressed as `g[j-1,i-1]`. An N-band multispectral image is stored in BIP format as an $r \times c \times N$ array in `numpy`, in BIL format as an $r \times N \times c$ and in BSQ format as an $N \times r \times c$ array. So, for example, in BIP format the value $g_k(i,j)$ is stored at `g[j-1,i-1,k-1]`.

Auxiliary information, such as image acquisition parameters and georeferencing, is sometimes included with the image data on the same file, and the format may or may not make use of compression algorithms. Examples

Listing 1.1: Reading and displaying an image band in Python.

```python
#!/usr/bin/env python3
#  Name:     ex1_1.py
import  numpy as np
import sys
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly
import matplotlib.pyplot as plt

def disp(infile, n):
    '''read a multispectral image
       and display n'th band'''
    gdal.AllRegister()
    inDataset = gdal.Open(infile, GA_ReadOnly)
    cols = inDataset.RasterXSize
    rows = inDataset.RasterYSize
    bands = inDataset.RasterCount
    image = np.zeros((bands, rows, cols))
    for b in range(bands):
        band = inDataset.GetRasterBand(b+1)
        image[b,:,:] = band.ReadAsArray(0,0,cols,rows)
    inDataset = None
    band = image[n-1, :, :]
    mn = np.amin(band)
    mx = np.amax(band)
    plt.imshow((band-mn)/(mx-mn), cmap='gray')
    plt.show()

if __name__ == '__main__':
    infile = sys.argv[1]
    bandnumber = int(sys.argv[2])
    disp(infile, bandnumber)
```

are the GeoTIFF[*] file format used, for instance, by Space Imaging Inc. for distributing Carterra© imagery and which includes lossless compression, the HDF-EOS (Hierarchical Data Format-Earth Observing System) files in which ASTER images are distributed, and the PCIDSK format employed by PCI Geomatics© with its image processing software, in which auxiliary information is in plain ASCII and the image data are not compressed. ENVI (©Harris

[*]GeoTIFF is an open-source specification and refers to TIFF files which have geographic (or cartographic) data embedded as tags within the file. The geographic data can be used to position the image in the correct location and geometry on the screen of a geographic information display.

Geospatial Solutions) uses a simple "flat binary" file structure with an additional ASCII header file.

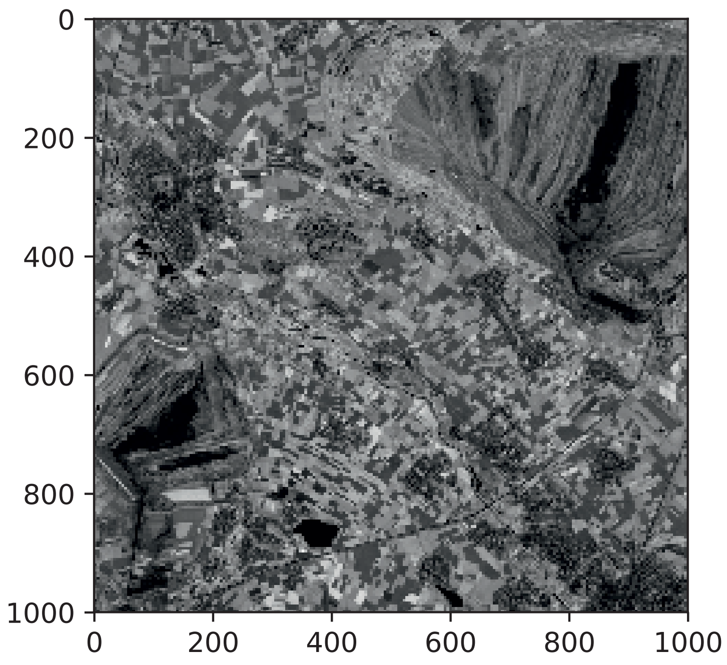


FIGURE 1.3
Output of the Python script in Listing 1.1 in a Jupyter notebook.

Listing 1.1 is a simple and fairly self-explanatory Python script which reads a multispectral image into a `numpy` array in BSQ interleave format with the aid of GDAL (the Geospatial Data Abstraction Library) and then displays a spectral band using the `matplotlib.pyplot` package. The script takes two arguments, the image filename, and the band number to be displayed and is run from the command prompt in Windows or from a console window on Unix-like systems with the command `python ex1_1.py *args`. In the Unix case, the "shebang" `#!` in the first line allows it to be run simply by typing the filename, assuming the path to the `env` utility is `/usr/bin/env`. In this book, we will prefer to work almost exclusively from within Jupyter notebooks, where the script can be executed with the so-called *line magic* `%run` without the `.py` extension, producing the output shown in Figure 1.3:

```
%run scripts/ex1_1 imagery/AST_20070501.tif 3
```

For commonly used line magics like `%run`, the % is optional. We shall be making extensive reference to the Jupyter notebooks which accompany each of the chapters. Software installation and Python scripts are documented in Appendix C.

1.2 Synthetic aperture radar images

Synthetic aperture radar (SAR) systems differ significantly from optical/ infrared sensor-based platforms. Richards (2009) and Oliver and Quegan (2004) provide thorough introductions to SAR remote sensing, SAR image statistics, image analysis, and interpretation. Here we only treat briefly those aspects relevant to our focus on classification and change detection.

The power received by a radar transmitting/receiving antenna reflected from a distributed (as opposed to point) target at a distance D from the antenna is given by (Richards, 2009)

$$P_R = \frac{P_T G_T G_R \lambda^2 \sigma^o \Delta_a \Delta_r}{(4\pi)^3 D^4}\ [W], \tag{1.2}$$

where P_T is the transmitted power $[W \cdot m^{-2}]$, λ is the operating wavelength $[m]$, $G_T(G_R)$ is the transmitting (receiving) antenna gain, $\Delta_a(\Delta_r)$ is the azimuth (ground range) resolution $[m]$, and σ^o is the unitless scattering coefficient (referred to as the *radar cross section*) of the target surface. The scattering coefficient is related to the (bio)physical properties of the surface being irradiated, notably its water content.

In later chapters, we will be concerned primarily with fully and partially polarized SAR data. A full, or *quad*, polarimetric SAR measures a 2×2 *scattering matrix* $\boldsymbol{S}$ at each resolution cell on the ground. The scattering matrix relates the incident and the backscattered electric fields $\boldsymbol{E}^i$ and $\boldsymbol{E}^b$ according to

$$\boldsymbol{E}^b = \boldsymbol{S}\boldsymbol{E}^i \quad \text{or} \quad \begin{pmatrix} E_h^b \\ E_v^b \end{pmatrix} = \begin{pmatrix} s_{hh} & s_{hv} \\ s_{vh} & s_{vv} \end{pmatrix} \begin{pmatrix} E_h^i \\ E_v^i \end{pmatrix}. \tag{1.3}$$

Here $E_h^{i(b)}$ and $E_v^{i(b)}$ denote the horizontal and vertical components of the incident (backscattered) oscillating electric fields directly at the target. These can be deduced from the transmitted and received radar signals via the so-called *far-field* approximations; see Richards (2009). If both horizontally and vertically polarized radar pulses are emitted and discriminated, then they determine, from Equation (1.3), the four complex scattering matrix elements.

Complex numbers provide a convenient representation of the amplitude E of an electric field:

$$E = |E| cos(\omega t + \phi) = \mathrm{Re}\left(|E| e^{\mathbf{i}(\omega t + \phi)}\right),$$

Out[2]:

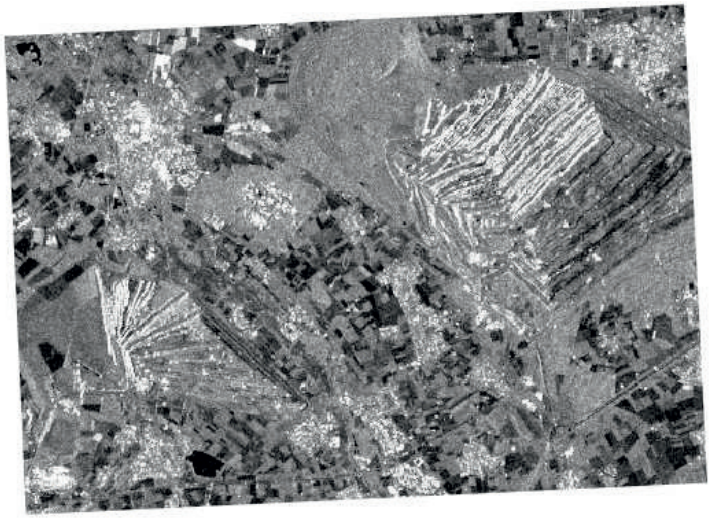


FIGURE 1.4
A Sentinel-1 image extracted from the GEE database and displayed in the Jupyter notebook `Chapter1.ipynb`.

where $\omega = 2\pi f$ and ϕ are the angular frequency and phase of the radiation and Re denotes "real part".[*] It is usually convenient to work exclusively with complex amplitudes $|E|e^{\mathbf{i}(\omega t+\phi)}$, bearing in mind that only the real part is physically significant. When the oscillating electric fields are described by complex numbers in this way, the scattering matrix elements are also complex. A full polarimetric, or *quad polarimetric* SAR image then consists of four complex bands s_{hh}, s_{hv}, s_{vh}, and s_{vv}, one for each pixel-wise determination of an element of the scattering matrix. So-called *reciprocity* (Richards, 2009), which normally applies to natural targets, implies that $s_{hv} = s_{vh}$. The squared amplitudes of the scattering coefficients, i.e., $|s_{hh}|^2$, $|s_{hv}|^2$, etc., constitute the radar cross sections for each polarization combination. These in turn replace σ^o in Equation (1.2) and determine the received power in each polarization channel.

The dual polarimetric Sentinel-1 sensor[†] transmits in only one polarization and receives in two, thus measuring only the bands s_{vv} and s_{vh} or s_{hh} and s_{hv}. To see an example we make our first encounter with the Google Earth Engine (GEE) (Gorelick et al., 2017). Here we use GEE's Python API (application programming interface) to access and display a Sentinel-1 scene from the GEE public data catalogue:

[*] A brief introduction to complex numbers is given in Appendix A.
[†] https://www.esa.int/Applications/Observing_the_Earth/Copernicus/Sentinel-1

```
%matplotlib inline
import IPython.display as disp
import ee
ee.Initialize()

minlon = 6.31
minlat = 50.83
maxlon = 6.58
maxlat = 50.95
rect=ee.Geometry.Rectangle([minlon,minlat,maxlon,maxlat])

collection = ee.ImageCollection('COPERNICUS/S1_GRD') \
.filterBounds(rect) \
.filterDate(ee.Date('2017-05-01'),ee.Date('2017-06-01'))\
.filter(ee.Filter.eq('transmitterReceiverPolarisation',
                                        ['VV','VH'])) \
.filter(ee.Filter.eq('resolution_meters', 10)) \
.filter(ee.Filter.eq('instrumentMode', 'IW'))

image = ee.Image(collection.first()).clip(rect)
url = image.select('VV').getThumbURL({'min':-20,'max':0})
disp.Image(url=url)
```

The code above runs in the Jupyter notebook `Chapter1.ipynb` which accompanies the text. It accesses and displays a spatial subset of the VV polarimetric band of a Sentinel-1 dual polarimetric image from May 2017, over roughly the same area as Figure 1.1. The archived image was acquired in interferometric wide swath (IW) mode and processed to the ground range detected (GRD) product with a pixel size of 10×10 m. The gray-scale values in the notebook output cell shown in Figure 1.4 correspond to the logarithms of the backscatter intensities after averaging (multi-looking) of the single-look complex backscattered signal. The projection used is "Maps Mercator" (EPSG:3857) which is the default for the GEE. The image is displayed in decibels.

1.3 Linear algebra of vectors and matrices

It is very convenient, in fact essential, to use a vector representation for multispectral or SAR image pixels. In this book, a pixel will be represented in the form

$$\boldsymbol{g}(i,j) = \begin{pmatrix} g_1(i,j) \\ \vdots \\ g_N(i,j) \end{pmatrix}, \tag{1.4}$$

which is understood to be a *column vector* of spectral intensities or gray-scale values at the image position (i, j). It can be thought of as a point in N-dimensional Euclidean space, commonly referred to as *input space* or *feature space*. In the case of SAR images, as explained in the preceding section, the vector components may be complex numbers.

Since we will be making extensive use of the vector notation of Equation (1.4), some of the basic properties of vectors, and of matrices which generalize them, will be reviewed here. One can illustrate these properties for 2-component vectors and 2×2 matrices, and this section is accordingly a very simple introduction to so-called *linear algebra*. A list of frequently used mathematical symbols is given at the end of the book.

1.3.1 Elementary properties

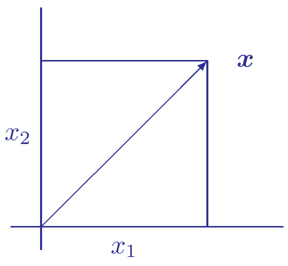


FIGURE 1.5
A vector with two components.

The *transpose* of the two-component column vector

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \tag{1.5}$$

shown in Figure 1.5 is the *row vector*

$$\boldsymbol{x}^\top = (x_1, x_2). \tag{1.6}$$

The *sum* of two column vectors is given by

$$\boldsymbol{x}+\boldsymbol{y} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}, \tag{1.7}$$

and their *inner product* by

$$\boldsymbol{x}^\top \boldsymbol{y} = (x_1, x_2) \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = x_1 y_1 + x_2 y_2. \tag{1.8}$$

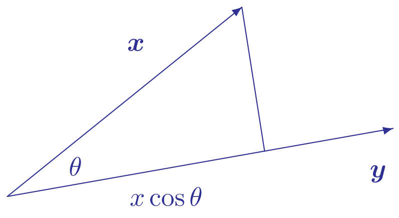


FIGURE 1.6
Illustrating the inner product.

The *length* or *Euclidean norm* of the vector $\boldsymbol{x}$ is

$$\|\boldsymbol{x}\| = \sqrt{x_1^2 + x_2^2} = \sqrt{\boldsymbol{x}^\top \boldsymbol{x}}\ . \tag{1.9}$$

The inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$ can be expressed in terms of the vector lengths and the angle θ between the two vectors:

$$\boldsymbol{x}^\top \boldsymbol{y} = \|\boldsymbol{x}\| \|\boldsymbol{y}\| \cos\theta; \tag{1.10}$$

see Figure 1.6 and Exercise 1. If $\theta = 90^o$, the vectors are said to be *orthogonal*, in which case $\boldsymbol{x}^\top \boldsymbol{y} = 0$.

Any vector can be expressed in terms of orthogonal *unit vectors*, e.g.,

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} = x_1 \boldsymbol{i} + x_2 \boldsymbol{j} \ , \tag{1.11}$$

where the symbols $\boldsymbol{i}$ and $\boldsymbol{j}$ denote vectors of unit length along the x and y directions, respectively.

A 2×2 *matrix* is written in the form

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}. \tag{1.12}$$

The first index of a matrix element indicates its row, the second its column. Numerical Python (Python plus `numpy`) indexes two-dimensional array and matrix objects in this way. When a matrix is multiplied with a vector, the result is another vector, e.g.,

$$\boldsymbol{A}\boldsymbol{x} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{pmatrix} = x_1 \begin{pmatrix} a_{11} \\ a_{21} \end{pmatrix} + x_2 \begin{pmatrix} a_{12} \\ a_{22} \end{pmatrix}.$$

In general, for $\boldsymbol{A} = (\boldsymbol{a}_1, \boldsymbol{a}_2 \dots \boldsymbol{a}_N)$, where the vectors $\boldsymbol{a}_i$ are the columns of $\boldsymbol{A}$,

$$\boldsymbol{A}\boldsymbol{x} = x_1 \boldsymbol{a}_1 + x_2 \boldsymbol{a}_2 + \cdots + \boldsymbol{x}_N \boldsymbol{a}_N. \tag{1.13}$$

The Python `numpy` package is very efficient in manipulating arrays, vectors and matrices. The scalar multiplication operator $*$ is interpreted as matrix multiplication for matrix operands. Running the Python interpreter in a command window as opposed to a Jupyter notebook, for example:

```
>>> import numpy
>>> X = numpy.mat([[1],[2]])
>>> A = numpy.mat([[1,2],[3,4]])
>>> print(A)
[[1 2]
 [3 4]]
>>> print(A*X)
[[ 5]
 [11]]
```

The product of two 2×2 matrices is given by

$$\boldsymbol{A}\boldsymbol{B} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

and is another matrix. More generally, the matrix product $\boldsymbol{A}\boldsymbol{B}$ is allowed whenever $\boldsymbol{A}$ has the same number of columns as $\boldsymbol{B}$ has rows. So if $\boldsymbol{A}$ has dimension $\ell \times m$ and $\boldsymbol{B}$ has dimension $m \times n$, then $\boldsymbol{A}\boldsymbol{B}$ is $\ell \times n$ with elements

$$(\boldsymbol{A}\boldsymbol{B})_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \quad i = 1 \dots \ell, \ j = 1 \dots n. \tag{1.14}$$

Matrix multiplication is not *commutative*, i.e., $\boldsymbol{AB} \neq \boldsymbol{BA}$ in general. However, it is *associative*:

$$(\boldsymbol{AB})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{BC}) \tag{1.15}$$

so we can write, for example,

$$(\boldsymbol{AB})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{BC}) = \boldsymbol{ABC} \tag{1.16}$$

without ambiguity. The *outer product* of two vectors of equal length, written $\boldsymbol{xy}^\top$, is a matrix, e.g.,

$$\boldsymbol{xy}^\top = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} (y_1, y_2) = \begin{pmatrix} x_1 & 0 \\ x_2 & 0 \end{pmatrix} \begin{pmatrix} y_1 & y_2 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} x_1y_1 & x_1y_2 \\ x_2y_1 & x_2y_2 \end{pmatrix}. \tag{1.17}$$

Matrices, like vectors, have a transposed form, obtained by interchanging their rows and columns:

$$\boldsymbol{A}^\top = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix}. \tag{1.18}$$

Transposition has the properties

$$\begin{aligned} (\boldsymbol{A}+\boldsymbol{B})^\top &= \boldsymbol{A}^\top + \boldsymbol{B}^\top \\ (\boldsymbol{AB})^\top &= \boldsymbol{B}^\top \boldsymbol{A}^\top. \end{aligned} \tag{1.19}$$

The analogous operation to transposition for complex vectors and matrices is *conjugate transposition* (see Appendix A), for which a_{ij} is replaced by a^*_{ji}, where the asterisk denotes complex conjugation. (The complex conjugate of $a + \mathbf{i}b$ is $a - \mathbf{i}b$.) We will write the conjugate transpose of $\boldsymbol{A}$ as $\boldsymbol{A}^\dagger$. Conjugate transposition has the same properties as given above for ordinary transposition.

1.3.2 Square matrices

A *square* matrix $\boldsymbol{A}$ has equal numbers of rows and columns. The *determinant* of a $p \times p$ square matrix is written $|\boldsymbol{A}|$ and defined as

$$|\boldsymbol{A}| = \sum_{(j_1 \ldots j_p)} (-1)^{f(j_1 \ldots j_p)} a_{1j_1} a_{2j_2} \cdots a_{pj_p}. \tag{1.20}$$

The sum is taken over all permutations $(j_1 \ldots j_p)$ of the integers $(1 \ldots p)$ and $f(j_1 \ldots j_p)$ is the number of transpositions (interchanges of two integers) required to change $(1 \ldots p)$ into $(j_1 \ldots j_p)$. The determinant of a 2×2 matrix, for example, is given by

$$|\boldsymbol{A}| = a_{11}a_{22} - a_{12}a_{21}.$$

The determinant has the properties

$$\begin{aligned}|\boldsymbol{AB}| &= |\boldsymbol{A}||\boldsymbol{B}| \\ |\boldsymbol{A}^\top| &= |\boldsymbol{A}|.\end{aligned} \tag{1.21}$$

The *identity matrix* is a square matrix with ones along its diagonal and zeroes everywhere else. For example,

$$\boldsymbol{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

and for any $\boldsymbol{A}$,

$$\boldsymbol{IA} = \boldsymbol{AI} = \boldsymbol{A}. \tag{1.22}$$

The *matrix inverse* $\boldsymbol{A}^{-1}$ of a square matrix $\boldsymbol{A}$ is defined in terms of the identity matrix by the requirements

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}. \tag{1.23}$$

For example, it is easy to verify that a 2×2 matrix has inverse

$$\boldsymbol{A}^{-1} = \frac{1}{|\boldsymbol{A}|}\begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix}.$$

In Python, continuing the previous dialogue:

```
>>> print(numpy.linalg.det(A)) # determinant
-2.0
>>> print(A.I) # shorthand for inverse
[[-2.   1. ]
 [ 1.5 -0.5]]
>>> print(A.I*A) # yields identity matrix
[[  1.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   1.00000000e+00]]
>>>
```

The matrix inverse has the properties

$$\begin{aligned}(\boldsymbol{AB})^{-1} &= \boldsymbol{B}^{-1}\boldsymbol{A}^{-1} \\ (\boldsymbol{A}^{-1})^\top &= (\boldsymbol{A}^\top)^{-1}.\end{aligned} \tag{1.24}$$

If the transpose of a square matrix is its inverse, i.e., if

$$\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{I}, \tag{1.25}$$

then it is referred to as an *orthonormal matrix.*

A system of n linear equations of the form

$$y_i = \sum_{j=1}^{n} a_j x_j(i), \quad i = 1 \ldots n, \tag{1.26}$$

can be written in matrix notation as

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{a}, \tag{1.27}$$

where $\boldsymbol{y} = (y_1 \dots y_n)^\top$, $\boldsymbol{a} = (a_1 \dots a_n)^\top$, and $A_{ij} = x_j(i)$. Provided $\boldsymbol{A}$ is non singular (see below), the solution for the parameter vector $\boldsymbol{a}$ is given by

$$\boldsymbol{a} = \boldsymbol{A}^{-1}\boldsymbol{y}. \tag{1.28}$$

The *trace* of a square matrix is the sum of its diagonal elements, e.g., for a 2×2 matrix,

$$\mathrm{tr}(\boldsymbol{A}) = a_{11} + a_{22}. \tag{1.29}$$

The trace has the properties

$$\begin{aligned} \mathrm{tr}(\boldsymbol{A} + \boldsymbol{B}) &= \mathrm{tr}\boldsymbol{A} + \mathrm{tr}\boldsymbol{B} \\ \mathrm{tr}(\boldsymbol{A}\boldsymbol{B}) &= \mathrm{tr}(\boldsymbol{B}\boldsymbol{A}). \end{aligned} \tag{1.30}$$

1.3.3 Singular matrices

If $|\boldsymbol{A}| = 0$, then $\boldsymbol{A}$ has no inverse and is said to be a *singular matrix.* If $\boldsymbol{A}$ is non singular, then the equation

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0} \tag{1.31}$$

only has the so-called *trivial solution* $\boldsymbol{x} = \boldsymbol{0}$. To see this, multiply from the left with $\boldsymbol{A}^{-1}$. Then $\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{I}\boldsymbol{x} = \boldsymbol{x} = \boldsymbol{0}$.

If $\boldsymbol{A}$ is singular, Equation (1.31) has at least one *non-trivial solution* $\boldsymbol{x} \neq \boldsymbol{0}$. This, again, is easy to see for a 2×2 matrix. Suppose $|\boldsymbol{A}| = 0$. Writing Equation (1.31) out fully:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= 0 \\ a_{21}x_1 + a_{22}x_2 &= 0. \end{aligned}$$

To get a non-trivial solution, assume without loss of generality that $a_{12} \neq 0$. Just choose $x_1 = 1$. Then the above two equations imply that

$$x_2 = -\frac{a_{11}}{a_{12}} \quad \text{and} \quad a_{21} - a_{22}\frac{a_{11}}{a_{12}} = 0.$$

The latter equality is satisfied because $|\boldsymbol{A}| = a_{11}a_{22} - a_{12}a_{21} = 0$.

1.3.4 Symmetric, positive definite matrices

The *covariance matrix*, which we shall meet in the next chapter and which plays a central role in digital image analysis, is both *symmetric* and *positive definite.*

DEFINITION 1.1 *A square matrix is said to be* symmetric *if* $\boldsymbol{A}^\top = \boldsymbol{A}$. *The* $p \times p$ *matrix* $\boldsymbol{A}$ *is* positive definite *if*

$$\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} > 0 \tag{1.32}$$

for all p-dimensional vectors $\boldsymbol{x} \neq \boldsymbol{0}$.

The expression $\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}$ in the above definition is called a *quadratic form*. If $\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} \geq 0$ for all $\boldsymbol{x}$, then $\boldsymbol{A}$ is *positive semi-definite*. Definition 1.1 can be generalized to complex matrices; see Exercise 9.

We can extract the covariance matrix from a multispectral image in the accompanying Jupyter notebook as follows:

```python
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly

gdal.AllRegister()
infile = 'imagery/AST_20070501.tif'
inDataset = gdal.Open(infile,GA_ReadOnly)
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize

#  data matrix
G = np.zeros((rows*cols,3))
k = 0
for b in range(3):
    band = inDataset.GetRasterBand(b+1)
    tmp = band.ReadAsArray(0,0,cols,rows).ravel()
    G[:,b] = tmp - np.mean(tmp)

#  covariance matrix
C = np.mat(G).T*np.mat(G)/(cols*rows-1)
C
```

The image bands, after subtraction of their mean values, are read into an array `G`, in which each pixel vector constitutes a row of the array. We will refer to such an array as a *centered data design matrix* or simply *data matrix*; see Section 2.3.1. Many of our Python programs will involve manipulations of the data matrix. Here, the data matrix is transposed, multiplied by itself and divided by the number of observations minus one to obtain (more precisely: *to estimate*) the covariance matrix. This will be explained in Chapter 2. The result for the three VNIR bands 1, 2, and 3N of the ASTER image in Figure 1.1 is a symmetric 3×3 matrix:

```
matrix([[ 407.13229638,  442.18038527,  -78.32374081],
        [ 442.18038527,  493.57036427, -120.64197555],
        [ -78.32374081, -120.64197555,  438.95704379]])
```

We will see how to show that this matrix is positive definite in Section 1.4.

1.3.5 Linear dependence and vector spaces

Vectors are said to be *linearly dependent* when any one can be expressed as a linear combination of the others. Here is a formal definition:

DEFINITION 1.2 *A set S of vectors $\boldsymbol{x}_1 \ldots \boldsymbol{x}_r$ is said to be* linearly dependent *if there exist scalars $c_1 \ldots c_r$, not all of which are zero, such that $\sum_{i=1}^{r} c_i \boldsymbol{x}_i = \boldsymbol{0}$. Otherwise, they are* linearly independent*.*

A matrix is said to have *rank* r if the maximum number of linearly independent columns is r. If the $p \times p$ matrix $\boldsymbol{A} = (\boldsymbol{a}_1 \ldots \boldsymbol{a}_p)$, where $\boldsymbol{a}_i$ is its ith column, is nonsingular, then it has full rank p. If this were not the case, then there must exist a set of scalars $c_1 \ldots c_p$, not all of which are zero, for which

$$c_1\boldsymbol{a}_1 + \ldots + c_p\boldsymbol{a}_p = \boldsymbol{A}\boldsymbol{c} = \boldsymbol{0}.$$

In other words, there would be a nontrivial solution to $\boldsymbol{A}\boldsymbol{c} = \boldsymbol{0}$, contradicting the fact that $\boldsymbol{A}$ is non singular.

The set S in Definition 1.2 is said to constitute a *basis* for a *vector space* V, comprising all vectors that can be expressed as a linear combination of the vectors in S. The number r of vectors in the basis is called the *dimension* of V. The vector space V is also an *inner product space* by virtue of the inner product definition Equation (1.8) in Subsection 1.3.1. Inner product spaces are elaborated upon in Appendix A.

1.3.6 Eigenvalues and eigenvectors

In image analysis, it is frequently necessary to solve an *eigenvalue problem.* In the simplest case, and the one which will concern us primarily, the eigenvalue problem consists of finding *eigenvectors* $\boldsymbol{u}$ and *eigenvalues* λ that satisfy the matrix equation

$$\boldsymbol{A}\boldsymbol{u} = \lambda\boldsymbol{u}, \tag{1.33}$$

where $\boldsymbol{A}$ is both symmetric and positive definite. Geometrically, we seek special vectors $\boldsymbol{u}$ that, when matrix multiplied with $\boldsymbol{A}$, change at most their sign and length but not their direction: these are the "own" or "eigen" vectors of $\boldsymbol{A}$. Equation (1.33) can be written equivalently as

$$(\boldsymbol{A} - \lambda\boldsymbol{I})\boldsymbol{u} = \boldsymbol{0}, \tag{1.34}$$

so for a nontrivial solution for $\boldsymbol{u}$, we must have

$$|\boldsymbol{A} - \lambda\boldsymbol{I}| = 0. \tag{1.35}$$

This is known as the *characteristic equation* for the matrix $\boldsymbol{A}$. For instance, in the case of a 2×2 matrix eigenvalue problem,

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \lambda \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}, \tag{1.36}$$

the characteristic equation is, for a symmetric matrix,

$$(a_{11}-\lambda)(a_{22}-\lambda)-a_{12}^2=0,$$

which is a quadratic equation in λ with solutions

$$\begin{aligned}\lambda^{(1)} &= \frac{1}{2}\left(a_{11}+a_{22}+\sqrt{(a_{11}+a_{22})^2-4(a_{11}a_{22}-a_{12}^2)}\right)\\ \lambda^{(2)} &= \frac{1}{2}\left(a_{11}+a_{22}-\sqrt{(a_{11}+a_{22})^2-4(a_{11}a_{22}-a_{12}^2)}\right).\end{aligned} \tag{1.37}$$

Thus there are two eigenvalues and, correspondingly, two eigenvectors $\boldsymbol{u}^{(1)}$ and $\boldsymbol{u}^{(2)}$.* The eigenvectors can be obtained by first substituting $\lambda^{(1)}$ and then $\lambda^{(2)}$ into Equation (1.36) and solving for u_1 and u_2 each time. It is easy to show (Exercise 10) that the eigenvectors are orthogonal

$$(\boldsymbol{u}^{(1)})^\top \boldsymbol{u}^{(2)}=0. \tag{1.38}$$

Moreover, since the left- and right-hand sides of Equation (1.33) can be multiplied by any constant, the eigenvectors can always be chosen to have unit length, $\|\boldsymbol{u}^{(1)}\| = \|\boldsymbol{u}^{(2)}\| = 1$. The matrix formed by two such eigenvectors, e.g.,

$$\boldsymbol{U}=(\boldsymbol{u}^{(1)},\boldsymbol{u}^{(2)})=\begin{pmatrix}u_1^{(1)} & u_1^{(2)}\\ u_2^{(1)} & u_2^{(2)}\end{pmatrix}, \tag{1.39}$$

is said to *diagonalize* the matrix $\boldsymbol{A}$. That is, if $\boldsymbol{A}$ is multiplied from the left by $\boldsymbol{U}^\top$ and from the right by $\boldsymbol{U}$, the result is a diagonal matrix with the eigenvalues along the diagonal:

$$\boldsymbol{U}^\top\boldsymbol{A}\boldsymbol{U}=\boldsymbol{\Lambda}=\begin{pmatrix}\lambda^{(1)} & 0\\ 0 & \lambda^{(2)}\end{pmatrix}, \tag{1.40}$$

as can easily be verified. Note that $\boldsymbol{U}$ is an orthonormal matrix: $\boldsymbol{U}^\top\boldsymbol{U}=\boldsymbol{I}$ and therefore Equation (1.40) can be written as

$$\boldsymbol{A}\boldsymbol{U}=\boldsymbol{U}\boldsymbol{\Lambda}. \tag{1.41}$$

All of the above statements generalize in a straightforward fashion to square matrices of any size.

Suppose that $\boldsymbol{A}$ is a $p\times p$ symmetric matrix. Then its eigenvectors $\boldsymbol{u}^{(j)}$, $j=1\dots p$, are orthogonal and any p-component vector $\boldsymbol{x}$ can be expressed as a linear combination of them,

$$\boldsymbol{x}=\sum_{j=1}^{p}\beta_j\boldsymbol{u}^{(j)}. \tag{1.42}$$

*The special case of degeneracy, which arises when the two eigenvalues are equal, will be ignored here.

Multiplying from the right with $\boldsymbol{u}^{(i)\top}$,

$$\boldsymbol{u}^{(i)\top}\boldsymbol{x}\ (= \boldsymbol{x}^\top \boldsymbol{u}^{(i)}) = \sum_{j=1}^{p} \beta_j \boldsymbol{u}^{(i)\top}\boldsymbol{u}^{(j)} = \beta_i, \tag{1.43}$$

so that we have

$$\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} = \boldsymbol{x}^\top \sum_i \beta_i \lambda_i \boldsymbol{u}^{(i)} = \sum_i \beta_i^2 \lambda_i, \tag{1.44}$$

where λ_i, $i = 1 \ldots p$, are the eigenvalues of $\boldsymbol{A}$. We can conclude, from the definition of positive definite matrices given in Section 1.3.4, that $\boldsymbol{A}$ is positive definite if and only if all of its eigenvalues λ_i, $i = 1 \ldots N$, are positive. This holds also for Hermitian matrices. (See Appendix A. The reader is asked to show in Exercise 9 that the eigenvalues of an Hermitian matrix $\boldsymbol{A}^\dagger = \boldsymbol{A}$ are real numbers.)

The eigenvalue problem for symmetric and Hermitian matrices can be solved in Python with the built-in function `numpy.linalg.eigh()`. This function returns arrays for the eigenvalues and eigenvectors, the latter as the columns of the matrix $\boldsymbol{U}$:

```
eigenvalues, eigenvectors = np.linalg.eigh(C)
print(eigenvalues)
print(eigenvectors)

[   4.77425683  399.58595201  935.2994956 ]
[[-0.73352328  0.22653637 -0.64080018]
 [ 0.67736254  0.16613156 -0.71664517]
 [ 0.05588906  0.95972995  0.27530862]]

U = eigenvectors
print(U.T*U)

[[  1.00000000e+00  -7.63278329e-17   1.75207071e-16]
 [ -7.63278329e-17   1.00000000e+00   0.00000000e+00]
 [  1.75207071e-16   0.00000000e+00   1.00000000e+00]]
```

Notice that, due to rounding errors, $\boldsymbol{U}^\top\boldsymbol{U}$ has finite but very small off-diagonal elements. The eigenvalues are all positive, so the covariance matrix calculated in the script is positive definite.

1.3.7 Singular value decomposition

Rewriting Equation (1.41), we get a special form of *singular value decomposition* (SVD) for symmetric matrices:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top. \tag{1.45}$$

This says that any symmetric matrix $\boldsymbol{A}$ can be factored into the product of an orthonormal matrix $\boldsymbol{U}$ times a diagonal matrix $\boldsymbol{\Lambda}$, whose diagonal elements are the eigenvalues of $\boldsymbol{A}$, times the transpose of $\boldsymbol{U}$. This is also called the *eigendecomposition* or *spectral decomposition* of $\boldsymbol{A}$ and can be written alternatively as a sum over outer products of the eigenvectors (Exercise 11),

$$\boldsymbol{A} = \sum_{i=1}^{p} \lambda_i \boldsymbol{u}^{(i)} \boldsymbol{u}^{(i)\top}. \tag{1.46}$$

For the general form for singular value decomposition of nonsymmetric or nonsquare matrices, see Press et al. (2002).

SVD is a powerful tool for the solution of systems of linear equations and is often used when a solution cannot be determined by other numerical algorithms. To invert a nonsingular symmetric matrix $\boldsymbol{A}$, we simply write

$$\boldsymbol{A}^{-1} = \boldsymbol{U}\boldsymbol{\Lambda}^{-1}\boldsymbol{U}^\top, \tag{1.47}$$

since $\boldsymbol{U}^{-1} = \boldsymbol{U}^\top$. The matrix $\boldsymbol{\Lambda}^{-1}$ is the diagonal matrix whose diagonal elements are the inverses of the diagonal elements of $\boldsymbol{\Lambda}$. If $\boldsymbol{A}$ is singular (has no inverse), clearly at least one of its eigenvalues is zero.

The matrix $\boldsymbol{A}$ is said to be *ill-conditioned* or *nearly singular* if one or more of the diagonal elements of $\boldsymbol{\Lambda}$ is close to zero. SVD detects this situation effectively, since the factorization in Equation (1.45) is always possible, even if $\boldsymbol{A}$ is truly singular. The `numpy` procedure for singular value decomposition is `numpy.linalg.svd`. For example, the code:

```python
import numpy as np
b = np.mat([1,2,3])
# an almost singular matrix
A = b.T*b + np.random.rand(3,3)*0.001
# a symmetric almost singular matrix
A = A + A.T
print('determinant:␣%f'%np.linalg.det(A))
# singular value decomposition
U,Lambda,V = np.linalg.svd(A)
print('Lambda␣=␣%s'%str(Lambda))
print('U␣=␣%s'%str(U))
print('V␣=␣%s'%str(V))
```

```
determinant: -0.000010
Lambda = [ 2.80019985e+01  6.92874899e-04  4.90453619e-04]
U = [[-0.26728335 -0.58367438 -0.76673582]
 [-0.5345376  -0.5722321   0.62194853]
 [-0.80176628  0.57608561 -0.15904779]]
V = [[-0.26728335 -0.5345376  -0.80176628]
 [ 0.58367438  0.5722321  -0.57608561]
 [-0.76673582  0.62194853 -0.15904779]]
```

indicates that $\boldsymbol{A}$ is ill-conditioned (two of the three diagonal elements of $\boldsymbol{\Lambda}$ are close to zero).

If $\boldsymbol{A}$ is singular and we order the eigenvalues and eigenvectors by decreasing eigenvalue, then the eigendecomposition reads

$$\boldsymbol{A} = \sum_{i=1}^{r} \lambda_i \boldsymbol{u}^{(i)} \boldsymbol{u}^{(i)\top}, \tag{1.48}$$

where r is the number of nonzero eigenvalues. Accordingly, Equation (1.45) becomes

$$\boldsymbol{A} = \boldsymbol{U}_r \boldsymbol{\Lambda}_r \boldsymbol{U}_r^\top, \tag{1.49}$$

where

$$\boldsymbol{U}_r = (\boldsymbol{u}^{(1)}, \dots, \boldsymbol{u}^{(r)}), \quad \boldsymbol{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_r \end{pmatrix}.$$

The *pseudo-inverse* of the symmetric, singular matrix $\boldsymbol{A}$ is then defined as

$$\boldsymbol{A}^+ = \boldsymbol{U}_r \boldsymbol{\Lambda}_r^{-1} \boldsymbol{U}_r^\top. \tag{1.50}$$

The pseudo inverse has the property that $\boldsymbol{A}^+\boldsymbol{A} - \boldsymbol{I}$, where $\boldsymbol{I}$ is the $p \times p$ identity matrix, is minimized (Press et al., 2002) and for a full rank (nonsingular) matrix, $\boldsymbol{A}^+\boldsymbol{A} - \boldsymbol{I} = \boldsymbol{0}$, or $\boldsymbol{A}^+ = \boldsymbol{A}^{-1}$.

1.3.8 Tensors

The vectors and matrices of the preceding sections are special cases of multidimensional arrays, which nowadays in the context of machine learning are often referred to as *tensors*. The `numpy` package, as we have seen, is good at handling such arrays, but it is not optimized for parallel computation on GPU hardware such as Google Colab. The `tensorflow` and `pytorch` packages are designed to run both on CPUs and GPUs and have established themselves in *deep learning* applications (which are essentially synonymous with large neural networks). The more "pythonic" of the two is `pytorch`; however, we will prefer to use `tensorflow` wherever appropriate since it is more closely integrated with the GEE. The `numpy` and `tensorflow` arrays are virtually interchangeable:

```
import tensorflow as tf
import numpy as np
data = [[1., 2.],[3., 4.]]
tf_data = tf.constant(data)
np_array = np.array(tf_data)
tf_np = tf.constant(np_array)
print(tf_np)
```

```
tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)
```

but note that the default precision in `tensorflow` (and `pytorch`) computations is `float32` as opposed to `float64` in `numpy`.*

1.4 Finding minima and maxima

In order to enhance some desirable properties of a remote sensing image, such as signal-to-noise ratio or spread in intensity, we often need to take derivatives with respect to vectors. A *vector partial derivative* operator is written in the form $\frac{\partial}{\partial \boldsymbol{x}}$. For instance, in two dimensions,

$$\frac{\partial}{\partial \boldsymbol{x}} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \frac{\partial}{\partial x_1} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \frac{\partial}{\partial x_2} = \boldsymbol{i} \frac{\partial}{\partial x_1} + \boldsymbol{j} \frac{\partial}{\partial x_2}.$$

Such an operator (also called a *gradient*) arranges the partial derivatives of any scalar function of the vector $\boldsymbol{x}$ with respect to each of the components of $\boldsymbol{x}$ into a column vector, e.g.,

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \frac{\partial f(\boldsymbol{x})}{\partial x_2} \end{pmatrix}.$$

Many of the operations with vector derivatives correspond exactly to operations with ordinary scalar derivatives (and can all be verified easily by writing out the expressions component by component):

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{x}^\top \boldsymbol{y}) = \boldsymbol{y} \quad \text{analogous to} \quad \frac{\partial}{\partial x} xy = y$$

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{x}^\top \boldsymbol{x}) = 2\boldsymbol{x} \quad \text{analogous to} \quad \frac{\partial}{\partial x} x^2 = 2x.$$

For quadratic forms, we have

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{y}) = \boldsymbol{A}\boldsymbol{y}$$
$$\frac{\partial}{\partial \boldsymbol{y}}(\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{y}) = \boldsymbol{A}^\top \boldsymbol{x}$$

and

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{A}^\top \boldsymbol{x}.$$

*See also https://www.tensorflow.org/guide/basics.

Note that, if $\boldsymbol{A}$ is a symmetric matrix, this last equation can be written

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}) = 2\boldsymbol{A}\boldsymbol{x}. \tag{1.51}$$

Suppose x^* is a stationary point of the function $f(x)$, by which is meant that its first derivative vanishes at that point:

$$\frac{d}{dx}f(x^*) = \frac{d}{dx}f(x)\Big|_{x=x^*} = 0; \tag{1.52}$$

see Figure 1.7. Then $f(x^*)$ is a local minimum if the second derivative at x^* is positive,

$$\frac{d^2}{dx^2}f(x^*) > 0.$$

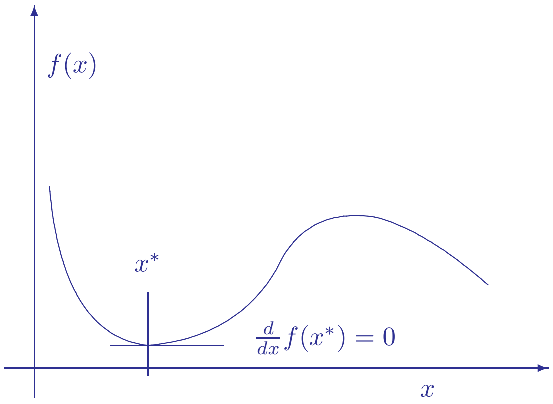


FIGURE 1.7
A function of one variable with a minimum at x^*.

This becomes obvious if $f(x)$ is expanded in a *Taylor series* about x^*,

$$f(x) = f(x^*) + (x - x^*)\frac{d}{dx}f(x^*) + \frac{1}{2}(x - x^*)^2\frac{d^2}{dx^2}f(x^*) + \dots\ . \tag{1.53}$$

The second term is zero, so for $|x - x^*|$ sufficiently small, $f(x)$ is approximately a quadratic function

$$f(x) \approx f(x^*) + \frac{1}{2}(x - x^*)^2\frac{d^2}{dx^2}f(x^*), \tag{1.54}$$

with a minimum at x^* when the second derivative is positive, a maximum when it is negative, and a point of inflection when it is zero.

The situation is similar for scalar functions of a vector. In this case, the Taylor expansion is

$$f(\boldsymbol{x}) = f(\boldsymbol{x}^*) + (\boldsymbol{x} - \boldsymbol{x}^*)^\top \frac{\partial f(\boldsymbol{x}^*)}{\partial \boldsymbol{x}} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^*)^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^*) + \dots, \quad (1.55)$$

where $\boldsymbol{H}$ is called the *Hessian matrix*. Its elements are given by

$$(\boldsymbol{H})_{ij} = \frac{\partial^2 f(\boldsymbol{x}^*)}{\partial x_i \partial x_j}, \quad (1.56)$$

and it is a symmetric matrix.* At the stationary point, the vector derivative with respect to $\boldsymbol{x}$ vanishes,

$$\frac{\partial f(\boldsymbol{x}^*)}{\partial \boldsymbol{x}} = \mathbf{0},$$

so we get the second-order approximation

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^*) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^*)^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^*). \quad (1.57)$$

Now the condition for a local minimum is clearly that the Hessian matrix be positive definite at the point $\boldsymbol{x}^*$. Note that the Hessian can be expressed in terms of an outer product of vector derivatives as

$$\frac{\partial}{\partial \boldsymbol{x}} \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}^\top} = \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x} \partial \boldsymbol{x}^\top}. \quad (1.58)$$

Suppose that we wish to find the position $\boldsymbol{x}^*$ of a minimum (or maximum) of a scalar function $f(\boldsymbol{x})$. If there are no constraints, then we solve the set of equations

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = 0, \quad i = 1, 2 \dots$$

or, in terms of our notation for vector derivatives,

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \mathbf{0}, \quad (1.59)$$

and examine the Hessian matrix at the solution point. However, suppose that $\boldsymbol{x}$ is *constrained* by the condition

$$h(\boldsymbol{x}) = 0. \quad (1.60)$$

For example, in two dimensions we might have the constraint

$$h(\boldsymbol{x}) = x_1^2 + x_2^2 - 1 = 0,$$

*Because the order of partial differentiation does not matter.

which requires $\boldsymbol{x}$ to lie on a circle of radius 1. An obvious procedure would be to solve the above equation for x_1, say, and then substitute the result into Equation (1.59) to determine x_2. This method is not always practical, however. It may be the case, for example, that the constraint Equation (1.60) cannot be solved analytically.

A more convenient and generally applicable way to find an extremum of $f(\boldsymbol{x})$ subject to $h(\boldsymbol{x}) = 0$ is to determine an *unconstrained* minimum or maximum of the expression

$$L(\boldsymbol{x}) = f(\boldsymbol{x}) + \lambda h(\boldsymbol{x}). \tag{1.61}$$

This is called a *Lagrange function* and λ is a *Lagrange multiplier*. The Lagrange multiplier is treated as though it were an additional variable. To find the extremum, we solve the set of equations

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{x}}(f(\boldsymbol{x}) + \lambda h(\boldsymbol{x})) &= \boldsymbol{0}, \\ \frac{\partial}{\partial \lambda}(f(\boldsymbol{x}) + \lambda h(\boldsymbol{x})) &= 0 \end{aligned} \tag{1.62}$$

for $\boldsymbol{x}$ and λ. To see this, note that a minimum or maximum need not generally occur at a point $\boldsymbol{x}^*$ for which

$$\frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{x}^*) = \boldsymbol{0}, \tag{1.63}$$

because of the presence of the constraint. This possibility is taken into account in the first of Equations (1.62), which at an extremum reads

$$\frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{x}^*) = -\lambda \frac{\partial}{\partial \boldsymbol{x}} h(\boldsymbol{x}^*). \tag{1.64}$$

It implies that, if $\lambda \neq 0$, any small change in $\boldsymbol{x}$ away from $\boldsymbol{x}^*$ causing a change in $f(\boldsymbol{x}^*)$, i.e., any change in $\boldsymbol{x}$ which is not orthogonal to the gradient of $f(\boldsymbol{x}^*)$, would be accompanied by a proportional change in $h(\boldsymbol{x}^*)$. This would necessarily violate the constraint, so $\boldsymbol{x}^*$ is an extremum. The second equation is just the constraint $h(\boldsymbol{x}^*) = 0$ itself. For more detailed justifications of this procedure, see Bishop (1995), Appendix C; Milman (1999), Chapter 14; or Cristianini and Shawe-Taylor (2000), Chapter 5.

As a first example illustrating the Lagrange method, let $f(\boldsymbol{x}) = ax_1^2 + bx_2^2$ and $h(\boldsymbol{x}) = x_1 + x_2 - 1$. Then we get the three equations

$$\begin{aligned} \frac{\partial}{\partial x_1}(f(\boldsymbol{x}) + \lambda h(\boldsymbol{x})) &= 2ax_1 + \lambda = 0 \\ \frac{\partial}{\partial x_2}(f(\boldsymbol{x}) + \lambda h(\boldsymbol{x})) &= 2bx_2 + \lambda = 0 \\ \frac{\partial}{\partial \lambda}(f(\boldsymbol{x}) + \lambda h(\boldsymbol{x})) &= x_1 + x_2 - 1 = 0. \end{aligned}$$

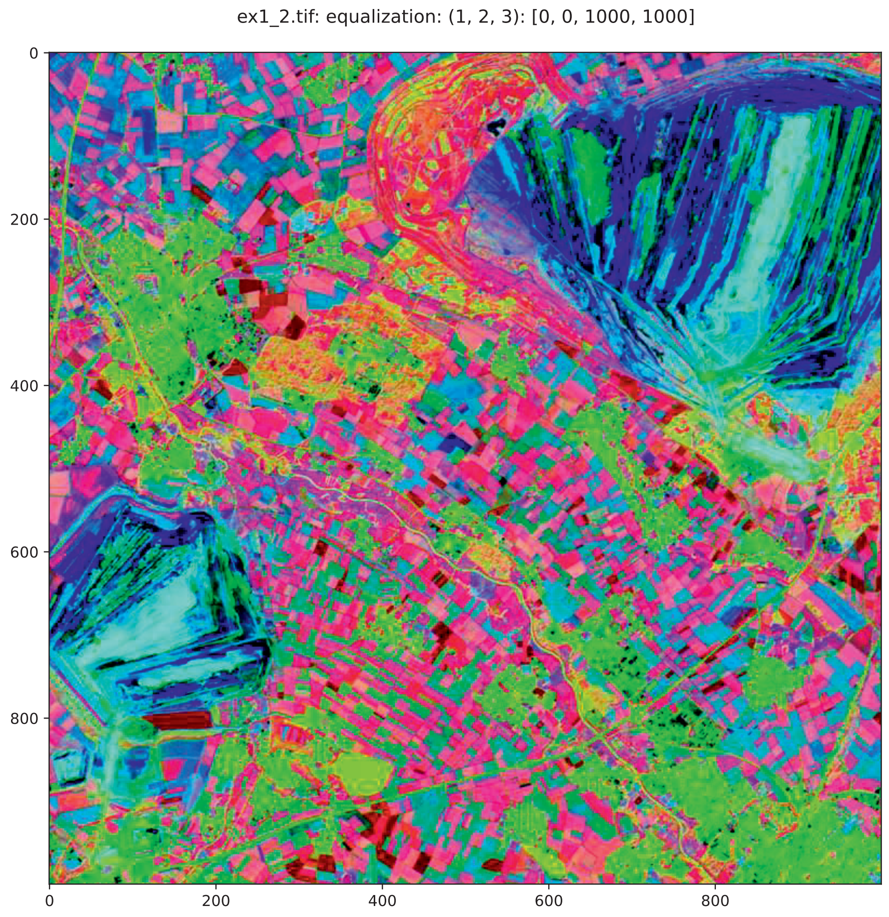


FIGURE 1.8
RGB composite of the first three principal components of the spectral bands for the ASTER image of Figure 1.1 calculated with the script `ex1_2.py` in Listing 1.2 and displayed with the Python script `dispms.py`.

The solution for $\boldsymbol{x}$ is

$$x_1 = \frac{b}{a+b}, \quad x_2 = \frac{a}{a+b}.$$

A second—and very important—example: Let us find the maximum of

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{C}\boldsymbol{x},$$

where $\boldsymbol{C}$ is symmetric positive definite, subject to the constraint

$$h(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{x} - 1 = 0.$$

Listing 1.2: Principal components analysis in Python.

```
#!/usr/bin/env python3
#Name:  ex1_2.py
import numpy as np
from osgeo import gdal
import sys
from osgeo.gdalconst import GA_ReadOnly, GDT_Float32

def pca(infile, outfile):
    gdal.AllRegister()
    inDataset = gdal.Open(infile, GA_ReadOnly)
    cols = inDataset.RasterXSize
    rows = inDataset.RasterYSize
    bands = inDataset.RasterCount
    #  data matrix
    G = np.zeros((rows*cols,bands))
    k = 0
    for b in range(bands):
        band = inDataset.GetRasterBand(b+1)
        tmp = band.ReadAsArray(0,0,cols,rows).ravel()
        G[:, b] = tmp - np.mean(tmp)
    #  covariance matrix
    C = np.mat(G).T*np.mat(G)/(cols*rows-1)
    #  diagonalize
    lams,U = np.linalg.eigh(C)
    #  sort
    idx = np.argsort(lams)[::-1]
    lams = lams[idx]
    U = U[:, idx]
    #  project
    PCs = np.reshape(np.array(G*U),(rows,cols,bands))
    #  write to disk
    if outfile:
        driver = gdal.GetDriverByName('Gtiff')
        outDataset = driver.Create(outfile,
                        cols,rows,bands,GDT_Float32)
        projection = inDataset.GetProjection()
        if projection is not None:
            outDataset.SetProjection(projection)
        for k in range(bands):
            outBand = outDataset.GetRasterBand(k+1)
            outBand.WriteArray(PCs[:, :, k], 0, 0)
            outBand.FlushCache()
        outDataset = None
    inDataset = None
```

The Lagrange function is

$$L(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{C}\boldsymbol{x} - \lambda(\boldsymbol{x}^\top \boldsymbol{x} - 1).$$

Any extremum of L must occur at a value of $\boldsymbol{x}$ for which

$$\frac{\partial L}{\partial \boldsymbol{x}} = 2\boldsymbol{C}\boldsymbol{x} - 2\lambda\boldsymbol{x} = 0,$$

which is the eigenvalue problem

$$\boldsymbol{C}\boldsymbol{x} = \lambda\boldsymbol{x}. \tag{1.65}$$

Let $\boldsymbol{u}$ be an eigenvector with eigenvalue λ. Then

$$f(\boldsymbol{u}) = \boldsymbol{u}^\top \boldsymbol{C}\boldsymbol{u} = \lambda\boldsymbol{u}^\top \boldsymbol{u} = \lambda.$$

So to maximize $f(\boldsymbol{x})$, we must choose the eigenvector of $\boldsymbol{C}$ with maximum eigenvalue.

If $\boldsymbol{C}$ is the covariance matrix of a multispectral image, then, as we shall see in Chapter 3, the above maximization corresponds to a *principal components analysis* (PCA). Listing 1.2 gives a rudimentary function for doing PCA on a multispectral image in `numpy`. After estimating the covariance matrix in line 22 (see Chapter 2), the eigenvalue problem, Equation (1.65), is solved in line 24 using the `numpy.linalg.eigh()` function. The order of eigenvalues `lams` and eigenvectors (the columns of `U`) is undefined, so the eigenvalues and eigenvectors are sorted into decreasing order in lines 26 and 27. Then the principal components are calculated by projecting the original image bands along the eigenvectors and the resulting data matrix is rearranged in BIP format (line 30). Finally, the principal components are stored on disk, preserving the map projection.

The following commands from within the Jupyter notebook run the script and display an RGB color composite of the first three principal components:

```
%run scripts/ex1_2 imagery/AST_20070501.tif \
                   imagery/ex1_2.tif
%run scripts/dispms -f 'imagery/ex1_2.tif' \
                    -p [1,2,3] -e 4
```

see Figure 1.8. See Appendix C for documentation of the script `dispms.py` invoked above. It can be used to view histogram-enhanced RGB composites of 3-band combinations of any multi-band image stored on disk.

1.5 Exercises

1. Demonstrate that the definition
$$\boldsymbol{x}^\top \boldsymbol{y} = \|\boldsymbol{x}\|\|\boldsymbol{y}\| \cos\theta$$
is equivalent to
$$\boldsymbol{x}^\top \boldsymbol{y} = x_1 y_1 + x_2 y_2 .$$
Hint: Use the trigonometric identity $\cos(\alpha-\beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$.

2. Show that the outer product of two two-dimensional vectors is a singular matrix. What is its rank?

3. Verify the matrix identity
$$(\boldsymbol{AB})^\top = \boldsymbol{B}^\top \boldsymbol{A}^\top$$
in Python. You must import the `numpy` package.

4. Show that three two-dimensional vectors representing three points, all lying on the same line, are linearly dependent.

5. Show that the determinant of a symmetric 2×2 matrix is given by the product of its eigenvalues.

6. Prove that the inverse of a symmetric nonsingular matrix is symmetric.

7. Prove that the eigenvectors of $\boldsymbol{A}^{-1}$ are the same as those of $\boldsymbol{A}$, but with reciprocal eigenvalues.

8. Prove the identity
$$\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} = \mathrm{tr}(\boldsymbol{A}\boldsymbol{x}\boldsymbol{x}^\top)$$
with the aid of the second of Equations (1.30).

9. A square complex matrix $\boldsymbol{A}$ is said to be *Hermitian* if $\boldsymbol{A}^\dagger = \boldsymbol{A}$, which is a generalization of Definition 1.1 for a symmetric real matrix. It is positive semi-definite if $\boldsymbol{x}^\dagger \boldsymbol{A}\boldsymbol{x} \geq 0$ for all non-zero complex vectors $\boldsymbol{x}$. The matrix $\boldsymbol{B} = \boldsymbol{A}^\dagger \boldsymbol{A}$ is obviously Hermitian (why?). Prove that, if the complex matrix $\boldsymbol{A}$ is Hermitian, its eigenvalues are real.

10. Prove that the eigenvectors of a 2×2 symmetric matrix are orthogonal.

11. Demonstrate the equivalence of Equations (1.45) and (1.46) for a symmetric 2×2 matrix.

12. Prove, from Equation (1.46), that the trace of a symmetric matrix is the sum of its eigenvalues.

13. Differentiate the function
$$\frac{1}{\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{y}}$$
with respect to $\boldsymbol{y}$.

14. Calculate the eigenvectors of the (nonsymmetric!) matrix
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$
with Python. You will need the `numpy.linalg` package.

15. Plot the function $f(\boldsymbol{x}) = x_1^2 - x_2^2$ using the `matplotlib` package. Find its minima and maxima subject to the constraint $h(\boldsymbol{x}) = x_1^2 + x_2^2 - 1 = 0$.

2

Image Statistics

In an optical/infrared or a synthetic aperture radar image, a given pixel value $g(i,j)$, derived from the measured radiation field at a satellite sensor, is never exactly reproducible. It is the outcome of a complex measurement influenced by instrument noise, atmospheric conditions, changing illumination, and so forth. It may be assumed, however, that there is an underlying random process or mechanism with an associated probability distribution which restricts the possible outcomes in some way. Each time we make an observation, we are sampling from that probability distribution or, put another way, we are observing a different possible *realization* of the random process. In this chapter, some basic statistical concepts for multispectral and SAR images viewed as random processes will be introduced.

2.1 Random variables

A *random variable* can be used to represent a quantity, in the present context an image gray-scale value, which changes in an unpredictable way each time it is observed. In order to make a precise definition, let us consider some chance experiment which has a set Ω of possible *outcomes*. This set is referred to as the *sample space* for the experiment. Subsets of Ω are called *events*. An event will be said to have *occurred* whenever the outcome of the experiment is contained within it.

To make this clearer, consider the random experiment consisting of the throw of two dice. The sample space is the set of 36 possible outcomes

$$\Omega = \{(1,1),(1,2),(2,1)\dots(6,6)\}.$$

An event is then, for example, that the sum of the points is 7. It is the subset

$$\{(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)\}$$

of the sample space. If, for instance, $(3,4)$ is thrown, then the event has occurred.

DOI: 10.1201/9781003503286-2

DEFINITION 2.1 *A* random variable $Z : \Omega \mapsto \mathbb{R}$ *is a function which maps all outcomes onto the set* $\mathbb{R}$ *of real numbers such that the set*

$$\{\omega \in \Omega \mid Z(\omega) \leq z\}$$

is an event, i.e., a subset of Ω. *This subset is usually abbreviated as* $\{Z \leq z\}$.

Thus, for the throw of two dice, the sum of points S is a random variable, since it maps all outcomes onto real numbers:

$$S(1,1) = 2,\ S(1,2) = S(2,1) = 3,\ \ldots\ S(6,6) = 12,$$

and sets such as

$$\{S \leq 4\} = \{(1,1),(1,2),(2,1),(1,3),(3,1),(2,2)\}$$

are subsets of the sample space. The set $\{S \leq 1\}$ is the empty set, whereas $\{S \leq 12\} = \Omega$, the entire sample space.

On the basis of the *probabilities* for the individual outcomes, we can associate a function $P(z)$ with the random variable Z as follows:

$$P(z) = \Pr(Z \leq z).$$

This is the probability of observing the event that the random variable Z takes on a value less than or equal to z. The probability of an event may be thought of as the relative frequency with which it occurs in n repetitions of a random experiment in the limit as $n \to \infty$. (For the complete, axiomatic definition, see, e.g., Freund (1992).) In the dice example, the probability of throwing a four or less is

$$P(4) = \Pr(S \leq 4) = 6/36 = 1/6,$$

for instance.

DEFINITION 2.2 *Given the random variable* Z, *then*

$$P(z) = \Pr(Z \leq z), \quad -\infty < z < \infty, \tag{2.1}$$

is called its distribution function.

2.1.1 Discrete random variables

When, as in the case of the dice throw, a random variable Z is discrete and takes on values $z_1 < z_2 < z_3 < \ldots$, then the probabilities of the separate outcomes

$$p(z_i) = \Pr(Z = z_i) = P(z_i) - P(z_{i-1}), \quad i = 1, 2 \ldots$$

are said to constitute the *mass function* for the random variable. This is best illustrated with a practical example. As we shall see in Chapter 7, the evaluation of a land cover classification model involves repeated trials with a finite number of independent test observations, keeping track of the number of times the model fails to predict the correct land cover category. This will lead us to consideration of a discrete random variable having the so-called *binomial distribution*. We can derive its mass function as follows.

Let θ be the probability of failure in a single trial. The probability of getting y misclassifications (and hence $n-y$ correct classifications) in n trials *in a specific sequence* is

$$\theta^y(1-\theta)^{n-y}.$$

In this expression, there is a factor θ for each of the y misclassifications and a factor $(1-\theta)$ for each of the $n-y$ correct classifications. Taking the product is justified by the assumption that the trials are independent of each other. The number of such sequences is just the number of ways of selecting y trials from n possible ones. This is given by the *binomial coefficient*

$$\binom{n}{y} = \frac{n!}{(n-y)!\;y!}, \tag{2.2}$$

so that the probability for y misclassifications in n trials is

$$\binom{n}{y}\theta^y(1-\theta)^{n-y}.$$

A discrete random variable Y is said to be *binomially distributed* with parameters n and θ if its mass function is given by

$$p_{n,\theta}(y) = \begin{cases} \binom{n}{y}\theta^y(1-\theta)^{n-y} & \text{for } y = 0,1,2\dots n \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

Note that the values of $p_{n,\theta}(y)$ are the terms in the binomial expansion of

$$[\theta + (1-\theta)]^n = 1^n = 1,$$

so the sum over the probabilities equals 1, as it should.

2.1.2 Continuous random variables

In the case of continuous random variables, which we are in effect dealing with when we speak of pixel intensities, the distribution function is not expressed in terms of the discrete probabilities of a mass function, but rather in terms of a *probability density function* $p(z)$.

DEFINITION 2.3 *A function with values $p(z)$, defined over the set of all real numbers, is called a probability density function of the continuous random* *variable Z if and only if*

$$\Pr(a \le Z \le b) = \int_a^b p(z)dz \tag{2.4}$$

for any real numbers $a \le b$.

The quantity $p(z)dz$ is the probability that the associated random variable Z lies within the infinitesimal interval $[z, z + dz]$. The integral (sum) over all such intervals is one:

$$\int_{-\infty}^{\infty} p(z)dz = 1. \tag{2.5}$$

The distribution function $P(z)$ can be written in terms of the density function and vice versa as

$$P(z) = \int_{-\infty}^{z} p(t)dt, \quad p(z) = \frac{d}{dz}P(z). \tag{2.6}$$

The distribution function has the limiting values

$$P(-\infty) = 0, \quad P(\infty) = \int_{-\infty}^{\infty} p(t)dt = 1.$$

The following theorem can often be used to determine the probability density of a function of some random variable whose density is known. For a proof, see Freund (1992).

THEOREM 2.1
Let $p_z(z)$ be the density function for random variable Z and

$$y = u(z) \tag{2.7}$$

a monotonic function of z for all values of z for which $p_z(z) \neq 0$. For these z values, Equation (2.7) can be solved for z to give $z = w(y)$. Then the density function of the random variable $Y = u(Z)$ is given by

$$p_y(y) = p_z(z)\left|\frac{dz}{dy}\right| = p_z(w(y))\left|\frac{dz}{dy}\right|. \tag{2.8}$$

As an example of the application of Theorem 2.1, suppose that a random variable Z has the exponential distribution (Section 2.1.5) with density function

$$p_z(z) = \begin{cases} e^{-z} & \text{for } z > 0 \\ 0 & \text{otherwise,} \end{cases}$$

and we wish to determine the probability density of the random variable $Y = \sqrt{Z}$. The monotonic function $y = u(z) = \sqrt{z}$ can be inverted to give

$$z = w(y) = y^2.$$

Thus

$$\left|\frac{dz}{dy}\right| = |2y|, \quad p_z(w(y)) = e^{-y^2},$$

and we obtain

$$p_y(y) = 2ye^{-y^2}, \quad y > 0.$$

For many practical applications, it is sufficient to characterize a distribution function by a small number of its *moments*. The *mean* or *expected value* of a continuous random variable Z is commonly written $\langle Z\rangle$ or $E(Z)$.[*] It is defined in terms of its density function $p_z(z)$ according to

$$\langle Z\rangle = \int_{-\infty}^{\infty} z \cdot p_z(z)dz. \tag{2.9}$$

The mean has the important property that, for two random variables Z_1 and Z_2 and real numbers a_0, a_1, and a_2,

$$\langle a_0 + a_1Z_1 + a_2Z_2\rangle = a_0 + a_1\langle Z_1\rangle + a_2\langle Z_2\rangle, \tag{2.10}$$

a fact which follows directly from Equation (2.9).

The *variance* of Z, written var(Z), describes how widely the realizations scatter around the mean. It is defined as

$$\text{var}(Z) = \left\langle (Z - \langle Z\rangle)^2\right\rangle, \tag{2.11}$$

that is, as the mean of the random variable $Y = (Z - \langle Z\rangle)^2$. In terms of the density function $p_y(y)$ of Y, the variance is given by

$$\text{var}(Z) = \int_{-\infty}^{\infty} y \cdot p_y(y)dy,$$

but in fact can be written ((Freund, 1992); Theorem 4.1) more conveniently as

$$\text{var}(Z) = \int_{-\infty}^{\infty} (z - \langle Z\rangle)^2 p_z(z)dz, \tag{2.12}$$

which is also referred to as the *second moment about the mean*. For discrete random variables, the integrals in Equations (2.9) and (2.12) are replaced by summations over the allowed values of Z and the probability density is replaced by the mass function.

[*] We will prefer to use the former.

As a simple example, consider a *uniformly distributed* random variable Z with density function

$$p(z) = \begin{cases} 1 & \text{if } 0 \le z \le 1 \\ 0 & \text{otherwise.} \end{cases}$$

We calculate the moments to be

$$\begin{aligned} \langle Z \rangle &= \int_0^1 z \cdot 1 \; dz = 1/2 \\ \text{var}(Z) &= \int_0^1 (z - 1/2)^2 \cdot 1 \; dz = 1/12. \end{aligned}$$

Since the populations we are dealing with in the case of actual measurements are infinite, it is clear that mean and variance can, in reality, never be known exactly. As we shall discuss Section 2.3, they must be estimated from the available data.

Two very useful identities follow from the definition of variance (Exercise 1):

$$\begin{aligned} \text{var}(Z) &= \langle Z^2 \rangle - \langle Z \rangle^2 \\ \text{var}(a_0 + a_1 Z) &= a_1^2 \; \text{var}(Z). \end{aligned} \tag{2.13}$$

2.1.3 Random vectors

The idea of a distribution function may be extended to more than one random variable. For convenience, we consider only two continuous random variables in the following discussion, but the generalization to any number of continuous or discrete random variables is straightforward.

Let $\boldsymbol{Z} = (Z_1, Z_2)^\top$ be a *random vector*, i.e., a vector the components of which are random variables. The *joint distribution function* of $\boldsymbol{Z}$ is defined by

$$P(\boldsymbol{z}) = P(z_1, z_2) = \Pr(Z_1 \le z_1 \text{ and } Z_2 \le z_2) \tag{2.14}$$

or, in terms of the *joint density function* $p(z_1, z_2)$,

$$P(z_1, z_2) = \int_{-\infty}^{z_1} \int_{-\infty}^{z_2} p(t_1, t_2) dt_1 dt_2 \tag{2.15}$$

and, conversely,

$$p(z_1, z_1) = \frac{\partial^2}{\partial z_1 \partial z_2} P(z_1, z_2). \tag{2.16}$$

The *marginal distribution function* for Z_1 is given by

$$P_1(z_1) = P(z_1, \infty) = \int_{-\infty}^{z_1} \left[\int_{-\infty}^{\infty} p(t_1, t_2) dt_2 \right] dt_1 \tag{2.17}$$

and similarly for Z_2. The *marginal density* is defined as

$$p_1(z_1) = \int_{-\infty}^{\infty} p(z_1, z_2) dz_2, \tag{2.18}$$

with a similar expression for $p_2(z_2)$. So to get a marginal density value at z_1 we integrate (sum) over all of the probabilities for z_2 at fixed z_1.

The mean of the random vector $\boldsymbol{Z}$ is the vector of mean values of Z_1 and Z_2,

$$\langle \boldsymbol{Z} \rangle = \begin{pmatrix} \langle Z_1 \rangle \\ \langle Z_2 \rangle \end{pmatrix},$$

where the vector components are calculated with Equation (2.9) using the corresponding marginal densities.

Next we formalize the concept of statistical independence.

DEFINITION 2.4 *Two random variables Z_1 and Z_2 are said to be* independent *when their joint distribution is the product of their marginal distributions:*

$$P(z_1, z_2) = P_1(z_1)P_2(z_2)$$

or, equivalently, when their joint density is the product of their marginal densities:

$$p(z_1, z_2) = p_1(z_1)p_2(z_2).$$

Thus we have, for the mean of the product of two independent random variables,

$$\begin{aligned}\langle Z_1 Z_2 \rangle &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} z_1 z_2 p(z_1, z_2) dz_1 dz_2 \\ &= \int_{-\infty}^{\infty} z_1 p_1(z_1) dz_1 \int_{-\infty}^{\infty} z_2 p_2(z_2) dz_2 = \langle Z_1 \rangle \langle Z_2 \rangle.\end{aligned}$$

In particular, if Z_1 and Z_2 have the same distribution function with mean $\langle Z \rangle$, then

$$\langle Z_1 Z_2 \rangle = \langle Z \rangle^2. \tag{2.19}$$

The following Theorem generalizes Theorem 2.1 to random vectors (Freund, 1992):

THEOREM 2.2
Let $p_z(z_1, z_2)$ be the joint probability density of the random variables Z_1 and Z_2 and the functions $y_1 = u_1(z_1, z_2)$ and $y_2 = u_2(z_1, z_2)$ be partially differentiable and represent a one-to-one transformation for all values within the range of Z_1 and Z_2 for which $p(z_1, z_2) \neq 0$. For these values of z_1 and z_2, the equations can be uniquely solved for z_1 and z_2 to give $z_1 = w_1(y_1, y_2)$ and $z_2 = w_2(y_1, y_2)$. For the corresponding values of y_1 and y_2, the joint probability density of the random variables $Y_1 = u_1(Z_1, Z_2)$ and $Y_2 = u_2(Z_1, Z_2)$ is given by

$$p_y(y_1, y_2) = p_z(w_1(y_1, y_2), w_2(y_1, y_2))|\boldsymbol{J}|,$$

where the Jacobian $\boldsymbol{J}$ is the determinant of the partial derivatives

$$\boldsymbol{J} = \begin{vmatrix} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_1}{\partial y_2} \\ \frac{\partial z_2}{\partial y_1} & \frac{\partial z_2}{\partial y_2} \end{vmatrix}.$$

The *covariance* of random variables Z_1 and Z_2 is a measure of how their realizations are dependent upon each other and is defined to be the mean of the random variable $(Z_1 - \langle Z_1 \rangle)(Z_2 - \langle Z_2 \rangle)$, i.e.,

$$\text{cov}(Z_1, Z_2) = \langle \ (Z_1 - \langle Z_1 \rangle)(Z_2 - \langle Z_2 \rangle) \ \rangle . \tag{2.20}$$

Their *correlation* is defined by

$$\rho_{12} = \frac{\text{cov}(Z_1, Z_2)}{\sqrt{\text{var}(Z_1)\text{var}(Z_2)}}. \tag{2.21}$$

The correlation is unitless and restricted to values $-1 \leq \rho_{12} \leq 1$. If $|\rho_{12}| = 1$, then Z_1 and Z_2 are *linearly dependent*. Two simple consequences of the definition of covariance are

$$\begin{aligned} \text{cov}(Z_1, Z_2) &= \langle Z_1 Z_2 \rangle - \langle Z_1 \rangle \langle Z_2 \rangle \\ \text{cov}(Z_1, Z_1) &= \text{var}(Z_1). \end{aligned} \tag{2.22}$$

A convenient way to represent the variances and covariances of the components of a random vector is in terms of the *variance–covariance matrix*. Let $\boldsymbol{a} = (a_1, a_2)^\top$ be any constant vector. Then the variance of the random variable $\boldsymbol{a}^\top \boldsymbol{Z} = a_1 Z_1 + a_2 Z_2$ is, according to the preceding definitions,

$$\begin{aligned} \text{var}(\boldsymbol{a}^\top \boldsymbol{Z}) &= \text{cov}(a_1 Z_1 + a_2 Z_2, a_1 Z_1 + a_2 Z_2) \\ &= a_1^2 \text{var}(Z_1) + a_1 a_2 \text{cov}(Z_1, Z_2) + a_1 a_2 \text{cov}(Z_2, Z_1) + a_2^2 \text{var}(Z_2) \\ &= (a_1, a_2) \begin{pmatrix} \text{var}(Z_1) & \text{cov}(Z_1, Z_2) \\ \text{cov}(Z_2, Z_1) & \text{var}(Z_2) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}. \end{aligned}$$

The matrix in the above equation is the variance–covariance matrix,* usually denoted by the symbol $\boldsymbol{\Sigma}$,

$$\boldsymbol{\Sigma} = \begin{pmatrix} \text{var}(Z_1) & \text{cov}(Z_1, Z_2) \\ \text{cov}(Z_2, Z_1) & \text{var}(Z_2) \end{pmatrix}. \tag{2.23}$$

Therefore, we have

$$\text{var}(\boldsymbol{a}^\top \boldsymbol{Z}) = \boldsymbol{a}^\top \boldsymbol{\Sigma} \boldsymbol{a}. \tag{2.24}$$

Note that, since $\text{cov}(Z_1, Z_2) = \text{cov}(Z_2, Z_1)$, $\boldsymbol{\Sigma}$ is a symmetric matrix. Moreover, since $\boldsymbol{a}$ is arbitrary and the variance of any random variable is (generally) positive, $\boldsymbol{\Sigma}$ is also positive definite, see Definition 1.1.

*For the sake of brevity, we will simply call it the *covariance matrix* from now on.

The covariance matrix can be written as an outer product:

$$\boldsymbol{\Sigma} = \langle\ (\boldsymbol{Z} - \langle\boldsymbol{Z}\rangle)(\boldsymbol{Z} - \langle\boldsymbol{Z}\rangle)^\top\ \rangle = \langle\boldsymbol{Z}\boldsymbol{Z}^\top\rangle - \langle\boldsymbol{Z}\rangle\langle\boldsymbol{Z}\rangle^\top, \tag{2.25}$$

as is easily verified. Indeed, if $\langle\boldsymbol{Z}\rangle = \boldsymbol{0}$, we can write simply

$$\boldsymbol{\Sigma} = \langle\boldsymbol{Z}\boldsymbol{Z}^\top\rangle. \tag{2.26}$$

The *correlation matrix* $\boldsymbol{R}$ is similar to the covariance matrix, except that each matrix element $(\boldsymbol{\Sigma})_{ij}$ is divided by $\sqrt{\text{var}(Z_i)\text{var}(Z_j)}$ as in Equation (2.21):

$$\boldsymbol{R} = \begin{pmatrix} 1 & \rho_{12} \\ \rho_{21} & 1 \end{pmatrix} = \begin{pmatrix} 1 & \frac{\text{cov}(Z_1,Z_2)}{\sqrt{\text{var}(Z_1)\text{var}(Z_2)}} \\ \frac{\text{cov}(Z_2,Z_1)}{\sqrt{\text{var}(Z_1)\text{var}(Z_2)}} & 1 \end{pmatrix}, \tag{2.27}$$

where $\rho_{12} = \rho_{21}$ is the correlation of Z_1 and Z_2.

2.1.4 The normal distribution

It is very often the case that random variables are well described by the *normal* or *Gaussian density function*

$$p(z) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2\sigma^2}(z-\mu)^2\right), \tag{2.28}$$

where $-\infty < \mu < \infty$ and $\sigma^2 > 0$. In that case, it follows from Equation (2.9) and Equation (2.12) that

$$\langle Z\rangle = \mu, \quad \text{var}(Z) = \sigma^2.$$

This is commonly abbreviated by writing

$$Z \sim \mathcal{N}(\mu, \sigma^2).$$

If Z is normally distributed, then the *standardized* random variable $(Z-\mu)/\sigma$ has the *standard normal distribution* $\Phi(z)$ with zero mean and unit variance

$$\Phi(z) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z}\exp(-t^2/2)dt = \int_{-\infty}^{z}\phi(t)dt, \tag{2.29}$$

where the *standard normal density* $\phi(t)$ is given by

$$\phi(t) = \frac{1}{\sqrt{2\pi}}\exp(-t^2/2). \tag{2.30}$$

Since it is not possible to express the normal distribution function $\Phi(z)$ in terms of simple analytical functions, it is tabulated (nowadays of course approximated in a software procedure or function). From the symmetry of the density function it follows that

$$\Phi(-z) = 1 - \Phi(z), \tag{2.31}$$

so it is sufficient to give tables (functions) only for $z \geq 0$. Note that

$$P(z) = \Pr(Z \leq z) = \Pr\left(\frac{Z-\mu}{\sigma} \leq \frac{z-\mu}{\sigma}\right) = \Phi\left(\frac{z-\mu}{\sigma}\right), \tag{2.32}$$

so that values for any normally distributed random variable can be read from the table.

Proofs of the following two important theorems are given in Freund (1992).

THEOREM 2.3
(Additivity) *If the random variables $Z_1, Z_2 \dots Z_m$ are independent (see Definition 2.4) and normally distributed, then the linear combination*

$$a_1 Z_1 + a_2 Z_2 + \ldots + a_m Z_m$$

is normally distributed with moments

$$\mu = a_1\mu_1 + a_2\mu_2 + \ldots + a_m\mu_m, \quad \sigma^2 = a_1^2\sigma_1^2 + a_2^2\sigma_2^2 + \ldots a_m^2\sigma_m^2.$$

THEOREM 2.4
(Central Limit Theorem) *If random variables $Z_1, Z_2 \dots Z_m$ are independent and have equal distributions with mean μ and variance σ^2, then the random variable*

$$\frac{1}{\sigma\sqrt{m}}\sum_{i=1}^m (Z_i - \mu) = \frac{\bar Z - \mu}{\sigma/\sqrt{m}}$$

with $\bar Z = (1/m)\sum_i Z_i$ is standard normally distributed in the limit $m \to \infty$.

Theorem 2.3 implies that, if Z_i, $i = 1 \dots m$, is a random sample drawn from a population which is distributed with mean μ and variance σ^2, then the *sample mean* (see Section 2.2),

$$\bar Z = \frac{1}{m}\sum_{i=1}^m Z_i \ ,$$

is normally distributed with mean μ and variance σ^2/m (Exercise 6). Theorem 2.4, on the other hand, justifies approximating the distribution of the mean $\bar Z$ with a normal distribution having mean μ and variance σ^2/m for large m, even when the Z_i are not normally distributed.

As an illustration of the Central Limit Theorem, the code:

```
import numpy as np
import matplotlib.pyplot as plt
r = np.random.rand(10000,12)
array = np.sum(r,1)
p = plt.hist(array,bins=12)
```

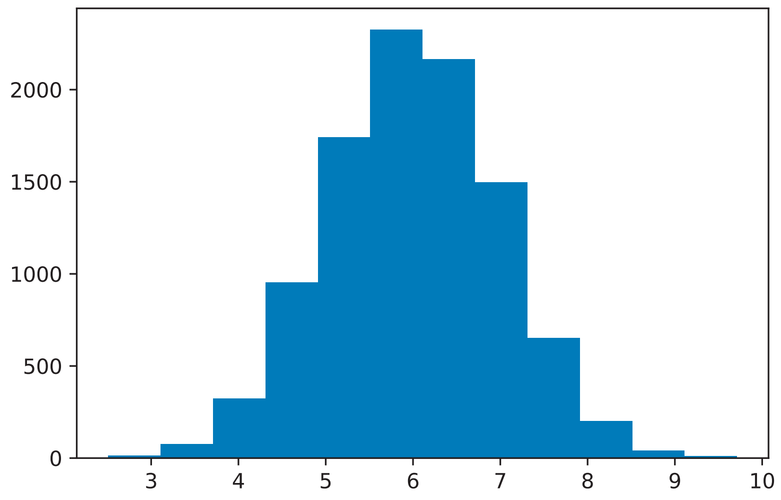


FIGURE 2.1
Histogram of sums of 12 uniformly distributed random numbers.

calculates 10,000 sums of $m = 12$ random numbers uniformly distributed on the interval $[0, 1]$ and plots their histogram; see Figure 2.1. Note the use of the NumPy function `np.sum(r,1)`, which sums the two-dimensional NumPy array `r` along its second dimension (dimensions are numbered from 0), i.e., along its column index. The histogram closely approximates a normal distribution.

2.1.5 The gamma distribution and its derivatives

A random variable Z is said to have a *gamma distribution* if its probability density function is given by

$$p_{G;\alpha,\beta}(z) = \begin{cases} \frac{1}{\beta^\alpha\Gamma(\alpha)} z^{\alpha-1}e^{-z/\beta} & \text{for } z > 0 \\ 0 & \text{elsewhere,} \end{cases} \tag{2.33}$$

where $\alpha > 0$ and $\beta > 0$ and where the *gamma function* $\Gamma(\alpha)$ is given by

$$\Gamma(\alpha) = \int_0^\infty z^{\alpha-1}e^{-z}dz, \quad \alpha > 0. \tag{2.34}$$

The gamma function has the recursive property

$$\Gamma(\alpha) = (\alpha-1)\Gamma(\alpha-1), \quad \alpha > 1,$$

and generalizes the notion of a factorial; see Exercise 7. It is easy to show (Exercise 8(a)) that the gamma distribution has mean and variance

$$\mu = \alpha\beta, \quad \sigma^2 = \alpha\beta^2. \tag{2.35}$$

The *regularized incomplete gamma function* is

$$\gamma(\alpha, z) = \frac{1}{\Gamma(\alpha)} \int_0^z t^{\alpha-1} e^{-t} dt \tag{2.36}$$

and must be approximated numerically.

A special case of the gamma distribution arises for $\alpha = 1$. Since $\Gamma(1) = \int_0^\infty e^{-z} dz = 1$, we obtain the *exponential distribution* with density function

$$p_{E;\beta}(z) = \begin{cases} \frac{1}{\beta} e^{-z/\beta} & \text{for } z > 0 \\ 0 & \text{elsewhere,} \end{cases} \tag{2.37}$$

where $\beta > 0$. According to Equation (2.35), the exponential distribution has mean β and variance β^2. In addition we have (Exercise 8(b)) the following theorem:

THEOREM 2.5

If random variables $Z_1, Z_2 \ldots Z_m$ are independent and exponentially distributed according to Equation (2.37), then the random variable $Z = \sum_{i=1}^m Z_i$ is gamma distributed with $\alpha = m$.

An immediate consequence of this Theorem is that if random variables Z_1 and Z_2 are are gamma distributed with $\alpha = m$ and $\alpha = n$ with the same values of β, then $Z_1 + Z_2$ is gamma distributed with $\alpha = m + n, \beta$. That is, $Z_1 + Z_2$ can be expressed as the sum of $m + n$ exponentially distributed random variables with parameter β; see also Theorem 2.7 below.

The *chi-square distribution with m degrees of freedom* is another special case of the gamma distribution. We get its density function with $\beta = 2$ and $\alpha = m/2$, i.e.,

$$p_{\chi^2;m}(z) = \begin{cases} \frac{1}{2^{m/2}\Gamma(m/2)} z^{(m-2)/2} e^{-z/2} & \text{for } z > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.38}$$

It follows that the chi-square distribution has mean $\mu = m$ and variance $\sigma^2 = 2m$. It is straightforward to show (Exercise 3) that the corresponding probability distribution can be written in terms of the incomplete gamma function, Equation (2.36), as

$$P_{\chi^2;m}(z) = \frac{1}{2^{m/2}\Gamma(m/2)} \int_0^z x^{(m-2)/2} e^{-x/2} dx = \gamma(m/2, z/2). \tag{2.39}$$

The reader is asked to prove a special case of the following theorem in Exercise 4.

THEOREM 2.6
If the random variables Z_i, $i = 1 \dots m$, are independent and standard normally distributed (i.e., with mean 0 *and variance* 1*), then the random variable* $Z = \sum_{i=1}^m Z_i^2$ *is chi-square distributed with* m *degrees of freedom.*

One can use the Python function `scipy.stats.chi2.cdf()` to calculate the chi-square probability distribution function or `scipy.stats.chi2.ppf()` to calculate its percentiles (values of z for given $P_{\chi^2;m}(z)$). They may be imported with the `scipy.stats` package. The Scipy statistics package also provides the function `chi2.pdf()` to calculate the chi-square probability *density*[*] function. Here we apply it in the Jupyter notebook to generate the plots shown in Figure 2.2.

```
import scipy.stats as st

z = np.linspace(1,20,200)
ax = plt.subplot(111)
for i in range(1,6):
    ax.plot(z,st.chi2.pdf(z,i),label = str(i))
ax.legend()
```

The gamma and exponential distributions will be essential for the characterization of SAR speckle noise in Chapter 5. The chi-square distribution plays a central role in the iterative change detection algorithm IR-MAD of Chapter 9.

Finally, we mention the *beta distribution* which has a probability density function defined only between zero and one:

$$p_{B:\alpha,\beta}(z) = \frac{1}{B(\alpha,\beta)} z^{\alpha-1}(1-z)^{\beta-1}, \quad 0 \le z \le 1,\ \alpha > 0,\ \beta > 0, \qquad (2.40)$$

where the *beta function* $B(\alpha,\beta)$ is given by

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}. \qquad (2.41)$$

The following result (Exercise 9) will play a role in the discussion of the sequential SAR change detection algorithm in Chapter 9:

THEOREM 2.7
If the random variables X *and* Y *are independent and gamma distributed with parameters* (m,β) *and* (n,β)*, respectively, then the random variables* $S = X + Y$ *and* $U = X/(X+Y)$ *are independent. Moreover,* S *is gamma distributed with parameters* $(m+n,\beta)$ *and* U *is* (m,n)*-beta distributed.*

[*]Note the unfortunate ambiguity of the abbreviation “pdf”!

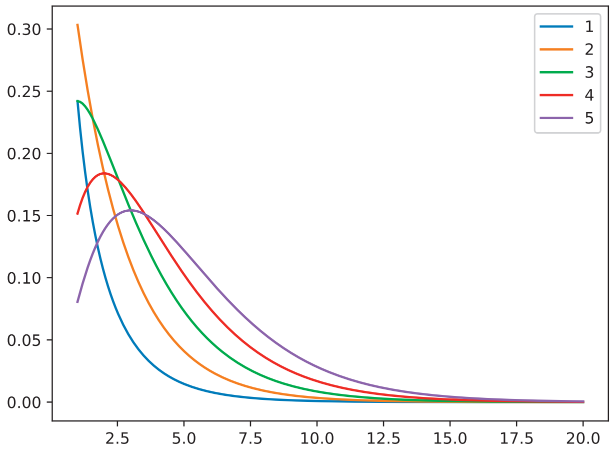


FIGURE 2.2
Plots of the chi-square probability density for $m = 1 \ldots 5$ degrees of freedom.

2.2 Parameter estimation

Having introduced distribution functions for random variables, the question arises as to how to estimate the parameters which characterize those distributions, most importantly their means, variances and covariances, from observations.

2.2.1 Random samples

Consider a multispectral image and a specific land cover category within it. We might choose n pixels belonging to that category and use them to estimate the moments of the underlying distribution. That distribution will be determined not only by measurement noise, atmospheric disturbances, etc., but also by the spread in reflectances characterizing the land cover category itself. For example, Figure 2.3 shows a region of interest (ROI) marking an area of mixed forest contained within the ASTER image of Figure 1.1. Recalling that the ASTER scene was acquired on May 1, 2007, we can extract the corresponding histogram from the GEE data archive with the script:

```
import ee
ee.Initialize()
im = ee.Image(ee.ImageCollection('ASTER/AST_L1T_003') \
             .filterBounds(ee.Geometry.Point([6.5,50.9])) \
             .filterDate('2007-04-30','2007-05-02') \
             .first()) \
             .select('B3N')
roi = ee.Geometry.Polygon(
    [[6.382713317871094,50.90736285477543],
     [6.3961029052734375,50.90130070888041],
     [6.4015960693359375,50.90519789328594],
     [6.388206481933594,50.91169247570916],
     [6.382713317871094,50.90736285477543]])
sample = im.sample(roi,scale=15) \
           .aggregate_array('B3N').getInfo()
p = plt.hist(sample,bins=20)
```

The histogram of the observations in the 3N band of the Aster image of Figure 1.1 under the mask is shown in Figure 2.4. It is, generously speaking, a normal distribution. The masked observations might thus be used to calculate an approximate mean and variance for a random variable describing mixed forest land cover.

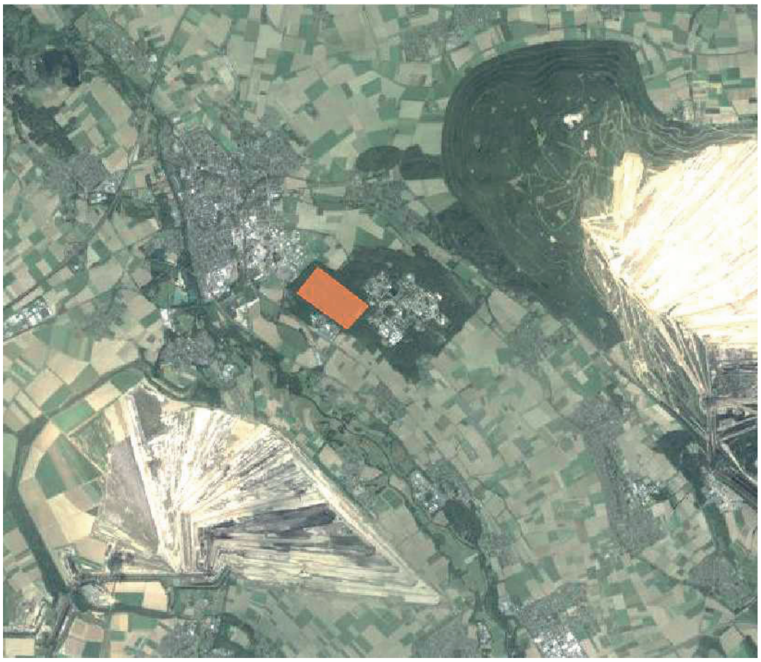


FIGURE 2.3
A region of interest covering an area of mixed forest. The polygon was set in the GEE code editor.

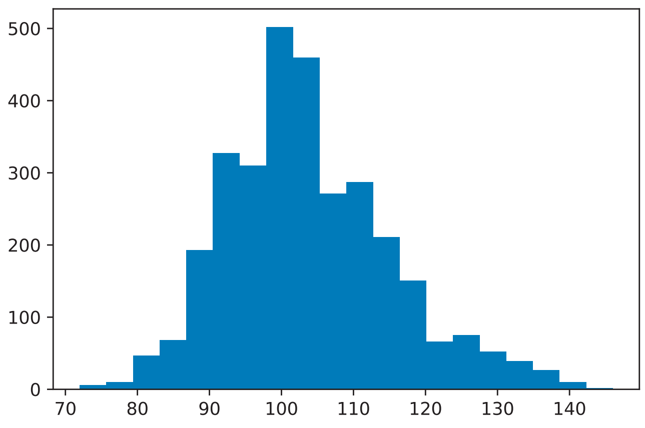


FIGURE 2.4
Histogram of the pixels in the 3N band of the ASTER image in the ROI of Figure 2.3.

More formally, let $Z_1, Z_2 \dots Z_m$ be independent random variables which all have the same distribution function $P(z)$ with mean $\langle Z\rangle$ and variance $\mathrm{var}(Z)$. These random variables are referred to as a *sample of the distribution* and are said to be *independent and identically distributed* (i.i.d.). Any function of them is called a *sample function* and is itself a random variable. The pixel intensities contributing to Figure 2.4 are a particular realization of some sample of the distribution corresponding to the land cover category mixed forest. For our present purposes, the sample functions of interest are those which can be used to estimate the mean and variance of the distribution $P(z)$. These are the *sample mean*

$$\bar Z = \frac{1}{m}\sum_{i=1}^m Z_i \tag{2.42}$$

and the *sample variance*

$$S = \frac{1}{m-1}\sum_{i=1}^m (Z_i - \bar Z)^2. \tag{2.43}$$

These two sample functions are also called *unbiased estimators* because their expected values are equal to the corresponding moments of $P(z)$, that is,

$$\langle \bar Z\rangle = \langle Z\rangle \tag{2.44}$$

and

$$\langle S\rangle = \mathrm{var}(Z). \tag{2.45}$$

The first result, Equation (2.44), follows immediately:

$$\langle \bar{Z}\rangle = \frac{1}{m}\sum_i \langle Z_i\rangle = \frac{1}{m}\, m\langle Z\rangle = \langle Z\rangle.$$

To see that Equation (2.45) holds, consider

$$\begin{aligned}(m-1)S &= \sum_i (Z_i-\bar{Z})^2 = \sum_i \left(Z_i^2 - 2Z_i\bar{Z} - \bar{Z}^2\right)\\ &= \sum_i Z_i^2 - 2m\bar{Z}^2 + m\bar{Z}^2.\end{aligned}$$

Therefore,

$$(m-1)S = \sum_i Z_i^2 - m\bar{Z}^2 = \sum_i Z_i^2 - m\frac{1}{m^2}\left(\sum_i Z_i\right)\left(\sum_i Z_i\right).$$

Expanding the product of sums yields

$$(m-1)S = \sum_i Z_i^2 - \frac{1}{m}\sum_i Z_i^2 - \frac{1}{m}\sum_{i\neq i'} Z_i Z_{i'}.$$

But Z_i and $Z_{i'}$ are independent random variables; see Definition 2.4 and Equation (2.19), so that

$$\langle Z_i Z_{i'}\rangle = \langle Z\rangle^2.$$

Therefore, since the double sum above has $m(m-1)$ terms,

$$(m-1)\langle S\rangle = m\langle Z^2\rangle - \frac{1}{m}\, m\langle Z^2\rangle - \frac{1}{m}\, m(m-1)\langle Z\rangle^2$$

or

$$\langle S\rangle = \langle Z^2\rangle - \langle Z\rangle^2 = \mathrm{var}(Z).$$

The denominator $(m-1)$ in the definition of the sample variance is thus seen to be required for unbiased estimation of the covariance matrix and to be due to the appearance of the sample mean $\bar{Z}$ rather than the distribution mean $\langle Z\rangle$ in the definition. The *maximum likelihood method*, which we will meet in Section 2.4, will lead to the same sample mean, but to a slightly different sample variance estimator.

2.2.2 Sample distributions and interval estimators

It follows from Theorem 2.3 that the sample mean $\bar{Z} = \sum_i Z_i/m$ for $Z_i \sim \mathcal{N}(\mu,\sigma^2)$ is normally distributed with mean μ and variance σ^2/m. For the sample variance, we have (Freund, 1992) the following theorem:

THEOREM 2.8
If S is the sample variance

$$S = \frac{1}{m-1}\sum_{i=1}^{m}(Z_i - \bar{Z})^2$$

of a random sample Z_i, $i = 1 \dots m$, drawn from a normally distributed population with mean μ and variance σ^2, then the random variable

$$(m-1)S/\sigma^2$$

is independent of $\bar{Z}$ and has the chi-square distribution with $m-1$ degrees of freedom.

The estimators in Equations (2.42) and (2.43) are referred to as *point estimators*, since their realizations involve real numbers. The estimated value of any distribution parameter will of course differ from the true value. Generally, one prefers to quote an interval within which, to some specified degree of confidence, the true value will lie.

To give an important example, consider again a $\mathcal{N}(\mu, \sigma^2)$-distributed random sample $Z_1 \dots Z_m$. Then

$$\bar{Z} \sim \mathcal{N}(\mu, \sigma^2/m),$$

and the variance in the sample mean decreases inversely with sample size m. Moreover, $(\bar{Z} - \mu)/(\sigma/\sqrt{m})$ is standard normally distributed. Therefore, for any $t > 0$, and with Equation (2.31),

$$\Pr\left(-t < \frac{\bar{Z} - \mu}{\sigma/\sqrt{m}} \le t\right) = \Phi(t) - \Phi(-t) = 2\Phi(t) - 1.$$

This may be written in the form

$$\Pr\left(\bar{Z} - t\,\frac{\sigma}{\sqrt{m}} \le \mu < \bar{Z} + t\,\frac{\sigma}{\sqrt{m}}\right) = 2\Phi(t) - 1.$$

Thus, we can say that the probability that the *random interval*

$$\left(\bar{Z} - t\,\frac{\sigma}{\sqrt{m}}, \bar{Z} + t\,\frac{\sigma}{\sqrt{m}}\right) \tag{2.46}$$

covers the unknown mean value μ is $2\Phi(t)-1$. Once a realization of the random interval has been obtained and reported, i.e., by plugging a realization of $\bar{Z}$ into Equation (2.46), then μ either lies within it or it doesn't. Therefore, one can no longer properly speak of probabilities. Instead, a *degree of confidence* for the reported interval is conventionally given and expressed in terms of a (usually small) quantity α defined by

$$1 - \alpha = 2\Phi(t) - 1.$$

This determines the value of t to be used in Equation (2.46) according to

$$\Phi(t) = 1 - \alpha/2. \tag{2.47}$$

For a given α, t can be read from the table of the normal distribution. One then says that the interval *covers the unknown parameter μ with confidence* $1 - \alpha$.

We can similarly derive a confidence interval for the estimated variance S. Define $\chi^2_{\alpha;m}$ by

$$\Pr(Z \geq \chi^2_{\alpha;m}) = \alpha$$

for a random variable Z having the chi-square distribution with m degrees of freedom. Then from Theorem 2.8,

$$\Pr\left(\chi^2_{1-\alpha/2;m-1} < \frac{(m-1)S}{\sigma^2} < \chi^2_{\alpha/2;m-1}\right) = 1 - \alpha$$

or

$$\Pr\left(\frac{(m-1)S}{\chi^2_{\alpha/2;m-1}} < \sigma^2 < \frac{(m-1)S}{\chi^2_{1-\alpha/2;m-1}}\right) = 1 - \alpha.$$

So we can say that, if s is the estimated variance (realization of S) of a random sample of size m from a normal population, then

$$\frac{(m-1)s}{\chi^2_{\alpha/2;m-1}} < \sigma^2 < \frac{(m-1)s}{\chi^2_{1-\alpha/2;m-1}}$$

is a $1 - \alpha$ confidence interval for σ^2.

The following Python script calculates 95% mean and variance confidence intervals for a uniformly distributed random sample with $m = 1000$:

```
from scipy.stats import norm,chi2

def x2(a,m):
    return chi2.ppf(1-a,m)

m = 1000
a = 0.05
g = np.random.random(m)
gbar = np.sum(g)/m
s = np.sum((g-gbar)**2)/(m-1)
print('sample variance: %f'%s)
lower = (m-1)*s/x2(a/2,m-1)
upper = (m-1)*s/x2(1-a/2,m-1)
print('%i percent confidence interval: (%f, %f)'\
               %(int((1-a)*100),lower,upper))
print('sample mean: %f'%gbar)
t = norm.ppf(1-a/2)
```

```
sigma = np.sqrt(s)
lower = gbar-t*sigma/np.sqrt(m)
upper = gbar+t*sigma/np.sqrt(m)
print('%i␣percent␣confidence␣interval:␣(%f,␣%f)'\
                %(int((1-a)*100),lower,upper))
```

The result is

```
sample variance: 0.083693
95 percent confidence interval: (0.076813, 0.091546)
sample mean: 0.508182
95 percent confidence interval: (0.490251, 0.526112)
```

According to the Central Limit Theorem, the mean approaches 0.5 and the variance tends to $1/12 = 0.08333$ as $m \to \infty$; see Section 2.1.2.

2.3 Multivariate distributions

As with continuous random variables, the continuous random vector $\boldsymbol{Z} = (Z_1, Z_2)^\top$ is often assumed to be described by a *bivariate normal density function* $p(\boldsymbol{z})$ given by

$$p(\boldsymbol{z}) = \frac{1}{(2\pi)^{N/2}\sqrt{|\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{z}-\boldsymbol{\mu})\right), \tag{2.48}$$

with $N = 2$. For $N > 2$, the same definition applies and we speak of a *multivariate normal density function*. The mean vector $\langle \boldsymbol{Z} \rangle$ is $\boldsymbol{\mu}$ and the covariance matrix is $\boldsymbol{\Sigma}$. This is indicated by writing

$$\boldsymbol{Z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}).$$

We have the following important result (see again Freund (1992)):

THEOREM 2.9
If two random variables Z_1, Z_2 have a bivariate normal distribution, they are independent if and only if $\rho_{12} = 0$, that is, if and only if they are uncorrelated.

In general, a zero correlation does not imply that two random variables are independent. However, this theorem says that it does if the variables are normally distributed.

A *complex Gaussian random variable* $Z = X + \mathbf{i}Y$ is a complex random variable whose real and imaginary parts are bivariate normally distributed. Under certain assumptions (Goodman, 1963), the real-valued form for the multivariate distribution, Equation (2.48), can be carried over straightforwardly to the domain of complex random vectors. In particular, it is assumed that the real and imaginary parts of each component of the complex random vector are uncorrelated and have equal variances, although the real and imaginary parts of different components can be correlated. As we shall see in Chapter 5, this corresponds closely to the properties of SAR amplitude data. The complex covariance matrix for a zero-mean complex random vector $\boldsymbol{Z}$ is, similar to Equation (2.26), given by

$$\boldsymbol{\Sigma} = \langle \boldsymbol{Z}\boldsymbol{Z}^\dagger \rangle, \tag{2.49}$$

where the $\dagger$ denotes conjugate transposition as defined in Chapter 1 and Appendix A. Note that $\boldsymbol{\Sigma}$ is Hermitian and positive semi-definite (see Exercise 9 in Chapter 1). The complex random vector $\boldsymbol{Z}$ is said to be *complex multivariate normally distributed* with zero mean and covariance matrix $\boldsymbol{\Sigma}$ if its density function is

$$p(\boldsymbol{z}) = \frac{1}{\pi^N|\boldsymbol{\Sigma}|}\exp(-\boldsymbol{z}^\dagger\boldsymbol{\Sigma}^{-1}\boldsymbol{z}). \tag{2.50}$$

This is indicated by writing $\boldsymbol{Z} \sim \mathcal{N}_C(\boldsymbol{0}, \boldsymbol{\Sigma})$.

2.3.1 Vector sample functions and the data matrix

Random vectors will, in our context, always represent pixel observation vectors in remote sensing images. The sample functions of interest are, as in the scalar case, those which can be used to estimate the vector mean and covariance matrix of a joint distribution $P(\boldsymbol{z})$, namely, the *vector sample mean*[*]

$$\bar{\boldsymbol{Z}} = \frac{1}{m}\sum_{\nu=1}^{m}\boldsymbol{Z}(\nu) \tag{2.51}$$

and the *sample covariance matrix*

$$\boldsymbol{S} = \frac{1}{m-1}\sum_{\nu=1}^{m}(\boldsymbol{Z}(\nu)-\bar{\boldsymbol{Z}})(\boldsymbol{Z}(\nu)-\bar{\boldsymbol{Z}})^\top. \tag{2.52}$$

These two sample functions are unbiased estimators because again, as in the scalar case, their expected values are equal to the corresponding parameters of $P(\boldsymbol{z})$, that is,

$$\langle\bar{\boldsymbol{Z}}\rangle = \langle\boldsymbol{Z}\rangle \tag{2.53}$$

and

$$\langle\boldsymbol{S}\rangle = \boldsymbol{\Sigma}. \tag{2.54}$$

[*]We shall prefer to use the Greek letter ν to index random vectors and their realizations from now on.

Suppose that we have made i.i.d. observations $\boldsymbol{z}(\nu)$, $\nu = 1 \ldots m$. They may be arranged into an $m \times N$ matrix $\boldsymbol{\mathcal{Z}}$ in which each N-component observation vector forms a row, i.e.,

$$\boldsymbol{\mathcal{Z}} = \begin{pmatrix} \boldsymbol{z}(1)^\top \\ \boldsymbol{z}(2)^\top \\ \vdots \\ \boldsymbol{z}(m)^\top \end{pmatrix}, \tag{2.55}$$

which, as mentioned in Section 1.3.4, is a very useful construct, and is called the *data matrix*. (Mardia et al. (1979) explain the theory of multivariate statistics entirely in terms of the data matrix!) The calligraphic font is chosen to avoid confusion with the random vector $\boldsymbol{Z}$. The unbiased estimate $\bar{\boldsymbol{z}}$ of the sample mean vector $\bar{\boldsymbol{Z}}$ is just the vector of the column means of the data matrix $\boldsymbol{\mathcal{Z}}$. It can be conveniently written as

$$\bar{\boldsymbol{z}} = \frac{1}{m}\boldsymbol{\mathcal{Z}}^\top \boldsymbol{1}_m, \tag{2.56}$$

where $\boldsymbol{1}_m$ denotes a column vector of m ones. If the column means have been subtracted out, then the data matrix is said to be *column centered*, in which case an unbiased estimate $\boldsymbol{s}$ for the covariance matrix $\boldsymbol{\Sigma}$ is given by

$$\boldsymbol{s} = \frac{1}{m-1}\boldsymbol{\mathcal{Z}}^\top \boldsymbol{\mathcal{Z}}. \tag{2.57}$$

Note that the order of transposition is reversed relative to Equation (2.26) because the observations are stored as rows. The rules of matrix multiplication, Equation (1.14), take care of the sum over all observations, so it only remains to divide by $m-1$ to obtain the desired estimate. If $\boldsymbol{d}$ is the diagonal matrix having the diagonal elements of $\boldsymbol{s}$,

$$(\boldsymbol{d})_{ij} = \begin{cases} (\boldsymbol{s})_{ij} & \text{for } i = j \\ 0 & \text{otherwise,} \end{cases}$$

then an unbiased estimate $\boldsymbol{r}$ of the correlation matrix $\boldsymbol{R}$, see Equation 2.27, is given by (Exercise 13)

$$\boldsymbol{r} = \boldsymbol{d}^{-1/2}\boldsymbol{s}\boldsymbol{d}^{-1/2}. \tag{2.58}$$

In terms of a column centered data matrix stored in the numerical Python matrix Z , for instance, the covariance and correlation matrices can be coded as

```
s = Z.T*Z/(Z.shape[0]-1)
d = np.mat(np.diag(np.sqrt(np.diag(s))))
r = d.I*s*d.I
```

Listing 2.1: A Python object class for the method of provisional means.

```python
class Cpm(object):
    '''Provisional means algorithm'''
    def __init__(self,N):
        self.mn = np.zeros(N)
        self.cov = np.zeros((N,N))
        self.sw = 0.0000001

    def update(self,Xs,Ws=None):
        lngth = len(np.shape(Xs))
        if lngth==2:
            n,N = np.shape(Xs)
        else:
            N = len(Xs)
            n = 1
        if Ws is None:
            Ws = np.ones(n)
        sw = ctypes.c_double(self.sw)
        mn = self.mn
        cov = self.cov
        provmeans(Xs,Ws,N,n,ctypes.byref(sw),mn,cov)
        self.sw = sw.value
        self.mn = mn
        self.cov = cov

    def covariance(self):
        c = np.mat(self.cov/(self.sw-1.0))
        d = np.diag(np.diag(c))
        return c + c.T - d

    def means(self):
        return self.mn
```

2.3.2 Provisional means

In Chapter 1, a Python script was given which estimated the covariance matrix of an image by sampling *all* of its pixels. As we will see in subsequent chapters, this operation is required for many useful transformations of multi-spectral images. For large datasets, however, this may become impractical since it requires that the image array be stored completely in memory. An alternative is to use an iterative algorithm reading and processing small portions of the image at a time. The procedure we describe here is referred to as the *method of provisional means*, and we give it in a form which includes the possibility of weighting each sample.

Let $\boldsymbol{g}(\nu) = (g_1(\nu), g_2(\nu) \dots g_N(\nu))^\top$ denote the νth sample from an N-band multispectral image with some distribution function $P(\boldsymbol{g})$. Set $\nu = 0$

and define the following quantities and their initial values:

$$\begin{aligned}\bar{g}_k(\nu=0)&=0, \quad k=1\dots N\\ c_{k\ell}(\nu=0)&=0, \quad k,\ell=1\dots N.\end{aligned}$$

The $\nu+1$st observation is given weight $w_{\nu+1}$ and we define the update constant

$$r_{\nu+1}=\frac{w_{\nu+1}}{\sum_{\nu'=1}^{\nu+1} w_{\nu'}}.$$

Each new observation $\boldsymbol{g}(\nu+1)$ leads to the following updates:

$$\begin{aligned}\bar{g}_k(\nu+1)&=\bar{g}_k(\nu)+(g_k(\nu+1)-\bar{g}_k(\nu))r_{\nu+1}\\ c_{k\ell}(\nu+1)&=c_{k\ell}(\nu)+w_{\nu+1}(g_k(\nu+1)-\bar{g}_k(\nu))(g_\ell(\nu+1)-\bar{g}_\ell(\nu)(1-r_{\nu+1}).\end{aligned}$$

Then, after m observations, $\bar{\boldsymbol{g}}(m)=(\bar{g}_1(m)\dots\bar{g}_N(m))^\top$ is a realization of the (weighted) sample mean, Equation (2.51), and $c_{k\ell}(m)$ is a realization of the (k,ℓ)th element of the (weighted) sample covariance matrix, Equation (2.52).

To see that this prescription gives the desired result, consider the case in which $w_\nu=1$ for all ν. The first two mean values are

$$\begin{aligned}\bar{g}_k(1)&=0+(g_k(1)-0)\cdot 1=g_k(1)\\ \bar{g}_k(2)&=g_k(1)+(g_k(2)-g_k(1))\cdot\frac{1}{2}=\frac{g_k(1)+g_k(2)}{2}\end{aligned}$$

as expected. The first two cross products are

$$\begin{aligned}c_{k\ell}(1)&=0+(g_k(1)-0)(g_\ell(1)-0)(1-1)=0\\ c_{k\ell}(2)&=0+(g_k(2)-g_k(1))(g_\ell(2)-g_\ell(1))(1-1/2)\\ &=\frac{1}{2}(g_k(2)-g_k(1))(g_\ell(2)-g_\ell(1))\\ &=\frac{1}{2-1}\sum_{\nu=1}^{2}\left(g_k(\nu)-\frac{g_k(1)+g_k(2)}{2}\right)\left(g_\ell(\nu)-\frac{g_\ell(1)+g_\ell(2)}{2}\right)\end{aligned}$$

again as expected.

Listing 2.1 shows the Python class `Cpm` , which is part of the `auxil` package. The `update()` method takes as input a single row of a multispectral image in the form of a data matrix `Xs`, together with weights `Ws` if desired. The `provmeans` function called in line 20 is coded in C and accessed with Python's `ctypes` package. It loops through the pixels in the `Xs` array, updating the mean `self.mn` and upper diagonal part of the covariance array `self.cov` as it goes. Thus, we can calculate the covariance matrix of the VNIR bands of the ASTER image as follows:

```
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly
import auxil.auxil1 as auxil

gdal.AllRegister()
infile = 'imagery/AST_20070501.tif'
inDataset = gdal.Open(infile,GA_ReadOnly)
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize
Xs = np.zeros((cols,3))
cpm = auxil.Cpm(3)
rasterBands=[inDataset.GetRasterBand(k+1)
             for k in range(3)]
for row in range(rows):
    for k in range(3):
        Xs[:,k]=rasterBands[k].ReadAsArray(0,row,cols,1)
    cpm.update(Xs)
print cpm.covariance()
```

The output is the same as in Section 1.3.4 apart from small rounding errors:

```
[[ 407.13229638  442.18038333   -78.32374081]
 [ 442.18038333  493.57036037  -120.6419761 ]
 [ -78.32374081 -120.6419761    438.95704379]]
```

2.3.3 Real and complex multivariate sample distributions

For the purposes of interval estimation and hypothesis testing, and also for image classification and change detection, we require not only sample vector estimators, but also their distributions. We concluded, from Theorem 2.3 in Section 2.1.4, that the mean of m normally distributed samples $Z_i \sim \mathcal{N}(\mu, \sigma^2)$,

$$\bar{Z} = \frac{1}{m}\sum_{i=1}^{m} Z_i,$$

is normally distributed with mean μ and variance σ^2/m. Similarly, according to Theorem 2.8, $(m-1)S/\sigma^2$, where S is the sample variance, is chi-square distributed with $m-1$ degrees of freedom.

In the multivariate case, we have a similar situation. The sample mean vector

$$\bar{\boldsymbol{Z}} = \frac{1}{m}\sum_{\nu=1}^{m} \boldsymbol{Z}(\nu)$$

is multivariate normally distributed with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}/m$ for samples $\boldsymbol{Z}(\nu) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, $\nu = 1 \ldots m$. The sample covariance matrix $\boldsymbol{S}$ is described by a *Wishart distribution* in the following sense. Suppose $\boldsymbol{Z}(\nu) \sim$

$\mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$, $\nu = 1 \dots m$. We then have, with Equation (2.52),

$$(m-1)\boldsymbol{S} = \sum_{\nu=1}^{m} \boldsymbol{Z}(\nu)\boldsymbol{Z}(\nu)^\top =: \boldsymbol{X}. \tag{2.59}$$

Realizations $\boldsymbol{x}$ of the random sample matrix $\boldsymbol{X}$, namely

$$\boldsymbol{x} = \sum_{\nu=1}^{m} \boldsymbol{z}(\nu)\boldsymbol{z}(\nu)^\top,$$

are symmetric and, for sufficiently large m, positive definite.

THEOREM 2.10
(Anderson, 2003) *The probability density function of* $\boldsymbol{X}$ *given by Equation (2.59) is the* Wishart density with m degrees of freedom

$$p_W(\boldsymbol{x}) = \frac{|\boldsymbol{x}|^{(m-N-1)/2} \exp(-\mathrm{tr}(\mathbf{\Sigma}^{-1}\boldsymbol{x})/2)}{2^{Nm/2}\pi^{N(N-1)/4}|\mathbf{\Sigma}|^{m/2} \prod_{i=1}^{N} \Gamma[(m+1-i)/2]} \tag{2.60}$$

for $\boldsymbol{x}$ *positive definite, and* 0 *otherwise.*

We write $\boldsymbol{X} \sim \mathcal{W}(\mathbf{\Sigma}, N, m)$. Since the gamma function $\Gamma(\alpha)$, Equation (2.34), is not defined for $\alpha \leq 0$, the Wishart distribution is undefined for $m < N$. Equation (2.60) generalizes the chi-square density, Equation (2.38), as may easily be seen by setting $N \to 1$, $\boldsymbol{x} \to x$ and $\mathbf{\Sigma} \to 1$.

Now define, in analogy to Equation (2.59), the random complex sample matrix

$$\boldsymbol{X} = \sum_{\nu=1}^{m} \boldsymbol{Z}(\nu)\boldsymbol{Z}(\nu)^\dagger, \tag{2.61}$$

where $\boldsymbol{Z}(\nu) \sim \mathcal{N}_C(\mathbf{0}, \mathbf{\Sigma})$. Its realizations are Hermitian and, again for sufficiently large m, positive definite.

THEOREM 2.11
(Goodman, 1963) *The probability density function of* $\boldsymbol{X}$ *given by Equation (2.61) is the* complex Wishart density with m degrees of freedom

$$p_{W_c}(\boldsymbol{x}) = \frac{|\boldsymbol{x}|^{(m-N)} \exp(-\mathrm{tr}(\mathbf{\Sigma}^{-1}\boldsymbol{x}))}{\pi^{N(N-1)/2}|\mathbf{\Sigma}|^{m} \prod_{i=1}^{N} \Gamma(m+1-i)}. \tag{2.62}$$

This is denoted $\boldsymbol{X} \sim \mathcal{W}_C(\mathbf{\Sigma}, N, m)$. The complex Wishart density function plays an important role in the discussions of polarimetric SAR imagery in Chapters 5, 7, and 9. In particular, we will need the following theorem:

THEOREM 2.12
If $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ are both complex Wishart distributed with covariance matrix $\boldsymbol{\Sigma}$ and m degrees of freedom, then the random matrix $\boldsymbol{X}_1 + \boldsymbol{X}_2$ is complex Wishart distributed with $2m$ degrees of freedom.

Proof. The theorem is an immediate consequence of Theorem 2.11. Let $\boldsymbol{Z}_1(\nu), \boldsymbol{Z}_2(\nu) \sim \mathcal{N}_C(\boldsymbol{0}, \boldsymbol{\Sigma})$, $\nu = 1 \dots m$, and

$$\boldsymbol{X}_1 = \sum_{\nu=1}^{m} \boldsymbol{Z}_1(\nu)\boldsymbol{Z}_1(\nu)^\dagger, \quad \boldsymbol{X}_2 = \sum_{\nu=1}^{m} \boldsymbol{Z}_2(\nu)\boldsymbol{Z}_2(\nu)^\dagger.$$

But since $\boldsymbol{Z}_1$ and $\boldsymbol{Z}_2$ have the same distribution, the sum $\boldsymbol{X}_1 + \boldsymbol{X}_2$ has the same form as Equation (2.61) with m replaced by $2m$. □

2.4 Bayes' Theorem, likelihood, and classification

If A and B are two events, i.e., two subsets of a sample space Ω, such that the probability of A and B occurring simultaneously is $\Pr(A, B)$, and if $\Pr(B) \neq 0$, then the *conditional probability of A occurring given that B occurs* is defined to be

$$\Pr(A \mid B) = \frac{\Pr(A, B)}{\Pr(B)}. \tag{2.63}$$

We have Theorem 2.13 (Freund, 1992):

THEOREM 2.13
(Theorem of Total Probability) *If $A_1, A_2 \dots A_m$ are disjoint events associated with some random experiment and if their union is the set of all possible events, then for any event B*

$$\Pr(B) = \sum_{i=1}^{m} \Pr(B \mid A_i)\Pr(A_i) = \sum_{i=1}^{m} \Pr(B, A_i). \tag{2.64}$$

It should be noted that both Equations (2.63) and (2.64) have their counterparts for probability density functions:

$$p(x, y) = p(x \mid y)p(y) \tag{2.65}$$

and

$$p(x) = \int_{-\infty}^{\infty} p(x \mid y)p(y)dy = \int_{-\infty}^{\infty} p(x, y)dy. \tag{2.66}$$

Bayes' Theorem* follows directly from the definition of conditional probability and is the basic starting point for inference problems using probability theory as logic.

THEOREM 2.14
(Bayes' Theorem) *If $A_1, A_2 \dots A_m$ are disjoint events associated with some random experiment, their union is the set of all possible events, and if $Pr(A_i) \neq 0$ for $i = 1 \dots m$, then for any event B for which $Pr(B) \neq 0$*

$$\Pr(A_k \mid B) = \frac{\Pr(B \mid A_k)\Pr(A_k)}{\sum_{i=1}^{m} \Pr(B \mid A_i)\Pr(A_i)}. \tag{2.67}$$

Proof: From the definition of conditional probability, Equation (2.63),

$$\Pr(A_k \mid B)\Pr(B) = \Pr(A_k, B) = \Pr(B \mid A_k)\Pr(A_k),$$

and therefore

$$\Pr(A_k \mid B) = \frac{\Pr(B \mid A_k)\Pr(A_k)}{\Pr(B)} = \frac{\Pr(B \mid A_k)\Pr(A_k)}{\sum_{i=1}^{m} \Pr(B \mid A_i)\Pr(A_i)}$$

from the Theorem of Total Probability. □

We will use Bayes' Theorem primarily in the following form: Let $\boldsymbol{g}$ be a realization of a random vector associated with a distribution of multi-spectral image pixel intensities (gray values), and let $\{k \mid k = 1 \dots K\}$ be a set of possible *class labels* (e.g., land cover categories) for all of the pixels. Then, the *a posteriori* conditional probability for class k, *given the observation* $\boldsymbol{g}$, may be written using Bayes' Theorem as

$$\Pr(k \mid \boldsymbol{g}) = \frac{\Pr(\boldsymbol{g} \mid k)\Pr(k)}{\Pr(\boldsymbol{g})}, \tag{2.68}$$

where $\Pr(k)$ is the *a priori* probability for class k, $\Pr(\boldsymbol{g} \mid k)$ is the conditional probability of observing the value $\boldsymbol{g}$ if it belongs to class k, and

$$\Pr(\boldsymbol{g}) = \sum_{k=1}^{K} \Pr(\boldsymbol{g} \mid k)\Pr(k) \tag{2.69}$$

is the total probability for $\boldsymbol{g}$.

One can formulate the problem of *classification* of multi-spectral imagery as the process of determining posterior conditional probabilities for all of the

*Named after Rev. Thomas Bayes, an 18th-century hobby mathematician who derived a special case.

classes. This is accomplished by specifying prior probabilities $\Pr(k)$ (if prior information exists), modeling the class-specific probabilities $\Pr(\boldsymbol{g} \mid k)$ and then applying Bayes' Theorem. Thus we might choose a class-specific density function as our model:

$$\Pr(\boldsymbol{g} \mid k) \sim p(\boldsymbol{g} \mid \theta_k),$$

where θ_k is a set of parameters for the density function describing the kth class (for a multivariate normal distribution, for instance, just the mean vector $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$). In this case Equation (2.68) takes the form

$$\Pr(k \mid \boldsymbol{g}) = \frac{p(\boldsymbol{g} \mid \theta_k)\Pr(k)}{p(\boldsymbol{g})}, \tag{2.70}$$

where $p(\boldsymbol{g})$ is the unconditional density function for $\boldsymbol{g}$,

$$p(\boldsymbol{g}) = \sum_{k=1}^{K} p(\boldsymbol{g} \mid \theta_k)\Pr(k), \tag{2.71}$$

which is independent of k. As we will see in Chapter 6, under reasonable assumptions the observation $\boldsymbol{g}$ should be assigned to the class k which maximizes $\Pr(k \mid \boldsymbol{g})$.

In order to find that maximum, we require estimates of the parameters θ_k, $k = 1 \dots K$. If we have access to measured values (realizations) that are known to be in class k, $\boldsymbol{g}(\nu)$, $\nu = 1 \dots m_k$, say, then we can form the product of probability densities

$$L(\theta_k) = \prod_{\nu=1}^{m_k} p(\boldsymbol{g}(\nu) \mid \theta_k), \tag{2.72}$$

which is called a *likelihood function*, and take its logarithm

$$\mathcal{L}(\theta_k) = \sum_{\nu=1}^{m_k} \log p(\boldsymbol{g}(\nu) \mid \theta_k), \tag{2.73}$$

which is the *log-likelihood*. Taking products is justified when the $\boldsymbol{g}(\nu)$ are realizations of independent random vectors. The parameter set $\hat{\theta}_k$ which maximizes the likelihood function or its logarithm, i.e., which gives the largest value for all of the realizations, is called the *maximum-likelihood estimate* of θ_k. For normally distributed random vectors, maximum-likelihood parameter estimators for $\theta_k = \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}$ turn out to correspond (almost) to the unbiased estimators, Equations (2.51) and (2.52).

To illustrate this for the class mean, write out Equation (2.73) for the multivariate normal distribution, Equation (2.48):

$$\mathcal{L}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = m_k \frac{N}{2} \log 2\pi - m_k \frac{1}{2} \log |\boldsymbol{\Sigma}_k| - \frac{1}{2} \sum_{\nu=1}^{m_k} (\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} (\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k).$$

To maximize with respect to $\boldsymbol{\mu}_k$, we set

$$\frac{\partial \mathcal{L}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\partial \boldsymbol{\mu}_k} = \sum_{\nu=1}^{m_k} \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k) = \boldsymbol{0},$$

giving

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{m_k}\sum_{i=1}^{m_k} \boldsymbol{g}(\nu), \tag{2.74}$$

which is the realization $\bar{\boldsymbol{g}}_k$ of the unbiased estimator for the sample mean, Equation (2.51). In a similar way (see, e.g., Duda et al. (2001)), one can show that

$$\hat{\boldsymbol{\Sigma}}_k = \frac{1}{m_k}\sum_{\nu=1}^{m_k} (\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)^\top, \tag{2.75}$$

which is (almost) the realization $\boldsymbol{s}$ of the unbiased sample covariance matrix estimator, Equation (2.52), except that the denominator $m_k - 1$ is replaced by m_k, a fact which can be ignored for large m_k.

The observations, of course, must be chosen from the appropriate class k in each case. For *supervised classification* (Chapters 6 and 7), there exists a set of training data with known class labels. Therefore, maximum-likelihood estimates can be obtained and posterior probability distributions for the classes can be calculated from Equation (2.70) and then used to generalize to all of the image data. In the case of *unsupervised classification*, the class memberships are not initially known. How this conundrum is solved will be discussed in Chapter 8.

2.5 Hypothesis testing

A *statistical hypothesis* is a conjecture about the distributions of one or more random variables. It might, for instance, be an assertion about the mean of a distribution, or about the equivalence of the variances of two different distributions. One distinguishes between *simple* hypotheses, for which the distributions are completely specified, for example: *the mean of a normal distribution with variance* σ^2 *is* $\mu = 0$, and *composite* hypotheses, for which this is not the case, e.g., *the mean is* $\mu \geq 0$.

In order to test such assertions on the basis of samples of the distributions involved, it is also necessary to formulate *alternative* hypotheses. To distinguish these from the original assertions, the latter are traditionally called *null* hypotheses. Thus, we might be interested in testing the simple null hypothesis $\mu = 0$ against the composite alternative hypothesis $\mu \neq 0$. An appropriate sample function for deciding whether or not to reject the null hypothesis in favor of its alternative is referred to as a *test statistic*, often denoted by the symbol Q. An appropriate *test procedure* will partition the possible realizations of the test statistic into two subsets: an acceptance region for the null hypothesis and a rejection region. The latter is customarily referred to as the *critical region.*

DEFINITION 2.5 *Referring to the null hypothesis as H_0, there are two kinds of errors which can arise from any test procedure:*

1. *H_0 may be rejected when in fact it is true. This is called an* error of the first kind *and the probability that it will occur is denoted α.*

2. *H_0 may be accepted when in fact it is false, which is called an* error of the second kind *with probability of occurrence β.*

The probability of obtaining a value of the test statistic within the critical region when H_0 is true is thus α. The probability α is also referred to as the *level of significance* of the test, in some contexts as the *false alarm probability.* It is generally the case that the lower the value of α, the higher is the probability β of making a second kind error. Traditionally, significance levels of 0.01 or 0.05 are used. Such values are obviously arbitrary, and for exploratory data analysis it is common to avoid specifying them altogether. Instead, the P-value (which we will represent with the calligraphic $\mathcal{P}$ to avoid confusion with the probability distribution function P) for the test is stated:

DEFINITION 2.6 *Given the observed value q of a test statistic Q, the P-value $\mathcal{P}$ is the lowest level of significance at which the null hypothesis could have been rejected.*

Formulated differently, the P-value is the probability of getting a test statistic that is at least as extreme as the one observed, given the null hypothesis, for example,

$$\mathcal{P} = 1 - \Pr(Q < q \mid H_0).$$

So if this probability is smaller than the prescribed significance level α, then the null hypothesis is rejected. Again, if the null hypothesis is true, and if the test statistic were determined many times, the fraction of P-values smaller than the significance level α would be just α. This implies that the P-values are distributed uniformly over the interval $[0, 1]$ under the null hypothesis. The reader is asked to give a more formal demonstration of this in Exercise 15. In any case, high P-values provide evidence in favor of accepting the null hypothesis, without actually forcing one to commit to a decision.

The theory of statistical testing specifies methods for determining the most appropriate test statistic for a given null hypothesis and its alternative. Fundamental to the theory is the *Neyman–Pearson Lemma*, which gives for simple hypotheses a prescription for finding the test procedure which maximizes the probability $1-\beta$ of rejecting the null hypothesis when it is false for a fixed level of significance α, see, e.g., Freund (1992). The following definition deals with the more general case of tests involving composite hypotheses H_0 and H_1.

DEFINITION 2.7 *Consider random samples $\boldsymbol{z}(\nu)$, $\nu = 1 \ldots m$, from a multivariate population whose density function is $p(\boldsymbol{z} \mid \theta)$. The likelihood function for the sample is (see Equation (2.72))*

$$L(\theta) = \prod_{\nu=1}^{m} p(\boldsymbol{z}(\nu) \mid \theta). \tag{2.76}$$

Let ω be space of all possible values of the parameter set θ and ω_0 be a subset of that space. The likelihood ratio test (LRT) *for the null hypothesis $\theta \in \omega_0$ against the alternative $\theta \in \omega - \omega_0$ has the critical region*

$$Q = \frac{\max_{\theta \in \omega_0} L(\theta)}{\max_{\theta \in \omega} L(\theta)} \le k. \tag{2.77}$$

This definition simply reflects the fact that, if H_0 is true, the maximum likelihood for θ when restricted to ω_0 should be close to the maximum likelihood for θ without that restriction. Therefore, if the likelihood ratio is small, (less than or equal to some small value k), then H_0 should be rejected.

To illustrate, consider scalar random samples $z(1), z(2) \ldots z(m)$ from a normal distribution with mean μ and known variance σ^2. The likelihood ratio test at significance level α for the simple hypothesis $H_0 : \ \mu = \mu_0$ against the alternative composite hypothesis $H_1 : \ \mu \ne \mu_0$ has, according to Definition 2.7, the critical region

$$Q = \frac{L(\mu_0)}{L(\hat{\mu})} \le k_\alpha,$$

where $\hat{\mu}$ maximizes the likelihood function and k_α depends on α. Therefore, we have

$$\hat{\mu} = \bar{z} = \frac{1}{m} \sum_{\nu=1}^{m} z(\nu).$$

The critical region is then (Exercise 17)

$$Q = \frac{L(\mu_0)}{L(\hat{\mu})} = \exp\left(-\frac{1}{2\sigma^2/m} (\bar{z} - \mu_0)^2 \right) \le k_\alpha. \tag{2.78}$$

Equivalently,

$$e^{-x^2} \le k_\alpha, \quad \text{where} \quad x = \frac{\bar{z} - \mu_0}{\sqrt{2}\sigma/\sqrt{m}}.$$

Since the above exponential function is maximum at $x = 0$ and vanishes asymptotically for large values of $|x|$, the critical region can also be written in

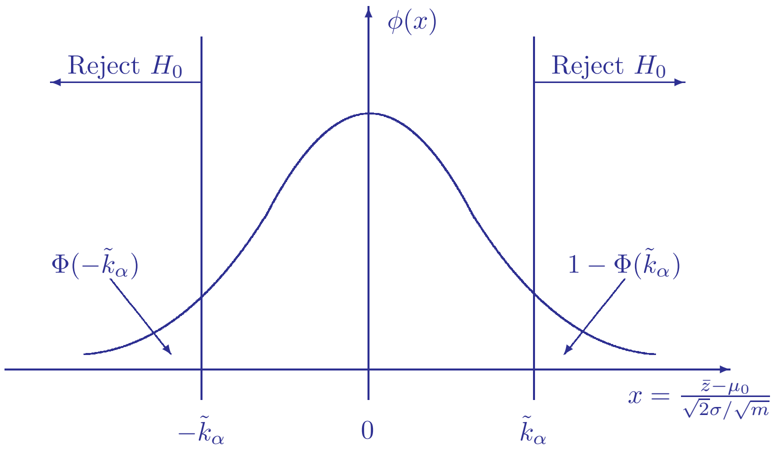

$\phi(x)$
Reject H_0
Reject H_0
$\Phi(-\tilde{k}_\alpha)$
$1-\Phi(\tilde{k}_\alpha)$
$x = \frac{\bar{z}-\mu_0}{\sqrt{2}\sigma/\sqrt{m}}$
$-\tilde{k}_\alpha$
0
$\tilde{k}_\alpha$

FIGURE 2.5
Critical region for rejecting the hypothesis $\mu = \mu_0$ at significance level α.

the form $|x| \geq \tilde{k}_\alpha$, where $\tilde{k}_\alpha$ also depends on α. But we know that $\bar{z}$ is normally distributed with mean μ_0 and variance σ^2/m. Therefore, x has the standard normal distribution $\Phi(x)$ with probability density $\phi(x)$; see Equations (2.29) and (2.30). Thus, $\tilde{k}_\alpha$ is determined by (see Figure 2.5)

$$\Phi(-\tilde{k}_\alpha) + 1 - \Phi(\tilde{k}_\alpha) = \alpha,$$

or

$$1 - \Phi(\tilde{k}_\alpha) = \alpha/2.$$

This example is straightforward because we assume that the variance σ^2 is known. Suppose, as is often the case, that the variance is unknown and that we wish nevertheless to make a statement about μ in terms of some realization of an appropriate test statistic. To treat this and similar problems, it is necessary to define some additional distribution functions.

If the random variables Z_i, $i = 0 \dots m$, are independent and standard normally distributed, then the random variable

$$T = \frac{Z_0}{\sqrt{\frac{1}{m}(Z_1^2 + \ldots + Z_m^2)}}$$

is said to be *Student-t distributed with m degrees of freedom*. The corresponding density is given by

$$p_{t;m}(z) = d_m \left(1 + \frac{z^2}{m}\right)^{-(m+1)/2}, \quad -\infty < z < \infty, \tag{2.79}$$

where d_m is a normalization factor. The Student-t distribution converges to the standard normal distribution for $m \to \infty$. The Python/SciPy functions `scipy.stats.t.cdf()` and `scipy.stats.t.ppf()` may be used to calculate the distribution and its percentiles.

The Student-t distribution is used to make statements regarding the mean when the variance is unknown. Thus if Z_i, $\nu = 1 \dots m$, is a sample from a normal distribution with mean μ and unknown variance, and

$$\bar{Z} = \frac{1}{m}\sum_{i=1}^{m} Z_i, \quad S = \frac{1}{m}\sum_{i=1}^{m}(Z_i - \bar{Z})^2,$$

then the random variable

$$T = \frac{\bar{Z} - \mu}{\sqrt{S/m}} \tag{2.80}$$

is Student-t distributed with $m-1$ degrees of freedom (Freund, 1992). $|T| \le k$ is the critical region for a likelihood ratio test for $H_0 : \mu = \mu_0$ against $H_1 : \mu \neq \mu_0$.

If X_i and Y_i, $i = 1 \dots m$, are samples from normal distributions with equal variance, then the random variable

$$T_d = \frac{\bar{X} - \bar{Y} - (\mu_X - \mu_Y)}{\sqrt{(S_X + S_Y)/m}} \tag{2.81}$$

is Student-t distributed with $2m - 2$ degrees of freedom. It may be used to test the hypothesis $\mu_X = \mu_Y$, for example.

Given independent and standard normally distributed random variables Y_i, $i = 1 \dots n$ and X_i, $i = 1 \dots m$, the random variable

$$F = \frac{\frac{1}{m}(X_1^2 + \dots + X_m^2)}{\frac{1}{n}(Y_1^2 + \dots + Y_n^2)} \tag{2.82}$$

is *F-distributed with m and n degrees of freedom*. Its density function is

$$p_{f;m,n}(z) = \begin{cases} c_{mn} z^{(m-2)/2}\left(1 + \frac{m}{n}z\right)^{-(m+n)/2} & \text{for } z > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{2.83}$$

with normalization factor c_{mn}. Note that F is the ratio of two chi-square distributed random variables. One can compute the F-distribution function and its percentiles in Python with `scipy.stats.f.cdf()` and `scipy.stats.f.ppf()`, respectively.

The F-distribution can be used for hypothesis tests regarding two variances. If S_X and S_Y are sample variances for samples of size n and m, respectively, drawn from normally distributed populations with variances σ_X^2 and σ_Y^2, then

$$F = \frac{\sigma_Y^2 S_X}{\sigma_X^2 S_Y} \tag{2.84}$$

is a random variable having an F-distribution with $n-1$ and $m-1$ degrees of freedom. $F \le k$ is the critical region for a likelihood ratio test for

$$H_0 : \sigma_X^2 = \sigma_Y^2, \quad H_1 : \sigma_X^2 < \sigma_Y^2.$$

In many cases, the likelihood ratio test will lead to a test statistic whose distribution is unknown. The LRT has, however, an important asymptotic property; see, for example, Mardia et al. (1979):

THEOREM 2.15
In the notation of Definition 2.7, if ω is a region of $\mathbb{R}^q$ and ω_0 is an r-dimensional subregion, then for each $\theta \in \omega_0$, $-2\log Q$ has an asymptotic chi-square distribution with $q-r$ degrees of freedom as $m \to \infty$.

Thus if we take minus twice the logarithm of the LRT statistic in Equation (2.78), we obtain

$$-2\log Q = -2 \cdot \left(-\frac{1}{2\sigma^2/m}(\bar z - \mu_0)^2 \right) = \frac{(\bar z - \mu_0)^2}{\sigma^2/m},$$

which is chi-square distributed with one degree of freedom (see Section 2.1.5). Since ω_0 consists of the single point μ_0, its dimension is $r = 0$, whereas ω is all real values of μ, so $q = 1$. Hence, $q - r = 1$. In this simple case, Theorem 2.15 holds for all values of m.

2.6 Ordinary linear regression

Many image analysis tasks involve fitting a set of data points with a straight line[*]

$$y(x) = a + bx. \tag{2.85}$$

We review here the standard procedure for determining the parameters a and b and their uncertainties, namely *ordinary linear regression*, in which it is assumed that the dependent variable y has a random error but that the independent variable x is exact. The procedure is then generalized to more than one independent variable and the concepts of regularization and duality are introduced.

Linear regression can also be carried out sequentially by updating the best fit after each new observation. This topic, as well as *orthogonal linear regression*, where is assumed that both variables have random errors associated with them, are discussed in Appendix A.

[*]Relative radiometric normalization, which we will meet in Chapter 9, as well as similarity warping, Chapter 5, are good examples.

2.6.1 One independent variable

Suppose that the dataset consists of m pairs $\{x(\nu), y(\nu) \mid \nu = 1 \dots m\}$. An appropriate statistical model for linear regression is

$$Y(\nu) = a + bx(\nu) + R(\nu), \quad \nu = 1 \dots m. \tag{2.86}$$

$Y(\nu)$ is a random variable representing the νth measurement of the dependent variable and $R(\nu)$, referred to as the *residual error*, is a random variable representing the measurement uncertainty. The $x(\nu)$ are exact. We will assume that the individual measurements are uncorrelated and that they all have the same variance:

$$\mathrm{cov}(R(\nu), R(\nu')) = \begin{cases} \sigma^2 & \text{for } \nu = \nu' \\ 0 & \text{otherwise.} \end{cases} \tag{2.87}$$

The realizations of $R(\nu)$ are $y(\nu) - a - bx(\nu)$, $\nu = 1 \dots m$, from which we define a least squares *goodness-of-fit* function

$$z(a,b) = \sum_{\nu=1}^{m} \left(\frac{y(\nu) - a - bx(\nu)}{\sigma} \right)^2. \tag{2.88}$$

If the residuals $R(\nu)$ are normally distributed, then we recognize Equation (2.88) as a realization of a chi-square distributed random variable. For the "best" values of a and b, Equation (2.88) is in fact chi-square distributed with $m - 2$ degrees of freedom (Press et al., 2002). The best values for the parameters a and b are obtained by minimizing $z(a,b)$, that is, by solving the equations

$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial b} = 0$$

for a and b. The solution is (Exercise 18)

$$\hat{b} = \frac{s_{xy}}{s_{xx}}, \quad \hat{a} = \bar{y} - \hat{b}\bar{x}, \tag{2.89}$$

where

$$\begin{aligned} s_{xy} &= \frac{1}{m} \sum_{\nu=1}^{m} (x(\nu) - \bar{x})(y(\nu) - \bar{y}) \\ s_{xx} &= \frac{1}{m} \sum_{\nu=1}^{m} (x(\nu) - \bar{x})^2 \\ \bar{x} &= \frac{1}{m} \sum_{\nu=1}^{m} x(\nu), \quad \bar{y} = \frac{1}{m} \sum_{\nu=1}^{m} y(\nu). \end{aligned} \tag{2.90}$$

The uncertainties in the estimates $\hat{a}$ and $\hat{b}$ are given by (Exercise 19)

$$\begin{aligned} \sigma_a^2 &= \frac{\sigma^2 \sum x(\nu)^2}{m \sum x(\nu)^2 - (\sum x(\nu))^2} \\ \sigma_b^2 &= \frac{m\sigma^2}{m \sum x(\nu)^2 - (\sum x(\nu))^2}. \end{aligned} \tag{2.91}$$

The goodness of fit can be determined by substituting the estimates $\hat{a}$ and $\hat{b}$ into Equation (2.88) which, as we have said, will then be chi-square distributed with $m-2$ degrees of freedom. The probability of finding a value $z = z(\hat{a}, \hat{b})$ *or higher* by chance (the P-value) is

$$\mathcal{P} = 1 - P_{\chi^2;m-2}(z),$$

where $P_{\chi^2;m-2}(z)$ is given by Equation (2.38). If $P < 0.01$, one would typically reject the fit as being unsatisfactory.

If σ^2 is not known *a priori*, then it can be estimated by

$$\hat{\sigma}^2 = \frac{1}{m-2}\sum_{\nu=1}^{m}(y(\nu) - \hat{a} - \hat{b}x(\nu))^2, \tag{2.92}$$

in which case the goodness-of-fit procedure cannot be applied, since we *assume* the fit to be good in order to estimate σ^2 with Equation (2.92) (Press et al., 2002).

2.6.2 Coefficient of determination (R^2)

The fitted or predicted values are

$$\hat{y}(\nu) = \hat{a} + \hat{b}x(\nu) \quad \nu = 1\dots m.$$

Consider the total variation of the observed variables $y(\nu)$, $\nu = 1\dots m$, about their mean value $\bar{y}$,

$$\begin{aligned}\sum_\nu (y(\nu) - \bar{y})^2 &= \sum_\nu (y(\nu) - \hat{y}(\nu) + \hat{y}(\nu) - \bar{y})^2 \\ &= \sum_\nu (y(\nu) - \hat{y}(\nu))^2 + (\hat{y}(\nu) - \bar{y})^2 + 2(y(\nu) - \hat{y}(\nu))(\hat{y}(\nu) - \bar{y}).\end{aligned}$$

The last term in the summation is, with $r(\nu) = y(\nu) - \hat{y}(\nu)$,

$$\sum_\nu 2r(\nu)(\hat{y}(\nu) - \bar{y}) = 2\sum_\nu r(\nu)\hat{y}(\nu) - 2\bar{y}\sum_\nu r(\nu) = 0,$$

since the errors are uncorrelated with the predicted values $\hat{y}(\nu)$ and have mean zero. Therefore, we have

$$\sum_\nu (y(\nu) - \bar{y})^2 = \sum_\nu (y(\nu) - \hat{y}(\nu))^2 + \sum_\nu (\hat{y}(\nu) - \bar{y})^2$$

or

$$1 = \frac{\sum_\nu (y(\nu) - \hat{y}(\nu))^2}{\sum_\nu (y(\nu) - \bar{y})^2} + \frac{\sum_\nu (\hat{y}(\nu) - \bar{y})^2}{\sum_\nu (y(\nu) - \bar{y})^2}$$

or

$$1 = \frac{\sum_\nu (y(\nu) - \hat{y}(\nu))^2}{\sum_\nu (y(\nu) - \bar{y})^2} + R^2,$$

where the *coefficient of determination* R^2,

$$R^2 = \frac{\sum_\nu (\hat{y}(\nu) - \bar{y})^2}{\sum_\nu (y(\nu) - \bar{y})^2}, \quad 0 \le R^2 \le 1, \tag{2.93}$$

is the fraction of the variance in Y that is explained by the regression model. It is easy to show (Exercise 22) that

$$R = \mathrm{corr}(y, \hat{y}) = \frac{\sum_\nu (y(\nu) - \bar{y})(\hat{y}(\nu) - \bar{y})}{\sqrt{\sum_\nu (y(\nu) - \bar{y})^2 \sum_\nu (\hat{y}(\nu) - \bar{y})^2}}. \tag{2.94}$$

In words: The coefficient of determination is the square of the sample correlation between the observed and predicted values of y.

2.6.3 More than one independent variable

The statistical model of the preceding section may be written more generally in the form

$$Y(\nu) = w_o + \sum_{i=1}^{N} w_i x_i(\nu) + R(\nu), \quad \nu = 1 \dots m, \tag{2.95}$$

relating m measurements of the N independent variables $x_1 \dots x_N$ to a measured dependent variable Y via the parameters $w_0, w_1 \dots w_N$. Equivalently, in vector notation, we can write

$$Y(\nu) = \boldsymbol{w}^\top \boldsymbol{x}(\nu) + R(\nu), \quad \nu = 1 \dots m, \tag{2.96}$$

where $\boldsymbol{x} = (x_0 = 1, x_1 \dots x_N)^\top$ and $\boldsymbol{w} = (w_0, w_1 \dots w_N)^\top$. The random variables $R(\nu)$ again represent the measurement uncertainty in the realizations $y(\nu)$ of $Y(\nu)$. We assume that they are independent and identically distributed with zero mean and variance σ^2, whereas the values $\boldsymbol{x}(\nu)$ are, as before, assumed to be exact. Now we wish to determine the best value for parameter vector $\boldsymbol{w}$.

Introducing the $m \times (N+1)$ data matrix

$$\boldsymbol{\mathcal{X}} = \begin{pmatrix} \boldsymbol{x}(1)^\top \\ \vdots \\ \boldsymbol{x}(m)^\top \end{pmatrix},$$

we can express Equation (2.95) or (2.96) in the form

$$\boldsymbol{Y} = \boldsymbol{\mathcal{X}} \boldsymbol{w} + \boldsymbol{R}, \tag{2.97}$$

where $\boldsymbol{Y} = (Y(1) \dots Y(m))^\top$, $\boldsymbol{R} = (R(1) \dots R(m))^\top$ and, by assumption,

$$\boldsymbol{\Sigma}_R = \langle \boldsymbol{R}\boldsymbol{R}^\top \rangle = \sigma^2 \boldsymbol{I}.$$

The identity matrix $\boldsymbol{I}$ is $m \times m$. The goodness-of-fit function analog to Equation (2.88) is

$$z(\boldsymbol{w}) = \sum_{\nu=1}^{m} \left[\frac{y(\nu) - \boldsymbol{w}^\top \boldsymbol{x}(\nu)}{\sigma} \right]^2 = \frac{1}{\sigma^2} (\boldsymbol{y} - \boldsymbol{\mathcal{X}}\boldsymbol{w})^\top (\boldsymbol{y} - \boldsymbol{\mathcal{X}}\boldsymbol{w}). \tag{2.98}$$

This is minimized by solving the equations

$$\frac{\partial z(\boldsymbol{w})}{\partial w_k} = 0, \quad k = 0 \dots N.$$

Using the rules for vector differentiation, we obtain (Exercise 20)

$$\boldsymbol{\mathcal{X}}^\top \boldsymbol{y} = (\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}}) \boldsymbol{w} \ . \tag{2.99}$$

Equation (2.99) is referred to as the *normal equation*. The estimated parameters of the model are obtained by solving for $\boldsymbol{w}$,

$$\hat{\boldsymbol{w}} = (\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}})^{-1} \boldsymbol{\mathcal{X}}^\top \boldsymbol{y} =: \boldsymbol{\mathcal{X}}^+ \boldsymbol{y}. \tag{2.100}$$

The matrix

$$\boldsymbol{\mathcal{X}}^+ = (\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}})^{-1} \boldsymbol{\mathcal{X}}^\top \tag{2.101}$$

is the pseudo inverse of the data matrix $\boldsymbol{\mathcal{X}}$.*

The following code snippet mimics the solution of the normal equation for some simulated data in Python/NumPy:

```
# biased data matrix X ( 3 independent variables)
X = np.random.rand(100,3)
X = np.mat(np.append(np.ones((100,1)),X,axis=1))
# a parameter vector
w = np.mat([[3.0],[4.0],[5.0],[6.0]])
# noisy dependent variable y with sigma = 0.1
y = X*w + np.random.normal(0,0.1,(100,1))
# pseudo inverse
Xp = (X.T*X).I*X.T
# estimated parameter vector
w = Xp*y
w

matrix([[3.01289152],
        [3.99940076],
        [4.99011592],
        [5.97549588]])
```

As a very trivial encounter with `tensorflow`, here is the same calculation:

*In terms of the singular value decomposition (SVD) of $\boldsymbol{\mathcal{X}}$, namely $\boldsymbol{\mathcal{X}} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{V}^\top$, the pseudo inverse is $\boldsymbol{\mathcal{X}}^+ = \boldsymbol{V}\boldsymbol{\Lambda}^{-1}\boldsymbol{U}^\top$, generalizing the definition for a symmetric square matrix given by Equation (1.50).

```
import tensorflow as tf
X1 = tf.constant(X)
y1 = tf.constant(y)
X1T = tf.transpose(X1)
X1p = tf.linalg.matmul(tf.linalg.inv(
        tf.linalg.matmul(tf.transpose(X1), X1)),
        tf.transpose(X1))
w = tf.linalg.matmul(X1p,y1)
print(w)

tf.Tensor(
[[2.95078213]
 [4.04382244]
 [5.02639515]
 [6.03419036]], shape=(4, 1), dtype=float64)
```

And once again the same calculation, now with the GEE Python API:

```
ee.Initialize()
# set up JSON description of the calculation
X1 = ee.Array(X.tolist())
y1 = ee.Array(y.tolist())
X1T = X1.matrixTranspose()
X1p = X1T.matrixMultiply(X1) \
         .matrixInverse() \
         .matrixMultiply(X1T)
w = X1p.matrixMultiply(y1)
# run on GEE server
np.round(w.getInfo(),4)

array([[3.0097],
       [4.0259],
       [5.0152],
       [5.9958]])
```

In order to obtain the uncertainty in the estimate $\hat{\boldsymbol{w}}$, Equation (2.100), we can think of $\boldsymbol{w}$ as a random vector with mean value $\langle\boldsymbol{w}\rangle$. Its covariance matrix is then given by

$$\begin{aligned}\boldsymbol{\Sigma}_w &= \big\langle(\boldsymbol{w}-\langle\boldsymbol{w}\rangle)(\boldsymbol{w}-\langle\boldsymbol{w}\rangle)^\top\big\rangle\\ &\approx \big\langle(\boldsymbol{w}-\hat{\boldsymbol{w}})(\boldsymbol{w}-\hat{\boldsymbol{w}})^\top\big\rangle\\ &= \big\langle(\boldsymbol{w}-\boldsymbol{\mathcal{X}}^+\boldsymbol{y})(\boldsymbol{w}-\boldsymbol{\mathcal{X}}^+\boldsymbol{y})^\top\big\rangle\\ &= \big\langle(\boldsymbol{w}-\boldsymbol{\mathcal{X}}^+(\boldsymbol{\mathcal{X}}\boldsymbol{w}+\boldsymbol{r}))(\boldsymbol{w}-\boldsymbol{\mathcal{X}}^+(\boldsymbol{\mathcal{X}}\boldsymbol{w}+\boldsymbol{r}))^\top\big\rangle.\end{aligned}$$

But from Equation (2.101), we see that $\boldsymbol{\mathcal{X}}^+\boldsymbol{\mathcal{X}}=\boldsymbol{I}$, so

$$\begin{aligned}\boldsymbol{\Sigma}_w &\approx \langle(-\boldsymbol{\mathcal{X}}^+\boldsymbol{r})(-\boldsymbol{\mathcal{X}}^+\boldsymbol{r})^\top\rangle = \boldsymbol{\mathcal{X}}^+\langle\boldsymbol{r}\boldsymbol{r}^\top\rangle\boldsymbol{\mathcal{X}}^{+\top}\\ &= \sigma^2\boldsymbol{\mathcal{X}}^+\boldsymbol{\mathcal{X}}^{+\top}.\end{aligned}$$

Again with Equation (2.101), we get finally the error covariance matrix as

$$\boldsymbol{\Sigma}_w \approx \sigma^2(\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}})^{-1}. \tag{2.102}$$

Continuing the example with simulated data, $\boldsymbol{\Sigma}_w$ evaluates as:

```
np.round(0.01*(X.T*X).I,6)

array([[ 1.194e-03, -6.110e-04, -6.510e-04, -8.350e-04],
       [-6.110e-04,  1.067e-03,  3.900e-05,  6.500e-05],
       [-6.510e-04,  3.900e-05,  1.315e-03, -2.200e-05],
       [-8.350e-04,  6.500e-05, -2.200e-05,  1.482e-03]])
```

To check that Equation (2.102) is indeed a generalization of ordinary linear regression on a single independent variable, identify the parameter vector $\boldsymbol{w}$ with the straight line parameters a and b, i.e.,

$$\boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}.$$

The matrix $\boldsymbol{\mathcal{X}}$ and vector $\boldsymbol{y}$ are correspondingly

$$\boldsymbol{\mathcal{X}} = \begin{pmatrix} 1 & x(1) \\ 1 & x(2) \\ \vdots & \vdots \\ 1 & x(m) \end{pmatrix}, \quad \boldsymbol{y} = \begin{pmatrix} y(1) \\ y(2) \\ \vdots \\ y(m) \end{pmatrix}.$$

Thus the best estimates for the parameters are

$$\hat{\boldsymbol{w}} = \begin{pmatrix} \hat{a} \\ \hat{b} \end{pmatrix} = (\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}})^{-1}(\boldsymbol{\mathcal{X}}^\top \boldsymbol{y}).$$

Evaluating:

$$(\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}})^{-1} = \begin{pmatrix} m & \sum x(\nu) \\ \sum x(\nu) & \sum x(\nu)^2 \end{pmatrix}^{-1} = \begin{pmatrix} m & m\bar{x} \\ m\bar{x} & \sum x(\nu)^2 \end{pmatrix}^{-1}.$$

Recalling the expression for the inverse of a 2×2 matrix in Chapter 1, we then have

$$(\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}})^{-1} = \frac{1}{m\sum x(\nu)^2 + m^2\bar{x}^2} \begin{pmatrix} \sum x(\nu)^2 & -m\bar{x} \\ -m\bar{x} & m \end{pmatrix}.$$

Furthermore,

$$\boldsymbol{\mathcal{X}}^\top \boldsymbol{y} = \begin{pmatrix} m\bar{y} \\ \sum x(\nu)y(\nu) \end{pmatrix}.$$

Therefore, the estimate for b is

$$\hat{b} = \frac{1}{m\sum x(\nu)^2 + m^2\bar{x}^2}\left(-m^2\bar{x}\bar{y} + m\sum x(\nu)y(\nu)\right) = \frac{-m\bar{x}\bar{y} + \sum x(\nu)y(\nu)}{m\sum x(\nu)^2 + m^2\bar{x}^2}. \tag{2.103}$$

From Equation (2.102), the uncertainty in b is given by σ^2 times the (2,2) element of $(\boldsymbol{\mathcal{X}}^\top\boldsymbol{\mathcal{X}})^{-1}$,

$$\sigma_b^2 = \sigma^2 \frac{m}{m\sum x(\nu)^2 + m^2\bar{x}^2}. \tag{2.104}$$

Equations (2.103) and (2.104) correspond to Equations (2.89) and (2.91).

2.6.4 Regularization, duality, and the Gram matrix

For *ill-conditioned* regression problems (e.g., large amount of noise, insufficient data or $\boldsymbol{\mathcal{X}}^\top\boldsymbol{\mathcal{X}}$ nearly singular), the solution $\hat{\boldsymbol{w}}$ in Equation (2.100) may be unreliable. A remedy is to restrict $\boldsymbol{w}$ in some way, the simplest one being to favor a small length or, equivalently, a small squared norm $\|\boldsymbol{w}\|^2$. In the modified goodness-of-fit function

$$z(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{\mathcal{X}}\boldsymbol{w})^\top(\boldsymbol{y} - \boldsymbol{\mathcal{X}}\boldsymbol{w}) + \lambda\|\boldsymbol{w}\|^2, \tag{2.105}$$

where we have assumed $\sigma^2 = 1$ for simplicity, the parameter λ defines a trade-off between minimum residual error and minimum norm. Equating the vector derivative with respect to $\boldsymbol{w}$ with zero as before then leads to the normal equation

$$\boldsymbol{\mathcal{X}}^\top\boldsymbol{y} = (\boldsymbol{\mathcal{X}}^\top\boldsymbol{\mathcal{X}} + \lambda\boldsymbol{I}_{N+1})\boldsymbol{w}, \tag{2.106}$$

where the identity matrix $\boldsymbol{I}_{N+1}$ has dimensions $(N+1)\times(N+1)$. The least squares estimate for the parameter vector $\boldsymbol{w}$ is now

$$\hat{\boldsymbol{w}} = (\boldsymbol{\mathcal{X}}^\top\boldsymbol{\mathcal{X}} + \lambda\boldsymbol{I}_{N+1})^{-1}\boldsymbol{\mathcal{X}}^\top\boldsymbol{y}. \tag{2.107}$$

For $\lambda > 0$, the matrix $\boldsymbol{\mathcal{X}}^\top\boldsymbol{\mathcal{X}} + \lambda\boldsymbol{I}_{N+1}$ can always be inverted. This procedure is known as *ridge regression* and Equation (2.107) may be referred to as its *primal solution*. Regularization will be encountered in Chapter 9 in the context of change detection.

With a simple manipulation, Equation (2.107) can be put in the form

$$\hat{\boldsymbol{w}} = \boldsymbol{\mathcal{X}}^\top\boldsymbol{\alpha} = \sum_{\nu=1}^{m}\alpha_\nu\boldsymbol{x}(\nu), \tag{2.108}$$

where $\boldsymbol{\alpha}$ is given by

$$\boldsymbol{\alpha} = \frac{1}{\lambda}(\boldsymbol{y} - \boldsymbol{\mathcal{X}}\hat{\boldsymbol{w}}); \tag{2.109}$$

see Exercise 23. Equation (2.108) expresses the unknown parameter vector $\hat{\boldsymbol{w}}$ as a linear combination of the observation vectors $\boldsymbol{x}(\nu)$. It remains to find a more suitable expression for the vector $\boldsymbol{\alpha}$. We can eliminate $\hat{\boldsymbol{w}}$ from Equation (2.109) by substituting Equation (2.108) and solving for $\boldsymbol{\alpha}$,

$$\boldsymbol{\alpha} = (\boldsymbol{\mathcal{X}}\boldsymbol{\mathcal{X}}^\top + \lambda\boldsymbol{I}_m)^{-1}\boldsymbol{y}, \tag{2.110}$$

where $\boldsymbol{I}_m$ is the $m \times m$ identity matrix. Equations (2.108) and (2.110) taken together constitute the *dual solution* of the ridge regression problem, and the components of $\boldsymbol{\alpha}$ are called the *dual parameters*. Once they have been determined from Equation (2.110), the solution for the original parameter vector $\hat{\boldsymbol{w}}$ is recovered from Equation (2.108). Note that in the primal solution, Equation (2.107), we are inverting a $(N+1) \times (N+1)$ matrix,

$$\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}} + \lambda \boldsymbol{I}_{N+1}$$

whereas in the dual solution, we must invert the (often much larger) $m \times m$ matrix

$$\boldsymbol{\mathcal{X}}\boldsymbol{\mathcal{X}}^\top + \lambda \boldsymbol{I}_m.$$

The matrix $\boldsymbol{\mathcal{X}}\boldsymbol{\mathcal{X}}^\top$ is called a *Gram matrix*. Its elements consist of all inner products of the observation vectors

$$\boldsymbol{x}(\nu)^\top \boldsymbol{x}(\nu'), \quad \nu, \nu' = 1 \ldots m,$$

and it is obviously a symmetric matrix. Moreover, it is positive semi-definite since, for any vector $\boldsymbol{z}$ with m components,

$$\boldsymbol{z}^\top \boldsymbol{\mathcal{X}}\boldsymbol{\mathcal{X}}^\top \boldsymbol{z} = \|\boldsymbol{\mathcal{X}}^\top \boldsymbol{z}\|^2 \ge 0.$$

The dual solution is interesting because the dual parameters are expressed entirely in terms of inner products of observation vectors $\boldsymbol{x}(\nu)$. Moreover, predicting values of y from new observations $\boldsymbol{x}$ can be expressed purely in terms of inner products as well. Thus

$$y = \hat{\boldsymbol{w}}^\top \boldsymbol{x} = \left(\sum_{\nu=1}^{m} \alpha_\nu \boldsymbol{x}(\nu)^\top \right) \boldsymbol{x} = \sum_{\nu=1}^{m} \alpha_\nu \ \boldsymbol{x}(\nu)^\top \boldsymbol{x}. \tag{2.111}$$

Later we will see that the inner products can be substituted by so-called *kernel functions*, which allow very elegant and powerful non-linear generalizations of linear methods such as ridge regression.

2.7 Entropy and information

Suppose we make an observation on a discrete random variable X with mass function

$$p\big(X = x(i)\big) = p(i), \quad i = 1 \ldots n.$$

Qualitatively speaking, the *amount of information* we receive on observing a particular realization $x(i)$ may be thought of as the "amount of surprise" associated with the result. The information content of the observation should be a monotonically decreasing function of the probability $p(i)$ for that observation: if the probability is unity, there is no surprise; if $p(i) \ll 1$, the surprise is large. The function chosen to express the information content of $x(i)$ is

$$h(x(i)) = -\log p(i). \tag{2.112}$$

This function is monotonically decreasing and zero when $p(i) = 1$. It also has the desirable property that, for two independent observations $x(i), x(j)$,

$$\begin{aligned} h(x(i), x(j)) &= -\log p(X = x(i), X = x(j)) \\ &= -\log\left[p(X = x(i))p(X = x(j))\right] \\ &= -\log\left[p(i)p(j)\right] = h(x(i)) + h(x(j)), \end{aligned}$$

that is, information gained from two independent observations is additive.

The *average amount of information* that we expect to receive on observing the random variable X is called the *entropy* of X and is given by

$$H(X) = -\sum_{i=1}^{n} p(i) \log p(i). \tag{2.113}$$

The entropy can also be interpreted as the *average amount of information required to specify the random variable.*

The discrete distribution with maximum entropy can be determined by maximizing the Lagrange function

$$L\left(p(1) \dots p(n)\right) = -\sum_i p(i) \log p(i) + \lambda \left(\sum_i p(i) - 1)\right).$$

Equating the derivatives to zero,

$$\frac{\partial L}{\partial p(i)} = -\log p(i) - 1 + \lambda = 0,$$

so $p(i)$ is independent of i. The condition $\sum_i p(i) = 1$ then requires that

$$p(i) = 1/n.$$

The Hessian matrix is easily seen to have diagonal elements given by

$$(\boldsymbol{H})_{ii} = \frac{\partial^2 L}{\partial p(i)^2} = -\frac{1}{p(i)},$$

and off-diagonal elements zero. It is therefore negative definite, i.e., for any $\boldsymbol{x} > \boldsymbol{0}$,

$$\boldsymbol{x}^\top \boldsymbol{H} \boldsymbol{x} = -\frac{1}{p(1)} x_1^2 - \dots - \frac{1}{p(n)} x_n^2 < 0.$$

Thus, the uniform distribution indeed maximizes the entropy.

If X is a continuous random variable with probability density function $p(x)$, then its entropy* is defined analogously to Equation (2.113) as

$$H(X) = -\int p(x)\log[p(x)]dx. \tag{2.114}$$

The continuous distribution function which has maximum entropy is the normal distribution; see Bishop (2006), Chapter 1, for a derivation of this fact.

If $p(x, y)$ is a joint density function for random variables X and Y, then the *conditional entropy* of Y given X is

$$\begin{aligned} H(Y \mid X) &= -\int p(x)\left(\int p(y \mid x)\log[p(y \mid x)]dy\right)dx \\ &= -\int\int p(x, y)\log[p(y \mid x)]dydx. \end{aligned} \tag{2.115}$$

For the second equality, we have used Equation (2.65). This is just the information $-\log[p(y \mid x)]$ gained on observing y given x, averaged over the joint probability for x and y. If Y is independent of X, then $p(y \mid x) = p(y)$ and $H(Y \mid X) = H(Y)$.

We can express the entropy $H(X, Y)$ associated with the random vector $(X, Y)^\top$ in terms of conditional entropy as follows:

$$\begin{aligned} H(X, Y) &= -\int\int p(x, y)\log[p(x, y)]dxdy \\ &= -\int\int p(x, y)\log[p(y \mid x)p(x)]dxdy \\ &= -\int\int p(x, y)\log[p(y \mid x)]dxdy - \int\int p(x, y)\log[p(x)]dxdy \\ &= H(Y \mid X) - \int\left(\int p(x, y)dy\right)\log[p(x)]dx \\ &= H(Y \mid X) - \int p(x)\log[p(x)]dx \end{aligned}$$

or

$$H(X, Y) = H(Y \mid X) + H(X). \tag{2.116}$$

2.7.1 Kullback–Leibler divergence

Let $p(x)$ be some unknown density function for a random variable X, and let $q(x)$ represent an approximation of that density function. Then, the information required to specify X when using $q(x)$ as an approximation for $p(x)$ is given by

$$-\int p(x)\log[q(x)]dx.$$

*More correctly, *differential entropy* (Bishop, 2006).

The *additional information* required relative to that for the correct density function is called the *Kullback–Leibler (KL) divergence* between density functions $p(x)$ and $q(x)$ and is given by

$$\begin{aligned}\mathrm{KL}(p,q) &= -\int p(x)\log[q(x)]dx - \left(-\int p(x)\log[p(x)]dx\right)\\ &= -\int p(x)\log\left[\frac{q(x)}{p(x)}\right]dx.\end{aligned} \tag{2.117}$$

The KL divergence can be shown to satisfy (Exercise 24)

$$\mathrm{KL}(p,q) > 0,\ p(x)\neq q(x),\quad \mathrm{KL}(p,p)=0,$$

and is thus a measure of the dissimilarity between $p(x)$ and $q(x)$.

2.7.2 Mutual information

Consider two gray-scale images represented by random variables X and Y. Their joint probability distribution is $p(x,y)$. If the images are completely independent, then

$$p(x,y) = p(x)p(y).$$

Thus the extent $I(X,Y)$ to which they are *not* independent can be measured by the KL divergence between $p(x,y)$ and $p(x)p(y)$:

$$I(X,Y) = \mathrm{KL}(p(x,y),p(x)p(y)) = -\int\int p(x,y)\log\left[\frac{p(x)p(y)}{p(x,y)}\right]dxdy, \tag{2.118}$$

which is called the *mutual information* between X and Y. Expanding:

$$\begin{aligned}I(X,Y) &= -\int\int p(x,y)\big[\log[p(x)]+\log[p(y)]-\log[p(x,y)]\big]dxdy\\ &= H(X)+H(Y)+\int\int p(x,y)\log[p(x\mid y)p(y)]dxdy\\ &= H(X)+H(Y)-H(X\mid Y)-H(Y),\end{aligned}$$

and thus

$$I(X,Y) = H(X) - H(X\mid Y). \tag{2.119}$$

Mutual information measures the degree of dependence between the two images, a value of zero indicating statistical independence. This is to be contrasted with correlation, where a value of zero implies statistical independence only for normally distributed quantities; see Theorem 2.9.

In practice, the images are quantized, so that if p_1 and p_2 are their normalized histograms (i.e., $\sum_i p_1(i) = \sum_i p_2(i) = 1$) and p_{12} is the normalized two-dimensional histogram, $\sum_{ij} p_{12}(i,j) = 1$, then the mutual information is approximately

$$
\begin{aligned}
I(1,2) &= -\sum_{ij} p_{12}(i,j)\big(\log[p_1(i)] + \log[p_2(j)] - \log[p_{12}(i,j)]\big) \\
&= \sum_{ij} p_{12}(i,j) \log \frac{p_{12}(i,j)}{p_1(i)p_2(j)}.
\end{aligned} \tag{2.120}
$$

The following code is a (naive; see Kraskov et al. (2004)) calculation of the mutual information of VNIR band combinations for the ASTER image of Figure 1.1:

```
def mi(arr1,arr2):
    '''mutual information of two uint8 arrays '''
    p12 = np.histogram2d(arr1,arr2,bins=256,
                          density=True)[0].ravel()
    p1  = np.histogram(arr1,bins=256,density=True)[0]
    p2  = np.histogram(arr2,bins=256,density=True)[0]
    p1p2 = np.outer(p1,p2).ravel()
    idx = p12>0
    return np.sum(p12[idx]*np.log(p12[idx]/p1p2[idx]))

gdal.AllRegister()
infile = 'imagery/AST_20070501.tif'

inDataset = gdal.Open(infile,GA_ReadOnly)
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize
image = np.zeros((3,rows*cols))
# VNIR bands
for b in range(3):
    band = inDataset.GetRasterBand(b+1)
    image[b,:]=np.byte(band.ReadAsArray(0,0,cols,rows))\
                                              .ravel()

print(mi(image[0,:],image[1,:]))
print(mi(image[0,:],image[2,:]))
print(mi(image[1,:],image[2,:]))
```

```
1.81539975948
0.507049223344
0.636113398912
```

The first two bands (visual spectrum) have a higher mutual information or dependency than either of the visual bands with the third (near infrared or vegetation) band.

2.8 Exercises

1. Derive Equations (2.13).

2. Let the random variable X be standard normally distributed with density function

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \exp(-x^2/2), \quad -\infty < x < \infty.$$

Show that the random variable $|X|$ has the density function

$$p(x) = \begin{cases} 2\phi(x) & \text{for } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

3. (a) Show that the chi-square distribution can be expressed in terms of the lower incomplete gamma function:

$$P_{\chi^2;m}(z) = \gamma(m/2, z/2).$$

(b) Program the `scipy.stats.ch2.ppf()` function in the GEE Python API by using the incomplete gamma function.

4. Use Equation (2.8) and the result of Exercise 2 to show that the random variable $Y = X^2$, where X is standard normally distributed, has the chi-square density function, Equation (2.38), with $m = 1$ degree of freedom.

5. If X_1 and X_2 are independent random variables, both standard normally distributed, show that $X_1 + X_2$ is normally distributed with mean 0 and variance 2. (*Hint*: Write down the joint density function $f(x_1, x_2)$ for X_1 and X_2. Then treat x_1 as fixed and apply Theorem 2.1.)

6. Show from Theorem 2.3 that the sample mean

$$\bar{Z} = \frac{1}{m}\sum_{i=1}^{m} Z_i \ ,$$

is normally distributed with mean μ and variance σ^2/m.

7. Prove that, for $\alpha > 1$, $\Gamma(\alpha) = (\alpha - 1)\Gamma(\alpha - 1)$ and hence that, for positive integers n, $\Gamma(n) = (n-1)!$ (Hint: Use integration by parts.)

8. (a) Show that the mean and variance of a random variable Z with the gamma probability density, Equation (2.33), are $\mu = \alpha\beta$ and $\sigma^2 = \alpha\beta^2$.

(b) (Proof of Theorem 2.5 for $m = 2$) Suppose that Z_1 and Z_2 are independent and exponentially distributed random variables with density functions as in Equation (2.37). Then, we can write the probability distribution of $Z = Z_1 + Z_2$ in the form

$$P(z) = \Pr(Z_1 + Z_2 < z) = \int_0^z \int_0^{z-z_2} \frac{1}{\beta} e^{-z_1/\beta} \frac{1}{\beta} e^{-z_2/\beta} dz_1 dz_2.$$

Evaluate this double integral and then take its derivative with respect to z to show that the probability density function for Z is the gamma density with $\alpha = 2$, i.e.,

$$p(z) = \begin{cases} \frac{1}{\beta^2\Gamma(2)} z e^{-z/\beta} & \text{for } z > 0 \\ 0 & \text{elsewhere.} \end{cases}$$

9. Prove Theorem 2.7 by noticing that the inverse transformation of $s = x + y$, $u = x/(x + y)$ is given by $x = su$, $y = s(1 - u)$ and applying Theorem 2.2.

10. For constant vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ and random vector $\boldsymbol{G}$ with covariance matrix $\boldsymbol{\Sigma}$, demonstrate that $\mathrm{cov}(\boldsymbol{a}^\top \boldsymbol{G}, \boldsymbol{b}^\top \boldsymbol{G}) = \boldsymbol{a}^\top \boldsymbol{\Sigma} \boldsymbol{b}$.

11. Write down the multivariate normal probability density function $p(\boldsymbol{z})$ for the case $\boldsymbol{\Sigma} = \sigma^2 \boldsymbol{I}$. Show that probability density function $p(z)$ for a one-dimensional random variable Z is a special case. Using the fact that $\int_{-\infty}^{\infty} p(z)dz = 1$, demonstrate that $\langle Z \rangle = \mu$.

12. Given the $m \times N$ (uncentered) data matrix $\boldsymbol{\mathcal{Z}}$, show that the covariance matrix estimate can be written in the form

$$(m-1)\boldsymbol{s} = \boldsymbol{\mathcal{Z}}^\top \boldsymbol{H} \boldsymbol{\mathcal{Z}},$$

where the *centering matrix* $\boldsymbol{H}$ is given by

$$\boldsymbol{H} = \boldsymbol{I}_{mm} - \frac{1}{m} \boldsymbol{1}_m \boldsymbol{1}_m^\top.$$

Show that $\boldsymbol{H}$ is not only symmetric, but also *idempotent* ($\boldsymbol{H}\boldsymbol{H} = \boldsymbol{H}$). Use this fact to prove that $\boldsymbol{s}$ is positive semi-definite.

13. Demonstrate Equation (2.58).

14. In the game *Lets Make a Deal!* a contestant is asked to choose between one of three doors. Behind one of the doors, the prize is an automobile. After the contestant has chosen, the quiz master opens one of the other two doors to show that the automobile is not there. He then asks the contestant if she wishes to change her mind and switch from her original choice to the other unopened door. Use Bayes' Theorem to prove that her correct answer is "yes."

15. Starting from the definition of the P-value, we can write

$$\mathcal{P} = 1 - \Pr(Q < q \mid H_0) = 1 - F_0(q),$$

where F_0 is the probability distribution function of the test statistic Q under the null hypothesis. Use the fact that F_0 is monotonic increasing to show that

$$\Pr(F_0(Q) < F_0(q)) = F_0(q)$$

and thus that $F_0(Q)$ (and hence $\mathcal{P}$) follows a uniform distribution.

16. Write a Python script to generate two normal distributions with the random number generator `numpy.random.randn()` and test them for equal means with the Student-t test (`scipy.stats.ttest_ind()`) and for equal variance with the F-test (`scipy.stats.bartlett()`).

17. Show that the critical region for the likelihood ratio test for $\mu = \mu_0$ against $\mu \neq \mu_0$ for known variance σ^2 and m samples can be written as

$$\exp\left(-\frac{1}{2\sigma^2/m}(\bar{z} - \mu_0)^2\right) \le k.$$

18. Prove Equations (2.89) for the regression parameter estimates $\hat{a}$ and $\hat{b}$. Demonstrate that these values correspond to a minimum of the goodness-of-fit function, Equation (2.88), and not to a maximum.

19. Derive the uncertainty for a in Equation (2.91) from the formula for error propagation for uncorrelated errors

$$\sigma_a^2 = \sum_{i=1}^{n} \sigma^2 \left(\frac{\partial a}{\partial y(i)}\right)^2 .$$

20. Derive Equation (2.99) by applying the rules for vector differentiation to minimize the goodness-of-fit function, Equation (2.98).

21. Write a Python program to calculate the regression coefficients of spectral band 2 on spectral band 1 of a multi-spectral image. (The built-in function for ordinary linear regression is `numpy.linalg.lstsq()` .)

22. Prove Equation (2.94) by replacing $(y(\nu) - \bar{y})$ in the numerator by $(y(\nu) + \hat{y}(\nu) - \hat{y}(\nu) + \bar{y})$ and expanding.

23. Show that Equations (2.107) and (2.108) are equivalent to Equation (2.109).

24. A *convex function* $f(x)$ satisfies $f(\lambda a + (1-\lambda)b) \leq \lambda f(a) + (1-\lambda)f(b)$. *Jensen's inequality* states that, for any convex function $f(x)$, any function $g(x)$ and any probability density $p(x)$,

$$\int f(g(x))p(x)dx \geq f\left(\int g(x)p(x)dx\right). \tag{2.121}$$

Use this to show that the KL divergence satisfies

$$\mathrm{KL}(p,q) > 0,\ p(x) \neq q(x), \quad \mathrm{KL}(p,p) = 0.$$

25. Explain why the KL divergence, Equation (2.117), although a measure of the difference between two probability distributions, is not an appropriate distance measure.

3

Transformations

Thus far, we have thought of visual/infrared and polarimetric SAR images as three-dimensional arrays of pixel intensities (columns $\times$ rows $\times$ bands) representing, more or less directly, measured radiances. In the present chapter, we consider other, more abstract representations which are useful in image interpretation and analysis and which will play an important role in later chapters.

The discrete Fourier and wavelet transforms that we treat in Sections 3.1 and 3.2 convert the pixel values in a given spectral band to linear combinations of orthogonal functions of spatial frequency and distance. They may therefore be classified as *spatial transformations.* The principal components, minimum noise fraction, and maximum autocorrelation factor transformations (Sections 3.3 to 3.5), on the other hand, create at each pixel location new linear combinations of the pixel intensities from all of the spectral bands and can properly be called *spectral transformations* (Schowengerdt, 2006).

3.1 The discrete Fourier transform

Let the function $g(x)$ represent the radiance at a point x focused along a row of push broom-geometry sensors, and $g(j)$ be the corresponding pixel intensities stored in a row of a digital image. We can think of $g(j)$ approximately as a discrete sample* of the function $g(x)$, taken c times at some sampling interval Δ, c being the number of columns in the image, i.e.,

$$g(j) = g(x = j\Delta), \quad j = 0 \ldots c-1.$$

For convenience, that is, to correspond to Python/NumPy array indexing, the pixels are numbered from zero, a convention that will be adhered to in the remainder of the book. The interval Δ is the sensor width or, projected back to the Earth's surface, the across-track ground sample distance (GSD).

*More correctly, $g(j)$ is a result of convolutions of the spatial and spectral response functions of the detector with the focused signal; see Chapter 4.

DOI: 10.1201/9781003503286-3

The theory of Fourier analysis states that the function $g(x)$ can be expressed in the form

$$g(x) = \int_{-\infty}^{\infty} \hat{g}(f) e^{\mathbf{i}2\pi fx} df, \tag{3.1}$$

where $\hat{g}(f)$ is called the *Fourier transform* of $g(x)$. Equation (3.1) describes a continuous superposition of periodic complex functions of x,

$$e^{\mathbf{i}2\pi fx} = \cos(2\pi fx) + \mathbf{i}\sin(2\pi fx),$$

having frequency f. Most often, frequency is associated with inverse time (cycles per second). Here, of course, we are speaking of *spatial frequency*, or cycles per meter.

In general, we require a continuum of periodic functions $e^{\mathbf{i}2\pi fx}$ to represent $g(x)$ in this way. However, if $g(x)$ is in fact itself periodic with period T, that is, if $g(x+T) = g(x)$, then the integral in Equation (3.1) can be replaced by an infinite sum of *discrete* periodic functions of frequency kf for $-\infty < k < \infty$, where f is the *fundamental frequency* $f = 1/T$:

$$g(x) = \sum_{k=-\infty}^{\infty} \hat{g}(k) e^{\mathbf{i}2\pi(kf)x}. \tag{3.2}$$

If we think of the sampled series of pixels $g(j)$, $j = 0 \dots c-1$, as also being periodic with period $T = c\Delta$, that is, repeating itself to infinity in both positive and negative directions, then we can replace x in Equation (3.2) by $j\Delta$ and express $g(j)$ in a similar way:

$$g(j) = g(j\Delta) = \sum_{k=-\infty}^{\infty} \hat{g}(k) e^{\mathbf{i}2\pi(kf)j\Delta} = \sum_{k=-\infty}^{\infty} \hat{g}(k) e^{\mathbf{i}2\pi kj/c}, \tag{3.3}$$

where in the last equality we have used $f\Delta = \Delta/T = 1/c$.

The limits in the summation in Equation (3.3) must, however, be truncated. This is due to the fact that there is a limit to the highest frequency $k_{max}f$ that can be measured by sampling at the interval Δ. The limit is called the *Nyquist critical frequency* f_N. It may be determined simply by observing that the minimum number of samples needed to describe a sine wave completely is two per period (e.g., at the maximum and minimum values). Therefore, the shortest period measurable is 2Δ and the Nyquist frequency is

$$f_N = \frac{1}{2\Delta} = \frac{cf}{2}.$$

Hence, $k_{max} = c/2$. Taking this into account in Equation (3.3), we obtain

$$g(j) = \sum_{k=-c/2}^{c/2} \hat{g}(k) e^{\mathbf{i}2\pi kj/c}, \quad j = 0 \dots c-1. \tag{3.4}$$

The effect of truncation depends upon the nature of the function $g(x)$ being sampled. According to the *Sampling Theorem*, see, e.g., Press et al. (2002), $g(x)$ is completely determined by the samples $g(j)$ in Equation (3.4) if it is *bandwidth limited* to frequencies smaller than f_N, i.e., provided that, in Equation (3.1), $\hat{g}(f) = 0$ for all $|f| \geq f_N$. If this is not the case, then any frequency component outside the interval $(-f_N, f_N)$ is spuriously moved into that range, a phenomenon referred to as *aliasing*.

To bring Equation (3.4) into a more convenient form, we have to make a few simple manipulations. To begin with, note that the exponents in the first and last terms in the summation are equal, i.e.,

$$e^{\mathbf{i}2\pi(-c/2)j/c} = e^{-\mathbf{i}\pi j} = (-1)^j = e^{\mathbf{i}\pi j} = e^{\mathbf{i}2\pi(c/2)j/c},$$

so we can lump those two terms together and write Equation (3.4) equivalently as

$$g(j) = \sum_{k=-c/2}^{c/2-1} \hat{g}(k)e^{\mathbf{i}2\pi kj/c}, \quad j = 0 \ldots c-1.$$

Rearranging further,

$$\begin{aligned}
g(j) &= \sum_{k=0}^{c/2-1} \hat{g}(k)e^{\mathbf{i}2\pi kj/c} + \sum_{k=-c/2}^{-1} \hat{g}(k)e^{\mathbf{i}2\pi kj/c} \\
&= \sum_{k=0}^{c/2-1} \hat{g}(k)e^{\mathbf{i}2\pi kj/c} + \sum_{k'=c/2}^{c-1} \hat{g}(k'-c)e^{\mathbf{i}2\pi(k'-c)j/c} \\
&= \sum_{k=0}^{c/2-1} \hat{g}(k)e^{\mathbf{i}2\pi kj/c} + \sum_{k'=c/2}^{c-1} \hat{g}(k'-c)e^{\mathbf{i}2\pi k'j/c}.
\end{aligned}$$

Thus, we have finally

$$g(j) = \sum_{k=0}^{c-1} \hat{g}(k)e^{\mathbf{i}2\pi kj/c}, \quad j = 0 \ldots c-1, \tag{3.5}$$

provided that we interpret $\hat{g}(k)$ as meaning $\hat{g}(k-c)$ when $k \geq c/2$.

Equation (3.5) is a set of c equations in the c unknown frequency components $\hat{g}(k)$. Its solution is called the *discrete Fourier transform* and is given by

$$\hat{g}(k) = \frac{1}{c}\sum_{j=0}^{c-1} g(j)e^{-\mathbf{i}2\pi kj/c}, \quad k = 0 \ldots c-1. \tag{3.6}$$

This follows (Exercise 2) from the orthogonality property of the exponentials:

$$\sum_{j=0}^{c-1} e^{\mathbf{i}2\pi(k-k')j/c} = c\delta_{k,k'}, \tag{3.7}$$

where $\delta_{k,k'}$ is the *delta function*,

$$\delta_{k,k'} = \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise.} \end{cases}$$

Equation (3.5) itself is the *discrete inverse Fourier transform*. We write

$$g(j) \Leftrightarrow \hat{g}(k),$$

to signify that $g(j)$ and $\hat{g}(k)$ constitute a *discrete Fourier transform pair*.

Determining the frequency components in Equation (3.6) from the original pixel intensities would appear to involve, in all, c^2 floating point multiplication operations. The *fast Fourier transform* (FFT) exploits the structure of the complex e-functions to reduce this to order $c \log c$, a very considerable saving in computation time for large arrays. For good explanations of the FFT algorithm see, e.g., Press et al. (2002) or Gonzalez and Woods (2017).

The discrete Fourier transform is easily generalized to two dimensions. Let $g(i,j)$, $i = 0 \dots c-1$, $j = 0, r-1$, represent a gray-scale image. Its discrete inverse Fourier transform is

$$g(i,j) = \sum_{k=0}^{c-1} \sum_{\ell=0}^{r-1} \hat{g}(k,\ell) e^{\mathbf{i}2\pi(ik/c + j\ell/r)} \tag{3.8}$$

and the discrete Fourier transform is

$$\hat{g}(k,\ell) = \frac{1}{cr} \sum_{i=0}^{c-1} \sum_{j=0}^{r-1} g(i,j) e^{-\mathbf{i}2\pi(ik/c + j\ell/r)}. \tag{3.9}$$

The frequency coefficients $\hat{g}(k,\ell)$ in Equations (3.8) and (3.9) are complex numbers. In order to represent an image in the frequency domain as a raster, one can calculate its *power spectrum*, which is defined as*

$$P(k,\ell) = |\hat{g}(k,\ell)|^2 = \hat{g}(k,\ell)\hat{g}^*(k,\ell). \tag{3.10}$$

Rather than displaying $P(k,\ell)$ directly, which, according to the usual display convention, would place zero frequency components $k = 0, \ell = 0$ in the upper left-hand corner, use can be made of the *translation property* (Exercise 4) of the Fourier transform:

$$g(i,j) e^{\mathbf{i}2\pi(k_0 i/c + \ell_0 j/r)} \Leftrightarrow \hat{g}(k - k_0, \ell - \ell_0). \tag{3.11}$$

In particular, for $k_0 = c/2$ and $\ell_0 = r/2$, we can write

$$e^{\mathbf{i}2\pi(k_0 i/c + \ell_0 j/r)} = e^{\mathbf{i}\pi(i+j)} = (-1)^{i+j}.$$

*The magnitude $|z|$ of a complex number $z = x + \mathbf{i}y$ is $\sqrt{x^2+y^2} = \sqrt{zz^*}$, where $z^* = x - \mathbf{i}y$ is the complex conjugate of z; see Appendix A.

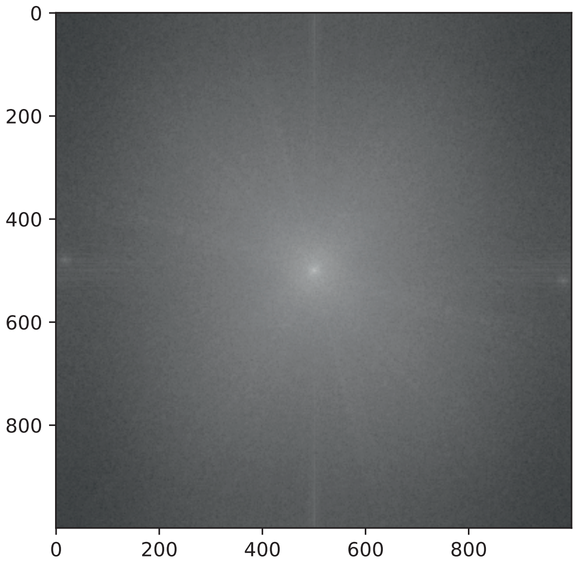


FIGURE 3.1
Logarithm of the power spectrum for the 3N band of the ASTER image of Figure 1.1.

Therefore

$$g(i,j)(-1)^{i+j} \Leftrightarrow \hat{g}(k - c/2, \ell - r/2),$$

so if we multiply an image by $(-1)^{i+j}$ before transforming, zero frequency will be at the center. This is illustrated in the following code:

```
import numpy as np
from numpy import fft
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly
import matplotlib.pyplot as plt

gdal.AllRegister()
infile = 'imagery/AST_20070501.tif'

inDataset = gdal.Open(infile,GA_ReadOnly)
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize

band = inDataset.GetRasterBand(2)
image = band.ReadAsArray(0,0,cols,rows)
```

```
#  arrays of i and j values
a = np.reshape(range(rows*cols),(rows,cols))
i = a % cols
j = a / cols
#  shift Fourier transform to center
image = (-1)**(i+j)*image
#  compute power spectrum and display
image = np.log(abs(fft.fft2(image))**2)
mn = np.amin(image)
mx = np.amax(image)
plt.imshow((image-mn)/(mx-mn), cmap='gray' )
```

The above script performs a fast Fourier transform of an image band using the Python function `numpy.fft()` and displays the logarithm of the power spectrum with zero frequency at the center; see Figure 3.1. We shall return to discrete Fourier transforms in the next chapter when we discuss convolutions and filters.

3.2 The discrete wavelet transform

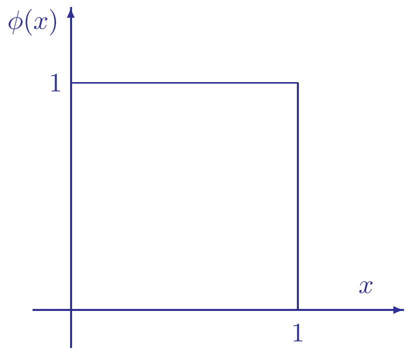


FIGURE 3.2
The Haar scaling function.

Unlike the Fourier transform, which represents an array of pixel intensities in terms of pure frequency functions, the wavelet transform expresses an image array in terms of functions which are restricted both in terms of frequency and spatial extent. In many image processing applications, this turns out to be particularly efficient and useful. The traditional (and most intuitive) way of introducing wavelets is in terms of the Haar scaling function (Strang, 1989), Aboufadel and Schlicker (1999), Gonzalez and Woods (2017), and we will adopt this approach here as well, in particular following the development in Aboufadel and Schlicker (1999).

Fundamental to the definition of wavelet transforms is the concept of an *inner product* of real-valued functions and the associated *inner product space* (Appendix A).

DEFINITION 3.1 *If f and g are two real functions on the set of real numbers $\mathbb{R}$, then their* inner product *is given by*

$$\langle f, g\rangle = \int_{-\infty}^{\infty} f(x)g(x)dx. \tag{3.12}$$

The inner product space $L_2(\mathbb{R})$ *is the collection of all functions $f : \mathbb{R} \mapsto \mathbb{R}$ with the property that*

$$\langle f, f\rangle = \int_{-\infty}^{\infty} f(x)^2 dx \quad \textit{is finite.} \tag{3.13}$$

3.2.1 Haar wavelets

The *Haar scaling function* is the function

$$\phi(x) = \begin{cases} 1 & \text{if } 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.14}$$

shown in Figure 3.2. We shall use it to represent pixel intensities.

The quantities $g(j)$, $j = 0, c-1$, representing a row of pixel intensities, can be thought of as a piecewise constant function of x. This is indicated in Figure 3.3. The abscissa has been normalized to the interval $[0, 1]$, so that j

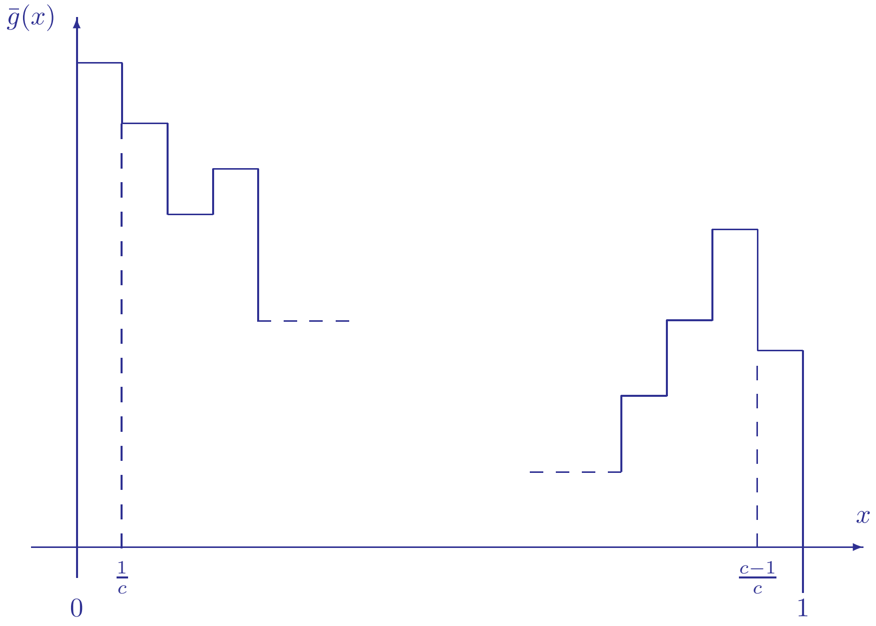


FIGURE 3.3
A row of c pixel intensities on the interval $[0, 1]$ as a piecewise constant function $\bar{g}(x) \in L_2(\mathbb{R})$. In the text, it is assumed that $c = 2^n$ for some integer n.

measures the distance in increments of $1/c$ along the pixel row, with the last pixel occupying the interval $[\frac{c-1}{c}, 1]$. We have called this piecewise constant function $\bar{g}(x)$ to distinguish it from $g(j)$. According to Definition 3.1, it is in $L_2(\mathbb{R})$.

Now let V_n be the collection of *all* piecewise constant functions on the interval $[0, 1]$ that have possible discontinuities at the rational points $j \cdot 2^{-n}$, where j and n are nonnegative integers. If the number of pixels c in Figure 3.3 is a power of two, $c = 2^n$ say, then $\bar{g}(x)$ clearly is a function which belongs to V_n, i.e., $\bar{g}(x) \in V_n$. The (possible) discontinuities occur at

$$x = 1 \cdot 2^{-n}, 2 \cdot 2^{-n} \ldots (c-1) \cdot 2^{-n}.$$

Certainly all members of V_n also belong to the function space $L_2(\mathbb{R})$, so that $V_n \subset L_2(\mathbb{R})$. Any function in V_n confined to the interval $[0, 1]$ in this way can be expressed as a linear combination of the *standard Haar basis functions.* These are scaled and shifted versions of the Haar scaling function of Figure 3.2 and comprise the set

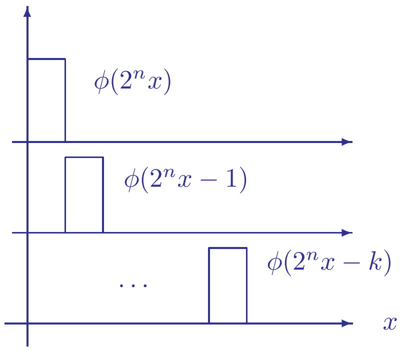


FIGURE 3.4
Basis functions C_n for space V_n.

$$C_n = \{\phi_{n,k}(x) = \phi(2^n x - k) \mid k = 0, 1 \ldots 2^n - 1\}, \tag{3.15}$$

see Figure 3.4.

Note that $\phi_{0,0}(x) = \phi(x)$. The index n corresponds to a compression or change of scale by a factor of 2^{-n}, whereas the index k shifts the basis function across the interval $[0, 1]$. The row of pixels in Figure 3.3 can be expanded in terms of the standard Haar basis trivially as

$$\begin{aligned}\bar{g}(x) &= g(0)\phi_{n,0}(x) + g(1)\phi_{n,1}(x) + \ldots + g(c-1)\phi_{n,c-1}(x) \\ &= \sum_{j=0}^{c-1} g(j)\phi_{n,j}(x).\end{aligned} \tag{3.16}$$

The Haar basis functions are clearly orthogonal:

$$\langle \phi_{n,k}, \phi_{n,k'} \rangle = \int_0^1 \phi_{n,k}(x)\phi_{n,k'}(x)dx = \frac{1}{2^n}\delta_{k,k'}. \tag{3.17}$$

The expansion coefficients $g(j)$ are therefore given formally by

$$g(j) = \frac{\langle \bar{g}, \phi_{n,j} \rangle}{\langle \phi_{n,j}, \phi_{n,j} \rangle} = 2^n \langle \bar{g}, \phi_{n,j} \rangle. \tag{3.18}$$

We will now derive a new and more interesting orthogonal basis for V_n.

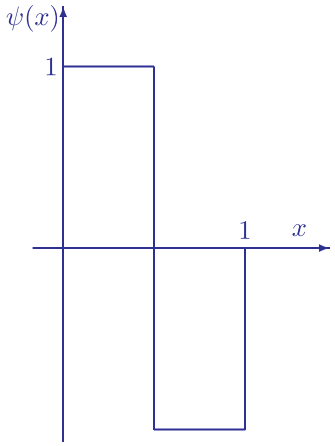


FIGURE 3.5
The Haar mother wavelet.

Consider, first of all, the function spaces V_0 and V_1 with standard Haar bases $\{\phi_{0,0}(x)\}$ and $\{\phi_{1,0}(x),\ \phi_{1,1}(x)\}$, respectively. According to the Orthogonal Decomposition Theorem (Appendix A, Theorem A.4), any function in V_1 can be expressed as a linear combination of the basis for V_0 plus some function in a *residual space* $V_0^\perp$ which is orthogonal to V_0 (i.e., any function in $V_0^\perp$ is orthogonal to any function in V_0). This is denoted formally by writing

$$V_1 = V_0 \oplus V_0^\perp. \tag{3.19}$$

For example, the basis function $\phi_{1,0}(x)$ of V_1 is also in V_1, and so can be written in the form

$$\phi_{1,0}(x) = \frac{\langle \phi_{1,0}, \phi_{0,0}\rangle}{\langle \phi_{0,0}, \phi_{0,0}\rangle}\ \phi_{0,0}(x) + r(x) = \frac{1}{2}\ \phi_{0,0}(x) + r(x).$$

The function $r(x)$ is in the residual space $V_0^\perp$. We see that, in this case,

$$r(x) = \phi_{1,0}(x) - \frac{1}{2}\ \phi_{0,0}(x) = \phi(2x) - \frac{1}{2}\ \phi(x).$$

But we can express $\phi(x)$ as $\phi(x) = \phi(2x) + \phi(2x-1)$ so that

$$r(x) = \phi(2x) - \frac{1}{2}(\phi(2x) + \phi(2x-1)) = \frac{1}{2}(\phi(2x) - \phi(2x-1)) =: \frac{1}{2}\ \psi(x).$$

The function

$$\psi(x) = \phi(2x) - \phi(2x-1) \tag{3.20}$$

is shown in Figure 3.5. It is orthogonal to $\phi(x)$ and is called the *Haar mother wavelet*. Thus, an *alternative basis* for V_1 is

$$B_1 = \{\phi_{0,0}, \psi_{0,0}\},$$

where for consistency we have defined $\psi_{0,0}(x) = \psi(x)$.

This argument can be repeated (Exercise 6) for $V_2 = V_1 \oplus V_1^\perp$ to obtain the basis

$$B_2 = \{\phi_{0,0}, \psi_{0,0}, \psi_{1,0}, \psi_{1,1}\}$$

for V_2, where now $\{\psi_{1,0}, \psi_{1,1}\}$ is an orthogonal basis for $V_1^\perp$ given by

$$\psi_{1,0} = \psi(2x),\ \psi_{1,1} = \psi(2x-1).$$

Indeed, the argument can be continued indefinitely, so in general the *Haar wavelet basis* for V_n is

$$B_n = \{\phi_{0,0}, \psi_{0,0}, \psi_{1,0}, \psi_{1,1} \dots \psi_{n-1,0}, \psi_{n-1,1} \dots \psi_{n-1,2^n-1}\},$$

where $\{\psi_{m,k} = \psi(2^m x - k) \mid k = 0 \dots 2^m - 1\}$ is an orthogonal basis for $V_m^\perp$, and

$$V_n = V_{n-1} \oplus V_{n-1}^\perp = V_0 \oplus V_0^\perp \oplus \dots \oplus V_{n-2}^\perp \oplus V_{n-1}^\perp.$$

In terms of this new basis, the function $\bar{g}(x)$ in Figure 3.3 can now be expressed as

$$\bar{g}(x) = \hat{g}(0)\phi_{0,0}(x) + \hat{g}(1)\psi_{0,0}(x) + \dots + \hat{g}(c-1)\psi_{n-1,c-1}(x), \tag{3.21}$$

where $c = 2^n$. The expansion coefficients $\hat{g}(j)$ are called the *wavelet coefficients*. They are still to be determined.

In the case of the Haar wavelets, their determination turns out to be quite easy because there is a simple correspondence between the basis functions (ϕ, ψ) and the space of 2^n-component vectors (Strang, 1989). Consider for instance $n = 2$. Then, the correspondence is

$$\boldsymbol{\phi}_{0,0} = \begin{pmatrix}1\\1\\1\\1\end{pmatrix},\ \boldsymbol{\phi}_{1,0} = \begin{pmatrix}1\\1\\0\\0\end{pmatrix},\ \boldsymbol{\phi}_{1,1} = \begin{pmatrix}0\\0\\1\\1\end{pmatrix},\ \boldsymbol{\phi}_{2,0} = \begin{pmatrix}1\\0\\0\\0\end{pmatrix},\ \dots$$

and

$$\boldsymbol{\psi}_{0,0} = \begin{pmatrix}1\\1\\-1\\-1\end{pmatrix},\ \boldsymbol{\psi}_{1,0} = \begin{pmatrix}1\\-1\\0\\0\end{pmatrix},\ \boldsymbol{\psi}_{1,1} = \begin{pmatrix}0\\0\\1\\-1\end{pmatrix}.$$

Thus, the orthogonal basis B_2 may be represented equivalently by the mutually orthogonal vectors

$$B_2 = \left\{ \begin{pmatrix}1\\1\\1\\1\end{pmatrix}, \begin{pmatrix}1\\1\\-1\\-1\end{pmatrix}, \begin{pmatrix}1\\-1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\-1\end{pmatrix} \right\}.$$

This gives us a more convenient representation of the expansion in Equation (3.21), namely

$$\bar{\boldsymbol{g}} = \boldsymbol{B}_n \hat{\boldsymbol{g}}, \tag{3.22}$$

where $\bar{\boldsymbol{g}} = \big(g(0) \dots g(c-1)\big)^\top$ is a column vector of the original pixel intensities, $\hat{\boldsymbol{g}} = \big(\hat{g}(0) \dots \hat{g}(c-1)\big)^\top$ is a column vector of the wavelet coefficients, and $\boldsymbol{B}_n$ is a transformation matrix whose columns are the basis vectors of B_n. The wavelet coefficients for the pixel vector are then given by inverting the representation:

$$\hat{\boldsymbol{g}} = \boldsymbol{B}_n^{-1}\bar{\boldsymbol{g}}. \tag{3.23}$$

A full gray-scale image is transformed by first applying Equation (3.23) to its columns and then to its rows. Wavelet coefficients for gray-scale images thus obtained tend to have "simple statistics" (Gonzalez and Woods, 2017), e.g., they might be approximately Gaussian with zero mean.

3.2.2 Image compression

The fact that many of the wavelet coefficients are close to zero makes the wavelet transformation useful for image compression. This is illustrated in the following script. First we define a function to return the Haar basis functions:

```
# The Haar mother wavelet
def psi_m(x):
    if x<0:return 0.0
    elif x<=0.5:return 1.0
    elif x<=1.0:return -1.0
    else:return 0.0
# The Haar basis functions
def psi(m,k,n):
    c = 2**n
    result = np.zeros(c)
    x = np.linspace(0,1,num=c)
    for i in range(c):
        result[i] = psi_m((2**m)*x[i]-k)
    return result
```

and use it to generate the basis for $n = 8$:

```
# Generate wavelet basis B_8
n = 8
B = np.ones((2**n,2**n))
i = 1
for m in range(n):
    for k in range(2**m):
        B[:,i] = psi(m,k,n)
        i += 1
B = np.mat(B)
```

Next we perform the wavelet transformation of a subset of the image used in the previous subsection:

```
# 256x256 subset
G = np.mat(image[200:456,200:456])

# Wavelet transformation
```

```
Gw = np.mat(np.zeros((256,256)))
# Filter the columns
for j in range(256):
    Gw[:,j] = B.I*G[:,j]
# Filter the rows
for i in range(256):
    Gw[i,:] = (B.I*Gw[i,:].T).T
# Histogram of wavelet coefficients
Gw = np.array(Gw).ravel()
p = plt.hist(Gw,bins=30,range=(-10,10))
```

The resulting histogram is shown in Figure 3.6, where it is apparent that the wavelet coefficients are tightly distributed about zero and that, in this case, most have absolute magnitudes less than about 2.5. The script is slow due to the many floating point operations involved in the transformations. This will be remedied in Chapter 4 when we treat filters and the fast wavelet transform.

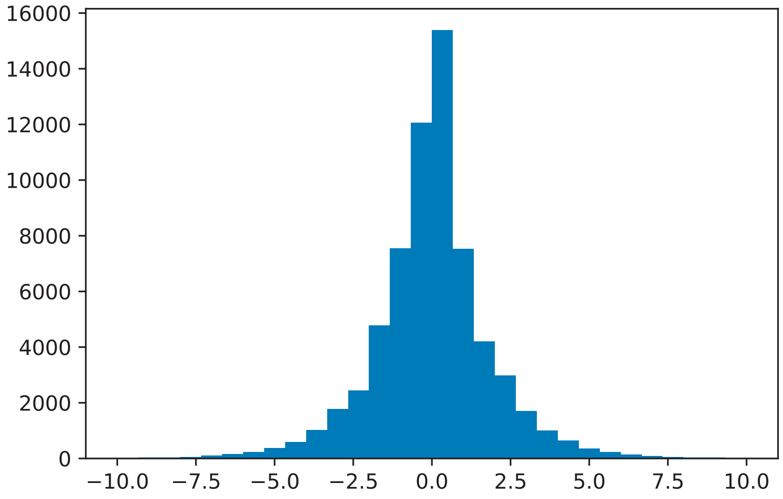


FIGURE 3.6
Histogram of the Haar wavelet coefficients.

Finally, we set all wavelet coefficients smaller than 2.0 equal to zero and invert the transformation:

```
# Truncate and reshape
Gw = np.reshape(np.where(np.abs(Gw)<2,0,Gw),(256,256))
```

```
# Invert the transformation
Gw = np.mat(Gw)
Gc = np.mat(np.zeros((256,256)))
for i in range(256):
    Gc[i,:] = (B*Gw[i,:].T).T
for j in range(256):
    Gc[:,j] = B*Gc[:,j]
f, ax = plt.subplots(1,2,figsize=(16,8))
ax[0].imshow(np.array(G)/255,cmap='gray')
ax[1].imshow(np.array(Gc)/255,cmap='gray')
```

The original and recovered images are shown in Figure 3.7.

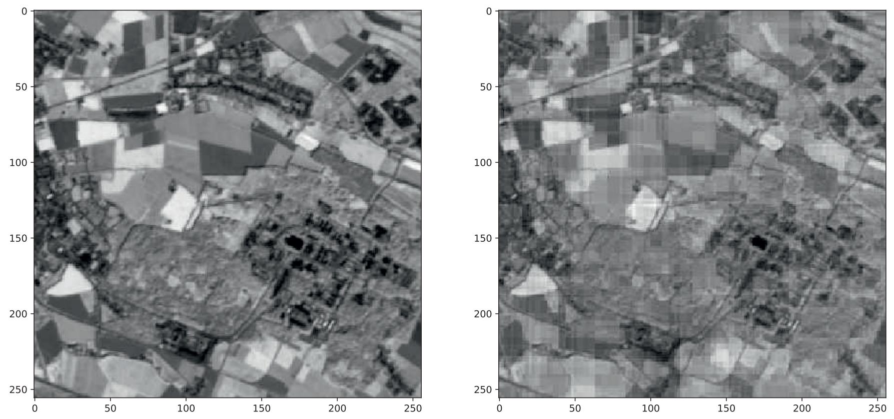


FIGURE 3.7
Left: a 256×256 spatial subset of the 3N band of the Jülich ASTER image of Figure 1.1. Right: the same subset after transformation to the Haar wavelet basis B_8, compression, and restoration with the inverse transformation.

By converting the original image and its wavelet representation to sparse matrix form we can see the amount of compression achievable:

```
from scipy import sparse

sG = sparse.csr_matrix(G)
sGw = sparse.csr_matrix(Gw)
print sG.data.nbytes
print sGw.data.nbytes
262144
117088
```

so about a factor of 2.3, at the expense of the considerable degradation visible in Figure 3.7.

3.2.3 Multiresolution analysis

So far we have represented only functions on the interval $[0, 1]$ with the standard basis $\phi_{n,k}(x) = \phi(2^n x - k)$, $k = 1 \ldots 2^n - 1$. We can extend this to functions defined on all real numbers in a straightforward way, still restricting ourselves, however, to functions with *compact support*. These are zero everywhere outside a closed, bounded interval. Thus

$$\{\phi(x - k) \mid k \in \mathbb{Z}\},$$

where $\mathbb{Z}$ is the set of *all* integers, is a basis for the space V_0 of all piecewise constant functions with compact support having possible breaks at integer values. Note that $\phi(x - k) = \phi_{0,k}$ is an *orthonormal* basis for V_0, that is,

$$\langle \phi(x - k), \phi(x - k') \rangle = \delta_{k,k'},$$

whereas $\phi(2^n x - k) = \phi_{n,k}$ with $n > 0$ is only an orthogonal basis for V_n, since the inner products for equal k are not unity. Quite generally then, an orthogonal basis for the set V_n of piecewise constant functions with possible breaks at $j \cdot 2^{-n}$ and compact support is

$$\{\phi(2^n x - k) \mid k \in \mathbb{Z}\}. \tag{3.24}$$

One can even allow $n < 0$. For example, $n = -1$ means that the possible breaks are at even integer values.

We can think of the collection of nested subspaces of piecewise constant functions as being *generated* by the Haar scaling function ϕ. Such a collection is an example of a *multi resolution analysis* (MRA). There are many other possible scaling functions that define or generate an MRA. Although the subspaces will no longer consist of simple piecewise constant functions, nevertheless, based on our experience with the Haar wavelets, we can appreciate the following definition (Aboufadel and Schlicker, 1999):

DEFINITION 3.2 *An MRA is a collection of nested subspaces*

$$\ldots \subseteq V_{-1} \subseteq V_0 \subseteq V_1 \subseteq V_2 \subseteq \ldots \subseteq L_2(\mathbb{R}),$$

with the following properties:

1. *For any function $f \in L_2(\mathbb{R})$, there exists a series of functions, one in each V_n, which converges to f.*

2. *The only function common to all V_n is $f(x) = 0$.*

3. *The function $f(x) \in V_n$ if and only if $f(2^{-n}x) \in V_0$.*

4. *The scaling function ϕ is an orthonormal basis for the function space V_0, i.e., $\langle \phi(x - k), \phi(x - k') \rangle = \delta_{kk'}$.*

Clearly, property 1 is met for the Haar MRA, since any function in $L_2(\mathbb{R})$ can be approximated to arbitrary accuracy with successively finer piecewise constant functions. Being common to all V_n means being piecewise constant on all intervals. The only function in $L_2(\mathbb{R})$ with this property and compact support is $f(x) = 0$, so property 2 is also satisfied for the Haar MRA. If $f(x) \in V_1$ then it is piecewise constant on intervals of length 1/2. Therefore, the function $f(2^{-1}x)$ is piecewise constant on intervals of length 1, that is, $f(2^{-1}x) \in V_0$, etc., and so property 3 is satisfied as well. Finally, property 4 also holds for the Haar scaling function.

3.2.3.1 The dilation equation and refinement coefficients

In the following, we will think of $\phi(x)$ as any scaling function which generates an MRA in the sense of Definition 3.2. Since $\{\phi(x-k) \mid k \in \mathbb{Z}\}$ is an orthonormal basis for V_0, it follows that $\{\phi(2x-k) \mid k \in \mathbb{Z}\}$ is an orthogonal basis for V_1. That is, let $f(x) \in V_1$. Then by property 3, $f(x/2) \in V_0$, hence

$$f(x/2) = \sum_k a_k \phi(x-k),$$

which implies that

$$f(x) = \sum_k a_k \phi(2x-k).$$

In particular, since $\phi(x) \in V_0 \subset V_1$, we have the *dilation equation*

$$\phi(x) = \sum_k c_k \phi(2x-k). \tag{3.25}$$

The constants c_k are called the *refinement coefficients*. For example, the dilation equation for the Haar scaling function is

$$\phi(x) = \phi(2x) + \phi(2x-1),$$

so that the refinement coefficients are $c_0 = c_1 = 1$, $c_k = 0$ otherwise. Note that $c_0^2 + c_1^2 = 2$. This is a general property of the refinement coefficients:

$$1 = \langle \phi(x), \phi(x) \rangle = \langle \sum_k c_k \phi(2x-k), \sum_{k'} c_{k'} \phi(2x-k') \rangle = \frac{1}{2} \sum_k c_k^2$$

and therefore,

$$\sum_{k=-\infty}^{\infty} c_k^2 = 2, \tag{3.26}$$

which is also called *Parseval's formula*. In a similar way, one can show (Exercise 7)

$$\sum_{k=-\infty}^{\infty} c_k c_{k-2j} = 0 \quad \text{for all} \quad j \neq 0. \tag{3.27}$$

3.2.3.2 The cascade algorithm

Some of the scaling functions which generate an MRA cannot be expressed as simple, analytical functions. Nevertheless, we can work with an MRA even when there is no simple representation for the scaling function which generates it. For instance, once we have the refinement coefficients for a scaling function, it can be approximated to any desired degree of accuracy using the dilation equation. The idea is to iterate the refinement equation with a so-called *cascade algorithm* until it converges to a sequence of points which approximates $\phi(x)$.

The following recursive scheme can be used to estimate a scaling function with up to five non-zero refinement coefficients $c_0, c_1 \dots c_4$:

$$\begin{aligned} f_0(x) &= \delta_{x,0} \\ f_i(x) &= c_0 f_{i-1}(2x) + c_1 f_{i-1}(2x-1) + c_2 f_{i-1}(2x-2) + c_3 f_{i-1}(2x-3) \\ &\quad + c_4 f_{i-1}(2x-4). \end{aligned}$$

In this scheme, x takes on values $j/2^n$, where j, n are any integers. The first definition is the termination condition for the recursion and approximates the scaling function to zeroth order as the delta function

$$\delta_{x,0} = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The second relation defines the ith approximation to the scaling function in terms of the $(i-1)$th approximation using the dilation equation. We can calculate the set of values

$$\phi \approx f_n\left(\frac{j}{2^n}\right) \quad \text{for } j = 0 \dots 4 \cdot 2^n$$

for some $n > 1$ as a point-wise approximation of ϕ on the interval $[0, 4]$. The cascade algorithm is in this case as follows:

```
def F(x,i,c):
    if i==0:
        if x==0:
            return 1.0
        else:
            return 0.0
    else:
        return c[0]*F(2*x,i-1,c)+c[1]*F(2*x-1,i-1,c) \
              +c[2]*F(2*x-2,i-1,c)+c[3]*F(2*x-3,i-1,c) \
              +c[4]*F(2*x-4,i-1,c)

# Haar refinement coefficients
c = np.zeros(5)
c[0] = 1.0; c[1] = 1.0
```

```
# fourth order approximation
n = 4
x = range(4*2**n)
FF = np.zeros(4*2**n)
for i in range(4*2**n):
    FF[i] = F(x[i]/2**n,n,c)

plt.plot(x,FF)
plt.ylim(-1,2)
```

The result using the Haar refinement coefficients is shown in Figure 3.8 and is seen to be an approximation of Figure 3.2.

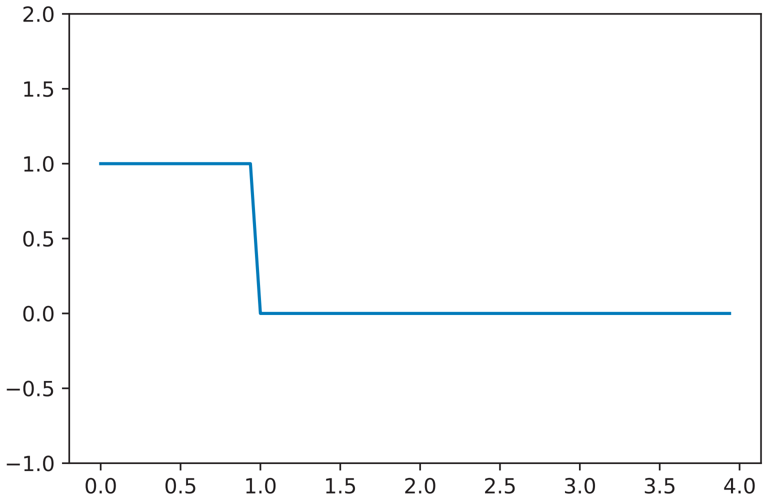


FIGURE 3.8
Approximation to the Haar scaling function with the cascade algorithm and $n = 4$ iterations.

3.2.3.3 The mother wavelet

For a general MRA, we also require a generalization of Equation (3.20), which relates the Haar mother wavelet to the scaling function. Let some MRA have a scaling function ϕ defined by the dilation Equation (3.25). Since

$$\langle \phi(2x-k), \phi(2x-k) \rangle = \frac{1}{2} \cdot \langle \phi(x), \phi(x) \rangle = \frac{1}{2} \ ,$$

the functions $\sqrt{2}\phi(2x-k)$ are both normalized and orthogonal. We can write Equation (3.25) in the form

$$\phi(x) = \sum_k h_k \sqrt{2}\phi(2x-k), \tag{3.28}$$

where

$$h_k = \frac{c_k}{\sqrt{2}} \; .$$

It follows from Equation (3.26) that

$$\sum_k h_k^2 = 1. \tag{3.29}$$

Now we assume, in analogy to Equation (3.28), that the mother wavelet ψ can also be expressed in terms of the scaling function as*

$$\psi(x) = \sum_k g_k \sqrt{2}\phi(2x-k). \tag{3.30}$$

Since $\phi \in V_0$ and $\psi \in V_0^{\perp}$, we have

$$\langle \phi, \psi \rangle = \sum_k h_k g_k = 0. \tag{3.31}$$

Similarly, with some simple index manipulations,

$$\langle \psi(x-k), \psi(x-m) \rangle = \sum_i g_i g_{i-2(k-m)} = \delta_{k,m}. \tag{3.32}$$

A set of coefficients that satisfies Equations (3.31) and (3.32) is given by

$$g_k = (-1)^k h_{1-k}. \tag{3.33}$$

So we obtain, finally, the general relationship between the mother wavelet and the scaling function:

$$\psi(x) = \sum_k (-1)^k h_{1-k}\sqrt{2}\phi(2x-k) = \sum_k (-1)^k c_{1-k}\phi(2x-k). \tag{3.34}$$

3.2.3.4 The Daubechies D4 scaling function

A family of MRAs which is very useful in digital image analysis is generated by the Daubechies scaling functions and their associated wavelets (Daubechies, 1988). The Daubechies D4 scaling function, for example, can be derived by placing the following two additional requirements on an MRA (Aboufadel and Schlicker, 1999):

*The coefficients g_k should not be confused with pixel intensities.

1. *Compact support:* The scaling function $\phi(x)$ is required to be zero outside the interval $0 < x < 3$. This means that the refinement coefficients c_k vanish for $k < 0$ and for $k > 3$. To see this, note that

$$c_{-3} = 2\langle \phi(x), \phi(2x+3)\rangle = \int_0^3 \phi(x)\phi(2x+3)dx = 0$$

and similarly for $k \le -4$ and for $k \ge 6$. Therefore, from the dilation equation,

$$\phi(-1/2) = 0 = c_{-2}\phi(-1+2) + c_{-1}\phi(-1+1) + \dots \quad \text{implying } c_{-2} = 0$$

and similarly for $k = -1, 4, 5$. Thus, from Equation (3.26), we can conclude that

$$c_0^2 + c_1^2 + c_2^2 + c_3^2 = 2 \tag{3.35}$$

and from Equation(3.27) with $j = 1$, that

$$c_0c_2 + c_1c_3 = 0. \tag{3.36}$$

2. *Regularity:* All constant and linear polynomials can be written as a linear combination of the basis $\{\phi(x-k) \mid k \in \mathbb{Z}\}$ for V_0. This implies that there is no residual in the orthogonal decomposition of $f(x) = 1$ and $f(x) = x$ onto the basis, that is,

$$\int_{-\infty}^{\infty} 1 \cdot \psi(x)dx = \int_{-\infty}^{\infty} x \cdot \psi(x)dt = 0. \tag{3.37}$$

With Equation (3.34), the mother wavelet is

$$\begin{aligned}\psi(x) &= -c_0\phi(2x-1) + c_1\phi(2x) - c_2\phi(2x+1) + c_3\phi(2x+2)\\ &= \sum_{k=0}^{3}(-1)^{k+1}c_k\phi(2x-1+k).\end{aligned} \tag{3.38}$$

The first requirement in Equation (3.37) gives immediately

$$-c_0 + c_1 - c_2 + c_3 = 0. \tag{3.39}$$

From the second requirement, we have

$$\begin{aligned}0 = \int_{-\infty}^{\infty} x\psi(x)dx &= \sum_{k=0}^{3}(-1)^{k+1}c_k \int_{-\infty}^{\infty} x\phi(2x-1+k)dx\\ &= \sum_{k=0}^{3}(-1)^{k+1}c_k \int_{-\infty}^{\infty} \frac{u+1-k}{4}\phi(u)du\\ &= \frac{0}{4}\cdot\int_{-\infty}^{\infty} u\phi(u)du + \frac{-c_0+c_2-2c_3}{4}\int_{-\infty}^{\infty}\phi(u)du,\end{aligned}$$

using Equation (3.39). Thus

$$-c_0 + c_2 - 2c_3 = 0. \tag{3.40}$$

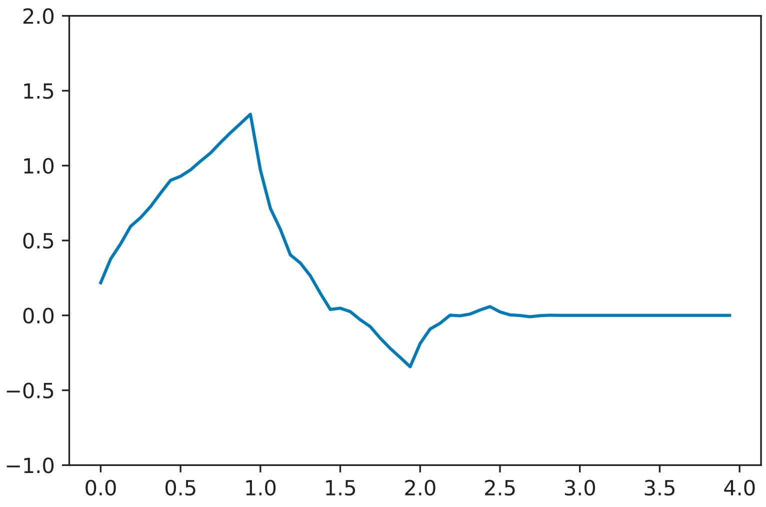


FIGURE 3.9
Approximation to the Daubechies D4 scaling function with the cascade algorithm after $n = 4$ iterations.

Equations (3.35), (3.36), (3.39), and (3.40) comprise a system of four (nonlinear) equations in four unknowns. A solution is given by

$$c_0 = \frac{1+\sqrt{3}}{4}, \quad c_1 = \frac{3+\sqrt{3}}{4}, \quad c_2 = \frac{3-\sqrt{3}}{4}, \quad c_3 = \frac{1-\sqrt{3}}{4},$$

which are known as the D4 refinement coefficients. Figure 3.9 shows the corresponding scaling function, determined with the cascade algorithm:

```
# Daubechies D4 refinement coefficients
c = np.zeros(5)
c[0] = (1+np.sqrt(3))/4; c[1] = (3+np.sqrt(3))/4
c[2] = (3-np.sqrt(3))/4; c[3] = (1-np.sqrt(3))/4
c[4] = 0.0

for i in range(4*2**n):
    FF[i] = F(x[i],n,c)

plt.plot(x,FF)
plt.ylim(-1,2)
```

The D4 scaling function and the subspaces that it generates are thus anything but simple. The scaling function is continuous but not everywhere differentiable and also self-similar (the tail is an exact but re-scaled copy of the entire function). Nevertheless, the D4 wavelets provide a much more useful representation of digital images than the Haar wavelets. We will return to them in Chapter 4 when we treat the fast wavelet transform and examine pyramid algorithms for image processing.

3.3 Principal components

The principal components transformation, also called *principal components analysis* (PCA), generates linear combinations of multispectral pixel intensities which are mutually uncorrelated and which have maximum variance. Specifically, consider a multispectral image represented by the random vector $\boldsymbol{G}$ (for vector of gray-scale values) and assume that $\langle \boldsymbol{G} \rangle = \boldsymbol{0}$, so that the covariance matrix is given by $\boldsymbol{\Sigma} = \langle \boldsymbol{G}\boldsymbol{G}^\top \rangle$. Let us seek a linear combination $Y = \boldsymbol{w}^\top \boldsymbol{G}$ whose variance $\boldsymbol{w}^\top \boldsymbol{\Sigma} \boldsymbol{w}$ is maximum. This quantity can trivially be made as large as we like by choosing $\boldsymbol{w}$ sufficiently large, so that the maximization only makes sense if we restrict $\boldsymbol{w}$ in some way. A convenient constraint is $\boldsymbol{w}^\top \boldsymbol{w} = 1$. According to the discussion in Section 1.6, we can solve this problem by maximizing the unconstrained Lagrange function

$$L = \boldsymbol{w}^\top \boldsymbol{\Sigma} \boldsymbol{w} - \lambda(\boldsymbol{w}^\top \boldsymbol{w} - 1).$$

This leads directly, see Equation (1.65), to the eigenvalue problem

$$\boldsymbol{\Sigma} \boldsymbol{w} = \lambda \boldsymbol{w}. \tag{3.41}$$

Denote the orthogonal and normalized eigenvectors of $\boldsymbol{\Sigma}$ obtained by solving the above problem by $\boldsymbol{w}_1 \dots \boldsymbol{w}_N$, sorted according to decreasing eigenvalue $\lambda_1 \ge \dots \ge \lambda_N$. These eigenvectors are the *principal axes* and the corresponding linear combinations $\boldsymbol{w}_i^\top \boldsymbol{G}$ are projections along the principal axes, called the *principal components* of $\boldsymbol{G}$. The individual principal components

$$Y_1 = \boldsymbol{w}_1^\top \boldsymbol{G},\ Y_2 = \boldsymbol{w}_2^\top \boldsymbol{G},\ \dots,\ Y_N = \boldsymbol{w}_N^\top \boldsymbol{G}$$

can be expressed more compactly as a random vector $\boldsymbol{Y}$ by writing

$$\boldsymbol{Y} = \boldsymbol{W}^\top \boldsymbol{G}, \tag{3.42}$$

where $\boldsymbol{W}$ is the matrix whose columns comprise the eigenvectors, that is,

$$\boldsymbol{W} = (\boldsymbol{w}_1 \dots \boldsymbol{w}_N).$$

Since the eigenvectors are orthogonal and normalized, $\boldsymbol{W}$ is an orthonormal matrix:

$$\boldsymbol{W}^\top \boldsymbol{W} = \boldsymbol{I}.$$

Listing 3.1: Principal components analysis with the GEE Python API.

```python
#!/usr/bin/env python3
#  Name: eepca.py
import ee

def pca(image,scale=30,nbands=6,maxPixels=1e9):
    #  center the image
    bandNames=image.bandNames()
    meanDict=image.reduceRegion(ee.Reducer.mean(),
                  scale=scale,maxPixels=maxPixels)
    means=ee.Image.constant(meanDict.values(bandNames))
    centered=image.subtract(means)
    #  principal components analysis
    pcNames = ['pc'+str(i+1) for i in range(nbands)]
    centered=centered.toArray()
    covar=centered.reduceRegion(
            ee.Reducer.centeredCovariance(),
            scale=scale,maxPixels=maxPixels)
    covarArray=ee.Array(covar.get('array'))
    eigens=covarArray.eigen()
    # get eigenvalues from 1st column
    lambdas=eigens.slice(1, 0, 1)
    # get eigenvectors from rest of array
    eivs=eigens.slice(1, 1)
    centered=centered.toArray(1)
    pcs=ee.Image(eivs).matrixMultiply(centered) \
                        .arrayProject([0]) \
                        .arrayFlatten([pcNames])
    return (pcs,lambdas)

if __name__ == '__main__':
    pass
```

If the covariance matrix of the principal components vector $\boldsymbol{Y}$ is called $\boldsymbol{\Sigma}'$, then we have

$$\begin{aligned}\boldsymbol{\Sigma}' &= \langle \boldsymbol{Y}\boldsymbol{Y}^\top\rangle = \langle \boldsymbol{W}^\top\boldsymbol{G}\boldsymbol{G}^\top\boldsymbol{W}\rangle \\ &= \boldsymbol{W}^\top\boldsymbol{\Sigma}\boldsymbol{W} = \begin{pmatrix}\lambda_1 & 0 & \cdots & 0\\ 0 & \lambda_2 & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & \lambda_N\end{pmatrix} =: \boldsymbol{\Lambda}.\end{aligned} \tag{3.43}$$

The eigenvalues are thus seen to be the variances of the principal components, and all of the covariances are zero. The first principal component Y_1 has maximum variance $\mathrm{var}(Y_1) = \lambda_1$, the second principal component Y_2 has maximum variance $\mathrm{var}(Y_2) = \lambda_2$ subject to the condition that it is uncorrelated with Y_1, and so on. The fraction of the total variance in the original multispectral image which is accounted for by the first i principal components is

$$\frac{\lambda_1 + \ldots + \lambda_i}{\lambda_1 + \ldots + \lambda_i + \ldots + \lambda_N}.$$

3.3.1 Principal components on the GEE

The Google Earth Engine API can be programmed to perform principal components analysis on the Earth Engine servers. Listing 3.1 is a port to the Python API of the JavaScript code for PCA given in the GEE documentation. The script is included in the `auxil` package in the accompanying software and is intended to be imported as a Python module. It illustrates several aspects of scripting with the Python API and is worth looking at in detail.

In lines 8 and 9 the `reduceRegion()` method of an `Image` object is used together with the `Reducer.mean()` reducer to return a dictionary of mean values of the image bands. The dictionary keys are the image `bandNames`. In line 10 a constant image of band means is constructed from the dictionary values and then in line 11 subtracted from the original image to produce a centered image. In line 14, the centered image is converted to an array form which is analogous to what we have called a data matrix. Then the `reduceRegion()` method is invoked again (lines 15–17), this time with the `Reducer.centeredCovariance()` reducer to return the covariance matrix in a dictionary. The actual matrix is accessed with the dictionary key `array` in line 18. The covariance matrix is diagonalized (line 19) and the eigenvalues and eigenvectors extracted in lines 20–23 (see the GEE documentation for `ee.Array.slice()`). Finally (lines 24–27), the principal components are determined as the projection of the centered image array along the eigenvectors.

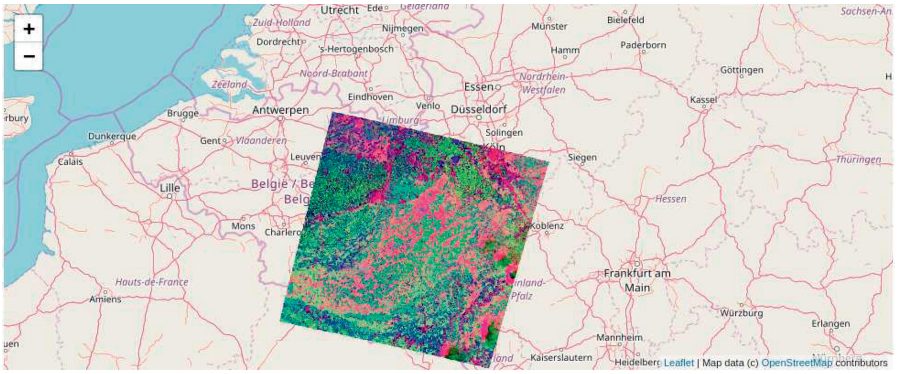


FIGURE 3.10
RGB composite of the first three principal components of a LANDSAT 7 ETM+ image calculated with the script in Listing 3.1.

It is important to realize that the function `pca()` in Listing 3.1 doesn't actually calculate anything. It returns, in the variables `pcs` and `lambdas`, *JavaScript Object Notation* (JSON) structures which merely describe the necessary calculations to be performed on the GEE servers. Only when the user wishes to see the result, e.g., display the principal components or print out the eigenvalues, are the JSON structures sent to the servers for execution. For example, the following Jupyter notebook code determines the principal components of the non-thermal bands of a LANDSAT 7 ETM+ full scene acquired on June 26, 2001 over an area southwest of Cologne, Germany. The script also illustrates the use of the `ipyleaflet` package to allow displaying interactive "slippy maps" in Jupyter notebook output cells:

```python
import ee
from ipyleaflet import (Map, DrawControl, TileLayer)
from auxil import eepca
ee.Initialize()

def GetTileLayerUrl(image):
    map_id = ee.Image(image).getMapId()
    return map_id["tile_fetcher"].url_format

# get the image
im = ee.Image(
  'LANDSAT/LE07/C02/T1_RT_TOA/LE07_197025_20010626') \
         .select('B1', 'B2', 'B3', 'B4', 'B5','B7')

# perform principal components analysis
pcs, lambdas = eepca.pca(im)

# display default base map and overlay the PCA image
m = Map(center=[50.7, 6.4], zoom=7)
m.add_layer(TileLayer(url=GetTileLayerUrl(
        pcs.select('pc1', 'pc2', 'pc3') \
       .visualize(min=-0.1, max=0.1, opacity=1.0)
    )
))

m
```

It is the very last command `m`, requesting display of the map and the overlay in the Jupyter notebook, that actually triggers the server-side calculation. The output cell is shown in Figure 3.10. In order to work with the principal components image, the user has the choice of exporting it to his or her GEE assets folder, cloud storage bucket or Google Drive for eventual local processing. For instance, in the last case:

```python
gdexporttask = ee.batch.Export.image.toDrive(pcs,
               description='driveExportTask',
               folder='EarthEngineImages',
               fileNamePrefix='ASTER_PCS',
                      scale=30,
                      maxPixels=1e9)
   gdexporttask.start()
```

3.3.2 Image compression and reconstruction

If the original multispectral channels are highly correlated, as is often the case, then the first few principal components will usually account for a very high percentage of the total variance in the image. For example, a color composite of the first three principal components of a LANDSAT 7 ETM+ scene displays essentially all of the information contained in the six non-thermal spectral components in one single image. The principal components transformation is therefore often used for dimensionality reduction of imagery prior, for instance, to land cover classification (Chapters 6, 7, and 8).

Alternatively, one can think of the first few principal components as being the main contributing factors to the observed image and then reconstruct the image from those factors. With Equation (3.42), we can recover the original image losslessly by inverting,

$$\boldsymbol{G} = (\boldsymbol{W}^\top)^{-1}\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{Y}.$$

Suppose that the first r principal components account for (explain) most of the variance in the data and let $\boldsymbol{W} = (\boldsymbol{W}_r, \boldsymbol{W}_{r-})$, where

$$\boldsymbol{W}_r = (\boldsymbol{w}_1, \dots \boldsymbol{w}_r), \quad \boldsymbol{W}_{r-} = (\boldsymbol{w}_{r+1}, \dots \boldsymbol{w}_N),$$

and similarly

$$\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{Y}_r \\ \boldsymbol{Y}_{r-} \end{pmatrix}.$$

Now reconstruct $\boldsymbol{G}$ from $\boldsymbol{Y}_r = (Y_1 \dots Y_r)^\top$ and $\boldsymbol{W}_r$,

$$\boldsymbol{G} \approx \boldsymbol{G}_r = \boldsymbol{W}_r\boldsymbol{Y}_r,$$

so we can write

$$\boldsymbol{G} = \boldsymbol{W}_r\boldsymbol{Y}_r + \boldsymbol{\epsilon}, \tag{3.44}$$

where the *reconstruction error* $\boldsymbol{\epsilon}$ is

$$\boldsymbol{\epsilon} = \boldsymbol{G} - \boldsymbol{G}_r = \boldsymbol{G} - \boldsymbol{W}_r\boldsymbol{W}_r^\top\boldsymbol{G} = (\boldsymbol{I} - \boldsymbol{W}_r\boldsymbol{W}_r^\top)\boldsymbol{G} = \boldsymbol{W}_{r-}\boldsymbol{W}_{r-}^\top\boldsymbol{G}.$$

The covariance matrix for the error term is given by

$$\begin{aligned} \langle\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top\rangle &= \langle\boldsymbol{W}_{r-}\boldsymbol{W}_{r-}^\top\boldsymbol{G}\boldsymbol{G}^\top\boldsymbol{W}_{r-}\boldsymbol{W}_{r-}^\top\rangle \\ &= \langle\boldsymbol{W}_{r-}\boldsymbol{Y}_{r-}\boldsymbol{Y}_{r-}^\top\boldsymbol{W}_{r-}\rangle \\ &= \boldsymbol{W}_{r-}\boldsymbol{\Lambda}_{r-}\boldsymbol{W}_{r-}^\top, \end{aligned} \tag{3.45}$$

Listing 3.2: Image reconstruction from principal components.

```python
#!/usr/bin/env python3
#Name:  ex3_1.py
import numpy as np
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly

def main():
    gdal.AllRegister()
    infile = 'imagery/AST_20050911'
    if infile:
        inDataset = gdal.Open(infile,GA_ReadOnly)
        cols = inDataset.RasterXSize
        rows = inDataset.RasterYSize
        bands = inDataset.RasterCount
    else:
        return
    #  transposed data matrix
    m = rows*cols
    G = np.zeros((bands,m))
    for b in range(bands):
        band = inDataset.GetRasterBand(b+1)
        tmp = band.ReadAsArray(0,0,cols,rows)\
                              .astype(float).ravel()
        G[b,:] = tmp - np.mean(tmp)
    G = np.mat(G)
    #  covariance matrix
    S = G*G.T/(m-1)
    #  diagonalize and sort eigenvectors
    lamda,W = np.linalg.eigh(S)
    idx = np.argsort(lamda)[::-1]
    lamda = lamda[idx]
    W = W[:,idx]
    #  get principal components and reconstruct
    r = 2
    Y = W.T*G
    G_r = W[:,:r]*Y[:r,:]
    #  reconstruction error covariance matrix
    print(((G-G_r)*(G-G_r).T/(m-1))[:3,:3])
    #  Equation (3.45)
    print((W[:,r:]*np.diag(lamda[r:])*W[:,r:].T)[:3,:3])
    inDataset = None

if __name__ == '__main__':
    main()
```

where

$$\mathbf{\Lambda}_{r-} = \begin{pmatrix} \lambda_{r+1} & 0 & \cdots & 0 \\ 0 & \lambda_{r+2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{pmatrix}.$$

Thus, the smaller the eigenvalues (variances) of the disregarded principal components are, the smaller is the reconstruction error. Obviously the dataset $\{\boldsymbol{W}_r, \boldsymbol{Y}_r\}$ can be considerably smaller than the original image $\boldsymbol{G}$. Equation (3.44) is in the form of a *factor analysis model*; see Mardia et al. (1979). In this case the *factors* are the first r principal components. The elements of $\boldsymbol{W}_r$ are called the *factor loadings*.

The Python script in Listing 3.2 illustrates the reconstruction process for $r = 2$ with an ASTER image over Isfahan in Iran (filename `AST_20050911`, non-thermal bands only, included in ENVI format in the `imagery` folder) without actually saving the reconstructed image; but see Exercise 11. For clarity, a transposed data matrix is used so that the code corresponds closely with the above equations. In line 38 part of the reconstruction error covariance matrix is printed. It is identical to Equation (3.45), printed for comparison in line 40. Running the script from the Jupyter notebook:

```
run scripts/ex3_1

[[ 48.276122    25.10956254 -33.83129098]
 [ 25.10956254  42.98112067 -33.23172804]
 [-33.83129098 -33.23172804  37.61442228]]
[[ 48.276122    25.10956254 -33.83129098]
 [ 25.10956254  42.98112067 -33.23172804]
 [-33.83129098 -33.23172804  37.61442228]]
<Figure size 640x480 with 0 Axes>
```

We can further illustrate (using the Python script `pca.py` included in the `scripts` subdirectory; see Appendix C) to perform the transformation:

```
%run scripts/pca -r 2 -n imagery/AST_20050911

------------ PCA ---------------
Wed Aug 14 18:48:03 2024
Input imagery/AST_20050911
Eigenvalues: [6.11586715e+03 1.77518619e+02
1.32004293e+02 3.14826406e+01  2.11086622e+01
1.57776347e+01 1.32005399e+01 3.77242660e+00
 2.65003835e+00]
PCs written to: imagery/AST_20050911_pca
Reconstruction written to: imagery/AST_20050911_recon
elapsed time: 0.36631035804748535
```

The flag `-n` suppresses graphical output and the flag `-r 2` instructs the script to reconstruct the original image from the first two principal components

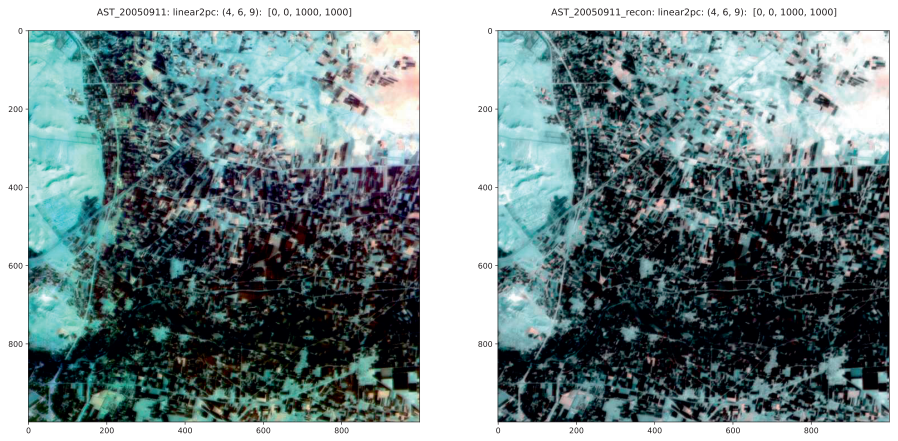

AST_20050911: linear2pc: (4, 6, 9): [0, 0, 1000, 1000]
AST_20050911_recon: linear2pc: (4, 6, 9): [0, 0, 1000, 1000]
0
200
400
600
800

FIGURE 3.11
RGB color composites (2% linear saturation) of three short wave infrared bands of an ASTER image acquired September 11, 2005 over Isfahan, Iran (left) and after reconstruction from the first two principal components (right).

(there are nine altogether). To display the result, we use the `dispms.py` script and show RGB composites of three of the short wave infrared bands, original and reconstructed:

```
%run scripts/dispms \
-f imagery/AST_20050911 -p [4,6,9] -e 3 \
-F imagery/AST_20050911_recon -P [4,6,9] -E 3
```

see Figure 3.11.

3.3.3 Primal solution

In order to perform PCA in practice, one calculates the estimate $\boldsymbol{s}$ of the covariance matrix $\boldsymbol{\Sigma}$ in terms of a data matrix $\boldsymbol{\mathcal{G}}$, see Equation (2.57), and then solves the eigenvalue problem, Equation (3.41), in the form

$$\boldsymbol{s}\boldsymbol{w} = \frac{1}{m-1}\boldsymbol{\mathcal{G}}^\top\boldsymbol{\mathcal{G}}\boldsymbol{w} = \lambda\boldsymbol{w}. \tag{3.46}$$

This is the *primal* problem for PCA (see the discussion of primal and dual formulations for ridge regression in Section 2.6.4). The Python programs in Listing 1.2 in Chapter 1 and in Listing 3.2 illustrate the procedure explicitly.

3.3.4 Dual solution

The normalized eigenvectors of the estimated covariance matrix $\boldsymbol{s}$ are $\boldsymbol{w}_i$, $i = 1\dots N$. These, as explained, are the principal vectors, and we now show how to express them in terms of the eigenvectors of the Gram matrix $\mathcal{G}\mathcal{G}^\top$, which was initially introduced in Section 2.6.4 in connection with ridge regression. Recall that the Gram matrix is symmetric, positive semi-definite.

Assume that the number of observations $m > N$ and consider an eigenvector-eigenvalue pair $(\boldsymbol{v}_i, \lambda_i)$ for the Gram matrix $\mathcal{G}\mathcal{G}^\top$. Then we can write with Equation 3.46

$$\boldsymbol{s}(\mathcal{G}^\top \boldsymbol{v}_i) = \frac{1}{m-1}(\mathcal{G}^\top \mathcal{G})(\mathcal{G}^\top \boldsymbol{v}_i) = \frac{1}{m-1}\mathcal{G}^\top(\mathcal{G}\mathcal{G}^\top)\boldsymbol{v}_i = \frac{1}{m-1}\lambda_i(\mathcal{G}^\top \boldsymbol{v}_i), \tag{3.47}$$

so that $(\mathcal{G}^\top \boldsymbol{v}_i, \lambda_i/(m-1))$ is an eigenvector-eigenvalue pair for $\boldsymbol{s}$. The norm of the eigenvector $\mathcal{G}^\top \boldsymbol{v}_i$ is

$$\|\mathcal{G}^\top \boldsymbol{v}_i\| = \sqrt{\boldsymbol{v}_i^\top \mathcal{G}\mathcal{G}^\top \boldsymbol{v}_i} = \sqrt{\lambda_i}. \tag{3.48}$$

For $\lambda_1 > \lambda_2 > \dots \lambda_N > 0$, the normalized principal vectors $\boldsymbol{w}_i$ can thus be expressed equivalently in the form

$$\boldsymbol{w}_i = \lambda_i^{-1/2}\mathcal{G}^\top \boldsymbol{v}_i, \quad i = 1\dots N.$$

In fact, the Gram matrix $\mathcal{G}\mathcal{G}^\top$ has exactly N positive eigenvalues, the rest being zero ($\mathcal{G}\mathcal{G}^\top$ has rank N). Informally, every positive eigenvalue for $\mathcal{G}\mathcal{G}^\top$ generates, via Equation (3.47), an eigenvector-eigenvalue pair for $\boldsymbol{s}$, and $\boldsymbol{s}$ has only N eigenvectors. For example, for $m = 100,\ N = 2$:

```
# column-centered data matrix for random 2D data
m, N = 100, 2
G = np.mat(2*np.random.rand(m, N))-1
# covariance matrix
S = G.T*G/(m-1)
# Gram matrix
K = G*G.T
lambda_s , _ = np.linalg.eigh(S)
lambda_k , _ = np.linalg.eigh(K)
# sort eigenvalues in decreasing oder
idx = np.argsort(lambda_s)[::-1]
lambda_s = lambda_s[idx]
idx = np.argsort(lambda_k)[::-1]
lambda_k = lambda_k[idx]
# compare
print(lambda_s)
print(lambda_k[0:3]/(m-1))
```

```
[0.33175673 0.27894934]
[3.31756725e-01 2.78949340e-01 6.71767647e-17]
```

The eigenvalues of $\boldsymbol{s} = \mathcal{G}^\top \mathcal{G}$ are equal to the first two eigenvalues of $\boldsymbol{k} = \mathcal{G}\mathcal{G}^\top$ when divided by $m-1$. The remaining $m-N$ eigenvalues are, apart from rounding errors, zero.

In terms of m-dimensional *dual vectors* $\boldsymbol{\alpha}_i = \lambda_i^{-1/2}\boldsymbol{v}_i$, we have

$$\boldsymbol{w}_i = \boldsymbol{\mathcal{G}}^\top \boldsymbol{\alpha}_i = \sum_{\nu=1}^{m} (\boldsymbol{\alpha}_i)_\nu \boldsymbol{g}(\nu), \quad i = 1 \dots N. \tag{3.49}$$

So, just as for ridge regression, we get the dual form by expressing the parameter vector as a linear combination of the observations. The projection of any observation $\boldsymbol{g}$ along a principal axis is then

$$\boldsymbol{w}_i^\top \boldsymbol{g} = \sum_{\nu=1}^{m} (\boldsymbol{\alpha}_i)_\nu (\boldsymbol{g}(\nu)^\top \boldsymbol{g}), \quad i = 1 \dots N. \tag{3.50}$$

Thus, we can alternatively perform PCA by finding eigenvalues and eigenvectors of the Gram matrix. The observations $\boldsymbol{g}(\nu)$ appear only in the form of inner products, both in the determination of the Gram matrix as well as in the projection of any new observations, Equation (3.50). This forms the starting point for non-linear, or *kernel* PCA. Non-linear kernels will be introduced in Chapter 4 with application to nonlinear PCA and appear again in connection with support vector machine classification in Chapter 6, hyper spectral anomaly detection in Chapter 7, and unsupervised classification in Chapter 8. Chapter 9 gives an example of kernel PCA for change detection.

3.4 Minimum noise fraction

Principal components analysis maximizes variance. This doesn't always lead to images of the desired quality (e.g., having minimal noise). The *minimum noise fraction* (MNF) transformation (Green et al., 1988) can be used to maximize the *signal-to-noise ratio* (SNR) rather than maximizing variance, so, if this is the desired criterion, it is to be preferred over PCA. In the following, we derive the MNF transformation directly by maximizing the ratio of signal variance to noise variance. This will be seen to involve the solution of a so-called *generalized eigenvalue problem*. Then we demonstrate how to perform the same transformation with principal components analysis alone.

3.4.1 Additive noise

A noisy multispectral image $\boldsymbol{G}$ may often be represented in terms of an *additive noise model*, i.e., as a sum of signal and noise contributions*

$$\boldsymbol{G} = \boldsymbol{S} + \boldsymbol{N}. \tag{3.51}$$

*The phenomenon of *speckle* in SAR imagery, on the other hand, can be treated as a form of multiplicative noise; see Chapter 5.

The signal $\boldsymbol{S}$ (not to be confused here with the covariance matrix estimator $\boldsymbol{s}$) is understood as the component carrying the information of interest. Noise, introduced most often by the sensors, corrupts the signal and masks that information. If both components are assumed to be normally distributed with respective covariance matrices $\boldsymbol{\Sigma}_S$ and $\boldsymbol{\Sigma}_N$, to have zero mean and, furthermore, to be uncorrelated, then the covariance matrix $\boldsymbol{\Sigma}$ for the image $\boldsymbol{G}$ is given by

$$\boldsymbol{\Sigma} = \langle \boldsymbol{G}\boldsymbol{G}^\top \rangle = \langle (\boldsymbol{S}+\boldsymbol{N})(\boldsymbol{S}+\boldsymbol{N})^\top \rangle = \langle \boldsymbol{S}\boldsymbol{S}^\top \rangle + \langle \boldsymbol{N}\boldsymbol{N}^\top \rangle,$$

the covariance $\langle \boldsymbol{N}\boldsymbol{S}^\top \rangle$ being zero by assumption. Thus the image covariance matrix is simply the sum of signal and noise contributions,

$$\boldsymbol{\Sigma} = \boldsymbol{\Sigma}_S + \boldsymbol{\Sigma}_N. \tag{3.52}$$

The signal-to-noise ratio in the ith band of a multi-spectral image is usually expressed as the ratio of the variance of the signal and noise components,

$$\mathrm{SNR}_i = \frac{\mathrm{var}(S_i)}{\mathrm{var}(N_i)}, \quad i = 1 \dots N.$$

Let us now seek a linear combination $Y = \boldsymbol{a}^\top \boldsymbol{G}$ of image bands for which this ratio is maximum. That is, we wish to maximize

$$\mathrm{SNR} = \frac{\mathrm{var}(\boldsymbol{a}^\top \boldsymbol{S})}{\mathrm{var}(\boldsymbol{a}^\top \boldsymbol{N})} = \frac{\boldsymbol{a}^\top \boldsymbol{\Sigma}_S \boldsymbol{a}}{\boldsymbol{a}^\top \boldsymbol{\Sigma}_N \boldsymbol{a}}. \tag{3.53}$$

The ratio of quadratic forms on the right is referred to as a *Rayleigh quotient.*

With Equation (3.52), we can write the Equation (3.53) equivalently in the form

$$\mathrm{SNR} = \frac{\boldsymbol{a}^\top \boldsymbol{\Sigma} \boldsymbol{a}}{\boldsymbol{a}^\top \boldsymbol{\Sigma}_N \boldsymbol{a}} - 1. \tag{3.54}$$

Setting its vector derivative with respect to $\boldsymbol{a}$ equal to zero, we get

$$\frac{\partial}{\partial \boldsymbol{a}}\mathrm{SNR} = \frac{1}{\boldsymbol{a}^\top \boldsymbol{\Sigma}_N \boldsymbol{a}} 2\boldsymbol{\Sigma}\boldsymbol{a} - \frac{\boldsymbol{a}^\top \boldsymbol{\Sigma} \boldsymbol{a}}{(\boldsymbol{a}^\top \boldsymbol{\Sigma}_N \boldsymbol{a})^2} 2\boldsymbol{\Sigma}_N \boldsymbol{a} = \boldsymbol{0}$$

or, equivalently,

$$(\boldsymbol{a}^\top \boldsymbol{\Sigma}_N \boldsymbol{a})\boldsymbol{\Sigma}\boldsymbol{a} = (\boldsymbol{a}^\top \boldsymbol{\Sigma} \boldsymbol{a})\boldsymbol{\Sigma}_N \boldsymbol{a}.$$

This condition is met when $\boldsymbol{a}$ solves the *symmetric generalized eigenvalue problem*

$$\boldsymbol{\Sigma}_N \boldsymbol{a} = \lambda \boldsymbol{\Sigma} \boldsymbol{a}, \tag{3.55}$$

as can easily be seen by substitution. In Equation (3.55), both $\boldsymbol{\Sigma}_N$ and $\boldsymbol{\Sigma}$ are symmetric and positive definite. The equation can be reduced to the standard eigenvalue problem that we are familiar with by performing a *Cholesky decomposition* of $\boldsymbol{\Sigma}$. As explained in Appendix A, Cholesky decomposition will factor $\boldsymbol{\Sigma}$ as $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^\top$, where $\boldsymbol{L}$ is a lower triangular matrix. Substituting this into Equation (3.55) gives

$$\boldsymbol{\Sigma}_N \boldsymbol{a} = \lambda \boldsymbol{L}\boldsymbol{L}^\top \boldsymbol{a}$$

or, multiplying both sides of the equation from the left by $\boldsymbol{L}^{-1}$ and inserting the identity $(\boldsymbol{L}^\top)^{-1}\boldsymbol{L}^\top$,

$$\boldsymbol{L}^{-1}\boldsymbol{\Sigma}_N(\boldsymbol{L}^\top)^{-1}\boldsymbol{L}^\top\boldsymbol{a} = \lambda\boldsymbol{L}^\top\boldsymbol{a}.$$

Now let $\boldsymbol{b} = \boldsymbol{L}^\top\boldsymbol{a}$. From the commutativity of the operations of inverse and transpose, it follows that

$$[\boldsymbol{L}^{-1}\boldsymbol{\Sigma}_N(\boldsymbol{L}^{-1})^\top]\boldsymbol{b} = \lambda\boldsymbol{b}, \tag{3.56}$$

a standard eigenvalue problem for the symmetric matrix $\boldsymbol{L}^{-1}\boldsymbol{\Sigma}_N(\boldsymbol{L}^{-1})^\top$. Let its (orthogonal and normalized) eigenvectors be $\boldsymbol{b}_i$, $i = 1\dots N$. Then

$$\boldsymbol{b}_i^\top\boldsymbol{b}_j = \boldsymbol{a}_i^\top\boldsymbol{L}\boldsymbol{L}^\top\boldsymbol{a}_j = \boldsymbol{a}_i^\top\boldsymbol{\Sigma}\boldsymbol{a}_j = \delta_{ij}.$$

Therefore, we see that the variances of the transformed components $Y_i = \boldsymbol{a}_i^\top\boldsymbol{G}$ are all unity:

$$\mathrm{var}(Y_i) = \boldsymbol{a}_i^\top\boldsymbol{\Sigma}\boldsymbol{a}_i = 1, \quad i = 1\dots N,$$

and that they are mutually uncorrelated:

$$\mathrm{cov}(Y_i, Y_j) = \boldsymbol{a}_i^\top\boldsymbol{\Sigma}\boldsymbol{a}_j = 0, \quad i, j = 1\dots N,\ i \neq j.$$

Listing 3.3 shows a Python routine for solving the generalized eigenvalue problem with Cholesky decomposition. The Cholesky algorithm is programmed explicitly,* but we could just as well have used the `numpy.linalg.cholesky()` function.

With the definition $\boldsymbol{A} = (\boldsymbol{a}_1, \boldsymbol{a}_2 \dots \boldsymbol{a}_N)$, the complete minimum noise fraction (MNF) transformation can be represented as

$$\boldsymbol{Y} = \boldsymbol{A}^\top\boldsymbol{G}, \tag{3.57}$$

in a manner similar to the principal components transformation, Equation (3.42). The covariance matrix of $\boldsymbol{Y}$ is (compare with Equation (3.43))

$$\boldsymbol{\Sigma}' = \boldsymbol{A}^\top\boldsymbol{\Sigma}\boldsymbol{A} = \boldsymbol{I}, \tag{3.58}$$

where $\boldsymbol{I}$ is the $N \times N$ identity matrix.

*As described in `http://en.wikipedia.org/wiki/Cholesky_decomposition`.

Listing 3.3: Solving the generalized eigenvalue problem in Python by Cholesky decomposition (excerpt from `auxil.auxil1.py`).

```python
def choldc(A):
    '''Cholesky-Banachiewicz algorithm,
       A is a numpy matrix'''
    L = A - A
    for i in range(len(L)):
        for j in range(i):
            sm = 0.0
            for k in range(j):
                sm += L[i,k]*L[j,k]
            L[i,j] = (A[i,j]-sm)/L[j,j]
        sm = 0.0
        for k in range(i):
            sm += L[i,k]*L[i,k]
        L[i,i] = math.sqrt(A[i,i]-sm)
    return L

def geneiv(A,B):
    '''solves A*x = lambda*B*x for numpy matrices A, B
       returns eigenvectors in columns'''
    Li = np.linalg.inv(choldc(B))
    C = Li*A*(Li.transpose())
    C = np.asmatrix((C + C.transpose())*0.5,np.float32)
    lambdas,V = np.linalg.eig(C)
    return lambdas, Li.transpose()*V
```

It follows from Equation (3.54) that the SNR for eigenvalue λ_i is just

$$\mathrm{SNR}_i = \frac{\boldsymbol{a}_i^\top \boldsymbol{\Sigma} \boldsymbol{a}_i}{\boldsymbol{a}_i^\top (\lambda_i \boldsymbol{\Sigma} \boldsymbol{a}_i)} - 1 = \frac{1}{\lambda_i} - 1. \tag{3.59}$$

Thus, the projection $Y_i = \boldsymbol{a}_i^\top \boldsymbol{G}$ corresponding to the *smallest* eigenvalue λ_i will have largest signal-to-noise ratio. Note that with Equation (3.55), we can write

$$\boldsymbol{\Sigma}_N \boldsymbol{A} = \boldsymbol{\Sigma} \boldsymbol{A} \boldsymbol{\Lambda}, \tag{3.60}$$

where $\boldsymbol{\Lambda}$ is the diagonal matrix with diagonal elements $\lambda_1 \dots \lambda_N$.

3.4.2 Minimum noise fraction via PCA

The MNF transformation can be carried out somewhat differently with the solution two successive ordinary eigenvalue problems, which are, as we shall now show, equivalent to the above derivation.

In the first step the noise contribution to the observation $\boldsymbol{G}$ is "whitened," that is, a transformation is performed after which the noise component $\boldsymbol{N}$ has covariance matrix $\boldsymbol{\Sigma}_N = \boldsymbol{I}$, the identity matrix. This can be accomplished by first doing a transformation which diagonalizes $\boldsymbol{\Sigma}_N$. Suppose that the transformation matrix for this operation is $\boldsymbol{C}$ and that $\boldsymbol{Z}$ is the resulting random vector. Then

$$\boldsymbol{Z} = \boldsymbol{C}^\top \boldsymbol{G}, \quad \boldsymbol{C}^\top \boldsymbol{\Sigma}_N \boldsymbol{C} = \boldsymbol{\Lambda}_N,\ \boldsymbol{C}^\top \boldsymbol{C} = \boldsymbol{I}, \tag{3.61}$$

where $\boldsymbol{\Lambda}_N$ is a diagonal matrix, the diagonal elements of which are the variances of the transformed noise component $\boldsymbol{C}^\top \boldsymbol{N}$. Next apply the transformation $\boldsymbol{\Lambda}_N^{-1/2}$ (the diagonal matrix whose diagonal elements are the square roots of the diagonal elements of $\boldsymbol{\Lambda}_N$) to give a new random vector $\boldsymbol{X}$,

$$\boldsymbol{X} = \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{Z} = \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{C}^\top \boldsymbol{G}.$$

Then the covariance matrix of the noise component $\boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{C}^\top \boldsymbol{N}$ is given by

$$\boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{\Lambda}_N \boldsymbol{\Lambda}_N^{-1/2} = \boldsymbol{I},$$

as desired, and the noise contribution has been "whitened." At this stage of affairs, the covariance matrix of the transformed random vector $\boldsymbol{X}$ is

$$\boldsymbol{\Sigma}_X = \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{C}^\top \boldsymbol{\Sigma} \boldsymbol{C} \boldsymbol{\Lambda}_N^{-1/2}. \tag{3.62}$$

In the second step, an ordinary principal components transformation is performed on $\boldsymbol{X}$, leading finally to the random vector $\boldsymbol{Y}$ representing the MNF components:

$$\boldsymbol{Y} = \boldsymbol{B}^\top \boldsymbol{X}, \quad \boldsymbol{B}^\top \boldsymbol{\Sigma}_X \boldsymbol{B} = \boldsymbol{\Lambda}_X,\ \boldsymbol{B}^\top \boldsymbol{B} = \boldsymbol{I}. \tag{3.63}$$

The overall transformation is thus

$$\boldsymbol{Y} = \boldsymbol{B}^\top \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{C}^\top \boldsymbol{G} = \boldsymbol{A}^\top \boldsymbol{G}, \tag{3.64}$$

where $\boldsymbol{A} = \boldsymbol{C} \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{B}$. To see that this transformation is indeed equivalent to solving the generalized eigenvalue problem, Equation (3.60), consider

$$\begin{aligned}
\boldsymbol{\Sigma}_N \boldsymbol{A} &= \boldsymbol{\Sigma}_N \boldsymbol{C} \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{B} \\
&= \boldsymbol{C} \boldsymbol{\Lambda}_N \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{B} \quad \text{from Equation (3.61)} \\
&= \boldsymbol{C} \boldsymbol{\Lambda}_N^{1/2} \boldsymbol{B} \\
&= \boldsymbol{C} \boldsymbol{\Lambda}_N^{1/2} (\boldsymbol{\Sigma}_X \boldsymbol{B} \boldsymbol{\Lambda}_X^{-1}) \quad \text{from Equation (3.63)} \\
&= \boldsymbol{C} \boldsymbol{\Lambda}_N^{1/2} \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{C}^\top \boldsymbol{\Sigma} \boldsymbol{C} \boldsymbol{\Lambda}_N^{-1/2} \boldsymbol{B} \boldsymbol{\Lambda}_X^{-1} \quad \text{from Equation (3.62)} \\
&= \boldsymbol{\Sigma} \boldsymbol{A} \boldsymbol{\Lambda}_X^{-1}.
\end{aligned} \tag{3.65}$$

This is the same as Equation (3.60) with $\boldsymbol{\Lambda}$ replaced by $\boldsymbol{\Lambda}_X^{-1}$, that is,

$$\lambda_{Xi} = \frac{1}{\lambda_i} = \mathrm{SNR}_i + 1, \qquad i = 1 \ldots N,$$

using Equation (3.59). Note that the MNF components returned here, unlike those of Section 3.4.1, do not have unit variance. Their variances are the eigenvalues λ_{Xi}. They are equal to the SNR plus one, so that values equal to one correspond to "pure noise."

3.5 Spatial correlation

Before the MNF transformation can be performed, it is of course necessary to estimate both the image and noise covariance matrices $\mathbf{\Sigma}$ and $\mathbf{\Sigma}_N$. The former poses no problem, but how does one estimate the noise covariance matrix? The spatial characteristics of the image can be used to estimate $\mathbf{\Sigma}_N$, taking advantage of the fact that the intensity of neighboring pixels is usually approximately constant. This property is quantified as the *autocorrelation* of an image. We shall first find a spectral transformation that maximizes the autocorrelation and then see how to relate it to the image noise statistics.

3.5.1 Maximum autocorrelation factor

Let $\boldsymbol{x} = (x_1, x_2)^\top$ represent the coordinates of a pixel within image $\boldsymbol{G}$ and assume that $\langle \boldsymbol{G} \rangle = 0$. The *spatial covariance* $\boldsymbol{C}(\boldsymbol{x}, \boldsymbol{h})$ is defined as the covariance of the original image, represented by $\boldsymbol{G}(\boldsymbol{x})$, with itself, but shifted by the amount $\boldsymbol{h} = (h_1, h_2)^\top$,

$$\boldsymbol{C}(\boldsymbol{x}, \boldsymbol{h}) = \langle \boldsymbol{G}(\boldsymbol{x}) \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})^\top \rangle. \tag{3.66}$$

We make the so-called *second-order stationarity assumption*, namely that $\boldsymbol{C}(\boldsymbol{x}, \boldsymbol{h}) = \boldsymbol{C}(\boldsymbol{h})$ is independent of $\boldsymbol{x}$. Then $\boldsymbol{C}(\boldsymbol{0}) = \langle \boldsymbol{G}\boldsymbol{G}^\top \rangle = \mathbf{\Sigma}$, and furthermore

$$\begin{aligned} \boldsymbol{C}(-\boldsymbol{h}) &= \langle \boldsymbol{G}(\boldsymbol{x}) \boldsymbol{G}(\boldsymbol{x}-\boldsymbol{h})^\top \rangle \\ &= \langle \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h}) \boldsymbol{G}(\boldsymbol{x})^\top \rangle \\ &= \langle (\boldsymbol{G}(\boldsymbol{x}) \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})^\top)^\top \rangle \\ &= \boldsymbol{C}(\boldsymbol{h})^\top. \end{aligned} \tag{3.67}$$

The *multivariate variogram*, $\mathbf{\Gamma}(\boldsymbol{h})$, is defined as the covariance matrix of the difference image $\boldsymbol{G}(\boldsymbol{x}) - \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})$,

$$\begin{aligned} \mathbf{\Gamma}(\boldsymbol{h}) &= \langle (\boldsymbol{G}(\boldsymbol{x}) - \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h}))(\boldsymbol{G}(\boldsymbol{x}) - \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})^\top \rangle \\ &= \langle \boldsymbol{G}(\boldsymbol{x}) \boldsymbol{G}(\boldsymbol{x})^\top \rangle + \langle \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h}) \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})^\top \rangle \\ &\quad - \langle \boldsymbol{G}(\boldsymbol{x}) \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})^\top \rangle - \langle \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h}) \boldsymbol{G}(\boldsymbol{x})^\top \rangle \\ &= 2\mathbf{\Sigma} - \boldsymbol{C}(\boldsymbol{h}) - \boldsymbol{C}(-\boldsymbol{h}). \end{aligned} \tag{3.68}$$

Now let us look at the covariance of projections $Y = \boldsymbol{a}^\top \boldsymbol{G}$ of the original and shifted images. This is given by

$$\begin{aligned}\mathrm{cov}(\boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}), \boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})) &= \boldsymbol{a}^\top \langle \boldsymbol{G}(\boldsymbol{x})\boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})^\top \rangle \boldsymbol{a} \\ &= \boldsymbol{a}^\top \boldsymbol{C}(\boldsymbol{h})\boldsymbol{a} \\ &= \boldsymbol{a}^\top \boldsymbol{C}(-\boldsymbol{h})\boldsymbol{a} \\ &= \frac{1}{2}\boldsymbol{a}^\top (\boldsymbol{C}(\boldsymbol{h}) + \boldsymbol{C}(-\boldsymbol{h}))\boldsymbol{a},\end{aligned} \tag{3.69}$$

where we have used Equation (3.67). From Equation (3.68), $\boldsymbol{C}(\boldsymbol{h})+\boldsymbol{C}(-\boldsymbol{h}) = 2\boldsymbol{\Sigma} - \boldsymbol{\Gamma}(\boldsymbol{h})$, and so we can write Equation (3.69) in the form

$$\mathrm{cov}(\boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}), \boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})) = \boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a} - \frac{1}{2}\boldsymbol{a}^\top \boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a}. \tag{3.70}$$

The *spatial autocorrelation* of the projections is therefore given by

$$\begin{aligned}\mathrm{corr}(\boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}), \boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})) &= \frac{\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a} - \frac{1}{2}\boldsymbol{a}^\top \boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a}}{\sqrt{\mathrm{var}(\boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}))\mathrm{var}(\boldsymbol{a}^\top \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h}))}} \\ &= \frac{\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a} - \frac{1}{2}\boldsymbol{a}^\top \boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a}}{\sqrt{(\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a})(\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a})}} \\ &= 1 - \frac{1}{2}\,\frac{\boldsymbol{a}^\top \boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a}}{\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a}}.\end{aligned} \tag{3.71}$$

The *maximum autocorrelation factor* (MAF) transformation determines the vector $\boldsymbol{a}$ which maximizes Equation (3.71). We obtain it by minimizing the Rayleigh quotient

$$R(\boldsymbol{a}) = \frac{\boldsymbol{a}^\top \boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a}}{\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a}}.$$

Setting the vector derivative equal to zero gives

$$\frac{\partial R}{\partial \boldsymbol{a}} = \frac{1}{\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a}}\ \frac{1}{2}\boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a} - \frac{\boldsymbol{a}^\top \boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a}}{(\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a})^2}\ \frac{1}{2}\boldsymbol{\Sigma}\boldsymbol{a} = \boldsymbol{0}$$

or

$$(\boldsymbol{a}^\top \boldsymbol{\Sigma}\boldsymbol{a})\boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a} = (\boldsymbol{a}^\top \boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a})\boldsymbol{\Sigma}\boldsymbol{a}.$$

This condition is met when $\boldsymbol{a}$ solves the generalized eigenvalue problem

$$\boldsymbol{\Gamma}(\boldsymbol{h})\boldsymbol{a} = \lambda \boldsymbol{\Sigma}\boldsymbol{a}, \tag{3.72}$$

which is seen to have the same form as Equation (3.55), with $\boldsymbol{\Gamma}(\boldsymbol{h})$ replacing the noise covariance matrix $\boldsymbol{\Sigma}_N$. Again, both $\boldsymbol{\Gamma}(\boldsymbol{h})$ and $\boldsymbol{\Sigma}$ are symmetric, and the latter is also positive definite. We obtain as before, via Cholesky decomposition, the standard eigenvalue problem

$$[\boldsymbol{L}^{-1}\boldsymbol{\Gamma}(\boldsymbol{h})(\boldsymbol{L}^{-1})^\top]\boldsymbol{b} = \lambda \boldsymbol{b}, \tag{3.73}$$

Listing 3.4: Estimation of the noise covariance matrix in Python from the difference of one-pixel shifts.

```
#!/usr/bin/env python3
#Name:  ex3_2.py
import numpy as np
from osgeo import gdal
import sys
from osgeo.gdalconst import GA_ReadOnly

def noisecovar(infile):
    gdal.AllRegister()
    inDataset = gdal.Open(infile,GA_ReadOnly)
    cols = inDataset.RasterXSize
    rows = inDataset.RasterYSize
    bands = inDataset.RasterCount
    #  data matrix for difference images
    D = np.zeros((cols*rows,bands))
    for b in range(bands):
        band = inDataset.GetRasterBand(b+1)
        tmp = band.ReadAsArray(0,0,cols,rows)
        D[:,b] = (tmp-np.roll(tmp,1,axis=0)).ravel()
    #  noise covariance matrix
    return np.mat(D).T*np.mat(D)/(2*(rows*cols-1))

if __name__ == '__main__':
    infile = sys.argv[1]
    S_N = noisecovar(infile)
    np.set_printoptions(precision=2, suppress=True)
    print('Noise covariance, file %s'%infile)
    print(S_N)
```

for the symmetric matrix $\boldsymbol{L}^{-1}\boldsymbol{\Gamma}(\boldsymbol{h})(\boldsymbol{L}^{-1})^\top$ with $\boldsymbol{b} = \boldsymbol{L}^\top\boldsymbol{a}$.

Let the eigenvalues of Equation (3.73) be ordered from smallest to largest, $\lambda_1 \leq \ldots \leq \lambda_N$, and the corresponding (orthogonal) eigenvectors be $\boldsymbol{b}_i$. We have

$$\boldsymbol{b}_i^\top\boldsymbol{b}_j = \boldsymbol{a}_i^\top\boldsymbol{L}\boldsymbol{L}^\top\boldsymbol{a}_j = \boldsymbol{a}_i^\top\boldsymbol{\Sigma}\boldsymbol{a}_j = \delta_{ij} \tag{3.74}$$

so that, like the components of the MNF transformation, the MAF components $Y_i = \boldsymbol{a}_i^\top\boldsymbol{G}$, $i = 1\ldots N$, are orthogonal (uncorrelated) with unit variance. Moreover, with Equation (3.71),

$$\mathrm{corr}(\boldsymbol{a}_i^\top\boldsymbol{G}(\boldsymbol{x}), \boldsymbol{a}_i^\top\boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})) = 1 - \frac{1}{2}\lambda_i, \quad i = 1\ldots N, \tag{3.75}$$

and the first MAF component has maximum autocorrelation.

3.5.2 Noise estimation

The similarity of Equation (3.72) and Equation (3.55) is a result of the fact that $\boldsymbol{\Gamma}(\boldsymbol{h})$ is, under fairly general circumstances, proportional to $\boldsymbol{\Sigma}_N$. We can demonstrate this as follows (Green et al., 1988). Let

$$\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{S}(\boldsymbol{x}) + \boldsymbol{N}(\boldsymbol{x})$$

and assume

$$\begin{aligned} \langle \boldsymbol{S}(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x})^\top \rangle &= \boldsymbol{0} \\ \langle \boldsymbol{S}(\boldsymbol{x})\boldsymbol{S}(\boldsymbol{x} \pm \boldsymbol{h})^\top \rangle &= b_h \boldsymbol{\Sigma}_S \\ \langle \boldsymbol{N}(\boldsymbol{x})\boldsymbol{N}(\boldsymbol{x} \pm \boldsymbol{h})^\top \rangle &= c_h \boldsymbol{\Sigma}_N, \end{aligned} \tag{3.76}$$

where b_h and c_h are constants. Under these assumptions, $\boldsymbol{C}(\boldsymbol{h}) = \boldsymbol{C}(-\boldsymbol{h})$ from Equation (3.67) and, from Equation (3.68), we can conclude that

$$\boldsymbol{\Gamma}(\boldsymbol{h}) = 2(\boldsymbol{\Sigma} - \boldsymbol{C}(\boldsymbol{h})). \tag{3.77}$$

But with Equation (3.66)

$$\boldsymbol{C}(\boldsymbol{h}) = \langle \big(\boldsymbol{S}(\boldsymbol{x}) + \boldsymbol{N}(\boldsymbol{x})\big)\big(S(\boldsymbol{x}+\boldsymbol{h}) + \boldsymbol{N}(\boldsymbol{x}+\boldsymbol{h})\big)^\top \rangle$$

or, with Equation (3.76),

$$\boldsymbol{C}(\boldsymbol{h}) = b_h \boldsymbol{\Sigma}_S + c_h \boldsymbol{\Sigma}_N. \tag{3.78}$$

Finally, combining Equations (3.77), (3.78), and (3.52) gives

$$\frac{1}{2}\boldsymbol{\Gamma}(\boldsymbol{h}) = (1 - b_h)\boldsymbol{\Sigma} + (b_h - c_h)\boldsymbol{\Sigma}_N. \tag{3.79}$$

For a signal with high spatial coherence and for random ("salt and pepper") noise, we expect that in Equation (3.76)

$$b_h \approx 1 \gg c_h$$

and therefore, from Equation (3.79), that

$$\boldsymbol{\Sigma}_N \approx \frac{1}{2}\boldsymbol{\Gamma}(\boldsymbol{h}). \tag{3.80}$$

Thus we can obtain an estimate for the noise covariance matrix by estimating the multivariate variogram

$$\boldsymbol{\Gamma}(\boldsymbol{h}) = \langle (\boldsymbol{G}(\boldsymbol{x}) - \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h}))(\boldsymbol{G}(\boldsymbol{x}) - \boldsymbol{G}(\boldsymbol{x}+\boldsymbol{h})^\top \rangle$$

and dividing the result by 2. This is illustrated in Listing 3.4. There, the matrix $\boldsymbol{\Gamma}(\boldsymbol{h})$ is determined by calculating the covariance matrix of the difference of the image with itself, shifted horizontally by one pixel.* Here is the result for the nine optical and infrared bands of the ASTER image of Figure 1.1.

*This will tend to overestimate the noise in an image in which the signal itself varies considerably. The noise determination should be restricted to regions having as little detailed structure as possible.

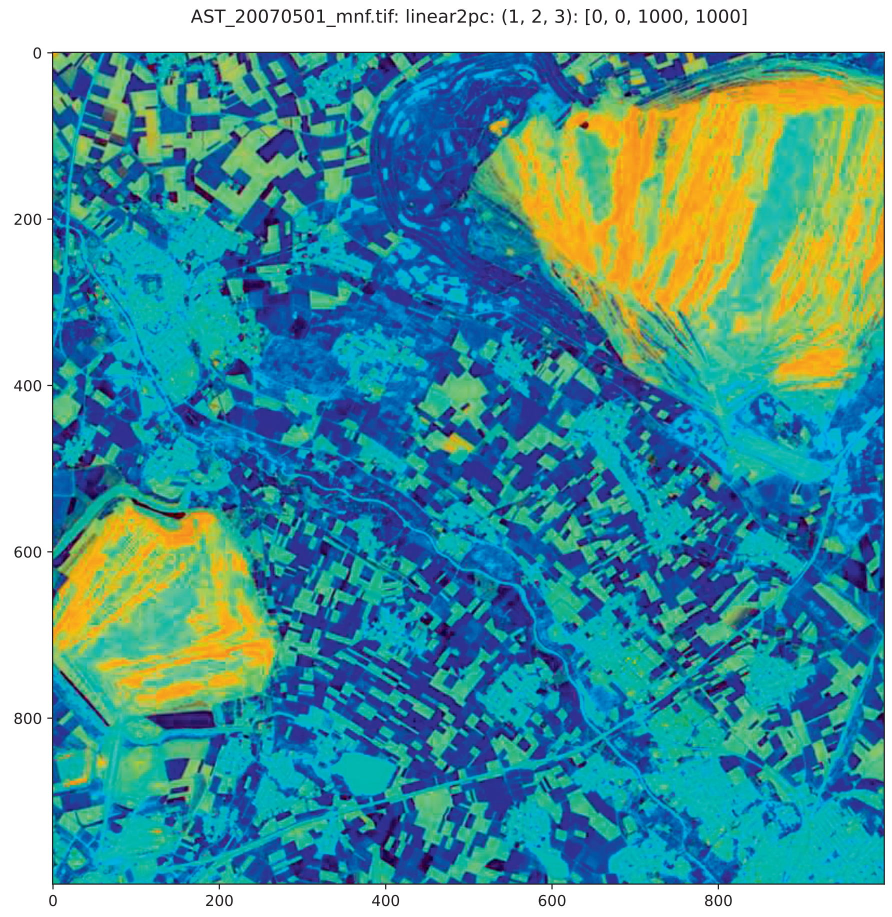

AST_20070501_mnf.tif: linear2pc: (1, 2, 3): [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800

FIGURE 3.12
RGB color composite of MNF bands 1, 2, 3 of the ASTER image of Figure 1.1.

```
run scripts/ex3_2 imagery/AST_20070501.tif
Noise covariance, file imagery/AST_20070501.tif
[[15.89 15.5   -0.21  1.45  0.45  0.51  0.44  0.36  0.24]
 [15.5  16.93 -2.69  1.58  0.5   0.57  0.49  0.4   0.26]
 [-0.21 -2.69 24.95 -0.17 -0.1  -0.11 -0.1  -0.08 -0.06]
 [ 1.45  1.58 -0.17  0.21  0.06  0.07  0.06  0.05  0.03]
 [ 0.45  0.5  -0.1   0.06  0.02  0.02  0.02  0.02  0.01]
 [ 0.51  0.57 -0.11  0.07  0.02  0.03  0.02  0.02  0.01]
 [ 0.44  0.49 -0.1   0.06  0.02  0.02  0.02  0.02  0.01]
 [ 0.36  0.4  -0.08  0.05  0.02  0.02  0.02  0.01  0.01]
 [ 0.24  0.26 -0.06  0.03  0.01  0.01  0.01  0.01  0.01]]
```

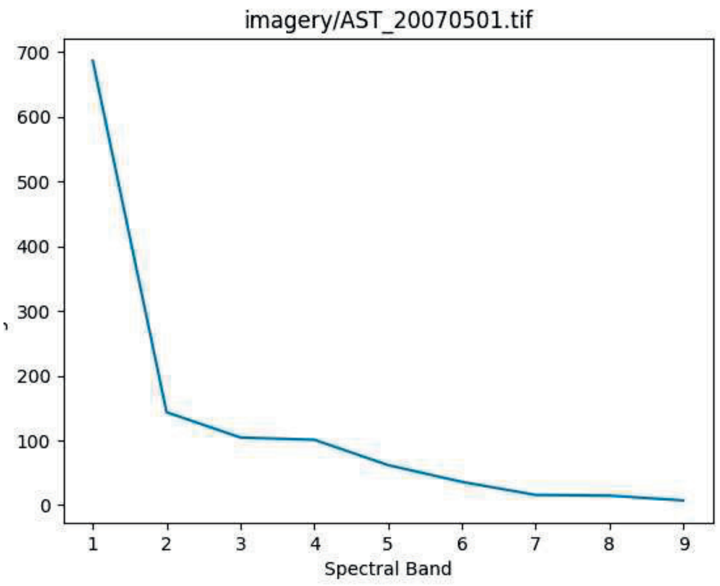


FIGURE 3.13
Signal to noise ratios in the nine MNF bands.

The program `mnf.py` in the `scripts` subdirectory uses the above ideas to perform the MNF transformation on multi-spectral imagery; see Appendix C. For example:

```
run scripts/mnf -n imagery/AST_20070501.tif
------------MNF ---------------
Wed Feb 21 12:17:54 2024
Input imagery/AST_20070501.tif
Signal to noise ratios: [
686.3224   143.42426  104.36482
100.94557   61.796833  35.90177
15.710253  14.701647   7.14548 ]
MNFs written to: imagery/AST_20070501_mnf.tif
elapsed time: 0.5313358306884766
```

The code

```
run scripts/dispms -f imagery/AST_20070501_mnf.tif
                                  -p [1,2,3] -e 3
```

generates the MNF image shown in Figure 3.12, and Figure 3.13 is a plot of the signal to noise ratios.

3.6 Exercises

1. Show for $g(x) = \sin(2\pi x)$ in Equation (3.1), that the corresponding frequency coefficients in Equation (3.4) are given by

$$\hat{g}(-1) = -\frac{1}{2\mathbf{i}}, \quad \hat{g}(1) = \frac{1}{2\mathbf{i}},$$

   and $\hat{g}(k) = 0$ otherwise.

2. Demonstrate Equation (3.6) with the help of Equation (3.7).

3. Calculate the discrete Fourier transform of the sequence $2, 4, 6, 8$ from Equation (3.5). You have to solve four simultaneous equations, the first of which is

$$2 = \hat{g}(0) + \hat{g}(1) + \hat{g}(2) + \hat{g}(3).$$

   Verify your result in Python with

```
>>> fft.fft([2,4,6,8])/4
```

4. Prove the Fourier translation property, Equation (3.11).

5. Derive the discrete form of *Parseval's Theorem*,

$$\sum_{k=0}^{c-1} |\hat{g}(k)|^2 = \frac{1}{c} \sum_{j=0}^{c-1} |g(j)|^2,$$

   using the orthogonality property, Equation (3.7).

6. Show that

$$B_2 = \{\phi_{0,0}(x), \psi_{0,0}(x), \psi_{1,0}(x), \psi_{1,1}(x)\},$$

   where

$$\psi_{1,0}(x) = \psi(2x), \;\; \psi_{1,1}(x) = \psi(2x-1)$$

   is an orthogonal basis for the subspace $V_1^{\perp}$.

7. Prove Equation (3.27).

8. It can be shown that, for any MRA, $\int_{-\infty}^{\infty} \phi(x)dx \neq 0$. Show that this implies that the refinement coefficients satisfy

$$\sum_k c_k = 2.$$

9. The cubic B-spline wavelet has the refinement coefficients $c_0 = c_4 = 1/8, \; c_1 = c_3 = 1/2, \; c_2 = 3/4$. Use the cascade algorithm (in notebook `Chapter3.ipynb`) to display the scaling function.

10. (a) (Strang and Nguyen, 1997) Given the dilation Equation (3.25) with n non-zero refinement coefficients $c_0 \dots c_{n-1}$, argue on the basis of the cascade algorithm, that the scaling function $\phi(x)$ must be zero outside the interval $[0, n-1]$.

    (b) Prove that $\phi(x)$ is supported on (extends over) the entire interval $[0, n-1]$.

11. Complete the Python script in Listing 3.2 to have it store the reconstructed image to disk.

12. As discussed in Section 3.3.3, for PCA, one estimates the covariance matrix $\boldsymbol{\Sigma}$ of an image in terms of the $m \times N$ data matrix $\boldsymbol{\mathcal{G}}$, solving the eigenvalue problem
$$\frac{1}{m-1}\boldsymbol{\mathcal{G}}^\top \boldsymbol{\mathcal{G}} \boldsymbol{w} = \lambda \boldsymbol{w}.$$
    Alternatively, consider the singular value decomposition of $\boldsymbol{\mathcal{G}}$ itself:
$$\boldsymbol{\mathcal{G}} = \boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^\top,$$
    where $\boldsymbol{U}$ is $m \times N$, $\boldsymbol{V}$ is $N \times m$, $\boldsymbol{W}$ is a diagonal $N \times N$ matrix, and
$$\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{V}^\top \boldsymbol{V} = \boldsymbol{I}.$$
    Explain why the columns of $\boldsymbol{V}$ are the principal axes (eigenvectors) of the transformation and the corresponding variances (eigenvalues) are proportional to the squares of the diagonal elements of $\boldsymbol{W}$.

13. The routine

```
# a 2D two-class image
n1 = np.random.randn(1000)
n2 = n1 + np.random.randn(1000)
B1 = np.zeros((1000,2))
B2 = np.zeros((1000,2))
B1[:,0] = n1
B1[:,1] = n2
B2[:,0] = n1+4
B2[:,1] = n2
G = np.concatenate((B1,B2))
# center the image
G[:,0] = G[:,0] - np.mean(G[:,0])
# estimate covariance and diagonalize
C = np.mat(G).T*np.mat(G)/2000
_,U = np.linalg.eigh(C)
# slopes of the principal axes
s1 = U[1,1]/U[0,1]
s2 = U[1,0]/U[0,0]
# plot
```

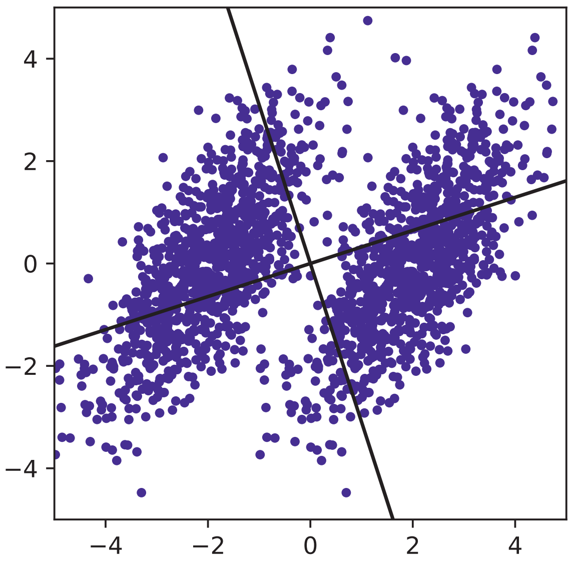


FIGURE 3.14
Two classes of observations in a two-dimensional feature space. The solid lines are the principal axes.

```
    plt.xlim((-5,5))
    plt.ylim((-5,5))
    plt.plot(G[:,0],G[:,1],'b.',
             [-5,5],[-5*s1,5*s1],'k',
             [-5,5],[-5*s2,5*s2],'k')
```

simulates two classes of observations B_1 and B_2 in a two-dimensional feature space and calculates the principal axes of the combined data; see Figure 3.14. While the classes are nicely separable in two dimensions, their one-dimensional projections along the x- or y-axes or along either of the principal axes are obviously not. A dimensionality reduction with PCA would thus result in a considerable loss of information about the class structure. *Fisher's linear discriminant* projects the observations $\boldsymbol{g}$ onto a direction $\boldsymbol{w}$, such that the ratio $J(\boldsymbol{w})$ of the squared difference of the class means of the projections $v = \boldsymbol{w}^\top \boldsymbol{g}$ to their overall variance is maximized (Duda and Hart, 1973). Specifically, define

$$\boldsymbol{m}_i = \frac{1}{n_i}\sum_{\boldsymbol{g}\in B_i} \boldsymbol{g}, \quad \boldsymbol{C}_i = \frac{1}{n_i}\sum_{\boldsymbol{g}\in B_i} (\boldsymbol{g}-\boldsymbol{m}_i)(\boldsymbol{g}-\boldsymbol{m}_i)^\top, \quad i = 1,2.$$

(a) Show that the objective function can be written in the form

$$J(\boldsymbol{w}) = \frac{\boldsymbol{w}^\top \boldsymbol{C}_B \boldsymbol{w}}{\boldsymbol{w}^\top (\boldsymbol{C}_1 + \boldsymbol{C}_2)\boldsymbol{w}}, \tag{3.81}$$

where $\boldsymbol{C}_B = (\boldsymbol{m}_1 - \boldsymbol{m}_2)(\boldsymbol{m}_1 - \boldsymbol{m}_2)^\top$.

(b) Show that the desired projection direction is given by

$$\boldsymbol{w} = (\boldsymbol{C}_1 + \boldsymbol{C}_2)^{-1}(\boldsymbol{m}_1 - \boldsymbol{m}_2). \tag{3.82}$$

(c) Modify the above script to calculate and plot the projection direction $\boldsymbol{w}$.

14. Using the code in Listings 3.1 and 3.3 as a starting point, write a Python module to perform the MNF transformation on the GEE Python API.

15. Show from Equation (3.50) that the variance of the principal components is given in terms of the eigenvalues λ_i of the Gram matrix by

$$\text{var}(\boldsymbol{w}_i^\top \boldsymbol{g}) = \frac{\lambda_i}{m-1}.$$

16. Formulate the primal and dual problems for the MNF transformation. (*Hint:* Similarly to the case for PCA, Equation (3.49), a dual problem can be obtained by writing $\boldsymbol{a} \propto \boldsymbol{\mathcal{G}}^\top \boldsymbol{\alpha}$.) Write the dual formulation in the form of a symmetric generalized eigenvalue problem. Can it be solved with Cholesky decomposition?

4

Filters, Kernels, and Fields

This chapter is somewhat of a catch-all, intended mainly to consolidate and extend material presented in the preceding chapters and to help lay the foundation for the rest of the book. In Sections 4.1 and 4.2, building on the discrete Fourier transform introduced in Chapter 3, the concept of discrete convolution is introduced and filtering, both in the spatial and in the frequency domain, is discussed. Frequent reference to filtering will be made in Chapter 5 when we treat enhancement and geometric and radiometric correction of visual/infrared and SAR imagery and in the discussion of convolutional neural networks in Chapter 7. In Section 4.3, it is shown that the discrete wavelet transform of Chapter 3 is equivalent to a recursive application of low- and high-pass filters (a filter bank) and a pyramid algorithm for multi-scale image representation is described and programmed in Python/Numpy. Wavelet pyramid representations are applied in Chapter 5 for panchromatic sharpening and in Chapter 8 for contextual clustering. Section 4.4 introduces so-called *kernelization*, in which the dual representations of linear problems described in Chapters 2 and 3 can be modified to treat nonlinear data. Kernel methods are illustrated with a non-linear version of the principal components transformation and they will be met again in Chapter 6 when we consider support vector machines for supervised classification, in Chapter 7 in connection with anomaly detection, and in Chapter 8 in the form of a kernel K-means clustering algorithm. Finally, Section 4.5 describes Gibbs–Markov random fields which are invoked in Chapter 8 in order to include spatial context in unsupervised classification.

4.1 The convolution theorem

The *convolution* of two continuous functions $g(x)$ and $h(x)$, denoted by $h*g$, is defined by the integral

$$(h*g)(x) = \int_{-\infty}^{\infty} h(t)g(x-t)dt. \tag{4.1}$$

This definition is symmetric, i.e., $h*g = g*h$, but often one function, $g(x)$ for example, is considered to be a *signal* and the other, $h(x)$, an *instrument* *response* or *kernel* which is more local than $g(x)$ and which "smears" the signal according to the above prescription.

In the analysis of digital images, of course, we are dealing mainly with discrete signals. In order to define the discrete analogue of Equation (4.1), we will again make reference to a signal consisting of a row of pixels $g(j)$, $j = 0 \dots c-1$. The discrete convolution kernel is any array of values $h(\ell)$, $\ell = 0 \dots m-1$, where $m < c$. The array h is referred to as a *finite impulse response* (FIR) filter kernel with duration m. The discrete convolution $f = h*g$ is then defined as

$$f(j) = \begin{cases} \sum_{\ell=0}^{m-1} h(\ell)g(j-\ell) & \text{for } m-1 \le j \le c-1 \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

Discrete convolution can be performed with the function `numpy.convolve()`. The restriction on j in Equation (4.2) is necessary because of edge effects: $g(j)$ is not defined for $j < 0$. This can be circumvented in `convolve()` by setting the keyword `mode = 'valid'`.

Suppose we extend the kernel $h(\ell)$ to have the same length $m = c$ as the signal $g(j)$ by padding it with zeroes, that is, $h(\ell) = 0, \quad \ell = m \dots c-1$. Then we can write Equation (4.2), assuming edge effects have been accommodated, simply as

$$f(j) = \sum_{\ell=0}^{c-1} h(\ell)g(j-\ell), \quad j = 0 \dots c-1. \tag{4.3}$$

The following theorem provides us with a useful alternative to performing this calculation explicitly.

THEOREM 4.1

(Convolution Theorem) *In the frequency domain, convolution is replaced by multiplication, that is, $h * g \Leftrightarrow c \cdot \hat{h} \cdot \hat{g}$.*

Proof: Taking the Fourier transform of Equation (4.3), we have

$$\hat{f}(k) = \frac{1}{c}\sum_{j=0}^{c-1} f(j)e^{-\mathbf{i}2\pi kj/c} = \frac{1}{c}\sum_{\ell=0}^{c-1} h(\ell)\sum_{j=0}^{c-1} g(j-\ell)e^{-\mathbf{i}2\pi kj/c}.$$

But from the translation property, Equation (3.11),

$$\frac{1}{c}\sum_{j=0}^{c-1} g(j-\ell)e^{-\mathbf{i}2\pi kj/c} = \hat{g}(k)e^{-\mathbf{i}2\pi k\ell/c},$$

therefore

$$\hat{f}(k) = \sum_{\ell=0}^{c-1} h(\ell)e^{-\mathbf{i}2\pi k\ell/c} \cdot \hat{g}(k) = c \cdot \hat{h}(k) \cdot \hat{g}(k). \qquad \square$$

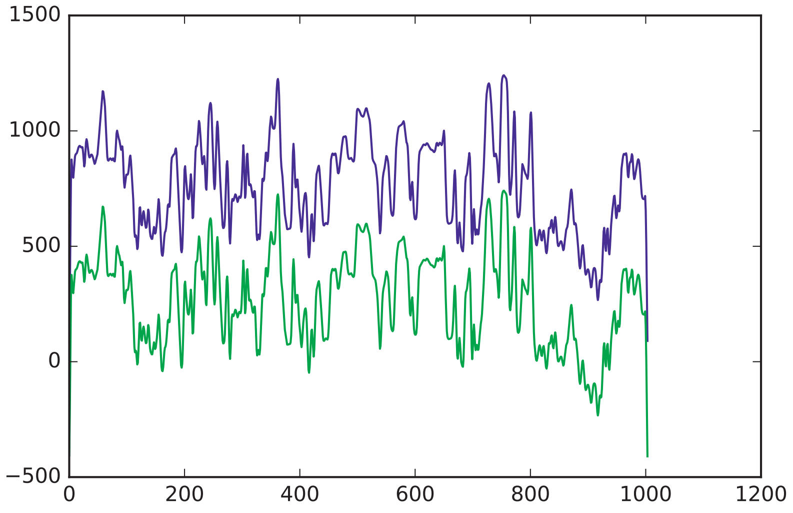


FIGURE 4.1
Illustrating the equivalence of convolution in the spatial (upper curve) and frequency (lower curve) domains.

A full statement of the theorem includes the fact that $h \cdot g \Leftrightarrow c \cdot \hat{h} * \hat{g}$, but that needn't concern us here. Theorem 4.1 says that we can carry out the convolution operation, Equation (4.3), by performing the following steps:

1. doing a Fourier transform on the signal and on the (padded) filter,
2. multiplying the two transforms together (and multiplying with c), and
3. performing the inverse Fourier transform on the result.

The FFT, as its name implies, is very fast and ordinary array multiplication is much faster than convolution. So, depending on the size of the arrays involved, convolving them in the frequency domain may be the better alternative. A pitfall when doing convolution in this fashion has to do with so-called *wraparound error*. The discrete Fourier transform assumes that both arrays are periodic. That means that the signal might overlap at the edges with a preceding or following period of the kernel, thus falsifying the result. The problem can be avoided by padding *both* arrays to $c + m - 1$, see, e.g., Gonzalez and Woods (2017), Chapter 4. The two alternative convolution procedures are illustrated in the following code, where a single row of image pixels is convolved with a smoothing filter kernel. They are seen to be completely equivalent; see Figure 4.1:

```
%matplotlib inline
import numpy as np
from numpy import fft
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly
import matplotlib.pyplot as plt

# get an image band
gdal.AllRegister()
infile = 'imagery/AST_20070501.tif'
inDataset = gdal.Open(infile,GA_ReadOnly)
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize

# pick out the middle row of pixels
band = inDataset.GetRasterBand(3)
G = band.ReadAsArray(0, rows/2, cols,1).flatten()

# define a FIR kernel of length m = 5
h = np.array([1 ,2 ,3 ,2 ,1])

# convolve in the spatial domain
Gs = np.convolve(h,G)

# pad the arrays to c + m - 1
G = np.concatenate((G,[0, 0, 0, 0]))
hp = G*0
hp[0:5] = h

# convolve in the frequency domain
Gf = fft.ifft(fft.fft (G)* fft.fft(hp))-500

x = np.array(range(1004))
plt.plot(x,Gs,x,Gf)
```

As a second example, we illustrate the use of convolution for *radar ranging*, which is also part of the SAR imaging process (Richards, 2009). In order to resolve ground features in the range direction (transverse to the direction of flight of the antenna) frequency modulated bursts (chirps), emitted and then received by the antenna after reflection from the Earth's surface, are convolved with the original signal. This allows discrimination of features on the ground even when the reflected bursts are not resolved from one another. This is mimicked in the following script; see Figure 4.2:

```
def chirp(t,t0):
    result = 0.0*t
    idx = np.array(range(2000))+t0
    tt = t[idx] - t0
    result[idx] = np.sin(2*np.pi*2e-3*(tt+1e-3*tt**2))
```

```
        return result

    t = np.array(range(5000))
    plt.plot(t,chirp(t,400)+9)
    plt.plot(t,chirp(t,800)+6)
    plt.plot(t,chirp(t,1400)+3)
    signal = chirp(t,400)+chirp(t,800)+chirp(t,1400)
    kernel = chirp(t,0)[:2000]
    kernel = kernel[::-1]
    plt.plot(t,signal)
    plt.plot(0.003*np.convolve(signal,kernel,\
                                    mode='same')-5)
    plt.xlabel('Time')
    plt.ylim((-8,12))
```

Convolution of a two-dimensional array is a straightforward extension of Equation (4.3). For a two-dimensional kernel $h(k,\ell)$ which has been appro-

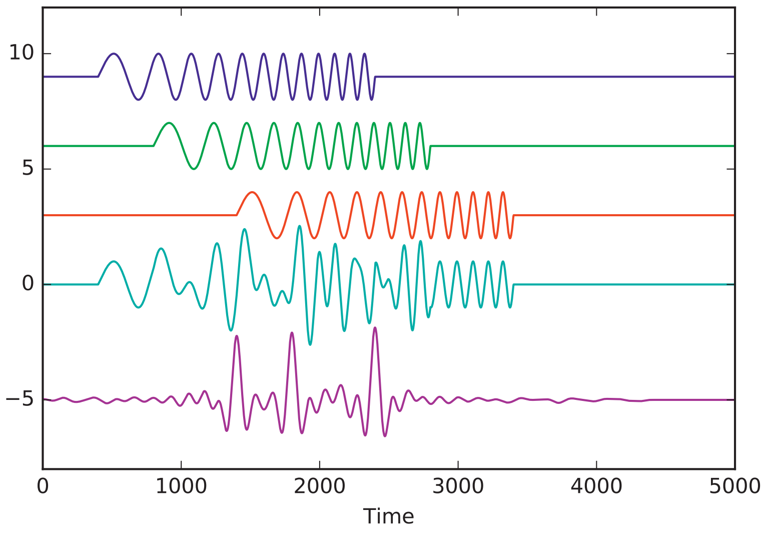


FIGURE 4.2
Illustrating radar ranging. The upper three signals represent reflections of a frequency modulated radar pulse (chirp) from three ground points lying close to one another. Their separation in time is proportional to the distances separating the ground features along the direction of pulse emission, that is, transverse to the flight direction. The fourth signal is the superposition actually received. By convolving it with the emitted signal waveform, the arrival times are resolved (bottom signal).

priately padded, the convolution with a $c \times r$ pixel array $g(i,j)$ is given by

$$f(i,j) = \sum_{k=0}^{c-1} \sum_{\ell=0}^{r-1} h(k,\ell) g(i-k, j-\ell). \tag{4.4}$$

The Convolution Theorem now reads

$$h * g \Leftrightarrow c \cdot r \cdot \hat{h} \cdot \hat{g}, \tag{4.5}$$

so that convolution can be carried out in the frequency domain using the Fourier and inverse Fourier transforms in two dimensions, Equations (3.9) and (3.8).

Convolution in two dimensions will be the basis of our discussion supervised semantic segmentation with convolutional neural networks in Chapter 7.

4.2 Linear filters

Linear filtering of images in the spatial domain generally involves moving a template across the image array, forming some specified linear combination of the pixel intensities within the template and associating the result with the coordinates of the pixel at the template's center. Specifically, for a rectangular template h of dimension $(2m+1) \times (2n+1)$,

$$f(i,j) = \sum_{k=-m}^{m} \sum_{\ell=-n}^{n} h(k,\ell) g(i+k, j+\ell), \tag{4.6}$$

where g represents the original image array and f the filtered result. The similarity of Equation (4.6) to convolution, Equation (4.4), is readily apparent and spatial filtering can be carried out in the frequency domain if desired. Whether or not the Convolution Theorem should be used to evaluate Equation (4.6) depends again on the size of the arrays involved. Richards (2012) gives a detailed discussion and calculates a cost factor

$$F = \frac{m^2}{2\log_2 c + 1} \cdot e$$

for an $m \times m$ template on a $c \times c$ image. If $F > 1$, it is economical to convolve in the frequency domain.

Linear smoothing templates are usually normalized so that $\sum_{k,\ell} h(k,\ell) = 1$. For example, the 3×3 "weighted average" filter

$$h = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{4.7}$$

might be used to suppress uninteresting small details or random noise in an image prior to intensity thresholding in order to identify larger objects. However the Convolution Theorem suggests the alternative approach of designing filters in the frequency domain right from the beginning. This is often more intuitive, since suppressing fine details in an image $g(i, j)$ is equivalent to attenuating high spatial frequencies in its Fourier representation $\hat{g}(k, \ell)$ (low-pass filtering). Conversely, enhancing detail, for instance for performing edge detection, can be done by attenuating the low frequencies in $\hat{g}(k, \ell)$ (high-pass filtering). Both effects are achieved by transforming $g(i, j)$ to $\hat{g}(k, \ell)$, choosing an appropriate form for $\hat{h}(k, \ell)$ in Equation (4.5) without reference to a spatial filter h, multiplying the two together and then doing the inverse transformation. The following code illustrates the procedure in the case of a Gaussian filter:

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from auxil import auxil1

# load the 3rd band from ASTER image
infile = 'imagery/AST_20070501.tif'
inDataset = gdal.Open(infile,GA_ReadOnly)
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize
band = inDataset.GetRasterBand(3)
G = band.ReadAsArray(0,0,cols,rows)

# Fourier transform
Gf = fft.fft2(G)

# create a Gaussian filter in frequency space
sigma = 50
Hf = auxil1.gaussfilter(sigma,1000,1000)

# low- and high-pass filtering in frequency domain
Gl = np.real(fft.ifft2(Gf*Hf))
Gh = np.real(fft.ifft2(Gf*(1.-Hf)))

# plot the filter
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
x, y = np.meshgrid(range(rows),range(cols))
ax.plot_surface(x, y, np.roll(Hf, (rows//2, cols//2),
               (0, 1)), cmap=cm.coolwarm)

# save and plot the filtered bands
from osgeo import gdal
from osgeo.gdalconst import GDT_Float32
```

```
driver = gdal.GetDriverByName('Gtiff')
outDataset = driver.Create('imagery/Gh.tif',
                     cols,rows,2,GDT_Float32)
outBand = outDataset.GetRasterBand(1)
outBand.WriteArray(Gl,0,0)
outBand = outDataset.GetRasterBand(2)
outBand.WriteArray(Gh,0,0)

%run scripts/dispms -f 'imagery/Gh.tif' -p [1,1,1] -e 3
%run scripts/dispms -f 'imagery/Gh.tif' -p [2,2,2] -e 3
```

Figure 4.3 shows the Gaussian low-pass filter

$$\hat{h}(k,\ell) = \exp(d^2/\sigma^2), \quad d^2 = \big((k-c/2)^2 + (\ell - r/2)^2\big)$$

generated by the script; see the `gaussfilter()` function in the `auxil.py` module. The high-pass filter is its complement $1-\hat{h}(k,\ell)$. Figures 4.4 and 4.5 display the result of applying them to a LANDSAT 7 ETM+ image band. We will return to the subject of low- and high-pass filters in Chapter 5, where we discuss image enhancement.

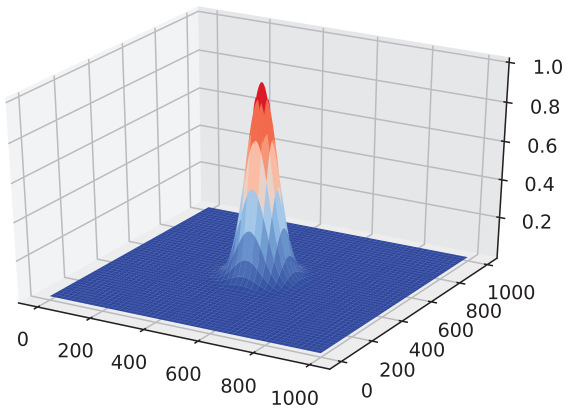


FIGURE 4.3
Gaussian filter in the frequency domain with $\sigma = 50$. Zero frequency is at the center.

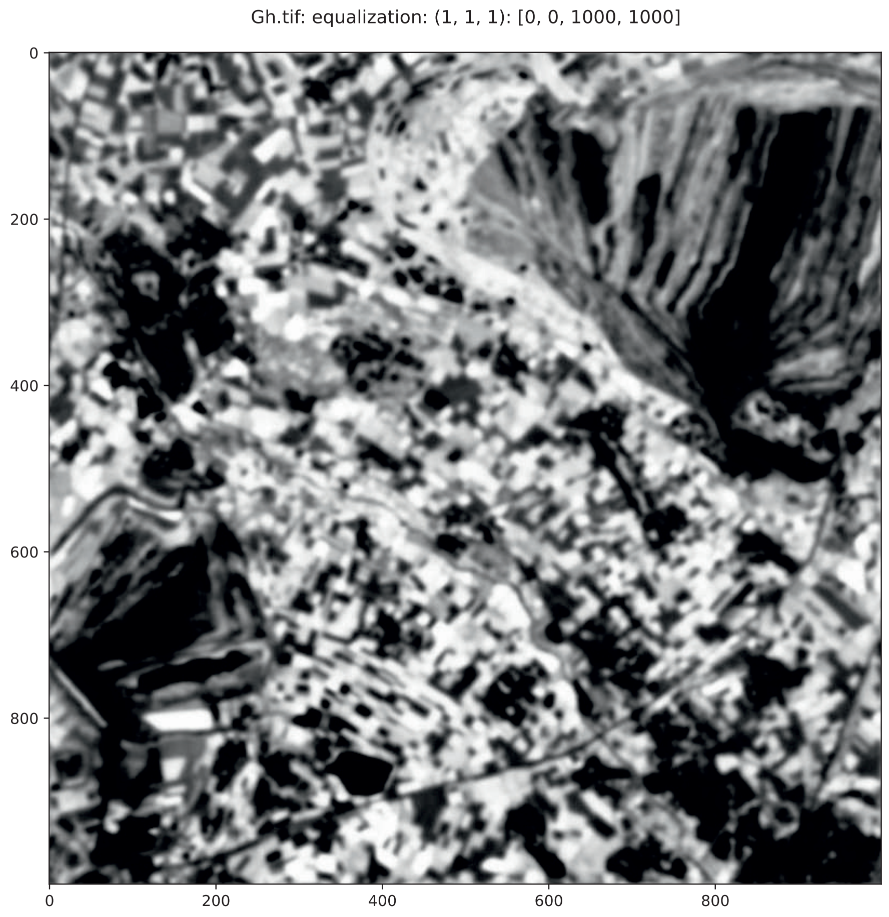


FIGURE 4.4
The third band of an ASTER image after filtering with the low-pass Gaussian filter of Figure 4.3.

4.3 Wavelets and filter banks

Using the Haar scaling function of Section 3.2.1, we were able to carry out the wavelet transformation in an equivalent vector space. In general, as was pointed out there, one can't represent scaling functions and wavelets in this way. In fact, usually all that we have to work with are the refinement coefficients of Equation (3.25) or (3.28). So how does one perform the wavelet transformation in this case?

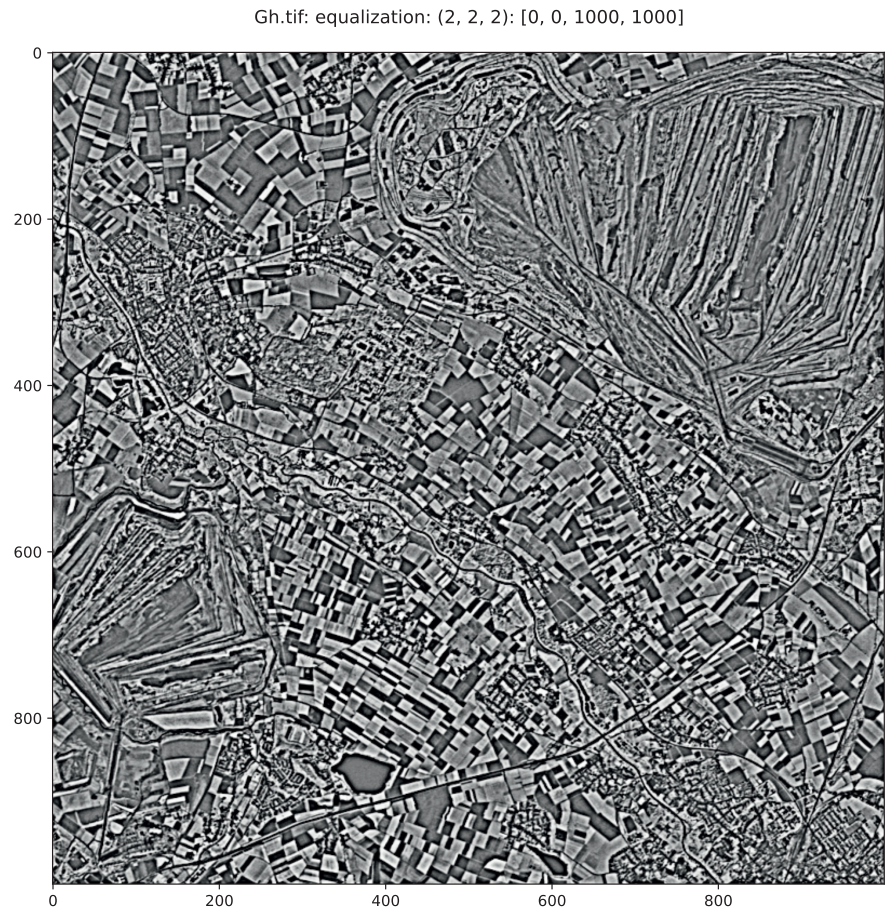


FIGURE 4.5
The third band of an ASTER image after filtering with a high-pass Gaussian filter (complement of Figure 4.3).

4.3.1 One-dimensional arrays

To answer this question, consider again a row of pixel intensities in a satellite image, which we now write in the form of a row vector $\boldsymbol{f} = (f_0, f_1, \ldots f_{c-1})$,[*] where we assume that $c = 2^n$ for some integer n.

In the multi resolution analysis (MRA) (see Definition 3.2) generated by a scaling function ϕ, such as the Daubechies D_4 scaling function of Figure 3.9, the signal $\boldsymbol{f}$ defines a function $f_n(x)$ in the subspace V_n on the interval $[0, 1]$

[*]Rather than our usual $\boldsymbol{g}$, in order to avoid confusion with the wavelet coefficients g_k.

according to

$$f_n(x) = \sum_{j=0}^{c-1} f_j \phi_{n,j}(x) = \sum_{j=0}^{c-1} f_j \phi(2^n x - j). \tag{4.8}$$

Assume that the V_n basis functions $\phi_{n,j}(x)$ are appropriately normalized:

$$\langle \phi_{n,j}(x), \phi_{n,j'}(x) \rangle = \delta_{j,j'}.$$

Now let us project $f_n(x)$ onto the subspace V_{n-1}, which has a factor of two coarser resolution. The projection is given by

$$f_{n-1}(x) = \sum_{k=0}^{c/2-1} \langle f_n(x), \phi(2^{n-1}x-k) \rangle \phi(2^{n-1}x-k) = \sum_{k=0}^{c/2-1} (H\boldsymbol{f})_k \phi(2^{n-1}x-k),$$

where we have introduced the quantity

$$(H\boldsymbol{f})_k = \langle f_n(x), \phi(2^{n-1}x - k) \rangle, \quad k = 0 \dots c/2 - 1. \tag{4.9}$$

This is the kth component of a vector $H\boldsymbol{f}$ representing the row of pixels in V_{n-1}. The notation implies that H is an operator. Its effect is to average the pixel vector $\boldsymbol{f}$ and to reduce its length by a factor of two. It is thus a kind of low-pass filter. More specifically, we have from Equation (4.8),

$$(H\boldsymbol{f})_k = \sum_{j=0}^{c-1} f_j \langle \phi(2^n x - j), \phi(2^{n-1}x - k) \rangle. \tag{4.10}$$

The dilation Equation (3.28) with normalized basis functions can be written in the form

$$\phi(2^{n-1}x - k) = \sum_{k'} h_{k'} \phi(2^n x - 2k - k'). \tag{4.11}$$

Substituting this into Equation (4.10), we have

$$\begin{aligned}(H\boldsymbol{f})_k &= \sum_{j=0}^{c-1} f_j \sum_{k'} h_{k'} \langle \phi(2^n x - j), \phi(2^n x - 2k - k') \rangle \\ &= \sum_{j=0}^{c-1} f_j \sum_{k'} h_{k'} \delta_{j,k'+2k}.\end{aligned}$$

Therefore, the filtered pixel vector has components

$$(H\boldsymbol{f})_k = \sum_{j=0}^{c-1} h_{j-2k} f_j, \quad k = 0 \dots \frac{c}{2} - 1 \ (= 2^{n-1} - 1). \tag{4.12}$$

Let us examine this vector in the case of the four non-vanishing refinement coefficients

$$h_0, h_1, h_2, h_3$$

of the Daubechies D4 wavelet. The elements of the filtered signal are

$$\begin{aligned}(H\boldsymbol{f})_0 &= h_0 f_0 + h_1 f_1 + h_2 f_2 + h_3 f_3\\(H\boldsymbol{f})_1 &= h_0 f_2 + h_1 f_3 + h_2 f_4 + h_3 f_5\\(H\boldsymbol{f})_2 &= h_0 f_4 + h_1 f_5 + h_2 f_6 + h_3 f_7\\&\vdots\end{aligned}$$

We recognize the above as the convolution $H * \boldsymbol{f}$ of the low-pass filter kernel $H = (h_3, h_2, h_1, h_0)$ (note the order!) with the vector $\boldsymbol{f}$ (see Equation (4.3)), except that *only every second result is retained*. This is referred to as *down sampling* or *decimation* and is illustrated in Figure 4.6.

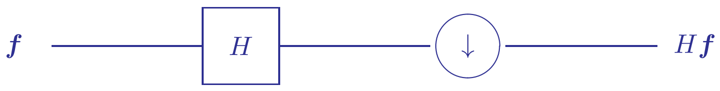


FIGURE 4.6
Schematic representation of Equation (4.12). The symbol ↓ indicates down sampling by a factor of two.

The residual, that is, the difference between the original vector $\boldsymbol{f}$ and its projection $H\boldsymbol{f}$ onto V_{n-1}, resides in the orthogonal subspace $V_{n-1}^{\perp}$. It is the projection of $f_n(x)$ onto the wavelet basis

$$\psi_{n-1,j}(x), \quad j = 0 \dots 2^{n-1} - 1.$$

A similar argument (Exercise 4) then leads us to the high-pass filter G, which projects $f_n(x)$ onto $V_{n-1}^{\perp}$ according to

$$(G\boldsymbol{f})_k = \sum_{j=0}^{c-1} g_{j-2k} f_j, \quad k = 0 \dots \frac{c}{2} - 1 = 2^{n-1} - 1. \tag{4.13}$$

The g_k are related to the h_k by Equation (3.33), i.e.,

$$g_k = (-1)^k h_{1-k}$$

so that the non-zero high-pass filter coefficients are actually

$$g_{-2} = h_3,\ g_{-1} = -h_2,\ g_0 = h_1,\ g_1 = -h_0. \tag{4.14}$$

The *concatenated vector*

$$(H\boldsymbol{f}, G\boldsymbol{f}) = (\boldsymbol{f}^1, \boldsymbol{d}^1) \tag{4.15}$$

is thus the projection of $f_n(x)$ onto $V_{n-1} \oplus V_{n-1}^{\perp}$. It has the same length as the original vector $\boldsymbol{f}$ and is an alternative representation of that vector. Its generation is illustrated in Figure 4.7 as a *filter bank.*

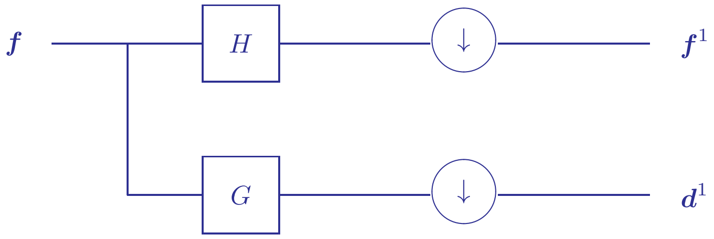


FIGURE 4.7
Schematic representation of the filter bank H, G.

The projections can be repeated on $\boldsymbol{f}^1 = H\boldsymbol{f}$ to obtain the projection

$$(H\boldsymbol{f}^1, G\boldsymbol{f}^1, G\boldsymbol{f}) = (\boldsymbol{f}^2, \boldsymbol{d}^2, \boldsymbol{d}^1) \tag{4.16}$$

onto $V_{n-2} \oplus V_{n-2}^{\perp} \oplus V_{n-1}^{\perp}$, and so on until the complete wavelet transformation has been obtained. Since the filtering process is applied recursively to arrays which are, at each application, reduced by a factor of two, the procedure is very fast. It constitutes *Mallat's algorithm* (Mallat, 1989) and is also referred to as the *fast wavelet transform, discrete wavelet transform* or *pyramid algorithm.*

The original vector can be reconstructed at any stage by applying the inverse operators H^* and G^*. For recovery from $(\boldsymbol{f}^1, \boldsymbol{d}^1)$, these are defined by

$$(H^*\boldsymbol{f}^1)_k = \sum_{j=0}^{c/2-1} h_{k-2j}\boldsymbol{f}_j^1, \quad k = 0 \ldots c-1 = 2^n - 1, \tag{4.17}$$

$$(G^*\boldsymbol{d}^1)_k = \sum_{j=0}^{c/2-1} g_{k-2j}\boldsymbol{d}_j^1, \quad k = 0 \ldots c-1 = 2^n - 1, \tag{4.18}$$

with analogous definitions for the other stages. To understand what's happening, consider the elements of the filtered vector in Equation (4.17). These are

$$\begin{aligned}
(H^*\boldsymbol{f}^1)_0 &= h_0 f_0^1\\
(H^*\boldsymbol{f}^1)_1 &= h_1 f_0^1\\
(H^*\boldsymbol{f}^1)_2 &= h_2 f_0^1 + h_0 f_1^1\\
(H^*\boldsymbol{f}^1)_3 &= h_3 f_0^1 + h_1 f_1^1\\
(H^*\boldsymbol{f}^1)_4 &= h_2 f_1^1 + h_0 f_2^1\\
(H^*\boldsymbol{f}^1)_5 &= h_3 f_1^1 + h_1 f_2^1\\
&\vdots
\end{aligned}$$

This is just the convolution of the filter $H^* = (h_0, h_1, h_2, h_3)$ with the vector

$$f_0^1, 0, f_1^1, 0, f_2^1, 0 \ldots f_{c/2-1}^1, 0,$$

which is called an *up sampled* array. The filter of Equation (4.17) is represented schematically in Figure 4.8.

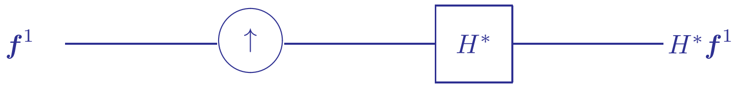


FIGURE 4.8
Schematic representation of the filter H^*. The symbol $\uparrow$ indicates up sampling by a factor of two.

Equation (4.18) is interpreted in a similar way. Finally we add the two results to get the original pixel vector:

$$H^*\boldsymbol{f}^1 + G^*\boldsymbol{d}^1 = \boldsymbol{f}. \tag{4.19}$$

To see this, write the equation out for a particular value of k:

$$(H^*\boldsymbol{f}^1)_k + (G^*\boldsymbol{d}^1)_k = \sum_{j=0}^{c/2-1} h_{k-2j}\left[\sum_{j'=0}^{c-1} h_{j'-2j} f_{j'} + g_{k-2j}\sum_{j'=0}^{c-1} g_{j'-2j} f_{j'}\right].$$

Combining terms and interchanging the summations, we get

$$(H^*\boldsymbol{f}^1)_k + (G^*\boldsymbol{d}^1)_k = \sum_{j'=0}^{c-1} f_{j'} \sum_{j=0}^{c/2-1} [h_{k-2j}h_{j'-2j} + g_{k-2j}g_{j'-2j}].$$

Now, using $g_k = (-1)^k h_{1-k}$,

$$(H^*\boldsymbol{f}^1)_k + (G^*\boldsymbol{d}^1)_k = \sum_{j'=0}^{c-1} f_{j'} \sum_{j=0}^{c/2-1} [h_{k-2j}h_{j'-2j} + (-1)^{k+j'} h_{1-k+2j}h_{1-j'+2j}].$$

With the help of Equations (3.26) and (3.27), it is easy to show that the second summation above is just $\delta_{j'k}$. For example, suppose k is even. Then

$$\sum_{j=0}^{c/2-1} [h_{k-2j}h_{j'-2j} + (-1)^{k+j'} h_{1-k+2j}h_{1-j'+2j}] =$$
$$h_0 h_{j'-k} + h_2 h_{j'-k+2} + (-1)^{j'} [h_1 h_{1-j'+k} + h_3 h_{3-j'+k}].$$

If $j' = k$, the right-hand side reduces to

$$h_0^2 + h_1^2 + h_2^2 + h_3^2 = 1,$$

from Equation (3.26) and the fact that $h_k = c_k/\sqrt{2}$. For any other value of j', the expression is zero. Therefore, we can write

$$(H^* \boldsymbol{f}^1)_k + (G^* \boldsymbol{d}^1)_k = \sum_{j'=0}^{c-1} f_{j'}\delta_{j'k} = f_k, \quad k = 0 \ldots c-1, \tag{4.20}$$

as claimed. The reconstruction of the original vector from $\boldsymbol{f}^1$ and $\boldsymbol{d}^1$ is shown in Figure 4.9 as a *synthesis bank.*

4.3.2 Two-dimensional arrays

The extension of the procedure to two-dimensional arrays is straightforward. Figure 4.10 shows an application of the filters H and G to the rows and columns of an image array $f_n(i,j)$ at scale n. The image is filtered and down sampled into four quadrants:

- f_{n-1}, the result of applying the low-pass filter H to both rows and columns plus down sampling;
- C^H_{n-1}, the result of applying the low-pass filter H to the columns and then the high-pass filter G to the rows plus down sampling;

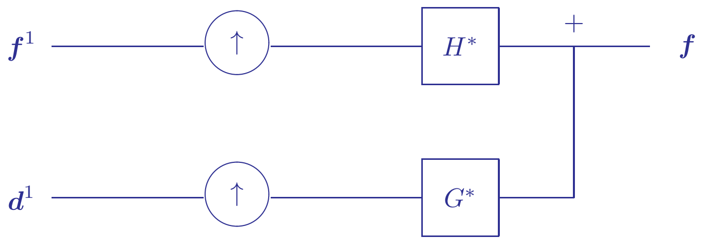


FIGURE 4.9
Schematic representation of the synthesis bank H^*, G^*.

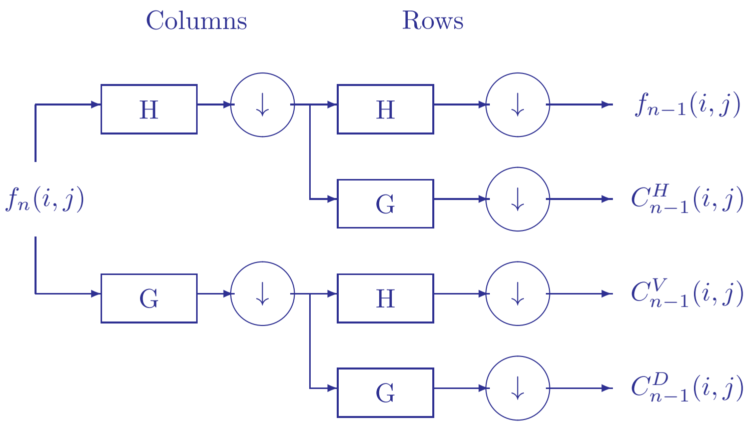


FIGURE 4.10
Wavelet filter bank. H is a low-pass filter and G a high-pass filter derived from the refinement coefficients of the wavelet transformation. The symbol ↓ indicates down sampling by a factor of two.

- C^V_{n-1}, the result of applying the high-pass filter G to the columns and then the low-pass filter H to the rows plus down sampling; and
- C^D_{n-1}, the result of applying the high-pass filter G to both the columns and the rows plus down sampling.

The original image can be losslessly recovered by inverting the filter as in Figure 4.8.

The filter bank of Figure 4.10 (and the corresponding synthesis filter bank) is implemented in Python as the object class `DWTArray`, which uses the Daubechies D4 MRA and is included in the `auxil.auxil1.py` module; see Appendix C. An excerpt is shown in Listing 4.1 in which the forward transformation, the method `filter()`, is implemented.

The following code illustrates the DWT filter with the ASTER image band 3 (in the variable `G`) from Section 4.2:

```python
from auxil.auxil import DWTArray

# instantiate a DWTArray object
dwtarr = DWTArray(G,1000,1000)
data0 = np.copy(dwtarr.data)

# filter once
dwtarr.filter()
data1 = np.copy(dwtarr.data)
quad1 = np.abs(dwtarr.get_quadrant(1))
```

Listing 4.1: The `filter` method for the Python object `DWTArray` (excerpt from `auxil.auxil1.py`).

```
    def filter(self):
    #   single application of filter bank
        if self.num_iter == self.max_iter:
            return 0
    #   get upper left quadrant
        m = self.lines//2**self.num_iter
        n = self.samples//2**self.num_iter
        f0 = self.data[:m, :n]
    #   temporary arrays
        f1 = np.zeros((m//2, n))
        g1 = np.zeros((m//2, n))
        ff1 = np.zeros((m//2, n//2))
        fg1 = np.zeros((m//2, n//2))
        gf1 = np.zeros((m//2, n//2))
        gg1 = np.zeros((m//2, n//2))
    #   filter columns and downsample
        ds = np.asarray(range(m//2))*2+1
        for i in range(n):
            temp = np.convolve(f0[:, i].ravel(),\
                                     self.H, 'same')
            f1[:, i] = temp[ds]
            temp = np.convolve(f0[:, i].ravel(),\
                                       self.G, 'same')
            g1[:, i] = temp[ds]
    #   filter rows and downsample
        ds = np.asarray(range(n//2))*2+1
        for i in range(m//2):
            temp=np.convolve(f1[i, :], self.H, 'same')
            ff1[i, :] = temp[ds]
            temp=np.convolve(f1[i, :], self.G, 'same')
            fg1[i, :] = temp[ds]
            temp=np.convolve(g1[i, :], self.H, 'same')
            gf1[i, :] = temp[ds]
            temp=np.convolve(g1[i, :], self.G, 'same')
            gg1[i, :]=temp[ds]
        f0[:m//2, :n//2] = ff1
        f0[:m//2, n//2:] = fg1
        f0[m//2:, :n//2] = gf1
        f0[m//2:, n//2:] = gg1
        self.data[:m, :n] = f0
        self.num_iter = self.num_iter+1
```

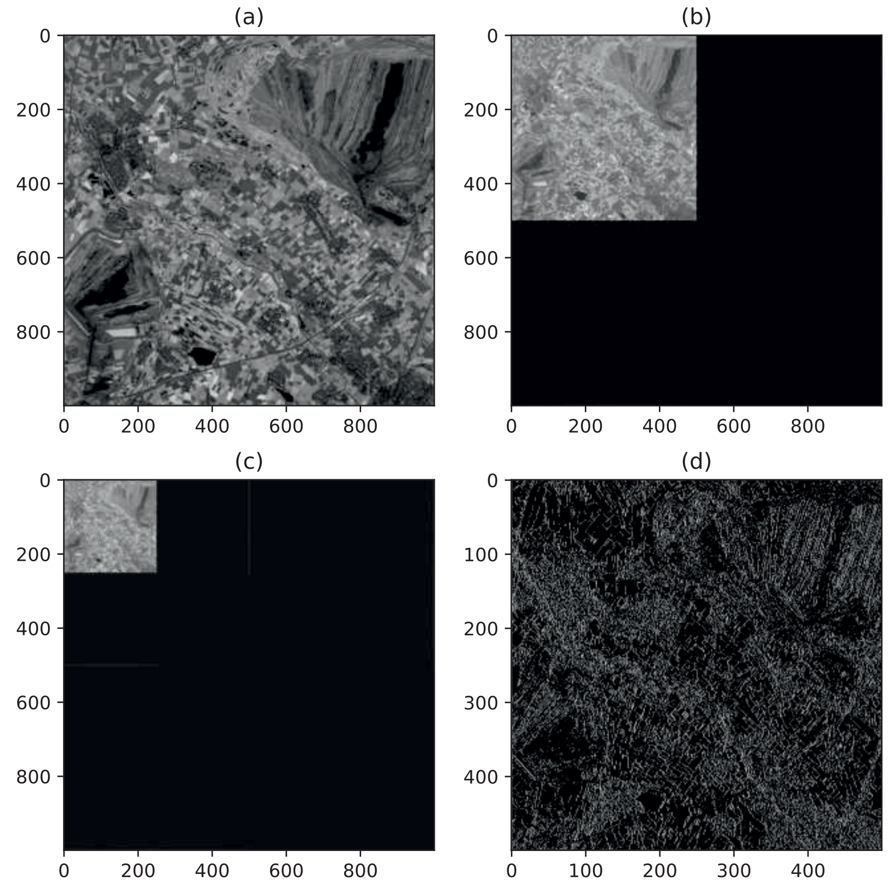


FIGURE 4.11
Application of the filter bank of Figure 4.10 to an image band. (a) The original 1000×1000 image. When padded out to 1024×1024 ($1024 = 2^{10}$), it is a two-dimensional function $f(x, y)$ in $V_{10} \otimes V_{10}$. (b) Single application of the filter bank. The result of low-pass filtering the rows and columns is in the upper left-hand quadrant, i.e., the projection of f onto $V_9 \otimes V_9$. The other three quadrants represent the high-pass projections onto the orthogonal subspaces $V_9^\perp \otimes V_9$ (upper right), $V_9 \otimes V_9^\perp$ (lower left), and $V_9^\perp \otimes V_9^\perp$ (lower right). (c) The result of a second application of the filter bank. (d) Logarithm of the wavelet coefficients in $V_9^\perp \otimes V_9$.

```
# filter again
dwtarr.filter()
data2 = dwtarr.data

# plot
f, ax = plt.subplots(2,2,figsize=(8,8))
ax[0,0].imshow(data0,cmap=cm.gray)
ax[0,0].set_title('(a)')
ax[0,1].imshow(data1,cmap=cm.gray)
ax[0,1].set_title('(b)')
ax[1,0].imshow(data2,cmap=cm.gray)
ax[1,0].set_title('(c)')
ax[1,1].imshow(np.log(quad1-np.min(quad1)+1e-6),
                      cmap=cm.gray)
ax[1,1].set_title('(d)')
```

The images generated are shown in Figure 4.11. Thus `DWTArray()` produces *pyramid representations* of image bands, which occupy the same amount of storage as the original image array.

The discrete wavelet transformation will be made use of in Chapter 5 for image sharpening and in Chapter 8 for unsupervised classification.

4.4 Kernel methods

In Section 2.6.4, it was shown that regularized linear regression (ridge regression) possesses a dual formulation in which observation vectors $\boldsymbol{x}(\nu)$, $\nu = 1 \dots m$, only enter in the form of inner products $\boldsymbol{x}(\nu)^\top \boldsymbol{x}(\nu')$. A similar dual representation was found in Section 3.3.4 for the principal components transformation. In both cases, the symmetric, positive semi-definite Gram matrix

$$\boldsymbol{\mathcal{X}}\boldsymbol{\mathcal{X}}^\top, \quad \boldsymbol{\mathcal{X}} = \text{ data matrix} = \begin{pmatrix} \boldsymbol{x}(1)^\top \\ \vdots \\ \boldsymbol{x}(m)^\top \end{pmatrix}$$

played a central role. It turns out that many linear methods in pattern recognition have dual representations, ones which can be exploited to extend well-established linear theories to treat non-linear data. One speaks in this context of *kernel methods* or *kernelization*. An excellent reference for kernel methods is Shawe-Taylor and Cristianini (2004). In the following, we outline the basic ideas and then illustrate them with a nonlinear, or kernelized, version of principal components analysis.

4.4.1 Valid kernels

Suppose that $\boldsymbol{\phi}(\boldsymbol{g})$ is some nonlinear function which maps the original N-dimensional Euclidean input space of the observation vectors $\boldsymbol{g}$ (image pixels) to some non-linear, usually higher-dimensional, inner product space $\mathcal{H}$ (see Definition3.1),

$$\boldsymbol{\phi} : \mathbb{R}^N \mapsto \mathcal{H}. \tag{4.21}$$

The mapping takes a data matrix $\boldsymbol{\mathcal{G}}$ into a new data matrix $\boldsymbol{\Phi}$ given by

$$\boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}\big(\boldsymbol{g}(1)\big)^\top \\ \vdots \\ \boldsymbol{\phi}\big(\boldsymbol{g}(m)\big)^\top \end{pmatrix}. \tag{4.22}$$

This matrix has dimension $m \times p$, where $p \geq N$ is the (possibly infinite) dimension of the nonlinear feature space $\mathcal{H}$.

DEFINITION 4.1 *A* valid kernel *is a function κ that, for all $\boldsymbol{g}, \boldsymbol{g}' \in \mathbb{R}^N$, satisfies*

$$\kappa(\boldsymbol{g}, \boldsymbol{g}') = \boldsymbol{\phi}(\boldsymbol{g})^\top \boldsymbol{\phi}(\boldsymbol{g}'), \tag{4.23}$$

where $\boldsymbol{\phi}$ is given by Equation (4.21). For a set of observations $\boldsymbol{g}(\nu)$, $\nu = 1 \dots m$, the $m \times m$ matrix $\boldsymbol{\mathcal{K}}$ with elements $\kappa(\boldsymbol{g}(\nu), \boldsymbol{g}(\nu'))$ is called the kernel matrix*.*

Note that with Equation (4.22), we can write the kernel matrix equivalently in the form

$$\boldsymbol{\mathcal{K}} = \boldsymbol{\Phi}\boldsymbol{\Phi}^\top. \tag{4.24}$$

In Section 2.6.4, we saw that the Gram matrix is symmetric and positive semi-definite. This is also the case for kernel matrices since, for any m-component vector $\boldsymbol{x}$,

$$\begin{aligned} \boldsymbol{x}^\top \boldsymbol{\mathcal{K}} \boldsymbol{x} &= \sum_{\nu,\nu'} x_\nu x_{\nu'} \mathcal{K}_{\nu\nu'} = \sum_{\nu,\nu'} x_\nu x_{\nu'} \boldsymbol{\phi}_\nu^\top \boldsymbol{\phi}_{\nu'} \\ &= (\sum_\nu x_\nu \boldsymbol{\phi}_\nu)^\top (\sum_{\nu'} x_{\nu'} \boldsymbol{\phi}_{\nu'}) = \Big\| \sum_\nu x_\nu \boldsymbol{\phi}_\nu \Big\|^2 \geq 0. \end{aligned}$$

Positive semi-definiteness of the kernel matrix is in fact a necessary and sufficient condition for any symmetric function $\kappa(\boldsymbol{g}, \boldsymbol{g}')$ to be a valid kernel in the sense of Definition 4.1. This is stated in the following theorem (Shawe-Taylor and Cristianini, 2004):

THEOREM 4.2

Let $k(\boldsymbol{g}, \boldsymbol{g}')$ be a symmetric function on the space of observations, that is, $k(\boldsymbol{g}, \boldsymbol{g}') = k(\boldsymbol{g}', \boldsymbol{g})$. Let $\{\boldsymbol{g}(\nu) \mid \nu = 1 \dots m\}$ be any finite subset of the input space $\mathbb{R}^N$ and define the matrix $\boldsymbol{\mathcal{K}}$ with elements

$$(\boldsymbol{\mathcal{K}})_{\nu\nu'} = k(\boldsymbol{g}(\nu), \boldsymbol{g}(\nu')), \quad \nu, \nu' = 1 \dots m.$$

Listing 4.2: Calculating a Gaussian (or linear) kernel matrix (excerpt from the package `auxil.auxil1.py`).

```
def kernelMatrix(X, Y=None, gam=None, k=0):
    if Y is None:
        Y = X
    if k == 0:
        X = np.mat(X)
        Y = np.mat(Y)
        return (X*(Y.T), 0)
    else:
        m = X[:, 0].size
        n = Y[:, 0].size
        onesm = np.mat(np.ones(m))
        onesn = np.mat(np.ones(n))
        # d = X^2
        d = np.mat(np.sum(X*X, axis=1)).T*onesn
        # d = X^2 + Y^2
        d = d + onesm.T*np.mat(np.sum(Y*Y, axis=1))
        # d = X^2 + Y^2 - 2XY
        d = d - 2*np.mat(X)*np.mat(Y).T
        if gam is None:
            sigma = np.sum(np.sqrt(abs(d)))/(m**2-m)
            gam = 1/(2*(n*sigma)**2)
        return (np.exp(-gam*d), gam)
```

Then $k(\boldsymbol{g}, \boldsymbol{g}')$ is a valid kernel if and only if $\boldsymbol{\mathcal{K}}$ is positive semi-definite.

The motivation for using valid kernels is that it allows us to apply known linear methods to nonlinear data simply by replacing the inner products in the dual formulation by an appropriate non-linear, valid kernel. One implication of Theorem 4.2 is that one can obtain valid kernels without even specifying the non-linear mapping $\boldsymbol{\phi}$ at all. In fact, it is possible to build up whole families of valid kernels in which the associated mappings are defined implicitly and are otherwise unknown. An important example, one which we will use later to illustrate kernel methods, is the *Gaussian kernel* given by

$$\kappa_{\text{rbf}}(\boldsymbol{g}, \boldsymbol{g}') = \exp(-\gamma \|\boldsymbol{g} - \boldsymbol{g}'\|^2). \tag{4.25}$$

This is an instance of a *homogeneous kernel*, also called *radial basis kernel*, one which depends only on the Euclidean distance between the observations. It is equivalent to the inner product of two infinite-dimensional feature vector mappings $\boldsymbol{\phi}(\boldsymbol{g})$ (Exercise 7). Listing 4.2 shows a Python function to calculate either a Gaussian or a linear kernel matrix. The parameter γ defaults to $1/2(n\sigma)^2$, where σ is equal to the mean distance between the observations in input space, i.e., the average over all test observations of $\|\boldsymbol{g}(\nu) - \boldsymbol{g}(\nu')\|$, $\nu \neq \nu'$ and n is an adjustable scaling factor.

If we wish to make use of kernels which, like the Gaussian kernel, imply non-linear mappings $\boldsymbol{\phi}(\boldsymbol{g})$ to which we have no direct access, then clearly we must work exclusively with inner products

$$\kappa(\boldsymbol{g},\boldsymbol{g}') = \boldsymbol{\phi}(\boldsymbol{g})^\top \boldsymbol{\phi}(\boldsymbol{g}').$$

Apart from dual formulations which only involve these inner products, it turns out that many other properties of the mapped observations $\boldsymbol{\phi}(\boldsymbol{g})$ can also be expressed purely in terms of the elements $\kappa(\boldsymbol{g},\boldsymbol{g}')$ of the kernel matrix. Thus the norm or length of the mapping of $\boldsymbol{g}$ in the nonlinear feature space $\mathcal{H}$ is

$$\|\boldsymbol{\phi}(\boldsymbol{g})\| = \sqrt{\boldsymbol{\phi}(\boldsymbol{g})^\top \boldsymbol{\phi}(\boldsymbol{g})} = \sqrt{\kappa(\boldsymbol{g},\boldsymbol{g})}. \tag{4.26}$$

The squared distance between any two points in $\mathcal{H}$ is given by

$$\begin{aligned}\|\boldsymbol{\phi}(\boldsymbol{g}) - \boldsymbol{\phi}(\boldsymbol{g}')\|^2 &= (\boldsymbol{\phi}(\boldsymbol{g}) - \boldsymbol{\phi}(\boldsymbol{g}'))^\top(\boldsymbol{\phi}(\boldsymbol{g}) - \boldsymbol{\phi}(\boldsymbol{g}')) \\ &= \boldsymbol{\phi}(\boldsymbol{g})^\top\boldsymbol{\phi}(\boldsymbol{g}) + \boldsymbol{\phi}(\boldsymbol{g}')^\top\boldsymbol{\phi}(\boldsymbol{g}') - 2\boldsymbol{\phi}(\boldsymbol{g})^\top\boldsymbol{\phi}(\boldsymbol{g}') \\ &= \kappa(\boldsymbol{g},\boldsymbol{g}) + \kappa(\boldsymbol{g}',\boldsymbol{g}') - 2\kappa(\boldsymbol{g},\boldsymbol{g}').\end{aligned} \tag{4.27}$$

The $1\times p$ row vector $\bar{\boldsymbol{\phi}}^\top$ of column means of the mapped data matrix $\boldsymbol{\Phi}$, Equation (4.22), can be written in matrix form as, see Equation (2.56),

$$\bar{\boldsymbol{\phi}}^\top = \mathbf{1}_m^\top \boldsymbol{\Phi}/m, \tag{4.28}$$

where $\mathbf{1}_m$ is a column vector of m ones. The norm of the mean vector $\bar{\boldsymbol{\phi}}$ is thus

$$\|\bar{\boldsymbol{\phi}}\| = \sqrt{\bar{\boldsymbol{\phi}}^\top\bar{\boldsymbol{\phi}}} = \frac{1}{m}\sqrt{\mathbf{1}_m^\top\boldsymbol{\Phi}\boldsymbol{\Phi}^\top\mathbf{1}_m} = \frac{1}{m}\sqrt{\mathbf{1}_m^\top\boldsymbol{\mathcal{K}}\mathbf{1}_m}.$$

We are even able to determine the elements of the kernel matrix $\tilde{\boldsymbol{\mathcal{K}}}$ which corresponds to a column centered data matrix $\tilde{\boldsymbol{\Phi}}$ (see Section 2.3.1). The rows of $\tilde{\boldsymbol{\Phi}}$ are

$$\tilde{\boldsymbol{\phi}}(\boldsymbol{g}(\nu)) = (\boldsymbol{\phi}(\boldsymbol{g}(\nu)) - \bar{\boldsymbol{\phi}})^\top = \boldsymbol{\phi}(\boldsymbol{g}(\nu))^\top - \bar{\boldsymbol{\phi}}^\top, \quad \nu = 1\dots m. \tag{4.29}$$

With Equation (4.28), the $m\times p$ matrix of m repeated column means of $\boldsymbol{\Phi}$ is given by

$$\begin{pmatrix}\bar{\boldsymbol{\phi}}^\top \\ \bar{\boldsymbol{\phi}}^\top \\ \vdots \\ \bar{\boldsymbol{\phi}}^\top\end{pmatrix} = \mathbf{1}_{mm}\boldsymbol{\Phi}/m,$$

where $\mathbf{1}_{mm}$ is an $m\times m$ matrix of ones, so that Equation (4.29) can be written in matrix form:

$$\tilde{\boldsymbol{\Phi}} = \boldsymbol{\Phi} - \mathbf{1}_{mm}\boldsymbol{\Phi}/m. \tag{4.30}$$

Therefore,

$$\begin{aligned}\tilde{\mathcal{K}} = \tilde{\mathbf{\Phi}}\tilde{\mathbf{\Phi}}^\top &= (\mathbf{\Phi} - \mathbf{1}_{mm}\mathbf{\Phi}/m)(\mathbf{\Phi} - \mathbf{1}_{mm}\mathbf{\Phi}/m)^\top \\ &= \mathbf{\Phi}\mathbf{\Phi}^\top - \mathbf{\Phi}\mathbf{\Phi}^\top\mathbf{1}_{mm}/m - \mathbf{1}_{mm}\mathbf{\Phi}\mathbf{\Phi}^\top/m - 1_{mm}\mathbf{\Phi}\mathbf{\Phi}^\top\mathbf{1}_{mm}/m^2 \\ &= \mathcal{K} - \mathcal{K}\mathbf{1}_{mm}/m - \mathbf{1}_{mm}\mathcal{K}/m + \mathbf{1}_{mm}\mathcal{K}\mathbf{1}_{mm}/m^2.\end{aligned} \tag{4.31}$$

Examining this equation component-wise we see that, to center the kernel matrix, we subtract from each element $(\mathcal{K})_{ij}$ the mean of the ith row and the mean of the jth column and add to that the mean of the entire matrix. The following Python function, taken from the module `auxil.auxil1.py`, takes a kernel matrix $\mathcal{K}$ as input and returns the column-centered kernel matrix $\tilde{\mathcal{K}}$:

```
def center(K):
    m = K[:,0].size
    Imm = np.mat(np.ones((m,m)))
    return K - (Imm*K + K*Imm - np.sum(K)/m)/m
```

4.4.2 Kernel PCA

In Section 3.3.4, it was shown that, in the dual formulation of the principal components transformation, the projection $P_i[\boldsymbol{g}]$ of an observation $\boldsymbol{g}$ along a principal axis $\boldsymbol{w}_i$ can be expressed as

$$P_i[\boldsymbol{g}] = \boldsymbol{w}_i^\top\boldsymbol{g} = \sum_{\nu=1}^{m}(\boldsymbol{\alpha}_i)_\nu\ \boldsymbol{g}(\nu)^\top\boldsymbol{g}, \quad i = 1\dots N. \tag{4.32}$$

In this equation, the dual vectors $\boldsymbol{\alpha}_i$ are determined by the eigenvectors $\boldsymbol{v}_i$ and eigenvalues λ_i of the Gram matrix $\mathcal{G}\mathcal{G}^\top$ according to

$$\boldsymbol{\alpha}_i = \lambda_i^{-1/2}\boldsymbol{v}_i, \quad i = 1\dots m. \tag{4.33}$$

Kernelization of principal components analysis simply involves replacing the Gram matrix $\mathcal{G}\mathcal{G}^\top$ by the (centered) kernel matrix $\tilde{\mathcal{K}}$, Equation (4.31), and the inner products $\boldsymbol{g}(\nu)^\top\boldsymbol{g}$ in Equation (4.32) by the corresponding kernel function. The projection along the ith principal axes in the nonlinear feature space $\mathcal{H}$ (the ith non-linear principal component) is then

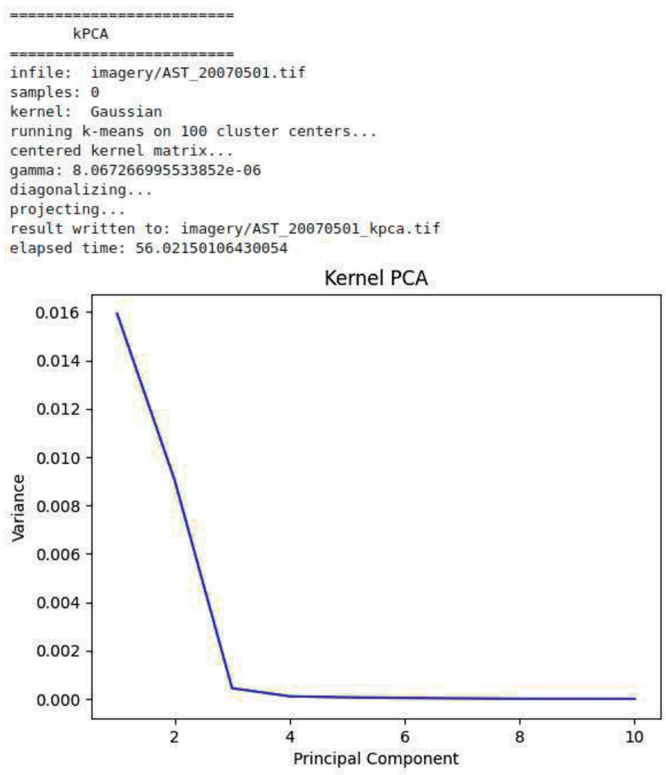
$$P_i[\boldsymbol{\phi}(\boldsymbol{g})] = \sum_{\nu=1}^{m}(\boldsymbol{\alpha}_i)_\nu\kappa(\boldsymbol{g}(\nu), \boldsymbol{g}), \quad i = 1\dots m, \tag{4.34}$$

where the dual vectors $\boldsymbol{\alpha}_i$ are still given by Equation (4.33), but $\boldsymbol{v}_i$ and λ_i, $i = 1\dots m$, are the eigenvectors and eigenvalues of the kernel matrix $\tilde{\mathcal{K}}$. The variance of the projections is given by (see Chapter 3, Exercise 15)

$$\mathrm{var}(P_i[\boldsymbol{\phi}(\boldsymbol{g})]) = \frac{\lambda_i}{m-1}, \quad i = 1\dots m.$$

Schölkopf et al. (1998) were the first to introduce kernel PCA and Shawe-Taylor and Cristianini (2004) analyze the method in detail. Canty and Nieslsen (2012) discuss it in the context of linear and kernel change detection methods.

In the analysis of remote sensing imagery, the number of observations is in general very large (order $10^6 - 10^8$), so that diagonalization of the kernel matrix is only feasible if the image is sampled. The sampled pixel vectors $\boldsymbol{g}(\nu)$, $\nu = 1 \dots m$, are then referred to as *training data* and the calculation of the sampled kernel matrix and its centering/diagonalization constitute the *training phase*. A realistic upper limit on m would appear to be about 2000. Diagonalization of a 2000×2000 symmetric matrix on, e.g., a PC workstation with Python/NumPy will generally not require page swapping and takes the order of minutes, with Python/TensorFlow and GPU hardware considerably faster. However, this is a very small sample ($\lessapprox 10^{-3}$). Kwon and Nasrabadi (2005) suggest extracting the most representative samples with a clustering



```
=========================
       kPCA
=========================
infile:  imagery/AST_20070501.tif
samples: 0
kernel:  Gaussian
running k-means on 100 cluster centers...
centered kernel matrix...
gamma: 8.067266995533852e-06
diagonalizing...
projecting...
result written to: imagery/AST_20070501_kpca.tif
elapsed time: 56.02150106430054
```

FIGURE 4.12
Applying kernel PCA on an ASTER image.

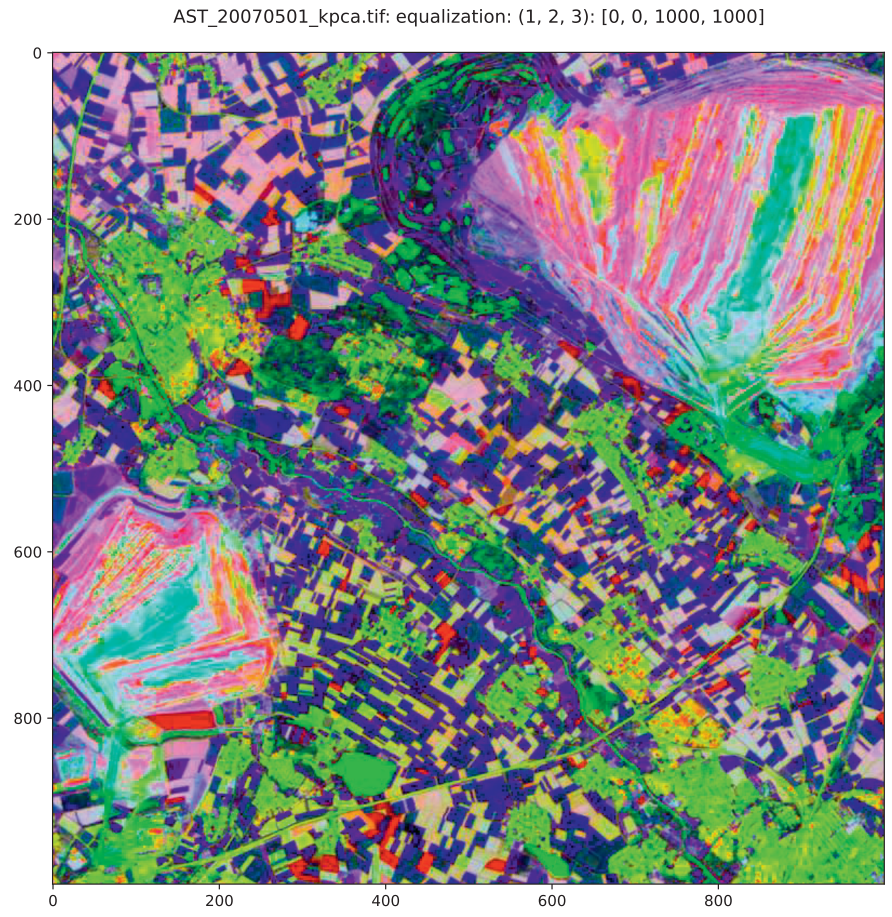

AST_20070501_kpca.tif: equalization: (1, 2, 3): [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800

FIGURE 4.13
RGB composite of the first 3 kernel principal components for an Aster image.

algorithm such as K-means (see Chapter 8). The cluster mean vectors serve as the training data and may just number a few hundred.

After diagonalization, the *generalization phase* involves the projection of each image pixel vector according to Equation (4.34). This means, for every pixel $\boldsymbol{g}$, recalculation of the kernels $\kappa(\boldsymbol{g}(\nu), \boldsymbol{g})$ for $\nu = 1 \dots m$. This is not the case for linear PCA, where we don't require the training data to project new observations.* For an image with n pixels, there are, therefore, $m \times n$ kernels involved, generally much too large an array to be held in memory, so that it is advisable to read in and project the image pixels row-by-row.

*For this reason, kernel PCA is said to be *memory-based*.

See Appendix C for documentation of the Python script `kpca.py` for performing kernel PCA on multispectral imagery using the Gaussian kernel, Equation (4.25). Figure 4.12 shows the Jupyter notebook output cell for kernel PCA performed on an ASTER image using K-means clustering for sampling:

```
run scripts/kpca -s 0 imagery/AST_20070501.tif
```

The option `-s 0` defines the desired number of random samples to be zero, which triggers as default the K-means algorithm with 100 clusters. An RGB composite of the first three kernel principal components is shown in Figure 4.13. In Chapter 8, we will look at a kernelized version of the K-means algorithm itself and Chapter 9 will illustrate the use of kernel PCA for change detection with (simulated) nonlinear data.

4.5 Gibbs–Markov random fields

Random fields are frequently invoked to describe prior expectations in a Bayesian approach to image analysis (Winkler, 1995; Li, 2001). The following brief introduction will serve to make their use more plausible in the unsupervised land cover classification context that we will meet in Chapter 8. The development adheres closely to Li (2001), but in a notation specific to that used later in the treatment of image classification.

Image classification is a problem of *labeling*: Given an observation, that is, a pixel intensity vector, we ask: “Which class label should be assigned to it?” If the observations are assumed to have no spatial context, then the labeling will consist of partitioning the pixels into K disjoint subsets according to some decision criterion, K being the number of land cover categories present. If spatial context within the image is also to be taken into account, the labeling will take place on what is referred to as a *regular lattice*.

A regular lattice representing an image with c columns and r rows is the discrete set of sites

$$\mathcal{I} = \{(i,j) \mid 0 \le i \le c-1, 0 \le j \le r-1\}.$$

By re-indexing, we can write this in a more convenient, linear form

$$\mathcal{I} = \{i \mid 1 \le i \le n\},$$

where $n = rc$ is the number of pixels. The interrelationship between the sites is governed by a *neighborhood system*

$$\mathcal{N} = \{\mathcal{N}_i \mid i \in \mathcal{I}\},$$

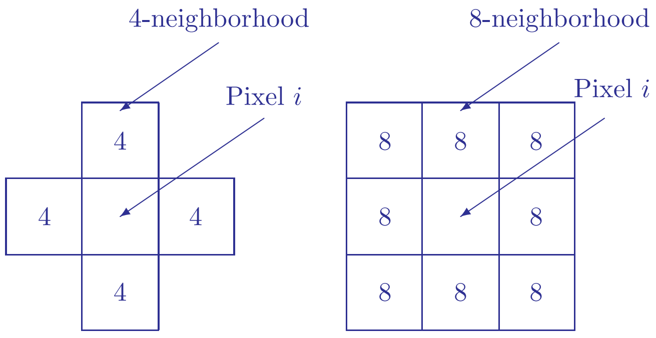


FIGURE 4.14
Pixel neighborhoods $\mathcal{N}_i$.

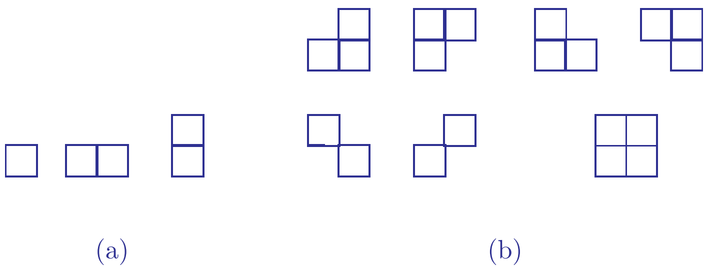


FIGURE 4.15
(a) Cliques for a 4-neighborhood. (a) and (b) Cliques for an 8-neighborhood.

where $\mathcal{N}_i$ is the set of pixels neighboring site i. Two frequently used neighborhood systems are shown in Figure 4.14.

The pair $(\mathcal{I}, \mathcal{N})$ may be thought of as constituting an *undirected graph* in which the pixels are nodes and the neighborhood system determines the edges between the nodes. Thus any two neighboring pixels are represented by two nodes connected by an edge. A *clique* is a single node or a subset of nodes which are all directly connected to one another in the graph by edges, i.e., they are mutual Neighbors. Figure 4.15 shows the possible cliques for the neighborhoods of Figure 4.14. (Note that one also distinguishes their orientation.) We will denote by the symbol $\mathcal{C}$ the set of all cliques in $\mathcal{I}$.

Next, let us introduce a set of *class labels*

$$\mathcal{K} = \{k \mid 1 \le k \le K\},$$

K being the number of possible classes present in the image. Assigning a class label to a site on the basis of measurement is a random experiment, so we associate with the ith site a discrete random variable L_i representing its label.

The set L of all such random variables

$$L = \{L_1 \dots L_n\}$$

is called a *random field* on $\mathcal{I}$. The possible realizations of L_i are labels $\ell_i \in \mathcal{K}$. A specific realization for the entire lattice, for example a classified image or thematic map, is a *configuration* ℓ, given by

$$\ell = \{\ell_1 \dots \ell_n\}, \quad \ell_i \in \mathcal{K},$$

and the space of all configurations is the Cartesian product set

$$\mathcal{L} = \overbrace{\mathcal{K} \otimes \mathcal{K} \dots \otimes \mathcal{K}}^{n \text{ times}}.$$

Thus there are K^n possible configurations. For each site i, the probability that the site has label ℓ_i is

$$\Pr(L_i = \ell_i) = \Pr(\ell_i),$$

and the joint probability for configuration ℓ is

$$\Pr(L_1 = \ell_1, L_2 = \ell_2 \dots L_n = \ell_n) = \Pr(L = \ell) = \Pr(\ell).$$

DEFINITION 4.2 *The random field L is said to be a* Markov random field (MRF) *on $\mathcal{I}$ with respect to neighborhood system $\mathcal{N}$ if and only if*

$$Pr(\ell) > 0 \quad \text{for all} \;\; \ell \in \mathcal{L}, \tag{4.35}$$

which is referred to as the positivity *condition, and*

$$Pr(\ell_i \mid \ell_1 \dots \ell_{i-1}, \ell_{i+1} \dots \ell_n) = Pr(\ell_i \mid \ell_j, j \in \mathcal{N}_i), \tag{4.36}$$

called the Markovianity *condition.*

The Markovianity condition in the above definition simply says that a label assignment can be influenced only by neighboring pixels.

DEFINITION 4.3 *The random field L constitutes a* Gibbs random field (GRF) *on $\mathcal{I}$ with respect to neighborhood system $\mathcal{N}$ if and only if it obeys a* Gibbs distribution. *A Gibbs distribution has density function*

$$p(\ell) = \frac{1}{Z}\exp(-\beta U(\ell)), \tag{4.37}$$

where β is a constant and Z is the normalization factor

$$Z = \sum_{\ell \in \mathcal{L}} \exp(-\beta U(\ell)), \tag{4.38}$$

and where the energy function $U(\ell)$ *is represented as a sum of contributing terms for each clique*

$$U(\ell) = \sum_{c \in \mathcal{C}} V_c(\ell). \tag{4.39}$$

$V_c(\ell)$ *is called a* clique potential *for clique c in configuration ℓ. If clique potentials are independent of the location of the clique within the lattice, then the GRF is said to be* homogeneous. *If V_c is independent of the orientation of c, then the GRF is* isotropic.

According to this definition, configurations in a GRF with low clique potentials are more probable than those with high clique potentials. The parameter β is an "inverse temperature." For small β (high temperature), all configurations become equally probable, irrespective of their associated energy.

A MRF is characterized by its local property (Markovianity) and a GRF by its global property (Gibbs distribution). It turns out that the two are equivalent:

THEOREM 4.3

(Hammersley–Clifford Theorem) *A random field L is an MRF on $\mathcal{I}$ with respect to $\mathcal{N}$ if and only if L is a GRF on $\mathcal{I}$ with respect to $\mathcal{N}$.*

Proof: The proof of the "if" part of the theorem, that a random field is an MRF if it is a GRF, is straightforward.* Write the conditional probability on the left-hand side of Equation (4.36) as

$$\Pr(\ell_i \mid \ell_1 \dots \ell_{i-1}, \ell_{i+1} \dots \ell_n) = \Pr(\ell_i \mid \ell_{\mathcal{I}-\{i\}}).$$

Here $\mathcal{I} - \{i\}$ is the set if all sites except the ith one. According to the definition of conditional probability, Equation (2.59),

$$\Pr(\ell_i \mid \ell_{\mathcal{I}-\{i\}}) = \frac{\Pr(\ell_i, \ell_{\mathcal{I}-\{i\}})}{\Pr(\ell_{\mathcal{I}-\{i\}})}.$$

Equivalently,

$$\Pr(\ell_i \mid \ell_{\mathcal{I}-\{i\}}) = \frac{\Pr(\ell)}{\sum_{\ell_i \in \mathcal{K}} \Pr(\ell')},$$

where $\ell' = \{\ell_1 \dots \ell_{i-1}, \ell'_i, \ell_{i+1} \dots \ell_n\}$ is any configuration which agrees with ℓ at all sites except possibly i. With Equations (4.37) and (4.39),

$$\Pr(\ell_i \mid \ell_{\mathcal{I}-\{i\}}) = \frac{\exp(-\sum_{c \in \mathcal{C}} V_c(\ell))}{\sum_{\ell_i \in \mathcal{K}} \exp(-\sum_{c \in \mathcal{C}} V_c(\ell'))}.$$

*The "only if" part is more difficult, see Li (2001).

Now divide the set of cliques $\mathcal{C}$ into two sets, namely $\mathcal{A}$, consisting of those cliques which contain site i, and $\mathcal{B}$, consisting of the rest. Then

$$\Pr(\ell_i \mid \ell_{\mathcal{I}-\{i\}}) = \frac{[\exp(-\sum_{c\in\mathcal{A}} V_c(\ell))][\exp(-\sum_{c\in\mathcal{B}} V_c(\ell))]}{\sum_{\ell_i\in\mathcal{K}}[\exp(-\sum_{c\in\mathcal{A}} V_c(\ell'))][\exp(-\sum_{c\in\mathcal{B}} V_c(\ell'))]}.$$

But for all cliques $c \in \mathcal{B}$, $V_c(\ell) = V_c(\ell')$ and the second factors in numerator and denominator cancel. Thus

$$\Pr(\ell_i \mid \ell_{\mathcal{I}-\{i\}}) = \frac{\exp(-\sum_{c\in\mathcal{A}} V_c(\ell))}{\sum_{\ell_i\in\mathcal{K}} \exp(-\sum_{c\in\mathcal{A}} V_c(\ell'))}.$$

This shows that the probability for ℓ_i is conditional only on the potentials of the cliques containing site i; in other words, that the random field is an MRF. □

The above theorem allows one to express the joint probability for a configuration ℓ of image labels in terms of the local clique potentials $V_c(\ell)$. We shall encounter an example in Chapter 8 in connection with unsupervised image classification.

4.6 Exercises

1. Give an explicit expression for the convolution of the array $(g(0)\dots g(5))$ with the array $(h(0), h(1), h(2))$ as it would be calculated with the Python function `numpy.convolve()`.

2. Modify the script on page 130 to convolve a two-dimensional array with the filter of Equation (4.7), both in the spatial and frequency domains.

3. Modify the `gaussfilter()` function in the `auxil.auxil1.py` module to implement the *Butterworth filter*

$$h(k,\ell) = \frac{1}{1+(d/d_0)^{2n}}, \tag{4.40}$$

   where d_0 is a width parameter and $n = 1, 2\dots$.

4. Demonstrate Equation (4.13).

5. (Press et al., 2002) Consider a row of $c = 8$ pixels represented by the row vector

$$\boldsymbol{f} = (f_0, f_1 \dots f_7).$$

Assuming that $\boldsymbol{f}$ is periodic (repeats itself), the application of low- and high-pass filter Equations (4.9) and (4.13) can be accomplished by multiplication of the column vector $\boldsymbol{f}^\top$ with the matrix

$$\boldsymbol{W} = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 & 0 & 0 & 0 & 0 \\ h_3 & -h_2 & h_1 & -h_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_0 & h_1 & h_2 & h_3 & 0 & 0 \\ 0 & 0 & h_3 & -h_2 & h_1 & -h_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 \\ 0 & 0 & 0 & 0 & h_3 & -h_2 & h_1 & -h_0 \\ h_2 & h_3 & 0 & 0 & 0 & 0 & h_0 & h_1 \\ h_1 & -h_0 & 0 & 0 & 0 & 0 & h_3 & -h_2 \end{pmatrix}.$$

(a) Prove that $\boldsymbol{W}$ is an orthonormal matrix (its inverse is equal to its transpose).

(b) In the transformed vector $\boldsymbol{W}\boldsymbol{f}^\top$, the components of $\boldsymbol{f}^1 = H\boldsymbol{f}$ and $\boldsymbol{d}^1 = G\boldsymbol{f}$ are interleaved. They can be sorted to give the vector $(\boldsymbol{f}^1, \boldsymbol{d}^1)$. When the matrix

$$\boldsymbol{W}^1 = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_3 & -h_2 & h_1 & -h_0 \\ h_2 & h_3 & h_0 & h_1 \\ h_1 & -h_0 & h_3 & -h_2 \end{pmatrix}$$

is then applied to the smoothed vector $(\boldsymbol{f}^1)^\top$ and the result again sorted, we obtain the complete discrete wavelet transformation of $\boldsymbol{f}$, namely $(\boldsymbol{f}^2, \boldsymbol{d}^2, \boldsymbol{d}^1)$. Moreover, by applying the inverse transformation to a unit vector, the D4 wavelet itself can be generated. Write a Python routine to plot the D4 wavelet by performing the inverse transformation on the vector $\underbrace{(0, 0, 0, 1, 0 \ldots 0)}_{1024}$.

6. Most filtering operations with the Fourier transform have their wavelet counterparts. Write a Python script to perform high-pass filtering with the discrete wavelet transformation. See Appendix C and the Python class `DWTArray` in the `auxil.auxil1` module to understand how code works, then proceed as follows:

   - Perform a single wavelet transform with the `DWTArray` object method `filter`.
   - Zero the upper left quadrant in the transformed image with a null array using the method `put_quadrant`.
   - Then invert the transformation with the method `invert()`.
   - Display the result with `dispms.py`.

7. Show that the Gaussian kernel, Equation (4.25), is equivalent to the inner product of infinite-dimensional mappings $\boldsymbol{\phi}(\boldsymbol{g})$.

8. Modify the program in Listing 4.2 to additionally calculate the kernel matrix for the *polynomial kernel function*

$$\kappa_{\text{poly}}(\boldsymbol{g}_i, \boldsymbol{g}_j) = (\gamma \boldsymbol{g}_i^\top \boldsymbol{g}_j + r)^d. \tag{4.41}$$

   The parameter r is called the *bias*, d the *degree* of the kernel function.

9. Shawe-Taylor and Cristianini (2004) prove (in their Proposition 5.2) that the centering operation minimizes the average eigenvalue of the kernel matrix. Write a Python script to do the following:

   (a) Generate m random N-dimensional observation vectors and, with routine in Listing 4.2, calculate the $m \times m$ Gaussian kernel matrix.

   (b) Determine its eigenvalues in order to confirm that the kernel matrix is positive semi-definite.

   (c) Center it (see the code following Equation (4.31)).

   (d) Re-determine the eigenvalues and verify that their average value has decreased.

   (e) Do the same using the polynomial kernel matrix from the preceding exercise.

10. A kernelized version of the dual formulation for the MNF transformation (see Chapter 3, Exercise 16) leads to a generalized eigenvalue problem having the form

$$\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{B}\boldsymbol{x}, \tag{4.42}$$

   where $\boldsymbol{A}$ and $\boldsymbol{B}$ are symmetric but not full rank. Let the symmetric $m \times m$ matrix $\boldsymbol{B}$ have rank $r < m$, eigenvalues $\lambda_1 \dots \lambda_m$, and eigenvectors $\boldsymbol{u}_1 \dots \boldsymbol{u}_m$. Show that Equation (4.42) can be re-formulated as an ordinary symmetric eigenvalue problem by writing $\boldsymbol{B}$ as a product of *matrix square roots* $\boldsymbol{B} = \boldsymbol{B}^{1/2}\boldsymbol{B}^{1/2}$. The square root is defined as

$$\boldsymbol{B}^{1/2} = \boldsymbol{P}\boldsymbol{\Lambda}^{1/2}\boldsymbol{P}^\top,$$

   where $\boldsymbol{P} = (\boldsymbol{u}_1 \dots \boldsymbol{u}_r)$ and $\boldsymbol{\Lambda} = \text{Diag}(\lambda_1 \dots \lambda_r)$.

11. Does a 24-neighborhood (the 5×5 array centered at a location i) have more clique types than are shown in Figure 4.15?

5 Image Enhancement and Correction

In preparation for the treatment of supervised/unsupervised classification and change detection, the subjects of the last four chapters of this book, the present chapter focuses on preprocessing methods. These fall into the two general categories of *image enhancement* (Sections 5.1 through 5.4) and *geometric correction* (Sections 5.5 and 5.6). Discussion mainly focuses on the processing of optical/infrared image data. However, Section 5.4 introduces polarimetric SAR imagery and treats the problem of speckle removal.

5.1 Lookup tables and histogram functions

Gray-level enhancements of an image are easily accomplished by means of *lookup tables*. For byte-encoded data, for example, the pixel intensities $g(i,j)$ are used to index into the array

$$LUT[k], \quad k = 0 \dots 255,$$

the entries of which also lie between 0 and 255. These entries can be chosen to implement simple histogram processing, such as linear stretching, saturation, equalization, etc. The pixel values $g(i,j)$ are replaced, for r rows and c columns, by

$$f(i,j) = LUT[g(i,j)], \quad 0 \le i \le r-1, \ 0 \le j \le c-1. \tag{5.1}$$

In deriving the appropriate transformations it is convenient to think of the normalized histogram of pixel intensities g as a probability density $p_g(g)$ of

Listing 5.1: Histogram equalization (from the `auxil.auxil1.py` module.)

```python
def histeqstr(x):
    x = bytestr(x)
#   histogram equalization stretch
    hist,bin_edges = np.histogram(x,256,(0,256))
    cdf = hist.cumsum()
    lut = 255*cdf/float(cdf[-1])
    return np.interp(x,bin_edges[:-1],lut)
```

DOI: 10.1201/9781003503286-5

a continuous random variable G, and the lookup table itself as a continuous transformation function, i.e.,

$$f(g) = LUT(g),$$

where both g and f are restricted to the interval $[0,1]$. We will illustrate the technique for the case of *histogram equalization.* First of all, we claim that the function

$$f(g) = LUT(g) = \int_0^g p_g(t)dt \tag{5.2}$$

corresponds to histogram equalization in the sense that a random variable $F = LUT(G)$ has uniform probability density. To see this, note that the function LUT in Equation (5.2) satisfies the monotonicity condition of Theorem 2.1. Therefore, with Equation (2.13), the PDF for F is

$$p_f(f) = p_g(g)\left|\frac{dg}{df}\right|.$$

Differentiating Equation (5.2),

$$\frac{df}{dg} = p_g(g), \quad \frac{dg}{df} = \frac{1}{p_g(g)}$$

and, since all probabilities are positive on the interval [0,1],

$$p_f(f) = p_g(g)\left|\frac{1}{p_g(g)}\right| = 1,$$

so F indeed has a uniform density. Histogram equalization can be approximated for byte-encoded data by first replacing $p_g(g)$ by the normalized histogram

$$p_g(g_k) = \frac{n_k}{n}, \quad k = 0\dots 255, \quad n = \sum_{j=0}^{255} n_j,$$

where n_k is the number of pixels with gray value g_k. Approximating the integral in Equation (5.2) by a summation and normalizing to the interval [0,255] this leads to the lookup table

$$LUT(g_k) = 255\cdot\sum_{j=0}^{k} p_g(g_k) = 255\cdot\sum_{j=0}^{k}\frac{n_j}{n}, \quad k = 0\dots 255. \tag{5.3}$$

The result is rounded down to the nearest integer. Because of the quantization, the resulting histogram will in general not be perfectly uniform, however the desired effect of spreading the intensities to span the full range of gray-scale values will be achieved.

The closely related procedure of *histogram matching*, which is the transformation of an image histogram to match the histogram of another image or some specified function, can be similarly derived using the probability density approximation; see Gonzalez and Woods (2017) for a detailed discussion.

Listing 5.1 shows a straightforward Python implementation of histogram equalization stretching. It makes efficient use of the `numpy.interp()` function to interpolate each intensity in the input array `x` between the histogram bin edges `bin_edges[:-1]` (just the sequence $[0, 1 \dots 255]$) and the normalized lookup table values `lut`, Equation (5.3). An example is given in the accompanying Jupyter notebook for Chapter 5.

5.2 High-pass spatial filtering and feature extraction

In Chapter 4, Section 4.2, we introduced filtering in the spatial domain, giving a simple example of a low-pass filter; see Equation (4.7). We shall now examine high-pass filtering for edge and contour detection, techniques that are used to implement low-level feature matching for, e.g., image co-registration. We will also see how to access the highly optimized algorithms of the Open Source Computer Vision Library OpenCV from the Python interpreter and look at similar functionality on the GEE Python API. Edge detection is also used in conjunction with feature extraction for scene analysis and for image segmentation, one of the subjects treated in Chapter 8. Localized image features and/or segments can often be conveniently characterized by their *geometric moments*. A short description of geometric moments together with a Python script to calculate them will also be presented in this section.

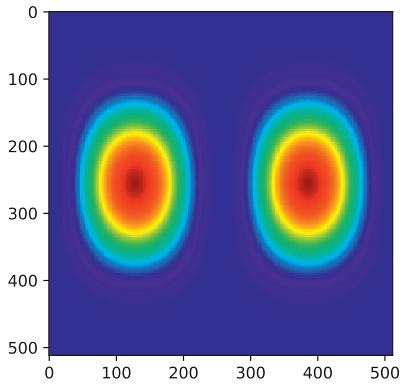


FIGURE 5.1
Power spectrum image of the Sobel filter h_1 of Equation (5.5). Low spatial frequencies are at the center of the image.

5.2.1 Sobel filter

We begin by introducing the *gradient operator* in two dimensions

$$\nabla = \frac{\partial}{\partial \boldsymbol{x}} = \boldsymbol{i}\frac{\partial}{\partial x_1} + \boldsymbol{j}\frac{\partial}{\partial x_2}, \tag{5.4}$$

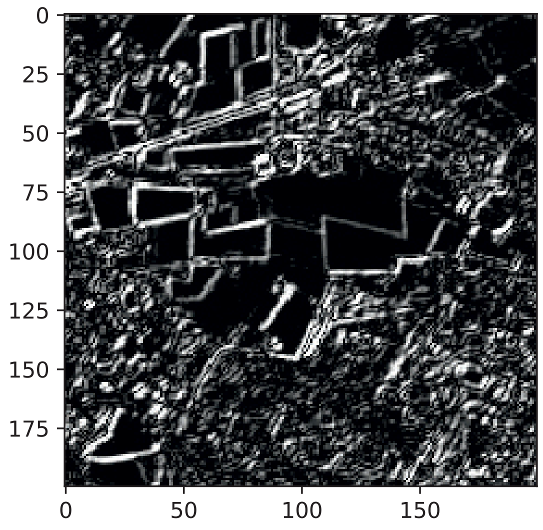


FIGURE 5.2
Sobel edge detection on a spatial subset of the 3N band of the Jülich ASTER image (Figure 1.1).

where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the image plane in the horizontal and vertical directions, respectively. In a two-dimensional image represented by the continuous scalar function $g(x_1, x_2) = g(\boldsymbol{x})$, $\nabla g(\boldsymbol{x})$ is a vector in the direction of the maximum rate of change of gray-scale intensity.

Of course, the intensity values are actually discrete, so the partial derivatives must be approximated. For example, we can use the *Sobel operators*:

$$\begin{aligned}\left.\frac{\partial g(\boldsymbol{x})}{\partial x_1}\right|_{\boldsymbol{x}=(i,j)} &\approx [g(i-1,j-1)+2g(i-1,j)+g(i-1,j+1)]\\ &\quad -[g(i+1,j-1)+2g(i+1,j)+g(i+1,j+1)] =: \ \nabla_1(\boldsymbol{x})\\ \left.\frac{\partial g(\boldsymbol{x})}{\partial x_2}\right|_{\boldsymbol{x}=(i,j)} &\approx [g(i-1,j-1)+2g(i,j-1)+g(i+1,j-1)]\\ &\quad -[g(i-1,j+1)+2g(i,j+1)+g(i+1,j+1)] =: \ \nabla_2(\boldsymbol{x}),\end{aligned}$$

which are equivalent to the two-dimensional spatial filters

$$h_1 = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad \text{and} \quad h_2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, \tag{5.5}$$

respectively; see Equation (4.6). The magnitude of the gradient at pixel $x = (i,j)$ is

$$\|\nabla g(i,j))\| = \sqrt{\nabla_1(\boldsymbol{x})^2 + \nabla_2(\boldsymbol{x})^2}.$$

Edge detection can be achieved by calculating the filtered image

$$f(i,j) = \|\nabla(g(i,j)\|$$

and setting an appropriate threshold. The Sobel filters, Equations (5.5), involve *differences*, a characteristic of high-pass filters. They have the property of returning near-zero values when traversing regions of constant intensity, and positive or negative values in regions of changing intensity. The following code computes the Fourier power spectrum of h_1; see Figure 5.1.

```
import numpy as np
from numpy import fft
import matplotlib.pyplot as plt
from matplotlib import cm
import auxil.auxil1 as auxil

# create filter
g = np.zeros((512, 512),dtype=float)
g[:3,:3] = np.array([[1,0,-1], [2,0,-2],[1,0,-1]])

#  shift Fourier transform to center
a = np.reshape(range(512**2),(512,512))
i = a % 512
j = a // 512
g = (-1)**(i+j)*g

# compute power spectrum and display in a linear stretch
p = np.abs(fft.fft2(g))**2
plt.imshow(auxil.linstr(p), cmap=cm.jet)
```

The Fourier spectrum of h_2 is the same, but rotated by 90 degrees.

The Sobel filter is available in the SciPy package. Here we apply it to a spatial subset of the 3N spectral band from the ASTER image; see Figure 5.2:

```
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly
from scipy import ndimage

gdal.AllRegister()
infile = 'imagery/AST_20070501.tif'
inDataset = gdal.Open(infile,GA_ReadOnly)
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize
band = inDataset.GetRasterBand(3)
image = band.ReadAsArray(0,0,cols,rows).astype(float)
edges0 = ndimage.sobel(image,axis=0)
edges1 = ndimage.sobel(image,axis=1)

# combine and perform 2% saturated linear stretch
edges = auxil.lin2pcstr(np.abs(edges0+edges1))
plt.imshow(edges[200:400,200:400], cmap=cm.gray)
```

Note that the edge widths vary considerably, depending upon the contrast and abruptness (strength) of the edge.

5.2.2 Laplacian-of-Gaussian filter

The magnitude of the gradient reaches a maximum at an edge in a gray-scale image. The second derivative, on the other hand, is zero at that maximum and has opposite signs immediately on either side. This offers the possibility to determine edge positions to the accuracy of one pixel by using second derivative filters. Thin edges, as we will see later, are useful in automatic determination of invariant features for image registration.

The second derivatives of the image intensities can be calculated with the *Laplacian* operator,

$$\nabla^2 = \nabla^\top \nabla = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2},$$

see Equation 5.4. The Laplacian $\nabla^2 g(\boldsymbol{x})$ is an isotropic scalar function which is zero whenever the gradient magnitude is maximum. Like the gradient operator, the Laplacian operator can also be approximated by a spatial filter, for example,

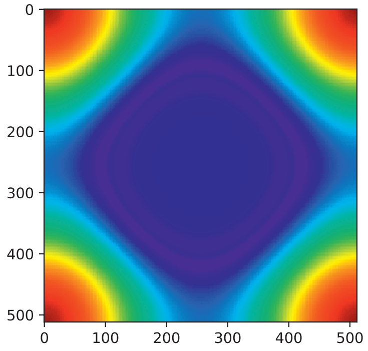


FIGURE 5.3
Power spectrum image of the Laplacian filter, Equation (5.6). Low spatial frequencies are at the center of the image.

$$h = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}. \tag{5.6}$$

Its power spectrum is depicted in Figure 5.3. This filter has the desired property of returning zero in regions of constant intensity and in regions of constantly varying intensity (e.g., ramps), but non-zero at their onset or termination. Laplacian filters tend to be very sensitive to image noise. Often a low-pass Gaussian filter is first used to smooth the image before the Laplacian filter is applied. This is equivalent to calculating the Laplacian of the Gaussian itself and then using the result to derive a high-pass filter. Recall from Chapter 2 that the normalized Gauss function in two dimensions is given by

$$\frac{1}{2\pi\sigma^2} \exp\left(-\frac{1}{2\sigma^2}(x_1^2 + x_2^2)\right),$$

where the parameter σ determines its extent. Taking the second partial derivatives gives the *Laplacian-of-Gaussian* (LoG) filter

$$\frac{1}{2\pi\sigma^6}(x_1^2 + x_2^2 - 2\sigma^2) \exp\left(-\frac{1}{2\sigma^2}(x_1^2 + x_2^2)\right). \tag{5.7}$$

The script below illustrates the use of a LoG filter (stored as a 16×16 array in the variable `filt`; see the Jupyter notebook) with determination of sign change to generate thin edges or contours from a gray-scale image.

```
# pad the ASTER image
impad = np.zeros((rows+16,cols+16))
impad[:rows,:cols] = image
# pad the filter as well
filtpad = impad*0.0
filtpad[:16,:16] = filt
# filter in frequency domain
im = np.real(fft.ifft2(fft.fft2(impad)*fft.fft2(filtpad)))
# get zero-crossings
idx = np.where( (im*np.roll(im,1,axis=0)<0) | \
                (im*np.roll(im,1,axis=1)<0) )
# get edge strengths
edges0 = ndimage.sobel(im,axis=0)
edges1 = ndimage.sobel(im,axis=1)
edges = auxil.lin2pcstr(np.abs(edges0+edges1))
# assign edge strengths at zero-crossings
im1 = 0.0*im
im1[idx] = edges[idx]
plt.imshow(im1[200:400,200:400],cmap='gray')
```

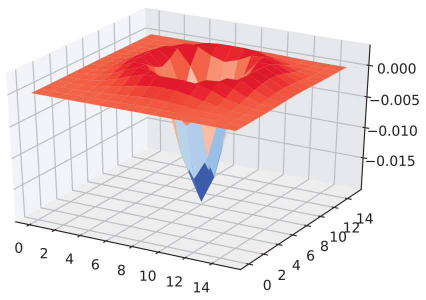


FIGURE 5.4
Laplacian-of-Gaussian filter on a 16×16 grid, with $\sigma = 2$.

The filtering is carried out, after appropriate padding, in the frequency domain. The zero crossings in the horizontal and vertical directions are determined from the products of the image with a copy of itself shifted by one pixel to the right and upward, respectively. Sign changes correspond to negative values in the product arrays and these define the contours. The contour pixel intensities are set equal to the magnitude of the local gradient as determined by a Sobel filter. This allows the application of subsequent thresholding to identify the more significant contours. The surface plot of the two-dimensional LoG filter is shown in Figure 5.4; the filtered image is displayed in Figure 5.5. Comparing with Figure 5.2, one sees that the contours are thinner. The "spaghetti" effect is characteristic.

5.2.3 OpenCV and GEE algorithms

The Open Source Computer Vision Library `OpenCV`, although primarily intended for real time vision applications, is a treasure chest of useful imaging processing routines, all of which are easily accessible from Python. Here is a quote from the website `opencv.org`:

OpenCV is released under a BSD license and hence it's free for both academic and commercial use. It has C++, C, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. OpenCV was designed for computational efficiency and with a strong focus on real-time applications.

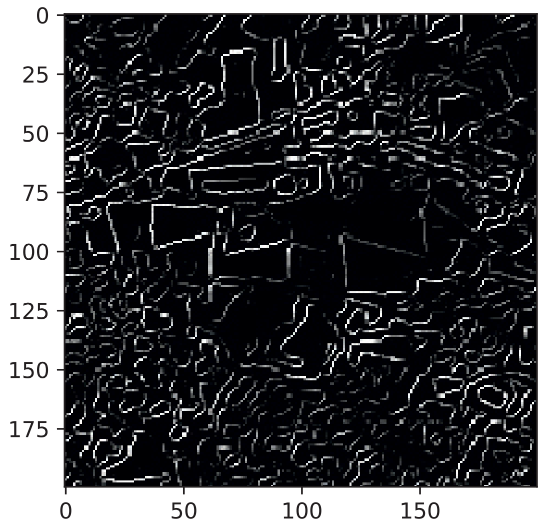


FIGURE 5.5
Image contours from a spatial subset of the 3N band of the Jülich ASTER image (Figure 1.1) calculated from the Laplacian of Gaussian filter.

In the following, we will demonstrate the use of OpenCV for remote sensing imagery analysis with two well-known feature detection algorithms, and in the remainder of the text, we will continue to make free use of the library for many of our Python scripts. There are also built-in functions on the Google Earth Engine platform for server-side feature extraction, and we will examine one of them briefly as well.

5.2.3.1 Corner detection

In a gray-scale image, the local (scalar) variogram in a neighborhood of the pixel position $\boldsymbol{x}$ can be estimated as

$$\gamma(\boldsymbol{h}) = \left\langle (g(\boldsymbol{x}) - g(\boldsymbol{x}+\boldsymbol{h}))^2 \right\rangle, \tag{5.8}$$

where $\langle\cdot\rangle$ signifies the average over the pixel's neighborhood; see Equation (3.68). A corner, or in general some interesting feature such as a localized bright or dark spot, will be characterized by a large value for $\gamma(\boldsymbol{h})$ in all directions of the displacement vector $\boldsymbol{h}$. An edge, on the other hand, would exhibit a large variation only in the directions nearly orthogonal to itself, and featureless regions would have small variations in all directions. Following Harris and Stephens (1988), we first of all write down the first-order Taylor series approximation to $g(\boldsymbol{x}+\boldsymbol{h})$ in Equation (5.8):

$$g(\boldsymbol{x}+\boldsymbol{h}) \approx g(\boldsymbol{x}) + \boldsymbol{h}^\top \frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}} = g(\boldsymbol{x}) + \boldsymbol{h}^\top \nabla g(\boldsymbol{x});$$

see Equation (1.55). With this, the variogram $\gamma(\boldsymbol{h})$ can be approximately written as

$$\gamma(\boldsymbol{h}) \approx \left\langle \left(\boldsymbol{h}^\top \nabla g(\boldsymbol{x})\right)^2 \right\rangle,$$

or, expanding the quadratic term,

$$\begin{aligned}
\gamma(\boldsymbol{h}) &\approx \boldsymbol{h}^\top \left\langle \nabla g(\boldsymbol{x}) \nabla g(\boldsymbol{x})^\top \right\rangle \boldsymbol{h} \\
&= \boldsymbol{h}^\top \left\langle \begin{pmatrix} \left(\frac{\partial g(\boldsymbol{x})}{\partial x_1}\right)^2 & \frac{\partial g(\boldsymbol{x})}{\partial x_1}\frac{\partial g(\boldsymbol{x})}{\partial x_2} \\ \frac{\partial g(\boldsymbol{x})}{\partial x_1}\frac{\partial g(\boldsymbol{x})}{\partial x_2} & \left(\frac{\partial g(\boldsymbol{x})}{\partial x_2}\right)^2 \end{pmatrix} \right\rangle \boldsymbol{h} \\
&\approx \boldsymbol{h}^\top \begin{pmatrix} \langle \nabla_1(\boldsymbol{x})^2 \rangle & \langle \nabla_1(\boldsymbol{x})\nabla_2(\boldsymbol{x}) \rangle \\ \langle \nabla_1(\boldsymbol{x})\nabla_2(\boldsymbol{x}) \rangle & \langle \nabla_2(\boldsymbol{x})^2 \rangle \end{pmatrix} \boldsymbol{h} = \boldsymbol{h}^\top \boldsymbol{A}\boldsymbol{h},
\end{aligned} \tag{5.9}$$

where $\nabla_i(\boldsymbol{x})$, $i = 1, 2$, are the Sobel gradient operators introduced in Section 5.2.1. The matrix $\boldsymbol{A}$ in Equation (5.9) is symmetric and also positive definite for sufficiently large neighborhoods. In its principal axis coordinate system, therefore, we can write Equation (5.9) as

$$\gamma(\boldsymbol{h}) \approx \boldsymbol{h}^\top \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \boldsymbol{h} = \lambda_1 h_1^2 + \lambda_2 h_2^2,$$

where λ_1 and λ_2 are the real positive eigenvalues of $\boldsymbol{A}$. If the variogram $\gamma(\boldsymbol{h})$ is to be large for all directions $\boldsymbol{h}$, then clearly both eigenvalues must be large. The OpenCV filter function `cornerMinEigenVal()` calculates the minimum eigenvalue of $\boldsymbol{A}$ for a square neighborhood of each pixel in a scene, returning the result as a floating point image. This image can then be thresholded to distinguish significant corners or other features.

5.2.3.2 Canny edge detector

The Canny edge detector (Canny, 1986),* which is similar to the corner detection algorithm, is based upon a gradient filter such as the Sobel filter. First a Gaussian smoothing filter is applied to suppress noise, followed by the gradient filter, whereby both the magnitude of the gradient

$$\|\nabla g(i,j))\| = \sqrt{\nabla_1(\boldsymbol{x})^2 + \nabla_2(\boldsymbol{x})^2}$$

*See also http://en.wikipedia.org/wiki/Canny_edge_detector.

Listing 5.2: Corner and Canny edge detection with OpenCV.

```
#!/usr/bin/env python3
#  Name:     ex5_1.py
import numpy as np
import os, sys, getopt
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly,GDT_Float32
import cv2 as cv

def main():
    options,args = getopt.getopt(sys.argv[1:],'b:a:')
    b = 1
    algorithm = 1
    for option, value in options:
        if option == '-b':
            b = eval(value)
        elif option == '-a':
            algorithm = eval(value)
    gdal.AllRegister()
    infile = args[0]
    path = os.path.dirname(infile)
    basename = os.path.basename(infile)
    root, ext = os.path.splitext(basename)
    inDataset = gdal.Open(infile,GA_ReadOnly)
    cols = inDataset.RasterXSize
    rows = inDataset.RasterYSize
    rasterBand = inDataset.GetRasterBand(b)
    band = rasterBand.ReadAsArray(0,0,cols,rows) \
                                  .astype(np.uint8)
    if algorithm==1:
    #   corner detection, window size 7x7
        result = cv.cornerMinEigenVal(band, 7)
        outfile = path+'/'+root+'_corner'+ext
    else:
    #   edge detection, window size 7x7
        result = cv.Canny(band,50,150)
        outfile = path+'/'+root+'_canny'+ext
    #  write to disk
    driver = inDataset.GetDriver()
    outDataset = driver.Create(outfile,
                     cols,rows,1,GDT_Float32)
    outBand = outDataset.GetRasterBand(1)
    outBand.WriteArray(result,0,0)
    outBand.FlushCache()
    outDataset = None; inDataset = None
    print('result written to %s'%outfile)
if __name__ == '__main__':
    main()
```

as well as its direction

$$\theta = \arctan\left(\frac{\nabla_2(\boldsymbol{x})}{\nabla_1(\boldsymbol{x})}\right)$$

are calculated. From here on we quote the excellent Wikipedia article:

> Given estimates of the image gradients, a search is then carried out to determine if the gradient magnitude assumes a local maximum in the gradient direction. ... From this stage, referred to as non-maximum suppression, a set of edge points, in the form of a binary image, is obtained. These are sometimes referred to as "thin edges."
>
> Large intensity gradients are more likely to correspond to edges than small intensity gradients. It is in most cases impossible to specify a threshold at which a given intensity gradient switches from corresponding to an edge into not doing so. Therefore Canny uses thresholding with hysteresis.*
>
> Thresholding with hysteresis requires two thresholds high and low. Making the assumption that important edges should be along continuous curves in the image allows us to follow a faint section of a given line and to discard a few noisy pixels that do not constitute a line but have produced large gradients. Therefore we begin by applying a high threshold. This marks out the edges we can be fairly sure are genuine. Starting from these, using the directional information derived earlier, edges can be traced through the image. While tracing an edge, we apply the lower threshold, allowing us to trace faint sections of edges as long as we find a starting point.
>
> Once this process is complete we have a binary image where each pixel is marked as either an edge pixel or a non-edge pixel. From complementary output from the edge tracing step, the binary edge map obtained in this way can also be treated as a set of edge curves.

A simple stand-alone script demonstrating corner and Canny edge detection is shown in Listing 5.2. For corner detection, the chosen image band is processed with `cornerMinEigenVal()` using a 7×7 window, line 31. The lower and upper hysteresis thresholds for the Canny edge detector are 50 and 150, respectively, line 35. An example is shown in Figure 5.6, which was created in the Jupyter notebook with the commands

```
%run scripts/ex5_1 -b 3 -a 1 imagery/AST_20070501.tif
%run scripts/ex5_1 -b 3 -a 2 imagery/AST_20070501.tif
%run scripts/dispms -f imagery/AST_20070501_corner.tif \
     -e 3 \
     -d [200,200,200,200] \
```

*In Section 5.6, we will mention a similar hysteresis approach for contour detection.

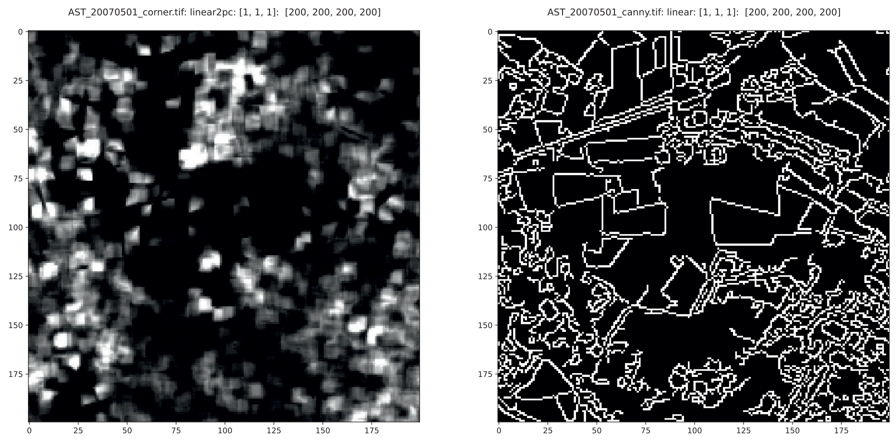

AST_20070501_corner.tif: linear2pc: [1, 1, 1]: [200, 200, 200, 200]
AST_20070501_canny.tif: linear: [1, 1, 1]: [200, 200, 200, 200]
0
25
50
75
100
125
150
175

FIGURE 5.6
Corners and edges detected from the same spatial subset of the 3N band of the ASTER image as Figure 5.5. Left: the minimum eigenvalue intensities in a linear 2% histogram stretch. Right: the Canny edges.

```
        -F imagery/AST_20070501_canny.tif -E 2 \
        -D [200,200,200,200]
```

In the first two lines, the spectral band is selected with the `-b` option and the algorithm with the `-a` option (1=corner, 2=Canny).

5.2.3.3 Canny edge detection on the GEE

The GEE has a Canny edge detector, too, and the following script applies it this time to a full scene LANDSAT 7 ETM+ acquisition:

```
import ee
ee.Initialize()
im = ee.Image(
  'LANDSAT/LE07/C02/T1_RT_TOA/LE07_197025_20010626') \
       .select('B4')
edges = ee.Algorithms.CannyEdgeDetector(im,0.2)
gdexporttask = ee.batch.Export.image.toAsset(edges,
       description='assetExportTask',
       assetId='projects/ee-mortcanty/assets/CRC5/canny',
       scale=30,
       maxPixels=1e9)
```

After determining the edge image from the VNIR band B4, it is exported to GEE assets. The GEE version only allows a single lower threshold. This was set to 0.2 since the image is TOA reflectance and intensities are in the interval $[0, 1]$. Figure 5.7 is a screen shot from the GEE code editor displaying part of the edge image superimposed onto the background satellite map.

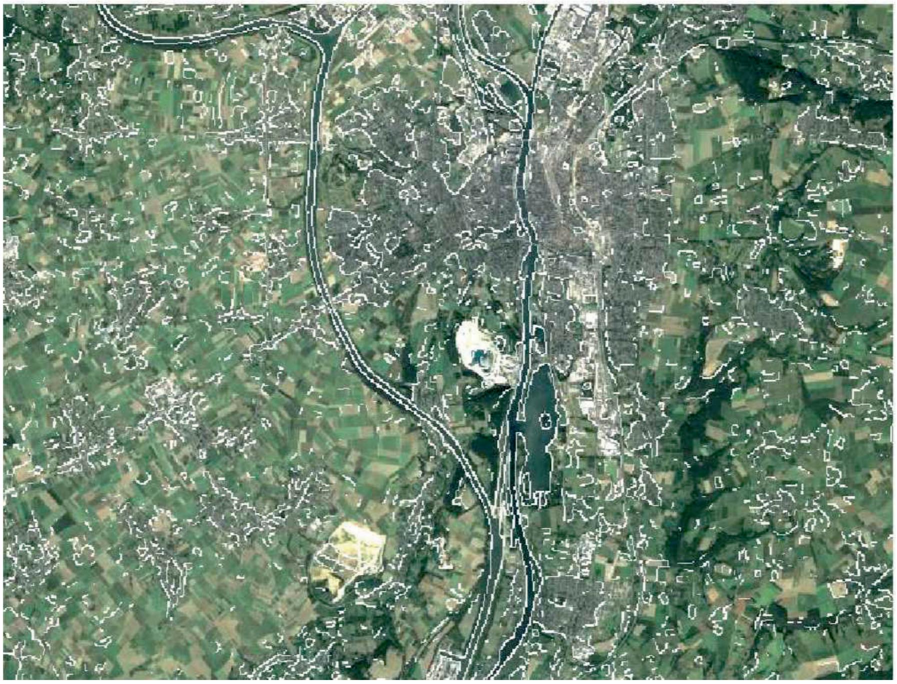


FIGURE 5.7
Edges from band 4 of a LANDSAT 7 ETM+ image with the GEE Canny edge detector. Map data (c)2023 Google.

5.2.4 Invariant moments

Moments and functions of moments are employed extensively in image classification, target identification and image scene analysis (Prokop and Reeves, 1992). A feature such as, for instance, a closed contour extracted from the image in Figure 5.5 can be described in terms of its *geometric moments*. Let S denote the set of pixels belonging to the feature. The (discrete) geometric moment of order p, q of the feature is defined by (see, e.g., Haberächer (1995); Gonzalez and Woods (2017))

$$m_{pq} = \sum_{i,j \in S} g(i,j) i^p j^q, \quad p, q = 0, 1, 2 \ldots. \tag{5.10}$$

Thus m_{00} is the total intensity of the feature or, in a binary representation in which $g(i,j) = 1$ if $(i,j) \in S$ and $g(i,j) = 0$; otherwise, m_{00} is the number of pixels. The center of gravity $(\bar{x}_1, \bar{x}_2)$ of the feature is

$$\bar{x}_1 = \frac{m_{10}}{m_{00}}, \quad \bar{x}_2 = \frac{m_{01}}{m_{00}}. \tag{5.11}$$

The translation-invariant *centralized moments* μ_{pq} are obtained by shifting the origin to the center of gravity,

$$\mu_{pq} = \sum_{i,j\in S} g(i-\bar{x}_1, j-\bar{x}_2)(i-\bar{x}_1)^p(j-\bar{x}_2)^q \tag{5.12}$$

and the *normalized centralized moments* η_{pq} by the following normalization; see Exercise 5,

$$\eta_{pq} = \frac{1}{\mu_{00}^{(p+q)/2+1}}\mu_{pq}. \tag{5.13}$$

The normalized centralized moments are, apart from effects of digital quantization, invariant under both translations and scale changes. For example, in a binary representation the moment η_{20} is

$$\eta_{20} = \frac{1}{\mu_{00}^2}\mu_{20} = \frac{1}{n^2}\sum_{i,j\in S}(i-\bar{x}_1)^2 = \frac{1}{n}\sum_{i\in S}(i-\bar{x}_1)^2,$$

where $n = |S|$, the cardinality of S. This is just the variance of the feature in the x_1 direction.

Finally, we can define *Hu moments*, which are functions of the normalized centralized moments of orders $p+q \le 3$ and which are invariant under rotations; see Hu (1962) or Gonzalez and Woods (2017). There are seven such moments in all, the first four of which are given by

$$\begin{aligned} h_1 &= \eta_{20}+\eta_{02} \\ h_2 &= (\eta_{20}-\eta_{02})^2 + 4\eta_{11}^2 \\ h_3 &= (\eta_{30}-3\eta_{12})^2 + (\eta_{03}-3\eta_{21})^2 \\ h_4 &= (\eta_{30}+\eta_{12})^2 + (\eta_{03}+\eta_{21})^2. \end{aligned} \tag{5.14}$$

To illustrate their rotational invariance, consider a rotation of the coordinate axes through the angle θ with origin at the center of gravity of a feature. A point (i,j) transforms according to

$$\begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} = \boldsymbol{A}\begin{pmatrix} i \\ j \end{pmatrix}.$$

Then, again in a binary image, the first invariant moment in the rotated coordinate system is

$$\begin{aligned} h_1 &= \frac{1}{n^2}\sum_{i',j'\in S}(i'^2+j'^2) = \frac{1}{n^2}\sum_{i',j'\in S}(i',j')\begin{pmatrix} i' \\ j' \end{pmatrix} \\ &= \frac{1}{n^2}\sum_{i,j\in S}(i,j)\boldsymbol{A}^\top\boldsymbol{A}\begin{pmatrix} i \\ j \end{pmatrix} = \frac{1}{n^2}\sum_{i,j\in S}(i^2+j^2), \end{aligned}$$

since $\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{I}$.

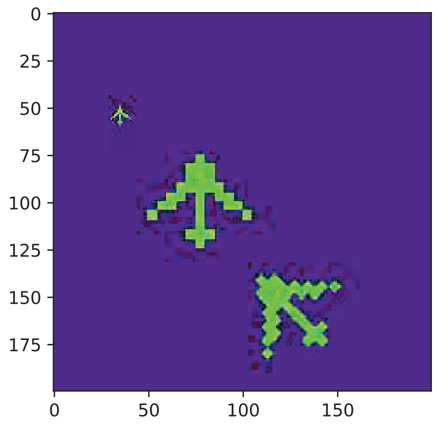


FIGURE 5.8
An "aircraft" feature translated, scaled and rotated.

The Hu moments are strictly invariant only in the limit of continuous shapes. The discrete nature of image features introduces errors which become especially severe for higher-order moments. Liao and Pawlak (1996) give correction formulae for the geometric moments, Equation (5.10), which reduce the error. In order to demonstrate the invariance, the following code generates the features shown in Figure 5.8 and then calls `cv.moments()` followed by `cv.HuMoments()` from the OpenCV package in order to calculate the logarithms of the invariant moments; see the explanation of these two functions below. Only the first four moments are printed.

```
from scipy.ndimage import rotate
# Airplanes
A = np.array([[0,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,1,1,1,0,0,0,0],
              [0,0,0,0,1,1,1,0,0,0,0],
              [0,0,0,1,1,1,1,1,0,0,0],
              [0,0,1,1,0,1,0,1,1,0,0],
              [0,1,1,0,0,1,0,0,1,1,0],
              [1,0,0,0,0,1,0,0,0,0,1],
              [0,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,1,1,1,0,0,0,0],
              [0,0,0,0,0,1,0,0,0,0,0]])
im = np.zeros((200,200))
im[50:60,30:41] = A
im1 = im*0
im1[75:125,50:105] = auxil.rebin(A,(50,55))
im2 = rotate(im1,45)
plt.imshow(im + im1 + im2[:200,:200])

import cv2 as cv
hu = cv.HuMoments(cv.moments(im)).ravel()
hu1 = cv.HuMoments(cv.moments(im1)).ravel()
hu2 = cv.HuMoments(cv.moments(im2)).ravel()
print(hu[:4])
print(hu1[:4])
print(hu2[:4])
```

```
[3.3262e-01 2.9152e-04 2.1044e-03 1.5381e-03]
[3.3796e-01 2.9152e-04 2.1044e-03 1.5381e-03]
[3.3794e-01 2.8995e-04 2.1018e-03 1.5369e-03]
```

As a less trivial example, the Python script `ex5_2.py` shown in Listing 5.3 extracts the "significant contours" from an image band, here band 1 of the ASTER image,

```
run scripts/ex5_2 -b 1 imagery/AST_20070501.tif,
```

calculates their invariant moments and plots histograms of the moment logarithms, making extensive use of OpenCV functions. The `findContours()` function in line 26 extracts contours from the binary image returned from the Canny edge detector, line 25. The `cd.CHAIN_APPROX_NONE` flag means that the returned contours will not be compressed, and the flag `cv.RETR_LIST` means that the contour hierarchy (which contour is enclosed in which other contour) is ignored. The seven Hu moments of the individual contours are calculated and stored in an array (lines 32–36). This occurs in two steps: First the translational and scale invariants are determined with `moments()`, line 35, then these are passed to `HuMoments()`, line 36. For very simple contour shapes the higher moments will be zero. This condition is checked for in line 39 prior to taking logarithms and plotting the histogram for the first three Hu moments.

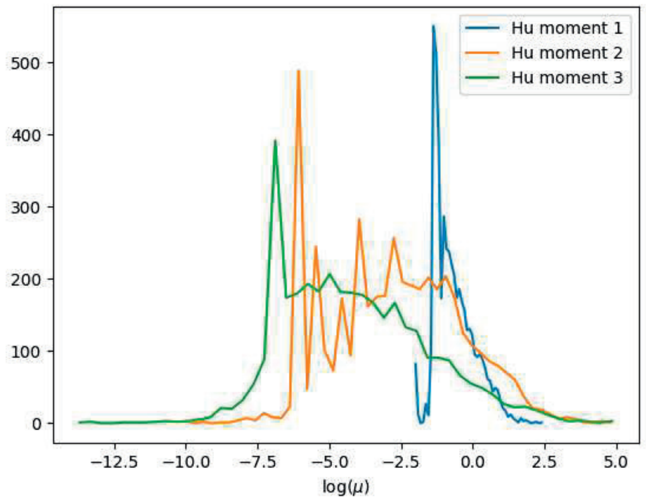


FIGURE 5.9
Histograms of the logarithms of the first three Hu moments of the image contours.

Listing 5.3: Hu invariant moments of image contours.

```
#!/usr/bin/env python3
#Name:  ex5_2.py
import sys, getopt
import numpy as np
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly
import cv2 as cv
import matplotlib.pyplot as plt
def main():
    options,args = getopt.getopt(sys.argv[1:],'b:')
    b = 1
    for option, value in options:
        if option == '-b':
            b = eval(value)
    gdal.AllRegister()
    infile = args[0]
    #  read band of an MS image
    inDataset = gdal.Open(infile,GA_ReadOnly)
    cols = inDataset.RasterXSize
    rows = inDataset.RasterYSize
    rasterBand = inDataset.GetRasterBand(b)
    band = np.uint8(rasterBand
                        .ReadAsArray(0,0,cols,rows))
    #  find significant contours
    edges = cv.Canny(band, 20, 80)
    contours,hierarchy = cv.findContours(edges,\
                 cv.RETR_LIST,cv.CHAIN_APPROX_NONE)
    #  determine Hu moments
    num_contours = len(hierarchy[0])
    hus = np.zeros((num_contours,7),dtype=np.float32)
    arr = np.zeros((rows, cols), dtype=np.uint8)
    for i in range(num_contours):
        arr = arr*0
        cv.drawContours(arr, contours, i, 1)
        m = cv.moments(arr)
        hus[i,:] = cv.HuMoments(m).ravel()
    #  plot histogram of logs of the first Hu moments
    for i in range(3):
        idx = np.where(hus[:,i]>0)
        hist,e=np.histogram(np.log(hus[idx,i]),50)
        plt.plot(e[1:],hist,label='Hu moment %i'%(i+1))
    plt.legend(); plt.xlabel('log($\mu$)'); plt.show()
if __name__ == '__main__':
    main()
```

See the OpenCV documentation for a more detailed explanation of the functions used and their parameters. The histograms generated by the script are depicted in Figure 5.9. The spikes in the three histograms correspond to simple line segments, i.e., with large first moment and small higher moments. Hu moment spectra such as those shown in Figure 5.9 may be used, e.g., to identify matching contours for image-image registration and we shall return to them briefly in Section 5.6.2.

The geometric moments are projections of pixel intensities onto the monomials $f_{pq}(i,j) = i^p j^q$. The continuous equivalents $f_{pq}(\boldsymbol{x}) = x_1^p x_2^q$ are not orthogonal, i.e.,

$$\int f_{pq}(\boldsymbol{x}) f_{p'q'}(\boldsymbol{x}) d\boldsymbol{x} \neq 0, \quad p,q \neq p'q',$$

so that geometric moments and their invariant derivatives carry redundant information. Alternative moment systems can be defined in terms of orthogonal Legendre or Zernike polynomials, which are particularly relevant for image reconstruction; see Teague (1980).

5.3 Panchromatic sharpening

The change detection and classification algorithms that we will meet in the next chapters exploit both the spatial and the spectral information provided by remote sensing imagery. Many common satellite platforms (e.g.,

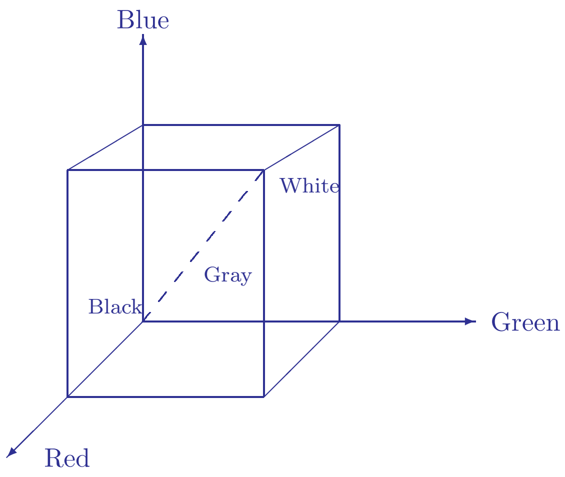


FIGURE 5.10
The RGB color cube.

LANDSAT 8 OLI, SPOT, IKONOS, QuickBird, GeoEye) supply co-registered panchromatic images with considerably higher ground resolution than that of the multispectral bands. However, without additional processing, application of spectral change detection or classification methods is restricted in the first instance to the poorer spatial resolution of the multispectral data.

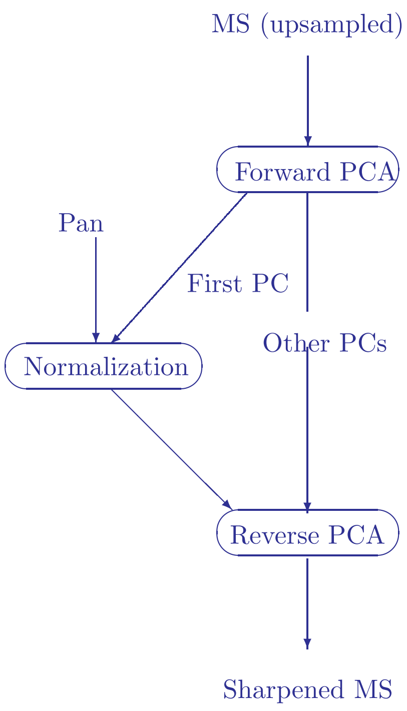


FIGURE 5.11
Panchromatic fusion with the principal components transformation.

Conventional image fusion techniques, such as the well-known HSV-transformation discussed below, can be used to sharpen the spectral components. However. the effect of mixing in the panchromatic image is often to "dilute" significantly the spectral information and, for instance in the case of classification, to reduce class separability in the multidimensional feature space. Another disadvantage of the HSV transformation is that one is restricted to using three of the available spectral channels. In the following, after outlining the HSV method, we consider a number of alternative (and better) fusion techniques. See Aanaes et al. (2008) for an excellent discussion of the issues involved in image fusion and Amro et al. (2011) for a review and comparison.

5.3.1 HSV fusion

In computers with 24-bit graphics (referred to as "true color" on Windows systems), three bands of a multispectral image can be displayed with 8 bits for each of the additive primary colors red, green, and blue. There are $2^{24} \approx 16$ million colors possible. The monitor displays the bands as an RGB color composite image which, depending on the choice of spectral bands and histogram functions, may or may not appear to be natural; see for instance Figure 1.1. The color of a given pixel in the image can be represented as a vector or point in a Cartesian coordinate system (the RGB color cube) as illustrated in Figure 5.10.

An alternative color representation is in terms of *hue*, *saturation* and *value* (HSV). Value (sometimes called *intensity*) can be thought of as the distance along an axis equidistant from the three orthogonal primary color axes in the RGB representation (the main diagonal of the RGB cube shown as the dashed line in Figure 5.10). Since all points on the axis have equal R, G, and B values, they appear as shades of gray. Hue refers to the actual color and is defined as an angle on a plane perpendicular to the value axis as measured from a reference line pointing in the direction the red vertex of the RGB cube. Saturation is the “amount” of color present and is represented by the radius of a circle described in the hue plane. A commonly used method for fusion of three low-resolution multispectral bands with a higher-resolution panchromatic image is to resample the former to the panchromatic resolution, transform it from RGB to HSV coordinates, replace the V component with the gray-scale panchromatic image, eventually after performing some kind of histogram matching or normalization of the two, and then to transform back to RGB space.

This procedure is referred to as *HSV panchromatic image sharpening* and is very easy to illustrate on the Earth Engine. The following is adapted from the GEE JavaScript documentation:

```
import ee
ee.Initialize()
# Load a Landsat 8 top-of-atmosphere reflectance image.
image = \
 ee.Image('LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318')
# Convert the RGB bands to the HSV color space.
hsv = image.select(['B4', 'B3', 'B2']).rgbToHsv()
# Swap in the panchromatic band and convert back to RGB.
sharpened = ee.Image.cat([
  hsv.select('hue'), hsv.select('saturation'),
  image.select('B8')]).hsvToRgb()
```

To make a before/after comparison, one only needs to export the sharpened image to Assets and then overlay it with the original scene; see the accompanying notebook.

5.3.2 Brovey fusion

In *Brovey* or *color normalized* fusion (Vrabel, 1996), each resampled multispectral pixel is multiplied by the ratio of the corresponding panchromatic pixel intensity to the sum of all of the multispectral intensities. The corrected pixel intensities $\bar{g}_k(i,j)$ in the kth fused multispectral channel are given by

$$\bar{g}_k(i,j) = g_k(i,j) \cdot \frac{g_p(i,j)}{\sum_{k'} g_{k'}(i,j)}, \tag{5.15}$$

where $g_k(i,j)$ is the (resampled) pixel intensity in the kth spectral band and $g_p(i,j)$ is the corresponding pixel intensity in the panchromatic image. This technique assumes that the spectral range spanned by the panchromatic image is essentially the same as that covered by the multispectral bands. This is seldom the case. Moreover, to avoid bias, the intensities used should be the radiances at the satellite sensors, implying use of the sensors’ calibration.

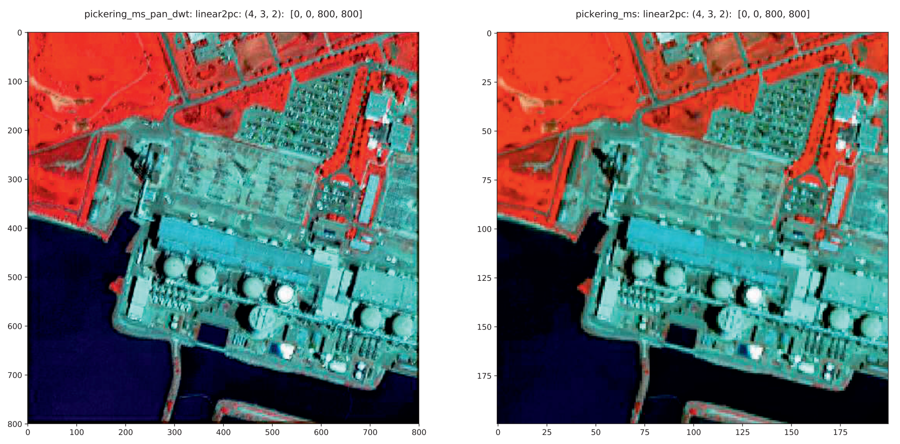

pickering_ms_pan_dwt: linear2pc: (4, 3, 2): [0, 0, 800, 800]
pickering_ms: linear2pc: (4, 3, 2): [0, 0, 800, 800]

FIGURE 5.12
Panchromatic sharpening of an IKONOS (©DigitalGlobe) multispectral image with the DWT filter bank. Left: panchromatically sharpened multispectral band of an image over a nuclear power reactor complex in Canada. Right: original band up sampled by a factor of 4.

5.3.3 PCA fusion

Panchromatic sharpening using principal components analysis (Welch and Ahlers, 1987) is similar to the HSV method. After the PCA transformation, the first principal component is replaced by the panchromatic image, again after some kind of normalization, and the transformation is inverted; see Figure 5.11. Image sharpening using HSV, Brovey, PCA and the closely related Gram–Schmidt transformation are available in most commercial image processing environments such as ENVI (Harris Geospatial Solutions) or Geomatica (PCI Geomatics).

5.3.4 DWT fusion

The discrete wavelet transform of a two-dimensional image was shown in Chapter 4 to be equivalent to an iterative application of the high-low-pass filter bank of Figure 4.10; see also Figure 4.11. The DWT filter bank can be used in an elegant way (Ranchin and Wald, 2000) to achieve panchromatic sharpening when the panchromatic and multispectral spatial resolutions differ by exactly a factor of 2^n for integer n. This is often the case, for example, in SPOT, LANDSAT 7 ETM+, LANDSAT 8 OLI, IKONOS, QuickBird, GeoEye, and ASTER imagery.

A DWT fusion procedure for IKONOS imagery for instance, in which the resolutions of panchromatic and the four multispectral bands differ exactly by a factor of 4, is as follows:

a) The panchromatic image, represented by $f_n(i,j)$, is filtered twice to give a degraded image $f_{n-2}(i,j)$ plus wavelet coefficients at two scales (see Figures 4.10 and 4.11).
b) Both the low-pass filtered portion of the panchromatic image as well as the four multispectral bands are filtered once again (the low-pass portions of all five bands now have 8 m pixel resolutions in the case of IKONOS data).
c) The high-frequency components C^z_{n-3}, $z = H, V, D$, are sampled to estimate radiometric normalization coefficients a^z and b^z for each multispectral band:

$$\begin{aligned} a^z &= \hat\sigma^z_{ms}/\hat\sigma^z_{pan} \\ b^z &= \hat m^z_{ms} - a^z \hat m^z_{pan}, \end{aligned} \tag{5.16}$$

where $\hat m^z$ and $\hat\sigma^z$ denote estimated mean and standard deviation, respectively.
d) These coefficients are then used to normalize the wavelet coefficients for the panchromatic image to those of each multispectral band at the $n-2$ and $n-1$ scales; see Exercise 7:

$$C^z_k(i,j) \to a^z C^z_k(i,j) + b^z, \quad z = H, V, D, \; k = n-2, n-1. \tag{5.17}$$

e) Finally, the degraded portion of the panchromatic image $f_{n-2}(i,j)$ is replaced with each of the four original multispectral bands in turn and the filter bank is inverted.

We thus obtain "what would be seen if the multispectral sensors had the resolution of the panchromatic sensor" (Ranchin and Wald, 2000). A Python script for panchromatic sharpening with the DWT is documented in Appendix C. Figure 5.12 shows an example of DWT pan-sharpening of IKONOS imagery, obtained in the Jupyter notebook for Chapter 5 with

```
run scripts/dwt -r 4 -b 4 -d [50,100,200,200] \
imagery/IKON_ms imagery/IKON_pan
=========================
   DWT Pansharpening
=========================
Sun Feb 25 14:44:30 2018
MS  file: imagery/pickering_ms
PAN file: imagery/pickering_pan
Wavelet correlations:
Band 1:     0.820   0.639   0.402
Band 2:     0.819   0.659   0.429
Band 3:     0.790   0.615   0.400
Band 4:     0.720   0.570   0.394
Result written to imagery/IKON_ms_pan_dwt

run scripts/dispms -f imagery/IKON_ms_pan_dwt \
                                   -p [4,3,2] -e 3 \
-F imagery/IKON_ms -D [50,100,200,200] \
                                   -P [4,3,2] -E 3
```

The achieved improvement more apparent in the notebook.

5.3.5 *À trous* fusion

The spectral fidelity of the pan-sharpened images obtained with the discrete wavelet transform is excellent, as will be shown below. However, the DWT is not *shift invariant* (Bradley, 2003), which means that small spatial displacements in the input array can cause major variations in the wavelet coefficients at their various scales. This has no effect on perfect reconstruction if one simply inverts the transformation. However, a small misalignment can occur when the multispectral bands are "injected" into the panchromatic image pyramid as described in the preceding section. This sometimes leads to spatial artifacts (blurring, shadowing, staircase effect) in the sharpened product (Yocky, 1996). These effects are visible in Figure 5.12 upon close inspection, again especially within the Jupyter notebook.

As an alternative to the DWT, the *à trous wavelet transform* (ATWT) has been proposed for image sharpening (Aiazzi et al., 2002). The ATWT is a multi resolution decomposition defined formally by a low-pass filter $H = \{h(0), h(1), \dots\}$ and a high-pass filter $G = \delta - H$, where δ denotes an all-pass filter. The high-frequency part is then just the difference between the original image and the low-pass filtered image. Not surprisingly, this transformation does not allow perfect reconstruction if the output is down sampled. Therefore down sampling is not performed at all. Rather, at the kth iteration of the low-pass filter, 2^{k-1} zeroes are inserted between the elements of H. This means that every other pixel is interpolated (averaged) on the first iteration:

$$H = \{h(0), 0, h(1), 0, \dots\},$$

while on the second iteration

$$H = \{h(0), 0, 0, h(1), 0, 0, \dots\},$$

etc. (hence, the name *à trous* = with holes). The low-pass filter is usually chosen to be symmetric (unlike the Daubechies wavelet filters for example). A good choice turns out to be the cubic B-spline filter (Núnez et al., 1999), $H = \{1/16, 1/4, 3/8, 1/4, 1/16\}$; see Exercise 9 in Chapter 3.

Figure 5.13 outlines the scheme implemented in the Python script `atwt.py` for ATWT panchromatic sharpening documented in Appendix C, assuming a difference in spatial resolution of a factor of two: The MS band is resampled to match the dimensions of the high-resolution band. The *à trous* transformation is applied to both bands (columns and rows are filtered with the up sampled cubic spline filter, with the difference from the original image determining the high-pass result). The high-frequency component of the panchromatic image is normalized to that of the MS image in a similar way as for DWT sharpening, Equations (5.16) and (5.17). Then, the smoothed panchromatic component is replaced by the filtered MS image and the transformation inverted.

The transformation is highly redundant and requires considerably more computer storage to implement. However, when used for image sharpening, it

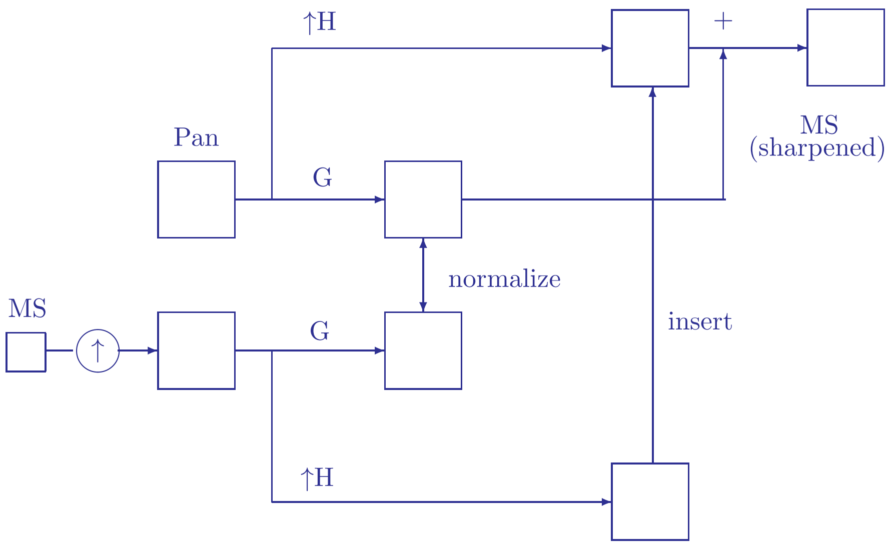


FIGURE 5.13
À trous image sharpening scheme for a multispectral to panchromatic resolution ratio of two. The symbol ↑H denotes the up sampled low-pass filter.

is much less sensitive to misalignment between the multispectral and panchromatic images. A comparison with the DWT method is made in Figure 5.14. Again, the differences are more clearly seen in the accompanying Jupyter notebook.

5.3.6 A quality index

Several quantitative measures have been suggested for determining the degree to which the spectral properties of the multispectral bands have been degraded in the pan-sharpening process (Tsai, 1982). Wang and Bovik (2002) suggest a particularly simple and intuitive measure of spectral fidelity between two image bands with pixel intensities represented by f and g:

$$\mathrm{Q} = \frac{\sigma_{\mathrm{fg}}}{\sigma_{\mathrm{f}}\sigma_{\mathrm{g}}} \cdot \frac{2\bar{\mathrm{f}}\bar{\mathrm{g}}}{\bar{\mathrm{f}}^2 + \bar{\mathrm{g}}^2} \cdot \frac{2\sigma_{\mathrm{f}}\sigma_{\mathrm{g}}}{\sigma_{\mathrm{f}}^2 + \sigma_{\mathrm{g}}^2} = \frac{4\sigma_{\mathrm{fg}}\bar{\mathrm{f}}\bar{\mathrm{g}}}{(\bar{\mathrm{f}}^2 + \bar{\mathrm{g}}^2)(\sigma_{\mathrm{f}}^2 + \sigma_{\mathrm{g}}^2)}, \tag{5.18}$$

where $\bar{f}$ $(\bar{g})$ and σ_f^2 (σ_g^2) are mean, and variance of band f (g) and σ_{fg} is the covariance of the two bands. The first factor in Equation (5.18) is seen to be the correlation coefficient between the two images, with values in $[-1, 1]$; the second factor compares their average brightness, with values in $[0, 1]$; and

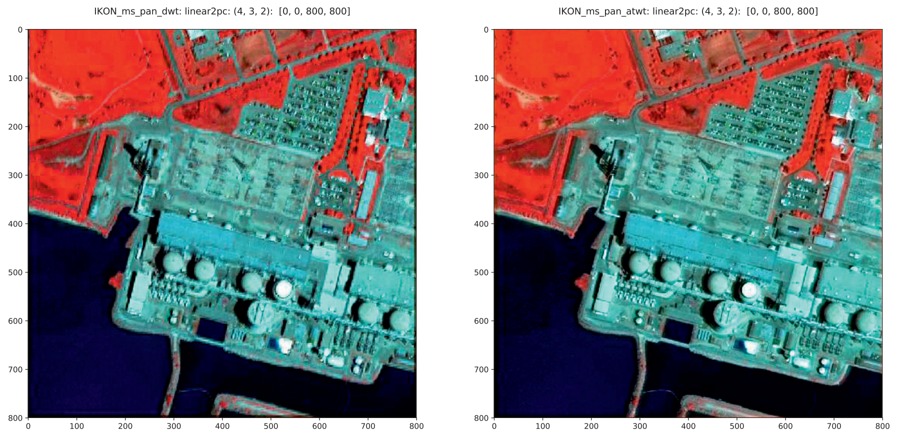

IKON_ms_pan_dwt: linear2pc: (4, 3, 2): [0, 0, 800, 800]
IKON_ms_pan_atwt: linear2pc: (4, 3, 2): [0, 0, 800, 800]

FIGURE 5.14
As Figure 5.12, except that the right-hand side now shows panchromatic sharpening with the *à trous* wavelet transform.

the third factor compares their contrasts, also in $[0, 1]$. Thus, perfect spectral correspondence would give a value $Q = 1$.[*]

TABLE 5.1
Quality indices for five panchromatic sharpening methods.

MS Band	HSV	PCA	G-S	DWT	ATWT
1	0.376	0.986	0.949	0.977	0.921
2	0.228	0.977	0.929	0.977	0.923
3	0.325	0.964	0.897	0.980	0.931
4	–	0.899	0.905	0.971	0.911
Mean	0.309	0.957	0.920	**0.976**	0.921

Since image quality is normally not spatially invariant, it is usual to compute Q_j in M sliding windows and then average over all of the windows:

$$\mathrm{Q} = \frac{1}{\mathrm{M}} \sum_{\mathrm{j}=1}^{\mathrm{M}} \mathrm{Q}_{\mathrm{j}}.$$

Table 5.1 gives the results for the HSV, PCA, Gram–Schmidt, DWT, and ATWT fusion methods for the image in Figures 5.12 and 5.14. The discrete

[*]It should be remarked that this index is "marginal" in the sense that the original and sharpened images are compared band-wise. A more stringent approach would be to test simultaneously for equal mean vectors and covariance matrices; see Anderson (2003).

wavelet transform performs best. The calculations were done with an ENVI/IDL script; see Canty (2014).

5.4 Radiometric correction of polarimetric SAR imagery

The most striking characteristic of SAR images, when compared to their visual/infrared counterparts, is the disconcerting "speckle" effect which makes visual interpretation very difficult. For the single-look data of Figure 1.2, for example, the effect is extreme. Speckle gives the appearance of random noise, but it is actually a deterministic consequence of the coherent nature of the radar signal. Although disadvantageous in some contexts, coherence effects in the amplitudes of the received radar signal constitute the basis of SAR interferometry and its many applications.

In our later treatment of classification and change detection, we will, however, be concerned primarily with SAR intensity (as opposed to amplitude) images, where speckle is indeed a nuisance. Further processing is necessary before the data can be used for the exacting tasks of thematic mapping or change detection. In this section we first examine the statistical properties of speckle in raw (single-look) polarimetric SAR images. We then consider the effect of averaging, or so-called *multi-looking*, and introduce the covariance matrix form for multilook polarimetric imagery. Finally, we describe two adaptive filtering procedures to further reduce speckle in the data.

5.4.1 Speckle statistics

Goodman (1984) gives a definitive treatment of the statistical distributions of the noise-like speckle patterns in coherent and partially coherent radiation reflected from rough surfaces. We consider speckle statistics here in the context of the measured radiation amplitude and intensity in a SAR image.

For single polarization transmission and reception, e.g., horizontal-horizontal (hh), the back scattered amplitude signal is, from Equation (1.3),

$$E_h^b = s_{hh} E_h^i. \tag{5.19}$$

The effect of the scattering amplitude s_{hh} is to introduce a change in amplitude and phase of the incident signal E_h^i, which is characteristic of the (bio)physical properties of the reflecting area. The area involved is determined by the ground resolution. For example, for TerraSAR-X in so-called *spotlight* acquisition mode, the pixel size may be as small as 1 m^2. The wavelength of the transmitted 9.65-GHz X-band signal is, on the other hand, only 31 mm. Therefore, even when the properties of the reflecting surface are uniform, random irregularities on the wavelength scale within the imaged pixel will act like a large number of *incremental scatterers*, leading to a random superposition of small scattering amplitudes. These in turn combine coherently to form the back scattered signal and are responsible for speckle. A simple model for this effect is to write Equation (5.19) in the form

$$E_h^b = \frac{E}{\sqrt{n}} \sum_{k=1}^{n} e^{\mathbf{i}\phi_k}. \tag{5.20}$$

Here we assume that the only effect of the incremental scatterers is to introduce local random phase shifts ϕ_k, $k = 1 \dots n$, in an overall scattered signal amplitude E.

We demonstrate now that the amplitude E_h^b has the complex normal distribution of Equation (2.50). To begin with, note that the phase shifts may be considered to be uniformly distributed on the interval $[-\pi, \pi]$, so that we have

$$|E_h^b|^2 = \frac{|E|^2}{n} \sum_k \sum_\ell e^{\mathbf{i}(\phi_k - \phi_\ell)} = \frac{|E|^2}{n} \sum_k e^{\mathbf{i}(\phi_k - \phi_k)} = |E|^2.$$

If we neglect the phase[*] of E, which is a characteristic of the entire pixel area, then we can express the real and imaginary parts of Equation (5.20) in the form

$$\mathrm{Re}(E_h^b) = \frac{E}{\sqrt{n}} \sum_k \cos\phi_k = \frac{E}{\sqrt{n}} X$$

$$\mathrm{Im}(E_h^b) = \frac{E}{\sqrt{n}} \sum_k \sin\phi_k = \frac{E}{\sqrt{n}} Y,$$

where $X = \sum_k \cos\phi_k$ and $Y = \sum_k \sin\phi_k$. Now the intensity of the back scattered signal can be written

$$|E_h^b|^2 = [\mathrm{Re}(E_h^b)]^2 + [\mathrm{Im}(E_h^b)]^2 = E^2 \frac{1}{n}(X^2 + Y^2). \tag{5.21}$$

We can think of X and Y as random variables. Since the sines and cosines contributing to X and Y are distributed symmetrically about zero, the means vanish: $\langle X \rangle = \langle Y \rangle = 0$. Furthermore,

$$\begin{aligned}
\mathrm{var}(X) &= \langle X^2 \rangle - \langle X \rangle^2 \\
&= \sum_k \sum_\ell \langle \cos\phi_k \cos\phi_\ell \rangle \\
&= \sum_k \langle \cos^2\phi_k \rangle = \sum_{k=1}^{n} \langle \frac{1}{2}(\cos 2\phi_k + 1) \rangle = \frac{n}{2},
\end{aligned}$$

with the same result for var(Y). Moreover, X and Y are uncorrelated:

$$\begin{aligned}
\mathrm{cov}(X, Y) &= \langle XY \rangle - \langle X \rangle \langle Y \rangle \\
&= \sum_k \sum_\ell \langle \cos\phi_k \sin\phi_\ell \rangle = 0.
\end{aligned}$$

[*]Or simply set it equal to zero.

Both X and Y are superpositions of a large number of identically distributed random variates ($\cos\phi_k$ and $\sin\phi_k$) and so the above results, together with the Central Limit Theorem 2.3, imply that X and Y are independently and normally distributed with zero mean and variance $\sigma^2 = n/2$. The joint probability density of X and Y is therefore

$$p(x,y) = \frac{1}{2\pi(n/2)} \exp\left(-\frac{1}{2(n/2)}(x^2+y^2)\right).$$

With $z = x + \mathbf{i}y$, this can be written in the form

$$p(z) = \frac{1}{\pi n} \exp\left(-z^* \left(\frac{1}{n}\right) z\right). \tag{5.22}$$

Comparison with Equation (2.50) shows that the complex random variable $Z = X + \mathbf{i}Y$, and hence the back scattered amplitude

$$E_h^b = \frac{E}{\sqrt{n}}(X + \mathbf{i}Y) = \frac{E}{\sqrt{n}}Z,$$

has a complex normal distribution as claimed.

As mentioned, the amplitude signals are interesting, e.g., for interferometric measurements, but what about the squared amplitude $|E_h^b|^2$, the quantity actually measured in the SAR intensity images which will concern us in classification and change detection applications? According to Theorem 2.6, the sum U of the squares of the standardized random variables X and Y,

$$U = \frac{X^2}{n/2} + \frac{Y^2}{n/2} = \frac{2}{n}(X^2+Y^2),$$

is chi-square distributed with 2 degrees of freedom, that is, with Equation (2.38),

$$p(u) = p_{\chi^2;2}(u) = \frac{1}{2}e^{-u/2}.$$

But this is also just the exponential probability density, Equation (2.37), with parameter $\beta = 1/2$. With Equation (5.21), the back scattered signal intensity is then

$$|E_h^b|^2 = E^2U/2. \tag{5.23}$$

A measurement of the radar cross section $|s_{hh}|^2$ derived from the emitted and received polarized signal intensities (see Equation (1.3)) will of course have the same exponential distribution. In other words, if the random variable G represents the measured intensity and $x = |s_{hh}|^2$ is the underlying signal, then

$$G \sim xU/2.$$

A simple application of Theorem 2.1 (Exercise 9(a)) now shows that G has an exponential probability density with $\beta = x$,

$$p(g) = \frac{1}{x}e^{-g/x}.$$

Since $\text{var}(G) = \langle G \rangle = x$, we can alternatively write

$$p(g) = \frac{1}{\langle G \rangle} e^{-g/\langle G \rangle}. \tag{5.24}$$

Thus, speckle behaves as an *exponentially distributed "noise" with mean and variance equal to the underlying signal strength.*

5.4.2 Multilook data

Multilook processing essentially corresponds to the averaging of neighborhood pixels, with the objective of reducing speckle and compressing the data. In practice, the averaging is often not performed in the spatial domain, but rather in the frequency domain during range/azimuth compression of the received signal.* The speckle reduction can be understood as follows. We showed above that a random variable G representing measurement of a physical quantity like $|s_{hh}|^2$ will have an exponential distribution with parameter $\beta = \langle G \rangle$. If these measurements are summed over m looks to give $\sum_{i=1}^m G_i$, then according to Theorem 2.5 the sums will be gamma distributed with $\beta = \langle G \rangle$ and $\alpha = m$, provided that the G_i are independent. The mean of the gamma distribution is $\alpha\beta = m\langle G \rangle$ and its variance is $\alpha\beta^2 = m\langle G \rangle^2$. Let us represent the look-averaged image by the random variable $\overline{G}$,

$$\overline{G} = \frac{1}{m} \sum_{i=1}^m G_i. \tag{5.25}$$

Then

$$\langle \overline{G} \rangle = \frac{1}{m} \cdot m\langle G \rangle = \langle G \rangle$$

and

$$\text{var}(\overline{G}) = \text{var}\left(\frac{1}{m} \sum_{i=1}^m G_i \right) = \frac{1}{m^2} \cdot m\langle G \rangle^2 = \frac{\langle G \rangle^2}{m} = \frac{\langle \overline{G} \rangle^2}{m}.$$

Hence, we see that *the variance of the look-averaged image decreases inversely as the number of looks.*

In real SAR imagery, the neighborhood pixel intensities contributing to the look average will be correlated to some extent. This is accounted for by defining an *equivalent number of looks* (ENL) whose definition is motivated by the above equation, that is, by solving it for m:

$$\text{ENL} = m = \frac{\langle \overline{G} \rangle^2}{\text{var}(\overline{G})}. \tag{5.26}$$

*Look averaging takes place at the cost of spatial resolution. See Richards (2009), Appendix D, for an excellent explanation of SAR image formation.

The ENL can be determined from Equation 5.26 by estimating $\langle\overline{G}\rangle^2$ and $\mathrm{var}(\overline{G})$ in homogeneous regions of an image, where the contribution of speckle to image statistics may be expected to dominate and is equivalent to the number of *independent* measurements averaged per pixel. For polarimetric data, one can simply average the values for each band, but see Exercise 10 and also Anfinsen et al. (2009a), who describe multivariate ENL estimators especially tuned to polarimetric SAR imagery. A Python script `enlml.py` for one of their methods is documented in Appendix C and an example of its use follows below. Now let's examine the case of multivariate polarimetric measurements in more detail.

The per-pixel polarimetric information in the full scattering matrix $\boldsymbol{S}$ of Equation (1.3), under the assumption of reciprocity ($s_{hv} = s_{vh}$), can be expressed as a three-component complex vector

$$\boldsymbol{s} = \begin{pmatrix} s_{hh} \\ \sqrt{2}s_{hv} \\ s_{vv} \end{pmatrix}, \tag{5.27}$$

where the $\sqrt{2}$ ensures that the total intensity (received signal power) is consistent. The total intensity is referred to as the *span*,

$$\mathrm{span} = \boldsymbol{s}^\dagger \boldsymbol{s} = |s_{hh}|^2 + 2|s_{hv}|^2 + |s_{vv}|^2, \tag{5.28}$$

and the corresponding gray-scale image as the *span image*.

Another representation of the polarimetric signal is the *complex covariance matrix* $\boldsymbol{c}$, obtained by taking the complex outer product of $\boldsymbol{s}$, Equation 5.27, with itself:

$$\boldsymbol{c} = \boldsymbol{s}\boldsymbol{s}^\dagger = \begin{pmatrix} |s_{hh}|^2 & \sqrt{2}\ s_{hh}s_{hv}^* & s_{hh}s_{vv}^* \\ \sqrt{2}\ s_{hv}s_{hh}^* & 2|s_{hv}|^2 & \sqrt{2}\ s_{hv}s_{vv}^* \\ s_{vv}s_{hh}^* & \sqrt{2}\ s_{vv}s_{hv}^* & |s_{vv}|^2 \end{pmatrix}. \tag{5.29}$$

The diagonal elements of $\boldsymbol{c}$ are real numbers, with span $= \mathrm{tr}(\boldsymbol{c})$, and the off-diagonal elements are complex. The covariance matrix representation contains all of the information in the polarized signal. It can also be averaged over the number of looks to give

$$\begin{aligned} \bar{\boldsymbol{c}} &= \frac{1}{m}\sum_{\nu=1}^{m} \boldsymbol{s}(\nu)\boldsymbol{s}(\nu)^\dagger = \langle \boldsymbol{s}\boldsymbol{s}^\dagger\rangle \\ &= \begin{pmatrix} \langle|s_{hh}|^2\rangle & \sqrt{2}\langle s_{hh}s_{hv}^*\rangle & \langle s_{hh}s_{vv}^*\rangle \\ \sqrt{2}\langle s_{hv}s_{hh}^*\rangle & \langle 2|s_{hv}|^2\rangle & \sqrt{2}\langle s_{hv}s_{vv}^*\rangle \\ \langle s_{vv}s_{hh}^*\rangle & \sqrt{2}\langle s_{vv}s_{hv}^*\rangle & \langle|s_{vv}|^2\rangle \end{pmatrix}. \end{aligned} \tag{5.30}$$

Rewriting the first equation above,

$$m\bar{\boldsymbol{c}} = \sum_{\nu=1}^{m} \boldsymbol{s}(\nu)\boldsymbol{s}(\nu)^\dagger. \tag{5.31}$$

This is seen to be a realization of a complex sample matrix like Equation (2.61). According to Theorem 2.11, $m\bar{\boldsymbol{c}}$ will have a complex Wishart distribution with 3×3 covariance matrix $\boldsymbol{c}$ and m degrees of freedom, provided that the vectors $\boldsymbol{s}(\nu)$ are independent and drawn from a complex multivariate normal distribution. This is ideally the case, as we saw in the one-dimensional situation, Equation (5.22); see also Oliver and Quegan (2004), Chapter 11. However, as already mentioned, the $\boldsymbol{s}(\nu)$ will generally be correlated. In order to account for this, the complex Wishart distribution is often parametrized with ENL (rather than m) degrees of freedom. The complex Wishart distribution of $m\bar{\boldsymbol{c}}$ will be exploited in Chapters 7 and 9 in the contexts of polarimetric SAR classification and change detection.

An equivalent and often preferred representation is in terms of the *Pauli decomposition*,

$$\boldsymbol{k} = \frac{1}{\sqrt{2}}\begin{pmatrix} s_{hh} + s_{vv} \\ s_{hh} - s_{vv} \\ 2s_{hv} \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix}$$

with the corresponding look-averaged polarimetric matrix form

$$\bar{\boldsymbol{t}} = \langle \boldsymbol{k}\boldsymbol{k}^\dagger \rangle = \begin{pmatrix} \langle |k_1|^2 \rangle & \langle k_1 k_2^* \rangle & \langle k_1 k_3^* \rangle \\ \langle k_2 k_1^* \rangle & \langle |k_2|^2 \rangle & \langle k_2 k_3^* \rangle \\ \langle k_3 k_1^* \rangle & \langle k_3 k_2^* \rangle & \langle |k_3|^* \rangle \end{pmatrix}.$$

This is sometimes preferred because the corresponding reflected intensities can be (roughly) interpreted as single bounce ($|s_{hh}+s_{vv}|^2$), e.g., from agricultural fields or grasslands, double bounce ($|s_{hh}-s_{vv}|^2$) from buildings and man-made structures and volume scattering ($|s_{hv}|^2$) from forest canopies.

The RADARSAT-2 image used in the following demonstrations consists of 9 bands, namely the three diagonal elements and the real and complex parts of the 3 elements above the diagonal in Equation (5.30.)* It is in the Pauli representation, so the matrix elements are interpreted accordingly. The image is shown in Figure 5.15, as generated in the Jupyter notebook with

```
run scripts/dispms -f imagery/RS2_20090829.tif \
                                        -p [6,9,1]
```

The ENL for this image can be estimated with the aforementioned multivariate method (Anfinsen et al., 2009a) which is implemented in the Python script `enlml.py`; see Appendix C. We choose here a spatial subset at center right in Figure 5.15 covering a forested area where the speckle statistics are well-developed:

```
run scripts/enlml -d [500,400,300,300] \
                    imagery/RS2_20090829.tif
```

*Because the RADARSAT-2 image is rather large, the corresponding examples in the accompanying Jupyter notebook for this Chapter make use of a smaller, dual pol Sentinel-1 image over the Frankfurt Airport.

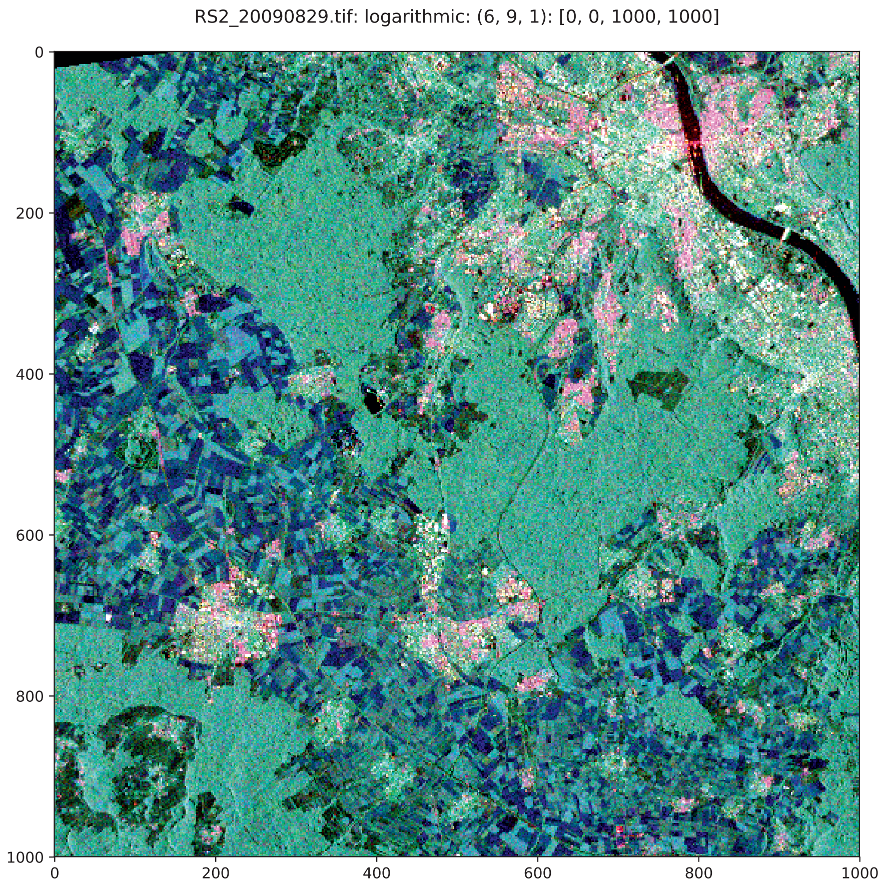
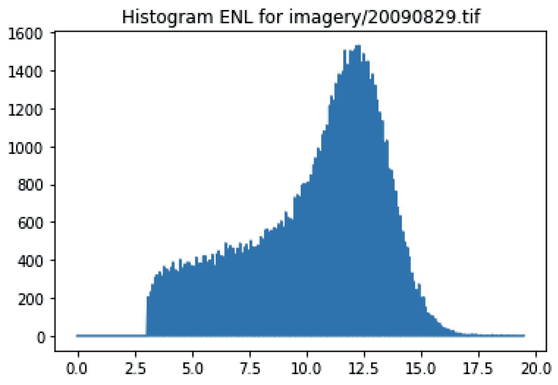

RS2_20090829.tif: logarithmic: (6, 9, 1): [0, 0, 1000, 1000]
0
200
400
600
800
1000
0
200
400
600
800
1000

FIGURE 5.15
RGB color composite of a quad polarimetric RADARSAT-2 image acquired over an area southwest of Bonn, Germany on August 29, 2009, in the Pauli representation RGB = $(|s_{hh} - s_{vv}|^2, |s_{hv}|^2, |s_{hh} + s_{vv}|^2)$. RADARSAT-2 Data and Products ©MacDonald, Dettwiler, and Associates Ltd. (2009–2101) – All Rights Reserved. RADARSAT is an official trademark of the Canadian Space Agency.

The output cell is shown in Figure 5.16. Figure 5.17 shows the ENL image obtained by sliding a 7×7 window across the subset and determining the value within the window. The ENL is taken to coincide with the mode (the most common value) of the histogram of the ENL image and is located at 12.5 in Figure 5.16.

The scattering vectors given in Equation (5.27) correspond to so-called full, or *quad polarimetric* SAR. Satellite-based SAR sensors often operate in

```
=========================
     ENL Estimation
=========================
Sat Mar  3 15:31:11 2018
infile:  imagery/20090829.tif
Quad polarimetry
filtering...
row:  100  200
```



```
 ENL image written to: imagery/20090829_enl.tif
elapsed time: 8.59535217285
```

FIGURE 5.16
Determining the ENL from a spatial subset of the image in Figure 5.15 with the Python script `enlml.py`.

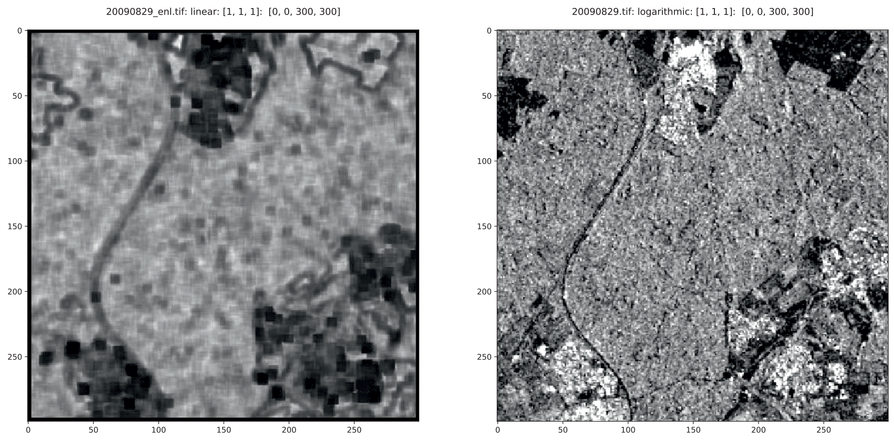

20090829_enl.tif: linear: [1, 1, 1]: [0, 0, 300, 300]
20090829.tif: logarithmic: [1, 1, 1]: [0, 0, 300, 300]
0
50
100
150
200
250

FIGURE 5.17
Left: ENL values determined in a 7×7 sliding window from a spatial subset of the polarimetric SAR image in Figure 5.15. Right: the first band of the original image showing the same subset.

reduced polarization modes, emitting only one polarization and receiving two (dual polarization) or one (single polarization). The look-averaged covariance matrices are reduced in dimension correspondingly: for dual polarization and horizontal transmission,

$$\bar{\boldsymbol{c}} = \begin{pmatrix} \langle |s_{hh}|^2 \rangle & \langle s_{hh} s_{hv}^* \rangle \\ \langle s_{hv} s_{hh}^* \rangle & \langle |s_{hv}|^2 \rangle \end{pmatrix}, \tag{5.32}$$

and, for single polarization and horizontal transmission, simply the intensity image

$$\langle \overline{G} \rangle = \langle |s_{hh}|^2 \rangle. \tag{5.33}$$

In the former case, the observations $m\bar{\boldsymbol{c}}$ are complex Wishart distributed (Equation (2.62)) with $N = 2$, in the latter case they are gamma distributed.

5.4.3 Speckle filtering

Let us represent an m look-averaged SAR scalar intensity image by the random variable G (dropping for convenience the overbar) with mean $\langle G \rangle = x$, where x is again the underlying signal. The variance falls off inversely with m, so $\mathrm{var}(G) = x^2/m$. Since G is gamma-distributed, its parameters α and β satisfy

$$\alpha\beta = x, \quad \alpha\beta^2 = x^2/m,$$

from which follows $\alpha = m$ and $\beta = x/m$. The density function of G for given value of x is therefore (see Equation (2.33))

$$p(g \mid x) = \frac{1}{(x/m)^m \Gamma(m)} g^{m-1} e^{-gm/x}. \tag{5.34}$$

Let $G = xV$. Then, again applying Theorem 2.1 (Exercise 9(b)), it follows that V has the density

$$p(v) = \frac{m^m}{\Gamma(m)} v^{m-1} e^{-vm}. \tag{5.35}$$

Therefore, in terms of the observed pixel intensities g (realizations of G), we can write

$$g = xv, \tag{5.36}$$

where v is a realization of a gamma-distributed random variable V with density given by Equation (5.35), and with mean 1 and variance $1/m$.

Because of this special *multiplicative noise* nature of speckle, conventional smoothing filters of the kind we met in Chapter 4 are not particularly suitable as an aid to SAR image interpretation. Moreover, since we will be especially concerned with polarimetric applications, a filtering algorithm which not only treats the statistical properties of the pixels correctly but also preserves the polarization properties is to be preferred. In the following, we will explain in some detail two speckle filters which try to meet these requirements, and also provide Python scripts which implement them.

5.4.3.1 Minimum mean square error (MMSE) filter

Lee et al. (1999) proposed an adaptive SAR speckle filter which is performed in two steps:

1. Using the span image only, Equation (5.28), filter weights are computed across the image for local windows of size $n \times n$, where, typically, $n = 7$. The filter used is an adaptive one which takes into account the local speckle statistics. It is derived below.
2. The filter weights are then applied uniformly and separately to the elements of the look-averaged covariance matrix, Equation (5.30). By ensuring that all elements of the covariance matrix are filtered by the same amount, the polarimetric properties are preserved.

The speckle filter in step 1 calculates a filtered pixel intensity estimate $\hat{x}$ of the signal x at the window center assuming to the linear relation

$$\hat{x} = a\bar{g} + bg, \tag{5.37}$$

where $\bar{g}$ is the mean of the (locally) observed pixel intensities in the window. The local mean $\bar{g}$ serves an initial estimate of the signal in the absence of speckle noise. This estimate is adaptively corrected in Equation (5.37) by choice of the parameters a and b and by the observed pixel intensity g to estimate the actual signal x at the window center. The filter parameters are chosen so as to minimize the mean square error

$$E = \langle(\hat{x} - x)^2\rangle = \langle(a\bar{g} + bg - x)^2\rangle,$$

where $\langle\ \cdot\ \rangle$ refers to mean value within the local window. Thus

$$\bar{g} = \langle g\rangle \approx \langle x\rangle.$$

To obtain the parameters, we set the partial derivatives of E equal to zero:

$$\begin{aligned}\frac{\partial E}{\partial a} &= \langle 2\bar{g}(a\bar{g} + bg - x)\rangle = 0\\ \frac{\partial E}{\partial b} &= \langle 2g(a\bar{g} + bg - x)\rangle = 0.\end{aligned} \tag{5.38}$$

The first of Equations (5.38) is equivalent to

$$a\bar{g} + b\bar{g} - \bar{g} = 0,$$

which implies

$$a = 1 - b. \tag{5.39}$$

Substituting this into the second of Equations (5.38) to eliminate a, we obtain

$$\langle g(x - \bar{g})\rangle + \langle b(\bar{g} - g)g\rangle = 0. \tag{5.40}$$

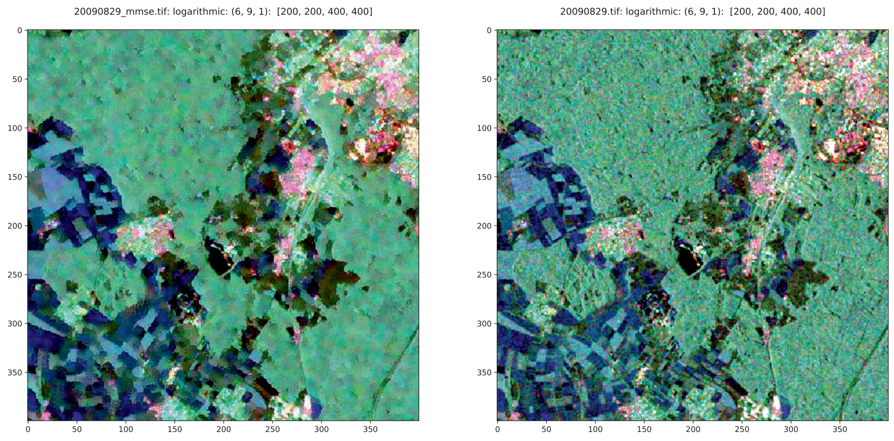

20090829_mmse.tif: logarithmic: (6, 9, 1): [200, 200, 400, 400]
20090829.tif: logarithmic: (6, 9, 1): [200, 200, 400, 400]

FIGURE 5.18
Left: MMSE filter of the RASARSAT-2 quad polarimetric image of Figure 5.15. Right: the original image.

From Equation (5.36), the first term on the left-hand side can be written

$$\langle g(x-\bar{g})\rangle = \langle xv(x-\bar{g})\rangle = \langle v\rangle\langle x(x-\bar{g})\rangle,$$

assuming that speckle noise v and signal intensity x are statistically independent. But $\langle v\rangle = 1$ and therefore

$$\langle g(x-\bar{g})\rangle = \langle x(x-\bar{g})\rangle.$$

Since $\langle \bar{g}(x-\bar{g})\rangle = \bar{g}\langle (x-\bar{g})\rangle = 0$, this can be rewritten in the form

$$\begin{aligned}\langle g(x-\bar{g})\rangle &= \langle x(x-\bar{g}) - \bar{g}(x-\bar{g})\rangle = \langle (x-\bar{g})^2\rangle \\ &= \langle (x-\bar{x})^2\rangle = \text{var}(x).\end{aligned}$$

In a similar way, we can write the second term in Equation (5.40) as

$$\langle b(\bar{g}-g)g\rangle = -b\langle (g-\bar{g})^2\rangle = -b\ \text{var}(g)$$

and we get

$$b = \frac{\text{var}(x)}{\text{var}(g)}. \tag{5.41}$$

Combining Equations (5.37), (5.39), and (5.41), we obtain finally the adaptive filter

$$\hat{x} = \bar{g} + \frac{\text{var}(x)}{\text{var}(g)}(g-\bar{g}). \tag{5.42}$$

In regions of the image where the signal variance var(x) is small compared to the overall variance (homogeneous regions), i.e., $\hat{x} \approx \bar{g}$, the pixels intensity is replaced by the window average. On the other hand, when the true signal intensity is varying strongly, $\mathrm{var}(x) \approx \mathrm{var}(g)$, then $\hat{x} \approx g$ and no filtering takes place at all.

While the overall variance within the window $\mathrm{var}(g)$ in Equation (5.42) can be determined at once, the signal variance $\mathrm{var}(x)$ is not directly available. However, we can again exploit the statistical independence of x and v and write, with Equation (5.36),

$$\langle g^2 \rangle = \langle x^2 \rangle \langle v^2 \rangle .$$

Equivalently,

$$\mathrm{var}(g) + \langle g \rangle^2 = (\mathrm{var}(x) + \langle x \rangle^2)(\mathrm{var}(v) + \langle v \rangle^2).$$

Solving for $\mathrm{var}(x)$ and using $\langle x \rangle = \langle g \rangle = \bar{g}$, $\mathrm{var}(v) = 1/m$, and $\langle v \rangle = 1$, we obtain

$$\mathrm{var}(x) = \frac{\mathrm{var}(g) - \frac{1}{m}\bar{g}^2}{1 + \frac{1}{m}}. \tag{5.43}$$

Appendix C documents the Python script `mmse_filter.py` that implements the MMSE adaptive filter for polarimetric SAR imagery and which includes a directionally sensitive averaging window; see Lee et al. (1999). Figure 5.18 shows an example of its application to quad polarimetric, multilook RADARSAT-2 data:

```
run scripts/mmse_filter imagery/RS2_20090829.tif 12.5

=========================
       MMSE_FILTER
=========================
Mon Mar  5 09:54:26 2018
infile:  imagery/RS2_20090829.tif
number of looks: 12.5
Determining filter weights from span image
row:
 50  100  ...  950   done
Filtering covariance matrix elements
band: 1
band: 2
...
band: 9
result written to: imagery/RS2_20090829_mmse.tif
elapsed time: 464.533332109

run scripts/dispms -f imagery/RS2_20090829_mmse.tif \
-p [6,9,1] -d [200,200,400,400] -F imagery/ \
 RS2_20090829.tif -P [6,9,1] -D [200,200,400,400]
```

Note that the last input parameter in the first command of the above listing is the ENL, which we already determined to be 12.5. The high degree of

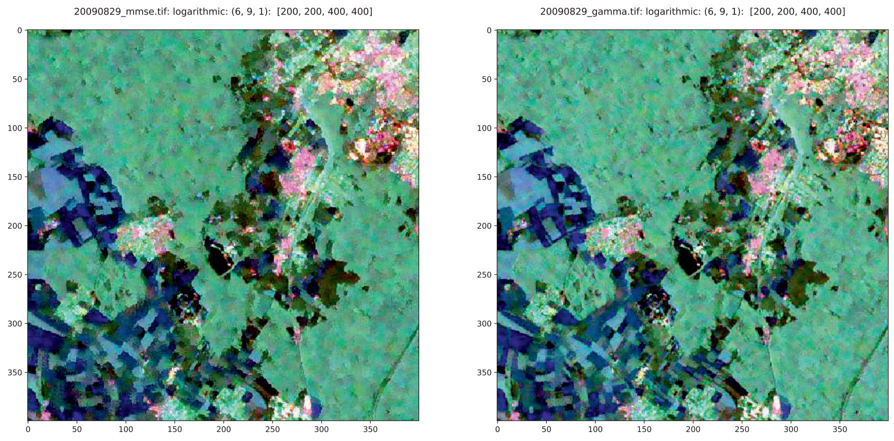

20090829_mmse.tif: logarithmic: (6, 9, 1): [200, 200, 400, 400]
20090829_gamma.tif: logarithmic: (6, 9, 1): [200, 200, 400, 400]

FIGURE 5.19
As Figure 5.18 but with the gamma MAP filter on the right-hand side.

variability of the filter over the SAR image means that the global ENL which results from the initial multilook processing will no longer be valid in the de-speckled image (Anfinsen et al., 2009b).

5.4.3.2 Gamma MAP filter

Oliver and Quegan (2004) discuss iterative versions of the MMSE filter and also other de-speckling algorithms that take into account explicit statistical models for the signal x. The simplest of these, referred to as *gamma maximum a posteriori* (gamma MAP) de-speckling, may be derived from Bayes' Theorem (Theorem 2.14). The *a posteriori* conditional probability for x, given intensity measurement g is (see Equation (2.70)),

$$\Pr(x \mid g) = \frac{p(g \mid x)\Pr(x)}{p(g)}, \tag{5.44}$$

where $p(g \mid x)$ is given by Equation (5.34), $\Pr(x)$ is the prior probability for x and $p(g)$ is the total probability density for g. This formulation allows us to include prior knowledge of the signal statistics (or texture) if available. An empirical statistical model for x is suggested by measurements of back scatter from ocean waves (Oliver and Quegan, 2004), namely

$$\Pr(x) \sim \left(\frac{\alpha}{\mu}\right)^{\alpha} \frac{x^{\alpha-1}}{\Gamma(\alpha)} e^{-\alpha x/\mu}. \tag{5.45}$$

This is just the gamma probability density with $\beta = \mu/\alpha$, and hence with mean $\alpha\beta = \mu$ and variance

$$\text{var}(x) = \alpha\beta^2 = \mu^2/\alpha. \tag{5.46}$$

The parameters μ and α can be estimated as follows. By passing an $n \times n$ window over the image, we can obtain $\bar{g} = \langle g \rangle$ and $\mathrm{var}(g)$. Then the estimates are

$$\hat{\mu} = \bar{g}, \tag{5.47}$$

and, with Equations (5.43) and (5.46),

$$\hat{\alpha} = \frac{\hat{\mu}^2}{\mathrm{var}(x)} = \frac{\bar{g}^2}{\mathrm{var}(x)} = \frac{1 + 1/m}{\mathrm{var}(g)/\bar{g}^2 - 1/m}. \tag{5.48}$$

Now, combining Equations (5.34), (5.44), and (5.45)[*], the posterior probability for x given measurement g is

$$\Pr(x \mid g) \sim \frac{1}{(x/m)^m \Gamma(m)} g^{m-1} e^{-gm/x} \left(\frac{\alpha}{\mu}\right)^{\alpha} \frac{x^{\alpha-1}}{\Gamma(\alpha)} e^{-\alpha x/\mu} =: L \tag{5.49}$$

Taking the logarithm,

$$\begin{aligned} \log L = {} & m \log m - m \log x + (m-1)\log g - \log \Gamma(m) - mg/x \\ & + \alpha \log \alpha - \alpha \log \mu + (\alpha - 1)\log x - \log \Gamma(\alpha) - \alpha x/\mu. \end{aligned}$$

We get the maximum *a posteriori* (MAP) value for x given the observed pixel intensity g by maximizing $\log L$ with respect to x:

$$\frac{d}{dx}\log L = -m/x + mg/x^2 + (\alpha - 1)/x - \alpha/\mu = 0.$$

This leads to a quadratic equation for the most probable signal intensity x,

$$\frac{\alpha}{\mu}x^2 + (m + 1 - \alpha)x - mg = 0, \tag{5.50}$$

where the parameters μ and α are estimated locally with Equations (5.47) and (5.48). Note, from Equation (5.48), that in homogeneous regions where $m \approx \bar{g}^2/\mathrm{var}(g)$, $\hat{\alpha} \to \infty$. In that case, from Equation (5.50), $x \approx \hat{\mu} = \bar{g}$, as in the MMSE algorithm.[†]

The gamma MAP filter is not appropriate to the complex off-diagonal matrix elements in Equation (5.30) as their *a priori* statistics are not well understood. Appendix C provides documentation for the Python routine `gamma_filter.py` for gamma MAP filtering of the diagonal terms. Figure 5.19 shows an example:

[*] And neglecting the total probability $p(g)$, which doesn't depend on x.

[†] If the signal intensity x is integrated out of Equation (5.49), one obtains the probability density function $p(g)$ for the measured intensity in the presence of both speckle and texture, namely the so-called K-distribution; see Oliver and Quegan (2004), Chapter 5.

```
run scripts/gamma_filter imagery/RS2_20090829.tif 12.5

=========================
    GAMMA MAP FILTER
=========================
Mon Mar  5 10:19:01 2018
infile:  imagery/RS2_20090829.tif
equivalent number of looks: 12.500000
Attempting parallel computation ...
available engines: [0, 1]
filtering 3 diagonal matrix element bands ...
result written to: imagery/RS2_20090829_gamma.tif
elapsed time: 287.622186899

run scripts/dispms -f imagery/RS2_20090829_mmse.tif \
                -p [6,9,1] -d [200,200,400,400] \
-F imagery/RS2_20090829_gamma.tif \
                -P [2,3,1] -D [200,200,400,400]
```

As the output hints, the `gamma_filter.py` script can take advantage of the parallel processing power of the Jupyter notebook to run several so-called *IPython engines* in parallel. In this case, since the computer used had only two cores available, only two engines were started. See Appendix C for details. Comparison of the images in Figure 5.19 indicates that the gamma MAP filter is somewhat more successful in preserving details than the MMSE filter.

5.4.3.3 Temporal filtering

The MMSE and gamma MAP filters are of course *spatial* filters. If a sequence of SAR images is conveniently available over a time period in which relevant changes are considered to be negligible, one can simply average them in order to reduce the effect of speckle. As we saw in the discussion of multilook averaging, we expect the variance of the temporally averaged image also to decrease in inverse proportion to the number of images in the sequence.

A sequence of Sentinel-1 SAR images can easily be obtained via the GEE and used to illustrate temporal speckle filtering. Below we choose a region near Jülich, Germany, and as a time interval, the month of May, 2017:

```
import ee, math
ee.Initialize()

# convert from decibels to linear scale
def linearize(current):
    return current.multiply(
      ee.Image.constant(math.log(10.0)/10.0)).exp()

# collect a time series
rect = ee.Geometry.Rectangle([6.31,50.83,6.58,50.95]);
collection = ee.ImageCollection('COPERNICUS/S1_GRD')\
```

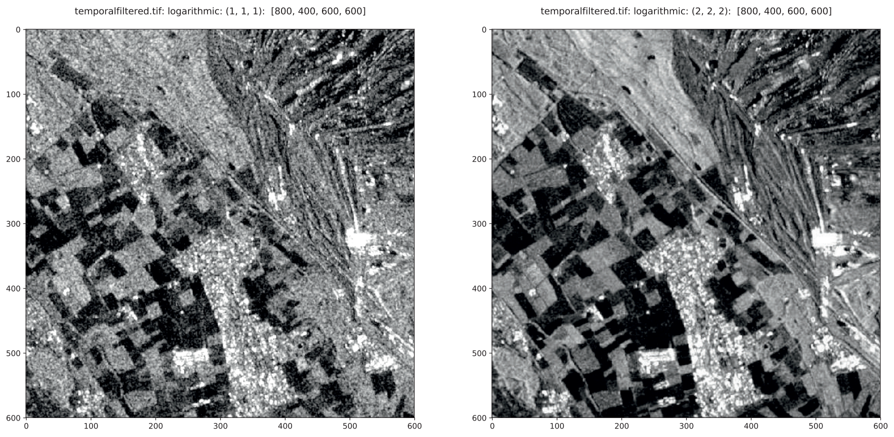

temporalfiltered.tif: logarithmic: (1, 1, 1): [800, 400, 600, 600]
temporalfiltered.tif: logarithmic: (2, 2, 2): [800, 400, 600, 600]

FIGURE 5.20
Illustrating temporal filtering. Left: Subset of the first band (VV) of the first image in a collection of 11 Sentinel-1 intensity images. Right: The mean of the 11 bands.

```
.filterBounds(rect)\
.filterDate(ee.Date('2017-05-01'),ee.Date('2017-06-01'))\
.filter(ee.Filter.eq('resolution_meters', 10)) \
.filter(ee.Filter.eq('orbitProperties_pass',
  'ASCENDING'))\
.map(linearize)
# series length
count =  collection.toList(100).length()
print('series␣length:␣%i'%count.getInfo())
# temporally filtered image band
filtered = collection.mean().select(0).clip(rect)
# unfiltered image band for comparison
unfiltered = ee.Image(collection.first()).select(0)\
                                        .clip(rect)
# export to Google Drive
outimage = ee.Image.cat(unfiltered,filtered)
gdexport = ee.batch.Export.image.toDrive(outimage,
     description='driveExportTask',
     folder = 'EarthEngineImages',
     fileNamePrefix='temporalfiltered',scale=10)
gdexport.start()
```

The exported result can be downloaded from Google Drive and then uploaded to the `imagery` directory for display in the Jupyter notebook with `dispms.py`; see Figure 5.20. Of course features which change over the time interval chosen for filtering are lost or falsified in the final result. This can be corrected with a sequential change detection procedure as we will see in Chapter 9.

5.5 Topographic correction

Satellite images are two-dimensional representations of the three-dimensional Earth surface. The inclusion of the third dimension—the elevation—is required for terrain modeling and accurate geo-referencing.

5.5.1 Rotation, scaling, and translation

Transformations of spatial coordinates in three dimensions, which involve only rotations, scaling and translations, can be conveniently represented by a 4×4 transformation matrix $\boldsymbol{A}$,

$$\boldsymbol{v}^* = \boldsymbol{A}\boldsymbol{v}, \tag{5.51}$$

where $\boldsymbol{v}$ is the column vector containing the original coordinates

$$\boldsymbol{v} = (X, Y, Z, 1)^\top$$

and $\boldsymbol{v}^*$ contains the transformed coordinates

$$\boldsymbol{v}^* = (X^*, Y^*, Z^*, 1)^\top.$$

For example, the translation

$$\begin{aligned} X^* &= X + X_0 \\ Y^* &= Y + Y_0 \\ Z^* &= Z + Z_0 \end{aligned}$$

corresponds to the transformation matrix

$$\boldsymbol{T} = \begin{pmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

a uniform scaling by 50% to

$$\boldsymbol{S} = \begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and a simple rotation θ about the Z-axis to

$$\boldsymbol{R}_\theta = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

etc. The combined rotation, scaling, translation (RST) transformation is in this case

$$\boldsymbol{v}^* = (\boldsymbol{R}_\theta \boldsymbol{S}\boldsymbol{T})\boldsymbol{v} = \boldsymbol{A}\boldsymbol{v}.$$

$\boldsymbol{A}$ is also referred to as a *similarity transformation* and is a special case of the more general *affine transformation*,

$$\boldsymbol{A} = \begin{pmatrix} a_1 & a_2 & a_3 & X_0 \\ a_4 & a_5 & a_6 & Y_0 \\ a_7 & a_8 & a_9 & Z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{a} & \boldsymbol{X} \\ \boldsymbol{0}^\top & 1 \end{pmatrix}, \tag{5.52}$$

where the 3×3 matrix $\boldsymbol{a}$ is non singular. The two-dimensional counterpart of the similarity transformation will be met in Sections 5.6.1 and 5.6.3 when we discuss image–image registration.

5.5.2 Imaging transformations

An imaging (or perspective) transformation projects 3D points onto a plane. It is used to describe the formation of a camera image and, unlike the RST transformation, is nonlinear since it involves division by coordinate values.

In Figure 5.21, a camera coordinate system (x, y, x) is shown, aligned with a *world coordinate system* (X, Y, Z) describing the terrain to be imaged. The camera focal length is λ. From simple geometry (similar triangles), we obtain expressions for the image plane coordinates in terms of the world coordinates:

$$\begin{aligned} x &= \frac{\lambda X}{\lambda - Z} \\ y &= \frac{\lambda Y}{\lambda - Z}. \end{aligned} \tag{5.53}$$

Solving for the X and Y world coordinates:

$$\begin{aligned} X &= \frac{x}{\lambda}(\lambda - Z) \\ Y &= \frac{y}{\lambda}(\lambda - Z). \end{aligned} \tag{5.54}$$

Thus, in order to extract the geographical coordinates (X, Y) of a point on the Earth's surface from its image coordinates (x, y), we require knowledge of the elevation Z. Correcting for the elevation in this way constitutes the process of *orthorectification*, resulting in an image without perspective distortion — every point appears as if viewed from directly above.

5.5.3 Camera models and RFM approximations

Equations (5.54) are very simple because they assume that the world and image coordinates coincide. But in order to apply them, one has first to transform the world coordinate system to the satellite coordinate system. This can be done in a straightforward way with the rotation and translation transformations introduced above. However, it requires accurate knowledge of the

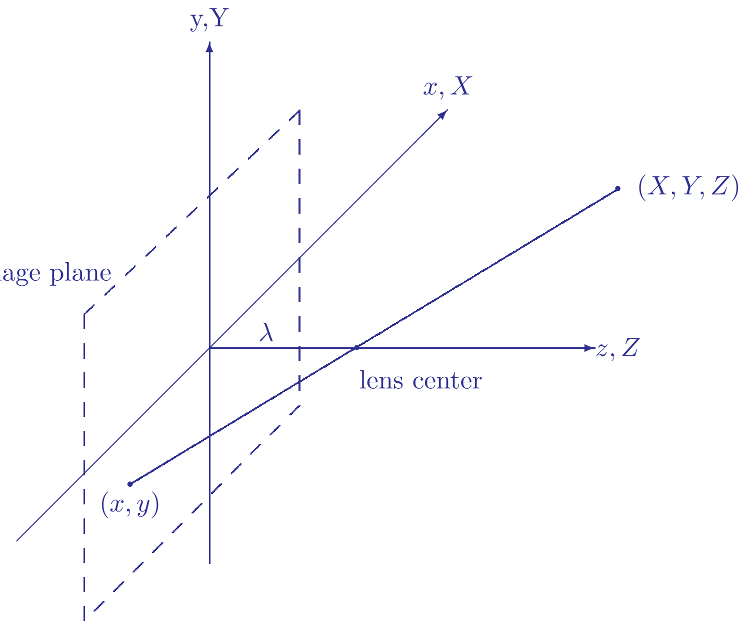


FIGURE 5.21
Basic imaging process. Camera coordinates are (x,y,z) and coincide with world coordinates (X,Y,Z). The focal length is λ.

altitude, orientation and detailed properties of the satellite imaging system at the time of the image acquisition (or, more precisely, *during* the acquisition, since the latter is never instantaneous). The resulting nonlinear equations that relate image and world coordinates, referred to as the *co-linearity equations*, constitute the *camera model* for that particular image.

Direct use of a camera model for image processing is complicated as it requires extremely exact, sometimes proprietary information about the sensor system and its orbit. An alternative exists if the image provider also supplies a so-called *rational function model* (RFM) with the imagery. The RFM approximates the camera model for each acquisition in terms of ratios of polynomials; see, e.g., Tao and Hu (2001). The RFMs have the form

$$\begin{aligned} r' &= f(X',Y',Z') = \frac{a(X',Y',Z')}{b(X',Y',Z')} \\ c' &= g(X',Y',Z') = \frac{c(X',Y',Z')}{d(X',Y',Z')}, \end{aligned} \tag{5.55}$$

where c' and r' are the column and row (x,y) coordinates in the image plane relative to an origin (c_0, r_0) and scaled by a factor c_s resp. r_s, i.e.,

$$c' = \frac{c - c_0}{c_s}, \quad r' = \frac{r - r_0}{r_s}.$$

Similarly X', Y', and Z' are relative, scaled world coordinates:

$$X' = \frac{X - X_0}{X_s}, \quad Y' = \frac{Y - Y_0}{Y_s}, \quad Z' = \frac{Z - Z_0}{Z_s}.$$

The polynomials a, b, c, and d in Equation (5.55) are typically to third order in the world coordinates, for example,

$$\begin{aligned}
&a(X, Y, Z) = \\
&a_0 + a_1X + a_2Y + a_3Z + a_4XY + a_5XZ + a_6YZ + a_7X^2 + a_8Y^2 + a_9Z^2 \\
&\ + a_{10}XYZ + a_{11}X^3 + a_{12}XY^2 + a_{13}XZ^2 + a_{14}X^2Y + a_{15}Y^3 + a_{16}YZ^2 \\
&\ + a_{17}X^2Z + a_{18}Y^2Z + a_{19}Z^3.
\end{aligned}$$

The advantage of using *ratios* of polynomials is that they are less subject to interpolation error than simple polynomials.

For a given acquisition, the provider will fit an RFM to its camera model with a least squares fitting procedure using a two- and three-dimensional grid of points covering the image resp. world spaces. The RFM is capable of representing the camera model extremely well and can be used as a replacement for it. Both Space Imaging Corp. and DigitalGlobe Inc., for example, provide RFMs with their so-called "ortho-ready" high-resolution imagery (IKONOS, QuickBird, WorldView-1 platforms).

To illustrate a simple use of RFM data, consider a vertical structure in a high-resolution image, such as a chimney or building facade. Suppose we determine the image coordinates of the bottom and top of the structure to be (r_b, c_b) and (r_t, c_t), respectively. Then, from Equations (5.55),

$$\begin{aligned}
r'_b &= f(X', Y', Z'_b) \\
c'_b &= g(X', Y', Z'_b) \\
r'_t &= f(X', Y', Z'_t) \\
c'_t &= g(X', Y', Z'_t),
\end{aligned} \tag{5.56}$$

since the (X', Y') coordinates must be the same. This would appear to constitute a set of four equations in four unknowns X', Y', Z'_b, and Z'_t, however the solution is unstable because of the close similarity of Z'_t to Z'_b. Nevertheless, the object height $Z'_t - Z'_b$ can be obtained by the following procedure:

1. Get (r_b, c_b) and (r_t, c_t) from the image and convert to scaled values (r'_b, c'_b) and (r'_t, c'_t).

2. Solve the first two of Equations (5.56), for instance, with Newton's method (Press et al., 2002), for X' and Y' with Z'_b set equal to the average elevation in the scene, i.e., Z_0/Z_s, if no digital elevation model (DEM) is available, otherwise to the true, properly scaled elevation.

3. For a range of Z'_t values increasing from Z'_b to some maximum value well exceeding the expected height, calculate (r'_t, c'_t) from the last two of Equations (5.56). Choose for Z'_t the value which gives closest agreement to the (r_t, c_t) values read in.

Quite generally, the RFM can be used as an alternative for providing end users with the necessary information to perform their own photogrammetric processing.

5.5.4 Stereo imaging and digital elevation models

The missing elevation information Z in Equations (5.53) or (5.54) can be obtained with stereoscopic imaging techniques. Figure 5.22 depicts two cameras viewing the same world point w from two positions. The separation B of the lens centers is the *baseline*. The objective is to find the coordinates of the point w if its image points have coordinates (x_1, y_1) and (x_2, y_2). We assume that the cameras are identical and that their image coordinate systems are perfectly aligned, differing only in the location of their origins. The Z coordinate of w is the same for both coordinate systems.

In Figure 5.22, the coordinate system of the first camera is shown as coinciding with the world coordinate system. Therefore, from Equation (5.54),

$$X_1 = \frac{x_1}{\lambda}(\lambda - Z).$$

Alternatively, if the second camera is brought to the origin of the world coordinate system,

$$X_2 = \frac{x_2}{\lambda}(\lambda - Z).$$

But, from the figure,

$$X_2 = X_1 + B,$$

where B is the baseline. We have from the above three equations:

$$Z = \lambda - \frac{\lambda B}{x_2 - x_1}. \tag{5.57}$$

Thus if the displacement of the image coordinates of the point w, namely $x_2 - x_1$, can be determined, then the Z coordinate can be calculated. The task is then to find two corresponding points in different images of the same scene. This is usually accomplished by spatial cross-correlation techniques and is closely related to the problem of image–image registration discussed later in this chapter.

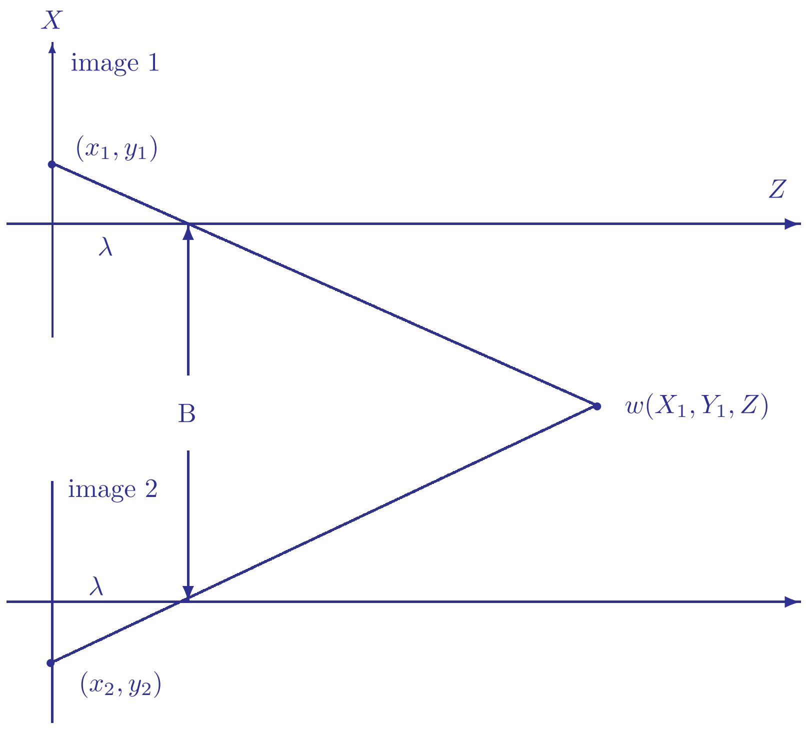


FIGURE 5.22
The stereo imaging process. The coordinates of the point w are relative to a world coordinate system coinciding with the upper camera.

Because the stereo images must be correlated, best results are obtained if they are acquired within a very short time of each other, preferably "along track" if a single platform is used; see Figure 5.23. This figure shows the orientation and imaging geometry of the VNIR 3N and 3B cameras on the ASTER platform for acquiring a stereo full scene. The satellite travels at a speed of 6.7 km/sec at a height of 705 km. A 60×60 km^2 full scene is scanned in 9 seconds. Then, 55 seconds later, the same scene is scanned by the back-looking camera, corresponding to a baseline of $B = 370$ km. The along-track geometry means that the *epipolar lines* (Solem, 2012) of the stereo pair are parallel, i.e., the displacements due to viewing angle are only along a common direction in the imaging planes of the two cameras, in this case the y axis. Therefore, the spatial cross-correlation algorithm used to match points can be one-dimensional. If carried out on a pixel-for-pixel basis, one obtains a DEM having approximately the same resolution as that of the stereo imagery.

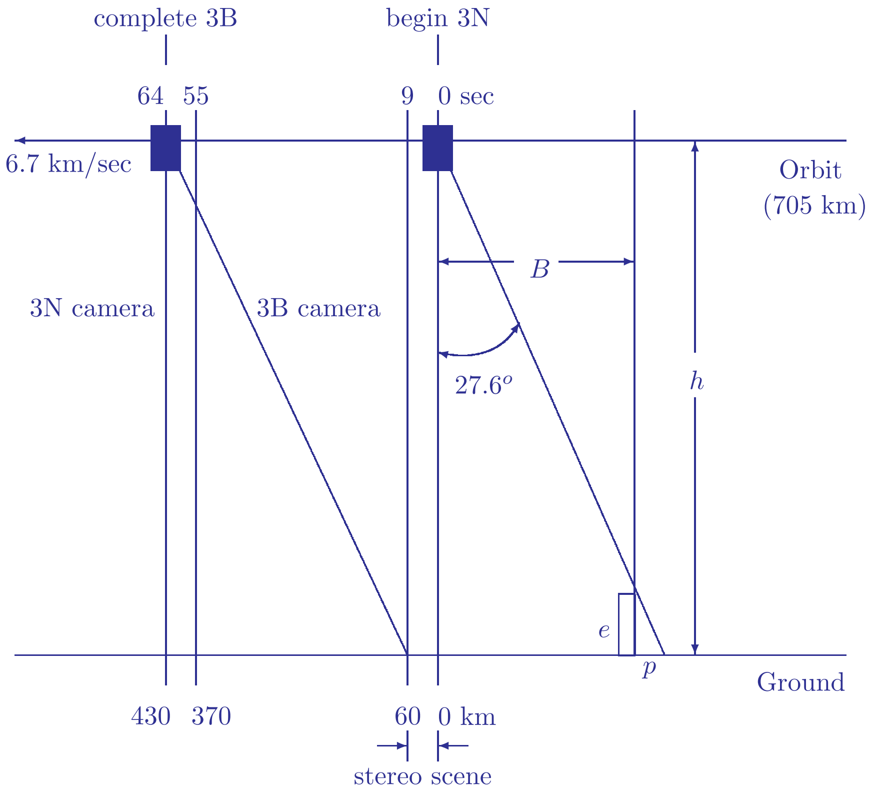


FIGURE 5.23
ASTER along-track stereo acquisition geometry, adapted from Lang and Welch (1999). Parallax p can be related to elevation e by similar triangles: $e/p = (h - e)/B \approx h/B$.

As an example, Figures 5.24 and 5.25 show an ASTER along-track stereo pair. The images are 600 × 600 pixels. A python script was used calculate a very rudimentary DEM: For each pixel in the nadir image a 15 × 15 window is moved along a 15 × 51 window (called the *epipolar segment*) in the back-looking image centered at the corresponding position, allowing for a maximum disparity of ±18 pixels. The point of maximum cross-correlation determines the parallax or disparity p. This is related to the elevation e of the pixel by

$$e = p \cdot \frac{h}{B} \times 15\text{m},$$

where h is the height of the sensor and B is the baseline; see Figure 5.23. The script is as follows:

```
from osgeo import gdal
from osgeo.gdalconst import GDT_Float32
```

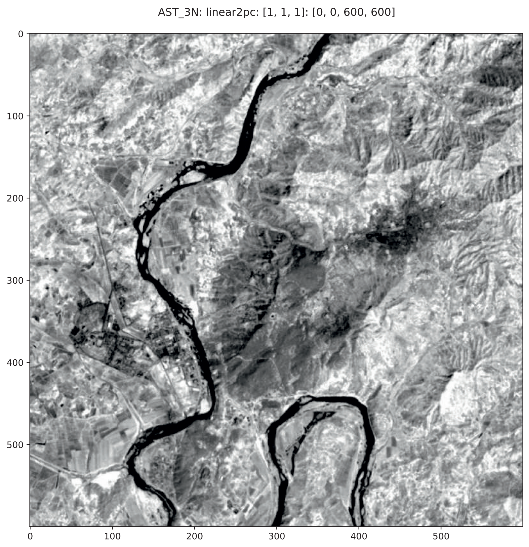


FIGURE 5.24
ASTER 3N band (nadir camera) over a hilly region in North Korea.

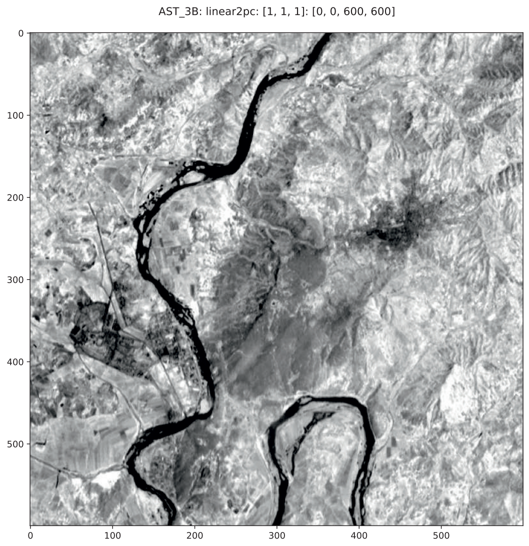


FIGURE 5.25
ASTER 3B band (back-looking camera) registered to Figure 5.24 by a first-order polynomial transformation (see Section 5.6.3).

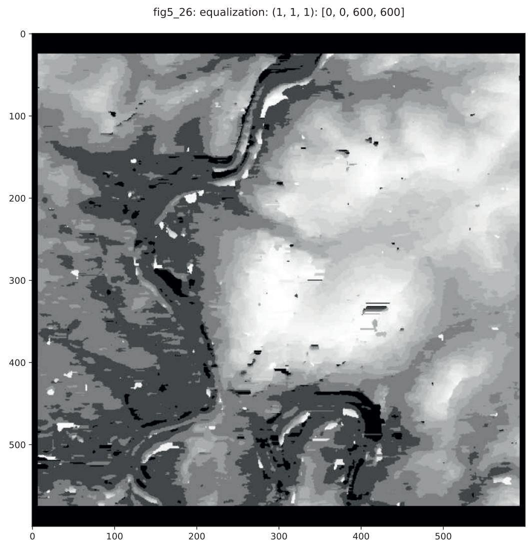


FIGURE 5.26
A rudimentary digital elevation model (DEM) from the ASTER stereo pair of Figures 5.24 and 5.25.

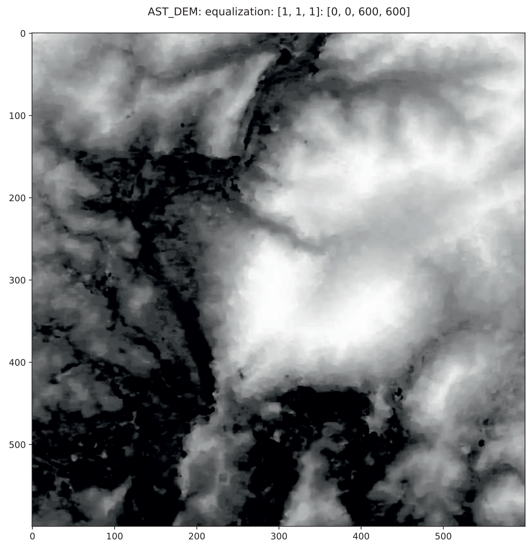


FIGURE 5.27
DEM generated with the commercial product AsterDTM (Sulsoft, 2003).

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.signal import correlate, correlation_lags
gdal.AllRegister()
# grab the ASTER nadir and back-looking images
inDataset1 = gdal.Open('imagery/AST_3N')
cols = inDataset1.RasterXSize
rows = inDataset1.RasterYSize
image1 = np.array(inDataset1.GetRasterBand(1) \
             .ReadAsArray(0,0,cols,rows), dtype=float)
inDataset2 = gdal.Open('imagery/AST_3B')
image2 = np.array(inDataset2.GetRasterBand(1) \
             .ReadAsArray(0,0,cols,rows), dtype=float)
# ASTER stereo geometry
height = 705.0  #km
base = 370.0    #km
res = 15.0      #m
# the parallax DEM
p = np.zeros([cols, rows], dtype = float) - 20
for i in range(25, rows-25):
    for j in range(7, cols-7):
        im1 = image1[i-7:i+7, j-7:j+7]
        im2 = image2[i-25:i+25, j-7:j+7]
        # center the image patches
        im1 = (im1 - np.mean(im1))
        im2 = (im2 - np.mean(im2))
        # 2-dimensional cross-correlation
        corr = correlate(im1, im2, mode='valid')
        # lag of maximum correlation index along y-axis
        lags = correlation_lags(im1.shape[0],
                      im2.shape[0], mode='valid')
        max_index = np.unravel_index(np.argmax(corr),
                      corr.shape)
        yoff =  lags[max_index[0]]
        if (yoff < -20) or (yoff > -5):
            p[i,j] = p[i,j-1]
        else:
            p[i,j] = yoff
dem =  p*res*(height/base)
```

Figure 5.26 shows the result. Clearly there are problems due to correlation errors; however, the relative elevations are approximately correct when compared to the DEM determined with the ENVI commercial add-on `AsterDTM` (Sulsoft, 2003); see Figure 5.27. This more sophisticated approach uses image pyramids to accumulate disparities at increasing scales (Quam, 1987).

In generating a DEM in this way, either the complete camera model or an RFM can be referred to, but usually neither is sufficient for determining absolute elevations. Most often, additional ground reference points within the image with known elevations are also required for absolute calibration. Ortho rectification of the image is carried out on the basis of the DEM and consists of referring the (X, Y, Z) coordinates of each pixel to the (X, Y) coordinates of a given map projection.

5.5.5 Slope and aspect

Topographic modeling, or terrain analysis, involves the processing of elevation data provided by a DEM. Specifically, we consider here the generation of *slope images*, which give the steepness of the terrain at each pixel, and *aspect images*, which give the direction relative to north of a vector normal to the landscape at each pixel.

A 3×3 pixel window can be used to determine both slope and aspect as follows; see Figure 5.28. Define

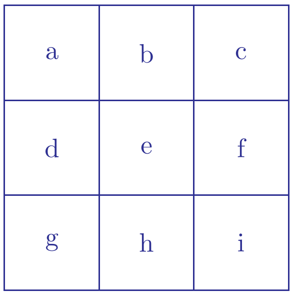


FIGURE 5.28
Pixel elevations in an 8-neighborhood. The letters represent elevations in meters.

$$\begin{aligned} \Delta x_1 &= c - a \quad & \Delta y_1 &= a - g \\ \Delta x_2 &= f - d \quad & \Delta y_2 &= b - h \\ \Delta x_3 &= i - g \quad & \Delta y_3 &= c - i \end{aligned}$$

and

$$\begin{aligned} \Delta x &= (\Delta x_1 + \Delta x_2 + \Delta x_3)/3 \\ \Delta y &= (\Delta y_1 + \Delta y_2 + \Delta y_3)/3. \end{aligned}$$

Then the slope angle in radians at the central pixel position is given by (Exercise 11)

$$\theta_p = \tan^{-1}\left(\frac{\sqrt{(\Delta x)^2 + (\Delta y)^2}}{2 \cdot \mathrm{GSD}}\right), \tag{5.58}$$

whereas the aspect in radians measured clockwise from north is

$$\phi_o = \tan^{-1}\left(\frac{\Delta x}{\Delta y}\right). \tag{5.59}$$

The GDAL utilities (Appendix C) include routines for slope/aspect determination from a DEM:

```
!gdaldem slope imagery/AST_DEM imagery/ASTslope.tif
0...10...20...30...40... ...90...100 - done.

!gdaldem aspect imagery/AST_DEM imagery/ASTaspect.tif
0...10...20...30...40... ...90...100 - done.

run scripts/dispms -f imagery/ASTslope.tif \
  -F imagery/ASTaspect.tif
```

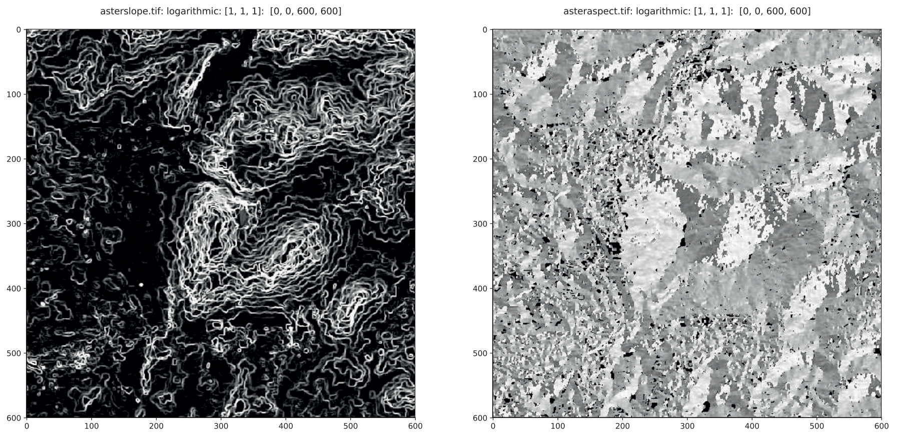

asterslope.tif: logarithmic: [1, 1, 1]: [0, 0, 600, 600]
asteraspect.tif: logarithmic: [1, 1, 1]: [0, 0, 600, 600]
0
100
200
300
400
500
600

FIGURE 5.29
Slope (left) and aspect (right) images calculated with the DEM of Figure 5.27.

The slope and aspect images are shown in Figure 5.29.

5.5.6 Illumination correction

Topographic modeling can be used to correct images for the effects of local solar illumination. The local illumination depends not only upon the sun's position (elevation and azimuth) but also upon the slope and aspect of the terrain being illuminated. Figure 5.30 shows the angles involved. The quantity to be determined is the local solar incidence angle γ_i, which determines the local irradiance. From trigonometry, we can calculate the relation

$$\cos\gamma_i = \cos\theta_p \cos\theta_z + \sin\theta_p \sin\theta_z \cos(\phi_a - \phi_o). \tag{5.60}$$

Image classification and change detection algorithms, the subject of the next chapters, will achieve better results if variable image properties extrinsic to the actual surface reflectance are first removed. For a Lambertian surface, the reflected radiance L_H from a horizontal surface toward a sensor (ignoring all atmospheric effects) is given by Equation (1.1), which we write in the simplified form

$$L_H = E \cdot \cos\theta_z \cdot R. \tag{5.61}$$

For a surface in rough terrain, the reflected radiance L_T is similarly

$$L_T = E \cdot \cos\gamma_i \cdot R, \tag{5.62}$$

thus giving the standard *cosine correction* relating the observed radiance L_T to that which would have been observed had the terrain been flat, namely

$$L_H = L_T \frac{\cos\theta_z}{\cos\gamma_i}. \tag{5.63}$$

The Lambertian assumption is in general a poor approximation, the actual reflectance being governed by a *bidirectional reflectance distribution function* (BRDF), which describes the dependence of reflectance on both illumination and viewing angles as well as on wavelength (Beisl, 2001). Particularly for the large range of incident angles involved with rough terrain, the cosine correction will over- or underestimate the extremes and lead to unwanted artifacts in the corrected imagery.

An example of an approach which takes better account of BRDF effects is the semi empirical cosine correction (C-correction) method suggested by Teillet et al. (1982). We replace Equations (5.61) and (5.62) by

$$L_H = m \cdot \cos\theta_z + b, \quad L_T = m \cdot \cos\gamma_i + b.$$

The parameters m and b can be estimated from a linear regression of observed radiance L_T vs. $\cos\gamma_i$ for a particular image band. The regression should be carried out separately for different land cover categories in order to take into account the variation of BRDF effects with land cover. Then, instead of Equation (5.63), one uses

$$L_H = L_T \left(\frac{\cos\theta_z + b/m}{\cos\gamma_i + b/m} \right) \tag{5.64}$$

as a correction formula.

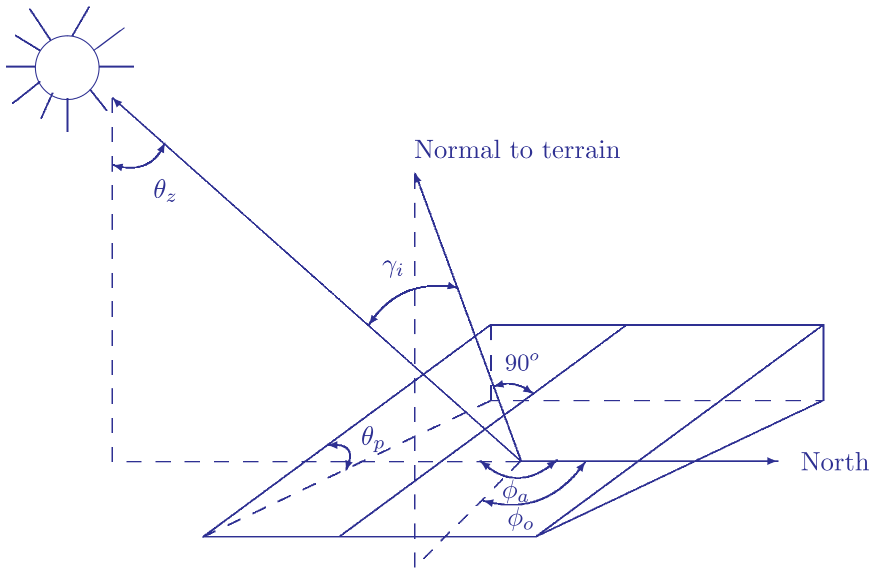


FIGURE 5.30
Angles involved in computation of local solar incidence (adapted from Riano et al. (2003)): θ_z = solar zenith angle, ϕ_a = solar azimuth, θ_p = slope, ϕ_o = aspect, γ_i = local solar incidence angle.

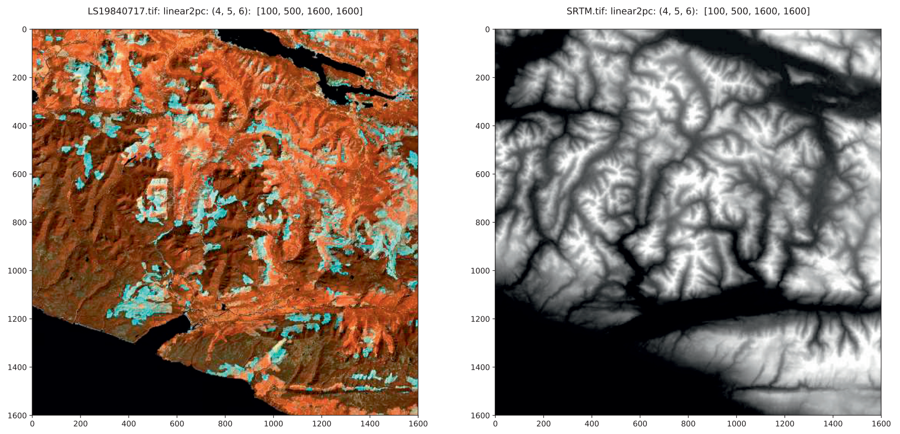

LS19840717.tif: linear2pc: (4, 5, 6): [100, 500, 1600, 1600]
SRTM.tif: linear2pc: (4, 5, 6): [100, 500, 1600, 1600]
0
200
400
600
800
1000
1200
1400
1600

FIGURE 5.31
Left: RGB composite of bands 4, 5, and 7 of a LANDSAT 5 TM image over Vancouver Island acquired 17 July, 1984. Right: STRM (Shuttle Radar Topography Mission) digital elevation data for the same region, 30 m resolution, acquired February 2000.

A Python script `c_corr.py` for illumination correction with the C-correction approximation, including land cover masking, is given in Appendix C. As an example of the procedure, consider the LANDSAT 5 TM image along with the associated DEM shown in Figure 5.31. The data were accessed from the GEE database with the commands

```
rect = ee.Geometry \
   .Rectangle([-124.705,48.414,-123.799,49.026])
image = ee.Image(
      'LANDSAT/LT05/C01/T1/LT05_048026_19840717') \
          .select('B1','B2','B3','B4','B5','B7') \
          .clip(rect)

dem = ee.Image('USGS/SRTMGL1_003').clip(rect)
```

and downloaded to the `imagery` directory for further processing in the Jupyter notebook.

The LANDSAT image was subjected to a principal components transformation and then classified with the *expectation maximization* (EM) clustering algorithm coded in the script `em.py`:[*]

[*]We have to anticipate here. See Chapter 8 for a discussion of unsupervised classification in general and the EM algorithm in particular.

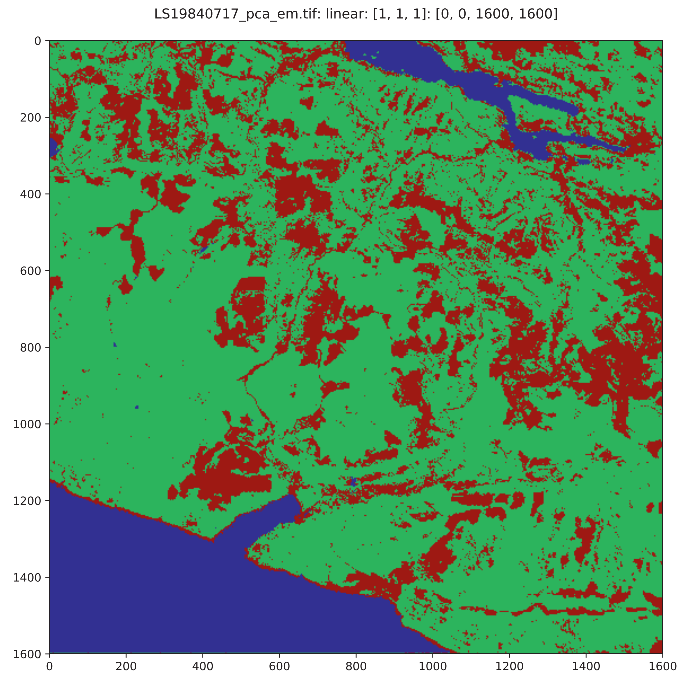

LS19840717_pca_em.tif: linear: [1, 1, 1]: [0, 0, 1600, 1600]
0
200
400
600
800
1000
1200
1400
1600

FIGURE 5.32
Unsupervised classification of the image of Figure 5.31. The three classes correspond to water (blue), forest canopy (green), and cut forest (brown).

```
# perform PCA
run scripts/pca -d [100,500,1600,1600] \
                   imagery/LS19840717.tif
PCs written to: myimagery/LS19840717_pca.tif

# classify on first 3 PCs only, assume 3 clusters
run scripts/em -p [1,2,3] -K 3 \
                   imagery/LS19840717_pca.tif
classified image: myimagery/LS19840717_pca_em.tif
```

The clustered image with three surface categories is shown in Figure 5.32. Finally, the C-correction code was invoked on the original LANDSAT scene and associated DEM, with the `-c` flag pointing to the class image and with the solar azimuth and elevation angles, 132.9 and 54.9 degrees, respectively, taken from the scene's metadata:

```
run scripts/c_corr -d [100,500,1600,1600] \
  -c imagery/LS19840717_pca_em.tif \
  132.9 54.9 imagery/LS19840717.tif imagery/SRTM.tif
```

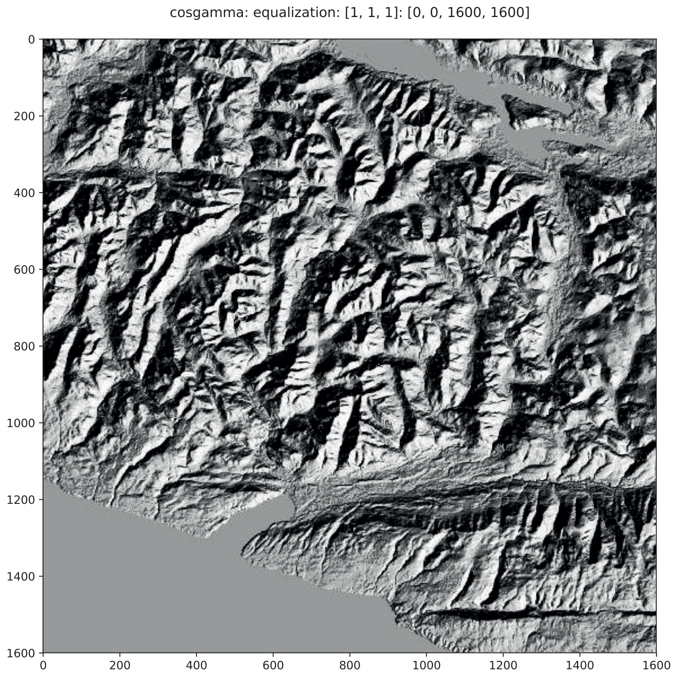


FIGURE 5.33
Cosine of the local solar incidence angle $\cos \gamma_i$ for the image of Figure 5.31.

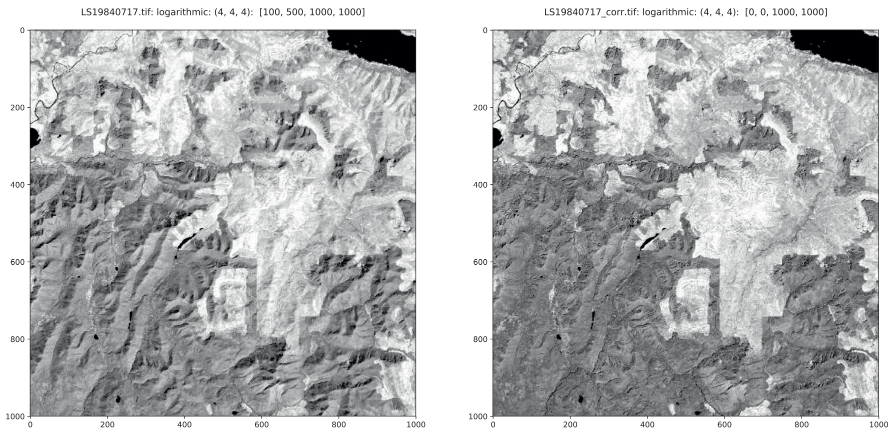

LS19840717.tif: logarithmic: (4, 4, 4): [100, 500, 1000, 1000]
LS19840717_corr.tif: logarithmic: (4, 4, 4): [0, 0, 1000, 1000]
0
200
400
600
800
1000

FIGURE 5.34
Left: spatial subset of band 4 for the image of Figure 5.31 in a logarithmic stretch. Right: with C-correction.

```
Band: 1 Class: 1 Pixels: 317119 Correlation: 0.035653
Band: 1 Class: 2 Pixels: 1637855 Correlation: 0.400593
---correcting band 1, class 2
Band: 1 Class: 3 Pixels: 605026 Correlation: 0.150770
Band: 2 Class: 1 Pixels: 317119 Correlation: 0.016266
Band: 2 Class: 2 Pixels: 1637855 Correlation: 0.490472
---correcting band 2, class 2
Band: 2 Class: 3 Pixels: 605026 Correlation: 0.259226
---correcting band 2, class 3
...
...
Band: 6 Class: 1 Pixels: 317119 Correlation: -0.013994
Band: 6 Class: 2 Pixels: 1637855 Correlation: 0.367915
---correcting band 6, class 2
Band: 6 Class: 3 Pixels: 605026 Correlation: 0.187557

c-corrected image written to: \
                      imagery/LS19840717_corr.tif
```

As can be inferred from the above output, for each image band only those pixels in a class with a sufficiently high correlation (> 0.2) between intensity and $\cos\gamma_i$ are corrected. The others are left as is. The $\cos\gamma_i$ image is shown in Figure 5.33. The original and C-corrected images are compared in Figure 5.34. Quite generally, hillsides sloped away from the sun are brighter after correction.

5.6 Image–image registration

Image registration, either to another image or to a map, is a fundamental task in remote sensing data processing. It is required for georeferencing, stereo imaging, accurate change detection, and indeed for any kind of multi temporal image analysis. A tedious task associated with manual co registration in the past has been the setting of tie-points or, as they are often called, ground control points (GCPs), since in general it was necessary to resort to manual entry. Fortunately, there now exist many reliable automatic or semi-automatic procedures for locating tie-points, Tondewad and Dale (2020) give a good overview. In general, registration procedures can be divided roughly into four classes (Reddy and Chatterji, 1996):

1. Algorithms that use pixel intensity values directly, such as correlation methods or methods that maximize mutual information
2. Frequency- or wavelet-domain methods that use, e.g., the fast Fourier transform

3. Feature-based methods that use low-level features such as shapes, edges, or corners to derive tie-points
4. Algorithms that use high-level features and the relations between them (object-based methods)

We will consider here examples of frequency-domain and feature-based algorithms which illustrate some of the principles involved.

5.6.1 Frequency domain registration

Consider two $c \times c$ gray-scale images $g_1(i,j)$ and $g_2(i,j)$, where g_2 is offset relative to g_1 by an integer number of pixels:

$$g_2(i,j) = g_1(i',j') = g_1(i-i_0, j-j_0).$$

Taking the Fourier transform we have,

$$\hat{g}_2(k,\ell) = \frac{1}{c^2}\sum_{ij} g_1(i-i_0, j-j_0)e^{-\mathbf{i}2\pi(ik+j\ell)/c},$$

or with a change of indices to $i'j'$,

$$\begin{aligned}\hat{g}_2(k,\ell) &= \frac{1}{c^2}\sum_{i'j'} g_1(i',j')e^{-\mathbf{i}2\pi(i'k+j'\ell)/c}e^{-\mathbf{i}2\pi(i_0k+j_0\ell)/c}\\ &= \hat{g}_1(k,\ell)e^{-\mathbf{i}2\pi(i_0k+j_0\ell)/c}.\end{aligned}$$

This is the Fourier translation property that we met in Chapter 3; see Equation (3.11). Therefore, we can write

$$\hat{g}_2(k,\ell)\hat{g}_1^*(k,\ell) = |\hat{g}_1(k,\ell))|^2 e^{-\mathbf{i}2\pi(i_0k+j_0\ell)/c},$$

where $\hat{g}_1^*$ is the complex conjugate of $\hat{g}_1$, and hence

$$\frac{\hat{g}_2(k,\ell)\hat{g}_1^*(k,\ell)}{|\hat{g}_1(k,\ell)|^2} = e^{-\mathbf{i}2\pi(i_0k+j_0\ell)/c}. \tag{5.65}$$

The inverse transform of the right-hand side of Equation (5.65) exhibits a delta function (spike) at the coordinates (i_0, j_0). Thus if two otherwise identical (or closely similar) images are offset by an integer number of pixels, the offset can be found by taking their Fourier transforms, computing the ratio on the left-hand side of Equation (5.65) (the so-called *cross-power spectrum*), and then taking the inverse transform of the result. The position of the maximum value in the inverse transform gives the offset values of i_0 and j_0. Here is an illustration:

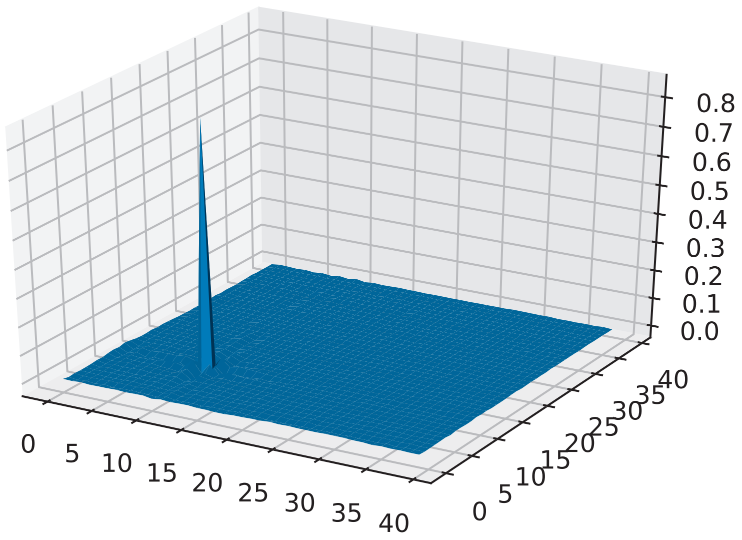


FIGURE 5.35
Phase correlation of two identical images shifted relative to one another by 10 pixels.

```
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly
import numpy as np
from numpy import fft
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# grab an image band
gdal.AllRegister()
inDataset = gdal.Open('imagery/AST_20070501')
cols = inDataset.RasterXSize
rows = inDataset.RasterYSize
band = inDataset.GetRasterBand(3) \
             .ReadAsArray(0,0,cols,rows)

# calculate and invert cross-power spectrum
g1 = band[10:522,10:522]
g2 = band[0:512,0:512]
f1 = fft.fft2(g1)
f2 = fft.fft2(g2)
g = fft.ifft2(f2*np.conj(f1)/np.absolute(f1)**2)
```

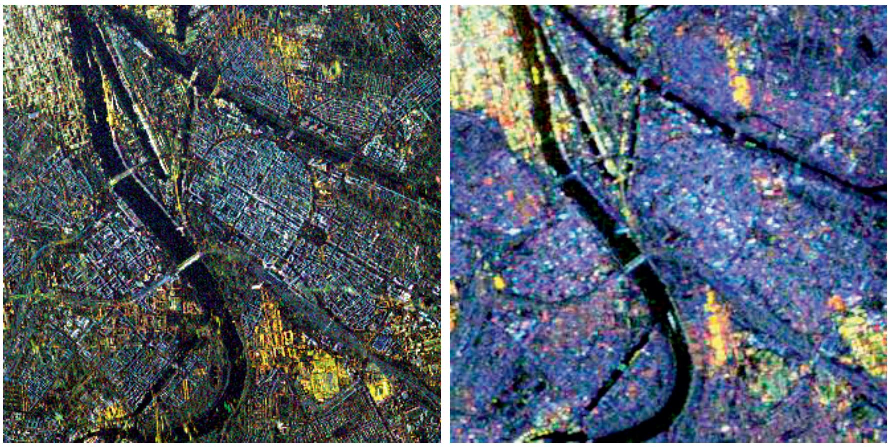


FIGURE 5.36
Frequency domain registration of quad polarimetric SAR imagery, RGB composites $(|s_{hh}|^2, |s_{hv}|^2, |s_{vv}|^2)$, logarithmic intensity values in a linear 2% stretch. Left: TerraSAR-X, right: RADARSAT-2.

```
# plot
fig = plt.figure()
ax = fig.gca(projection='3d')
x, y = np.meshgrid(range(40),range(40))
ax.plot_surface(x, y, np.real(g[0:40,0:40]))
```

The result is shown in Figure 5.35. Subpixel registration is also possible with a refinement of the above method (Shekarforoush et al., 1995).

Images which differ not only by an offset but also by a rigid rotation and/or change of scale can be registered similarly (Reddy and Chatterji, 1996). Both ENVI/IDL (Xie et al., 2003) and Python scripts are available which calculate RST or similarity transformations in the frequency domain. The Python function `similarity()` included in the `auxil.auxil1.py` module estimates the similarity transformation parameters between two gray-scale images in the frequency domain. It is a slight modification of code provided by C. Gohlke.[*] Two Python modules, `registerms.py` and `registersar.py` in the `auxil` package for registration of optical/infrared and polarimetric SAR images, respectively, make use of `similarity()` and are described in Appendix C. An example is shown in Figure 5.36 in which a RADARSAT-2 quad polarimetric image is registered to a TerraSAR-X quad polarimetric image. The latter was first re-sampled to the 15 m ground resolution of the RADARSAT-2 image.

[*]Laboratory for Fluorescence Dynamics.

5.6.2 Feature matching

Various techniques for automatic determination of tie-points based on low-level features have been suggested in the literature. Especially for scale and rotation invariant image matching of different views the Scale-Invariant Feature Transform (SIFT) (Lowe, 2004) is one of the best known methods. Here we outline a contour matching procedure proposed by Li et al. (1995). It functions especially well in bi-temporal remote sensing imagery in which vegetation changes do not dominate, and can of course be augmented by other automatic feature matching methods or by manual selection. The required steps are shown in Figure 5.37 and are described below.

The first step involves the application of a Laplacian-of-Gaussian filter to both images in the manner discussed in Section 5.2.2. After determining the contours by examining zero-crossings of the LoG-filtered image, the contour strengths are encoded in the pixel intensities. Strengths are taken to be proportional to the magnitude of the gradient at the zero-crossing determined by a Sobel filter as illustrated in Figure 5.2.

In the next step, all closed contours with strengths above some given threshold are determined by tracing the contours. Pixels which have been visited during tracing are set to zero so that they will not be visited again. A typical result is shown in Figure 5.38.

For subsequent matching purposes, all significant closed contours found in the preceding step are *chain encoded*. Any curve or contour can be represented by an integer sequence $\{a_1, a_2 \dots a_i \dots\}$, $a_i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$, depending on the relative position of the current pixel with respect to the previous pixel in the curve. A shift to the east is coded as 0, to the north-east as 1 and so on. This simple code has the drawback that some contours produce wraparound. For example, the line in the direction -22.5^o has the chain code $\{707070\dots\}$. Li et al. (1995) suggest the smoothing operation:

$$\{a_1a_2 \dots a_n\} \rightarrow \{b_1b_2 \dots b_n\},$$

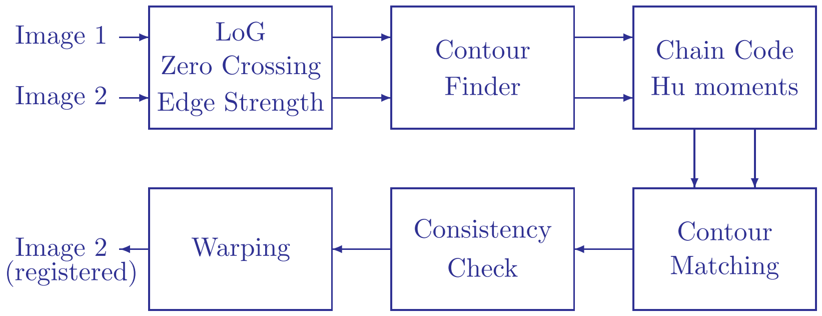


FIGURE 5.37
Image–image registration with contour matching.

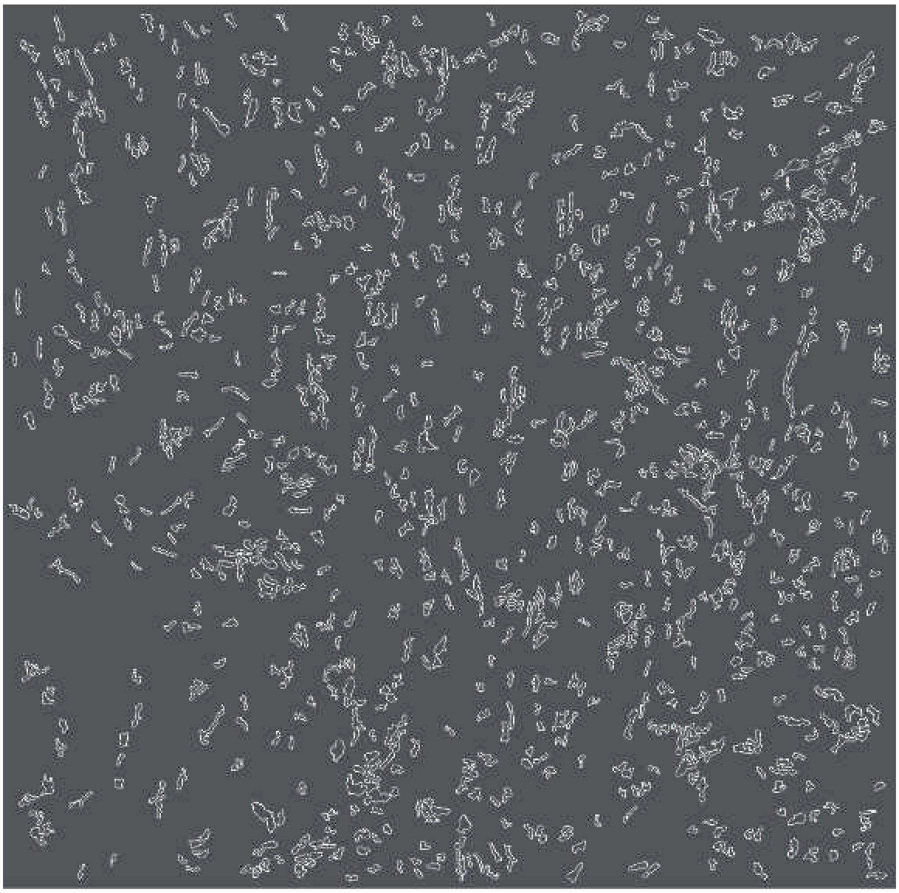
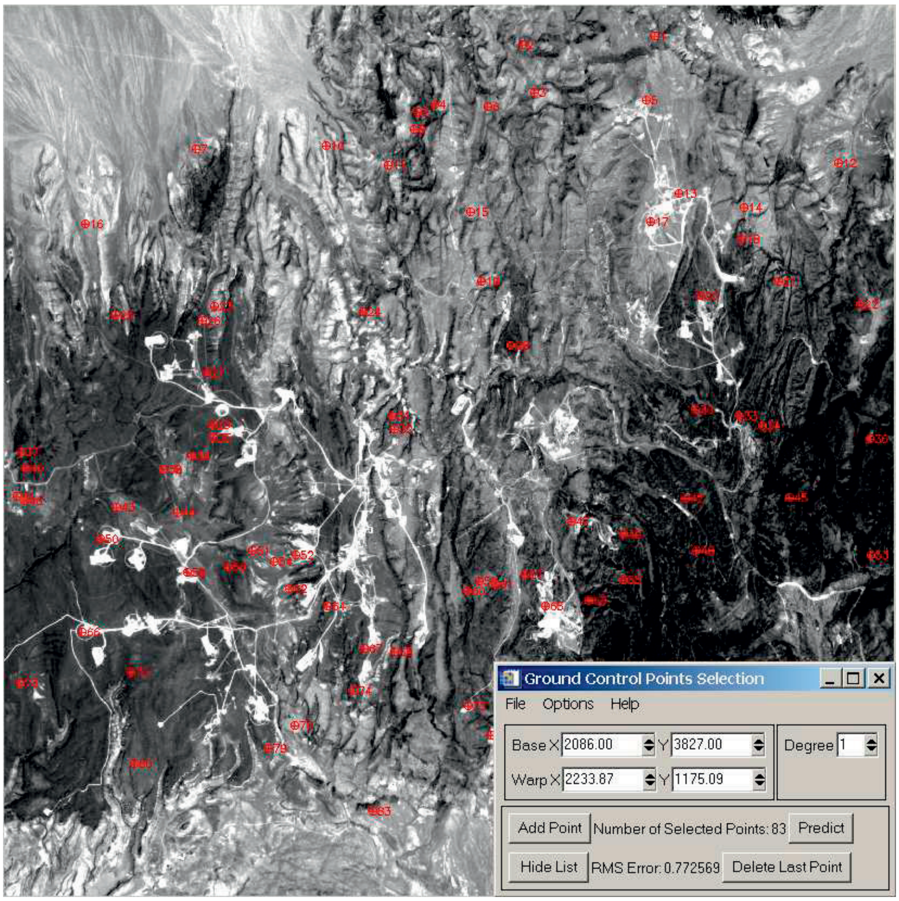


FIGURE 5.38
Closed contours derived from the 3N band of an ASTER image over Nevada acquired in July 2003.

where $b_1 = a_1$ and $b_i = q_i$. The integer q_i satisfies $(q_i - a_i) \bmod 8 = 0$ and $|q_i - b_{i-1}| \to \min$, $i = 2, 3 \ldots n$.* They also suggest applying the smoothing filter $\{0.1, 0.2, 0.4, 0.2, 0.1\}$ to the result. After both processing steps, two chain codes can be easily compared by "sliding" one over the other and determining their maximum correlation. The closed contours are further characterized by determining their first four Hu moments $h_1 \ldots h_4$, Equations (5.14).

*This is rather cryptic, so here is an example: For the wraparound sequence $\{707070\ldots\}$, we have $a_1 = b_1 = 7$ and $a_2 = 0$. Therefore, we must choose $q_2 = 8$, since this satisfies $(q_2 - a_2) \bmod 8 = 0$ and $|q_2 - b_1| = 1$. (For the alternatives $q_2 = 0, 16, 24\ldots$ the difference $|q_2 - b_1|$ is larger.) Continuing the same argument leads to the new sequence $\{787878\ldots\}$ with no wraparound.



Base X 2086.00 Y 3827.00
Warp X 2233.87 Y 1175.09
Degree 1
Number of Selected Points: 83
RMS Error: 0.772569

FIGURE 5.39
85 tie-points obtained by matching the contours of Figure 5.38 with those obtained from a similar image acquired in June 2001. The RMS error is 0.77 pixel for first-order polynomial warping; see text.

Each significant contour in one image is first matched with contours in the second image according to their invariant moments. This is done by setting a threshold on the allowed differences, for instance, one standard deviation. If one or more matches are found, the best candidate for a tie-point is then chosen to be that matched contour in the second image for which the chain code correlation with the contour in the first image is maximum. If the maximum correlation is less than some threshold, e.g., 0.9, then the match is rejected. The tie-point coordinates are taken to be the centers of gravity $(\bar{x}_1, \bar{x}_2)$ of the matched contour pairs; see Equations (5.11).

The contour matching procedure invariably generates some false tie-points, so a further processing step is required. In Li et al. (1995), use is made of the fact that distances are preserved under a rigid transformation. Let $\overline{A_1A_2}$ represent the distance between two points A_1 and A_2 in an image. For two sets of m matched contour centers $\{A_i \mid i = 1\ldots m\}$ and $\{B_i \mid i = 1\ldots m\}$ in image 1 and 2, the ratios

$$\overline{A_iA_j}/\overline{B_iB_j}, \quad i = 1\ldots m,\ j = i+1\ldots m,$$

are calculated. These should form a cluster, so that indices associated with ratios scattered away from the cluster center can be rejected as false matches.

An example is shown in Figure 5.39, determined with an ENVI/IDL extension written by the author and available on GitHub; see Canty (2014).

5.6.3 Re-sampling with ground control points

Having determined a valid set of tie-points, transformation parameters which map the target image to the base image may be estimated.

If uniform scaling s, rotation θ and shift (x_0, y_0) of the target image are sufficient for registering it to the base image, then the two-dimensional equivalent of the RST transformation discussed in Section 5.5.1 can be applied:

$$\boldsymbol{A} = \begin{pmatrix} s\cos\theta & -s\sin\theta & x_0 \\ s\sin\theta & s\cos\theta & y_0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a & -b & x_0 \\ b & a & y_0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{5.66}$$

where $a^2 + b^2 = s^2(\cos^2\theta + \sin^2\theta) = s^2$. The base image points (u, v) and target image points (x, y) are related by

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} a & -b & x_0 \\ b & a & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Equivalently,

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} a & -b \\ b & a \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix},$$

which can easily be rewritten as

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x & -y & 1 & 0 \\ y & x & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ x_0 \\ y_0 \end{pmatrix}.$$

Listing 5.4: Image–image registration with the similarity transform.

```
#!/usr/bin/env python3
#Name:  ex5_3.py
import  sys
import numpy as np
def parse_gcp(gcpfile):
    with open(gcpfile) as f:
        pts = []
        for i in range(6):
            line =f.readline()
        while line:
            pts.append(list(map(eval,line.split())))
            line = f.readline()
        f.close()
    pts = np.array(pts)
    return (pts[:,:2],pts[:,2:])
def main():
    infile = sys.argv[1]  # gcps
    if infile:
        pts1,pts2 = parse_gcp(infile)
    else:
        return
    n = len(pts1)
    y = pts1.ravel()
    A = np.zeros((2*n,4))
    for i in range(n):
        A[2*i,:] =   [pts2[i,0],-pts2[i,1],1,0]
        A[2*i+1,:] = [pts2[i,1], pts2[i,0],0,1]
    result = np.linalg.lstsq(A,y,rcond=-1)
    a,b,x0,y0 = result[0]
    RMS = np.sqrt(result[1]/n)[0]
    print ('RST␣transformation:')
    print (np.array([[a,-b,x0],[b,a,y0],[0,0,1]]) )
    print ('RMS:␣%f'%RMS)

if __name__ == '__main__':
    main()
```

Thus, for n tie-points, we obtain a multiple linear regression problem of the form given in Section 2.6.3, Equation (2.97), namely,

$$\begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ \vdots \\ u_n \\ v_n \end{pmatrix} = \begin{pmatrix} x_1 & -y_1 & 1 & 0 \\ y_1 & x_1 & 0 & 1 \\ x_2 & -y_2 & 1 & 0 \\ y_2 & x_2 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_n & -y_n & 1 & 0 \\ y_n & x_n & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ x_0 \\ y_0 \end{pmatrix},$$

from which the similarity transformation relating the target to the base image may be obtained.

To illustrate the procedure, the Python script shown in Listing 5.4 reads a tie-point file in ENVI format, and outputs the RST transformation matrix, Equation (5.66). For example, using a tie-point file for two LANDSAT 5 TM images over the Nevada Nuclear Test Site; see Figure 5.39:

```
run scripts/ex5_3 imagery/gcps.pts
RST transformation:
[[  9.99972110e-01  -2.02991329e-03   2.10315125e+02]
 [  2.02991329e-03   9.99972110e-01  -1.84336787e+02]
 [  0.00000000e+00   0.00000000e+00   1.00000000e+00]]
RMS: 0.504769
```

Apart from the shifts $x0$ and $y0$, the transformation is very close to the identity matrix, since the two images were already geo-referenced. The RMS (square root of the average of the squared residuals) is 0.5 pixels.

If the similarity transformation is not sufficient, then a polynomial map may be used: For instance, a second-order polynomial transformation of the target to the base image is given by

$$
\begin{aligned}
u &= a_0 + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2 \\
v &= b_0 + b_1 x + b_2 y + b_3 xy + b_4 x^2 + b_5 y^2.
\end{aligned}
$$

Since there are 12 unknown coefficients, at least six tie-point pairs are needed to determine the map (each pair generates two equations). If more than six pairs are available, the coefficients can again be found by least squares fitting. Similar considerations apply for lower- or higher-order polynomial maps.

Having determined the transformation coefficients, the target image can be registered to the base by re-sampling. *Nearest-neighbor re-sampling* simply chooses the pixel in the target image that has its transformed center nearest coordinates (i, j) in the warped image and transfers it to that location. This is often the preferred technique for classification or change detection, since the registered image consists of the original pixel intensities, simply rearranged in position to give a correct image geometry. Other commonly used re-sampling methods are *bilinear interpolation* and *cubic convolution interpolation*, see, e.g., Jensen (2005) for a good explanation. These methods interpolate, and therefore mix, the spectral intensities of neighboring pixels.

5.7 Exercises

1. Design a lookup table for byte-encoded data to perform 2% linear saturation (2% of the dark and bright pixels saturate to 0 and 255, respectively).

2. The *de-correlation stretch* generates a more color-intensive RGB composite image of highly correlated spectral bands than is obtained by simple linear stretching of the individual bands (Richards, 2012). Write a routine to implement it:

   (a) Do a principal components transformation of three selected image bands.

   (b) Then do a linear histogram stretch of the principal components.

   (c) Finally, invert the transformation and store the result to disk.

3. The *Roberts operator* or *Roberts filter* approximates intensity gradients in the diagonal directions:

$$\begin{aligned}\nabla_1(i,j) &= [g(i,j) - g(i+1,j+1)]\\ \nabla_2(i,j) &= [g(i+1,j) - g(i,j+1)]\end{aligned}$$

   Modify the Sobel filter script in Section 5.2.1 to calculate its power spectrum.

4. An edge detector due to Smith and Brady (1997) called SUSAN (*Smallest Univalue Segment Assimilating Nucleus*) employs a circular mask, typically approximated by 37 pixels, i.e.,

```
  000
 00000
0000000
0000000
0000000
 00000
  000
```

   Let r be any pixel under the mask, $g(r)$ its intensity and let r_0 be the central pixel. Define the function

$$c(r, r_0) = \begin{cases} 1 & \text{if } |g(r) - g(r_0)| \le t \\ 0 & \text{if } |g(r) - g(r_0)| > t, \end{cases}$$

where t is a threshold. Associate with r_0 the sum

$$n(r_0) = \sum_r c(r, r_0).$$

If the mask covers a region of sufficiently low contrast, $n(r_0) = n_{max} = 37$. As the mask moves toward an intensity "edge" having any orientation in an image, the quantity $n(r_0)$ will decrease, reaching a minimum as the center crosses the edge. Accordingly, an edge strength can be defined as

$$E(r_0) = \begin{cases} g(r_0) - n(r_0) & \text{if } n(r_0) < h \\ 0 & \text{otherwise.} \end{cases}$$

The parameter h is chosen (from experience) as $0.75 * n_{max}$.

(a) A convenient way to calculate $c(r, r_0)$ is to use the continuous approximation

$$c(r, r_0) = e^{-(g(r)-g(r_0))/t)^6}. \tag{5.67}$$

Write a Python script to plot this function for $g(r_0) = 127$ and for $g(r) = 0 \dots 255$.

(b) Write a Python program to implement the SUSAN edge detector for arbitrary gray-scale images. *Hint:* Create a lookup table to evaluate the expression (5.67):

```
LUT  = np.array(256
for i in range(256):
    LUT[i] = np.exp(-(i/t)**6)
```

5. One can approximate the centralized moments of a feature, Equation (5.12), by the integral

$$\mu_{pq} = \int\int (x - x_x)^p (y - y_c)^q f(x, y) dx dy,$$

where the integration is over the whole image and where $f(x, y) = 1$ if the point (x, y) lies on the feature and $f(x, y) = 0$ otherwise. Use this approximation to prove that the normalized centralized moments η_{pq} given in Equation (5.13) are invariant under scaling transformations of the form

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

6. Wavelet noise reduction (Gonzalez and Woods, 2017).

(a) Apply the discrete wavelet transformation to reduce the noise in a multispectral image by making use of the `DWTArray()` object class; see Listing 4.1, to perform the following steps:

- Select a multispectral image and determine the number of columns, rows, and spectral bands.

- Create a band sequential (BSQ) array of the same dimensions for output.
- For each band do the following:
  - Read the band into a new `DWTarray` object instance.
  - Filter once.
  - For each of the three quadrants containing the detail wavelet coefficients:
    - extract the coefficients with the method `get_quadrant()`,
    - determine their mean and standard deviation,
    - zero all coefficients with absolute value relative to the mean smaller than three standard deviations,
    - inject them back into the transformed image with the method `put_quadrant()`.
  - Expand back.
  - Store the modified band in the output array.
  - Destroy the object instance.
- Save the resulting array to disk.

Note: The coefficients are extracted as one-dimensional arrays. When injecting them back, they must be reformed to two-dimensional arrays. Use `<instance>.samples` and `<instance>.lines` to access these values.

(b) Test your program with a noisy 3-band image, for example, the last three components of the MNF transformation of a LANDSAT 7 TM+ image. Use the example program in Listing 3.4 to determine the noise covariance matrix before and after carrying through the above procedure.

7. Show that the means and standard deviations of the re-normalized panchromatic wavelet coefficients C_k^z in Equation (5.17) are equal to those of the multispectral bands.

8. Write a Python script to perform additive *à trous* fusion (see Núnez et al. (1999)).

9. Show, with the help of Theorem 2.1, that

   (a) if the random variable U has density $(1/2)e^{-u/2}$, then $G = xU/2$ has density $e^{-g/x}/x$, and

   (b) if the random variable G has density

$$\frac{1}{(x/m)^m\Gamma(m)} g^{m-1} e^{-gm/x}$$

and if $G = xV$, then V has density

$$\frac{m^m}{\Gamma(m)} v^{m-1} e^{-vm}.$$

10. Anfinsen et al. (2009b) suggest the following estimator, among others, for the ENL of a multilook polarimetric SAR image which takes into account the full sample covariance matrix:

$$\mathrm{ENL} = \frac{\mathrm{tr}(\langle \bar{\boldsymbol{c}} \rangle \langle \bar{\boldsymbol{c}} \rangle)}{\langle \mathrm{tr}(\bar{\boldsymbol{c}})^2 \rangle - \mathrm{tr}(\langle \bar{\boldsymbol{c}} \rangle)^2},$$

where $\bar{\boldsymbol{c}}$ is the look-averaged complex covariance matrix given by Equation (5.30), and $\langle \cdot \rangle$ indicates local average over a homogeneous region.

(a) Show that this expression reduces to Equation (5.28) for the single polarization case.

(b) Write a Python script to calculate it.

(c) (K. Conradsen (2013) private communication) Consider N-dimensional, complex-valued observations $\boldsymbol{z}(\nu) = \boldsymbol{x}(\nu) + \mathbf{i}\boldsymbol{y}(\nu)$, $\nu = 1 \dots m$, and organize them into real and imaginary parts in the $m \times N$ data matrices

$$\boldsymbol{\mathcal{X}} = \begin{pmatrix} \boldsymbol{x}(1)^\top \\ \vdots \\ \boldsymbol{x}(m)^\top \end{pmatrix}, \quad \boldsymbol{\mathcal{Y}} = \begin{pmatrix} \boldsymbol{y}(1)^\top \\ \vdots \\ \boldsymbol{y}(m)^\top \end{pmatrix}. \tag{5.68}$$

If the real and imaginary components $x(\nu)_1 \dots x(\nu)_N, y(\nu)_1 \dots y(\nu)_N$, $\nu = 1 \dots m$, are all standard normally distributed and independent, then

$$\boldsymbol{w} = \frac{1}{2}(\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{Y}}^\top \boldsymbol{\mathcal{Y}} - \mathbf{i}(\boldsymbol{\mathcal{X}}^\top \boldsymbol{\mathcal{Y}} - \boldsymbol{\mathcal{Y}}^\top \boldsymbol{\mathcal{X}})) \tag{5.69}$$

are realizations of a complex Wishart distributed random matrix $\boldsymbol{W} \sim \mathcal{W}_C(\boldsymbol{I}, N, m)$. Thus we may generate a complex Wishart distributed sample by generating $2Nm$ standardized Gaussian random samples, organizing them into the two data matrices, Equation (5.68), and then computing $\boldsymbol{w}$ as in Equation (5.69). Use this recipe to simulate a 500×500-pixel, quad polarimetric ($N = 3$) SAR image in covariance matrix format and verify the correctness of the script of part (b) above. Alternatively, if you wish to skip part (b), verify the script `enlml.py` (Appendix C) with the simulated image.

11. From the definition of the gradient, Equation (5.4) shows that the terrain slope angle θ_p can be approximated from a DEM by Equation (5.58).

6

Supervised Classification Part 1

Land cover classification of remote sensing imagery is an undertaking which falls into the general category of *pattern recognition*. Pattern recognition problems, in turn, are usually approached by developing appropriate *machine learning* algorithms. Broadly speaking, machine learning involves tasks for which there is no known direct, analytic method to compute a desired output from a set of inputs. The strategy adopted is for the computer to "learn" from a set of representative examples. In the case of supervised classification, the exercise can often be seen as one of modeling probability distributions. On the basis of representative data for, say, K land cover classes presumed to be present in a scene, the *a posteriori* probabilities for class k conditional on observation $\boldsymbol{g}$, $\Pr(k \mid \boldsymbol{g})$, $k = 1 \dots K$, are "learned" or approximated. This is usually called the *training phase* of the classification procedure. Then these probabilities are used to classify all of the pixels in the image, a step referred to as the *generalization* or *prediction phase*.

In the present chapter, we will consider three representative models for supervised classification which involve this sort of probability density estimation: a *parametric model* (the Bayes maximum likelihood classifier), a *non-parametric model* (Gaussian kernel classification), and a *semi parametric* or *mixture model* (the feed-forward neural network or FFN). In the case of the neural network, the *back propagation* training algorithm is by far the most commonly used. We make some considerable effort to explain it in detail and, in Appendix B, to develop more sophisticated and efficient analytical methods based on *scaled conjugate gradient* and the *Kalman filter*. All of these, however, will be seen to become unwieldy for so-called *deep learning* or *multiple hidden layer* neural network architectures. Fortunately, we can take advantage of the auto differentiation capability of modern deep learning libraries, in this instance TensorFlow, in order to work conveniently with multilayer FFN classifiers.

The Chapter concludes with a detailed explanation of the *support vector machine (SVM)* classifier. SVMs are also non-parametric in the sense that they make direct use of a subset of the labeled training data (the support vectors) to effect a partitioning of the feature space, however, unlike the aforementioned classifiers, without reference to the statistical distributions of the training data: the classes are discriminated on the basis of their most extreme members. The topics of accuracy assessment and model comparison will be treated in Chapter 7. There we also look at polarimetric SAR image

DOI: 10.1201/9781003503286-6

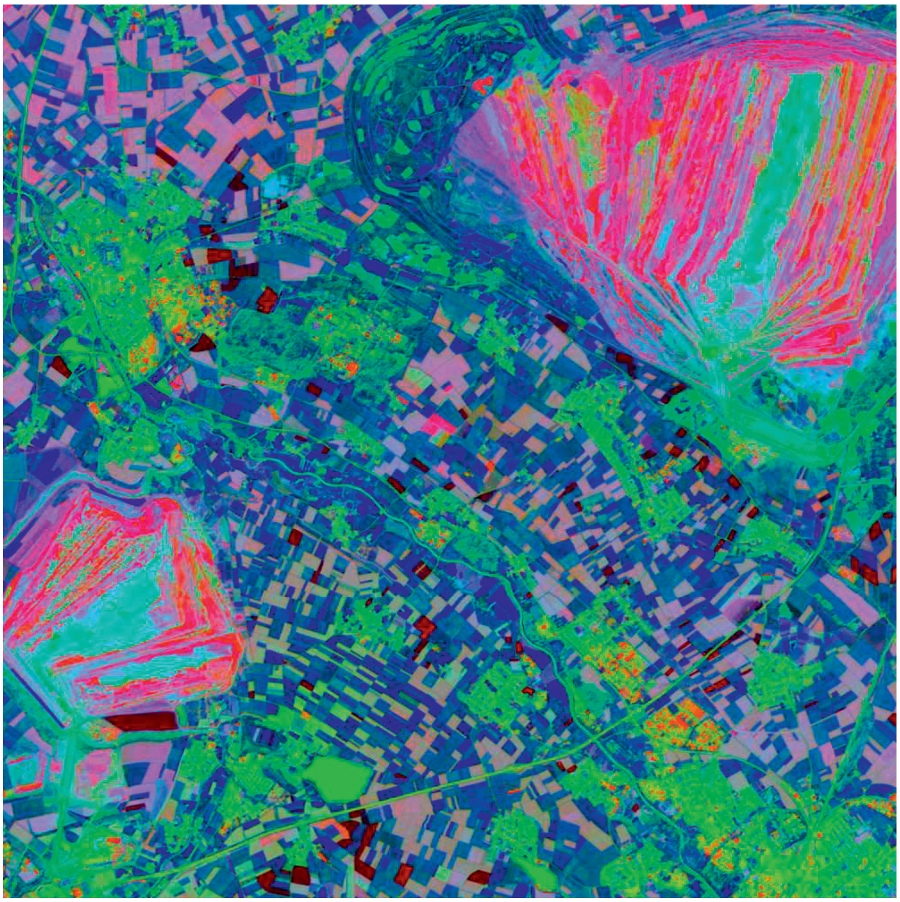


FIGURE 6.1
RGB color composite (1000×1000 pixels, linear 2% saturation stretch) of the first three principal components 1(red), 2(green), and 3(blue) of the nine non thermal bands of an ASTER scene acquired over the towns of Jülich (to the northwest) and Düren (southeast), Germany, on May 1, 2007.

classification, ensembles of classifiers as well as object classification with convolutional neural networks.

To illustrate the various algorithms developed here and in the following chapters on image classification, we will work with the ASTER scene shown in Figure 6.1 (see also Figure 1.1). In the ASTER dataset, the six SWIR bands have been sharpened to the 15-m ground resolution of the three VNIR bands with the *à trous* wavelet fusion method of Section 5.3.5 and a principal components analysis of the stacked nine-band image has been performed. The classification examples in the present chapter will be carried out with subsets of the principal components of the ASTER image.

6.1 Maximizing the *a posteriori* probability

The basis for most of the classifiers that we consider in this chapter is a decision rule based on the *a posteriori* probabilities $Pr(k \mid \boldsymbol{g})$, so this rule will be our starting point.

Let us begin by defining a *loss function* $L(k, \boldsymbol{g})$ which measures the cost of associating the observation $\boldsymbol{g}$ with the class k. Let λ_{kj} be the loss incurred if $\boldsymbol{g}$ in fact belongs to class k, and is classified as belonging to class j. It can reasonably be assumed that

$$\lambda_{kj} \begin{cases} = 0 & \text{if } k = j \\ > 0 & \text{otherwise,} \end{cases} \qquad k, j = 1 \dots K, \tag{6.1}$$

that is, correct classifications do not incur losses while misclassifications do. The loss function can then be expressed as a sum over the individual losses, weighted according to their probabilities of occurrence, $\Pr(j \mid \boldsymbol{g})$,

$$L(k, \boldsymbol{g}) = \sum_{j=1}^{K} \lambda_{kj} \Pr(j \mid \boldsymbol{g}). \tag{6.2}$$

Without further specifying λ_{kj}, a loss-minimizing decision rule for classification may be defined (ignoring the possibility of ties) as

$$\boldsymbol{g} \text{ is in class } k \text{ provided } L(k, \boldsymbol{g}) \le L(j, \boldsymbol{g}) \text{ for all } j = 1 \dots K. \tag{6.3}$$

So far we have been quite general. Now suppose the losses are independent of the kind of misclassification that occurs (for instance, the classification of a "forest" pixel into the class "meadow" is just as costly as classifying it as "urban area," etc.). Then we can write

$$\lambda_{kj} = 1 - \delta_{kj}, \tag{6.4}$$

where $\delta_{kj} = 1$ for $k = j$ and 0 otherwise. Thus any given misclassification ($j \ne k$) has unit cost, and a correct classification ($j = k$) costs nothing, as before. We then obtain from Equation (6.2)

$$L(k, \boldsymbol{g}) = \sum_{j=1}^{K} \Pr(j \mid \boldsymbol{g}) - \Pr(k \mid \boldsymbol{g}) = 1 - \Pr(k \mid \boldsymbol{g}), \; k = 1 \dots K, \tag{6.5}$$

and with (6.3) the following decision rule:

$$\boldsymbol{g} \text{ is in class } k \text{ provided } \Pr(k \mid \boldsymbol{g}) \ge \Pr(j \mid \boldsymbol{g}) \text{ for all } j = 1 \dots K; \tag{6.6}$$

in other words, assign each new observation to the class with the highest *a posteriori* probability As indicated in the introduction, our main task will therefore be to determine the posterior probabilities $\Pr(k \mid \boldsymbol{g})$.

6.2 Training data and separability

The choice of training data is arguably the most difficult and critical part of the supervised classification process. The standard procedure is to select areas within a scene which are representative of each class of interest. The areas are referred to as *training areas* or *regions of interest* (ROIs), from which the training observations are selected. Some fraction of the representative data may be retained for later accuracy assessment. These comprise the so-called *test data* and are withheld from the training phase in order not to bias the subsequent evaluation. We will refer to the set of labeled training data as *training pairs* or *training examples* and write it in the form

$$\mathcal{T} = \{\boldsymbol{g}(\nu), \ell(\nu)\}, \quad \nu = 1 \dots m, \tag{6.7}$$

where m is the number of observations and

$$\ell(\nu) \in \mathcal{K} = \{1 \dots K\} \tag{6.8}$$

is the class label of observation $\boldsymbol{g}(\nu)$.

Ground reference data used in the following discussions were collected on the same day as the acquisition of the ASTER image in Figure 6.1. Figure 6.2 shows photographs for four of the ten land cover categories used for classification. The others were water, suburban settlements, urban areas/industrial parks, herbivorous forest, coniferous forest, and open cast mining. In all, 30 regions of interest were identified in the scene as representative of the 10 classes, involving 7173 pixels. They are shown in Figure 6.3. Of these, 2/3 sampled uniformly across the ROIs were used for training and the remainder reserved for testing, i.e., estimating the generalization error on new observations.

The *degree of separability* of the training observations will give some indication of the prospects for success of the classification procedure and can help in deciding how the data should be processed prior to classification. A very commonly used separability measure may be derived by considering the *Bayes error*. Suppose that there are just two classes involved, $\mathcal{K} = \{1, 2\}$. If we apply the decision rule, Equation (6.6), for some pixel intensity vector $\boldsymbol{g}$, we must assign the class as that having maximum *a posteriori* probability. Therefore, the probability $r(\boldsymbol{g})$ of incorrectly classifying the pixel is given by

$$r(\boldsymbol{g}) = \min[\ \Pr(1 \mid \boldsymbol{g}), \Pr(2 \mid \boldsymbol{g})\ ].$$

The Bayes error ϵ is defined to be the average value of $r(\boldsymbol{g})$, which we can calculate as the integral of $r(\boldsymbol{g})$ times the probability density $p(\boldsymbol{g})$, taken over all of the observations $\boldsymbol{g}$:

$$\begin{aligned} \epsilon = \int r(\boldsymbol{g})p(\boldsymbol{g})d\boldsymbol{g} &= \int \min[\ \Pr(1 \mid \boldsymbol{g}), \Pr(2 \mid \boldsymbol{g})\ ]p(\boldsymbol{g})d\boldsymbol{g} \\ &= \int \min[\ p(\boldsymbol{g} \mid 1)\Pr(1), p(\boldsymbol{g} \mid 2)\Pr(2)\ ]d\boldsymbol{g}. \end{aligned} \tag{6.9}$$

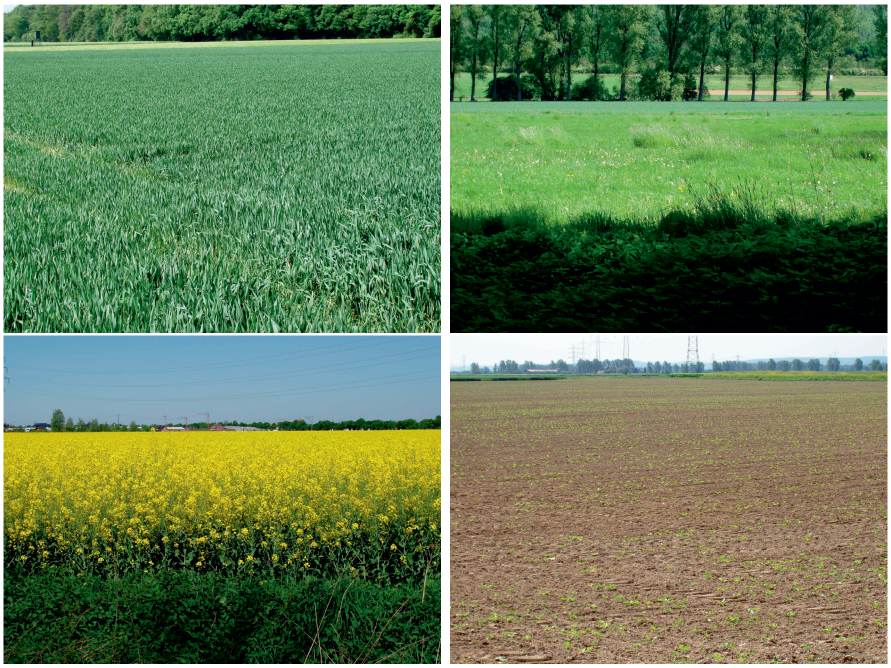


FIGURE 6.2
Ground reference data for four land cover categories, photographed on May 1st, 2007. Top left: cereal grain, top right: grassland, bottom left: rapeseed, bottom right: sugar beets.

Bayes' Theorem, Equation (2.70), was invoked in the last equality. The Bayes error may be used as a measure of the separability of the two classes: the smaller the error, the better the separability.

Calculating the Bayes error is in general difficult, but we can at least get an approximate upper bound on it as follows (Fukunaga, 1990). First note that, for any $a, b \geq 0$,

$$\min[\,a, b\,] \leq a^s b^{1-s}, \quad 0 \leq s \leq 1.$$

For example, if $a < b$, then the inequality can be written

$$a \leq a\left(\frac{b}{a}\right)^{1-s},$$

which is clearly true. Applying this inequality to Equation (6.9), we get the *Chernoff bound* ϵ_u on the Bayes error,

$$\epsilon \leq \epsilon_u = \Pr(1)^s \Pr(2)^{1-s} \int p(\boldsymbol{g} \mid 1)^s p(\boldsymbol{g} \mid 2)^{1-s} d\boldsymbol{g}. \tag{6.10}$$

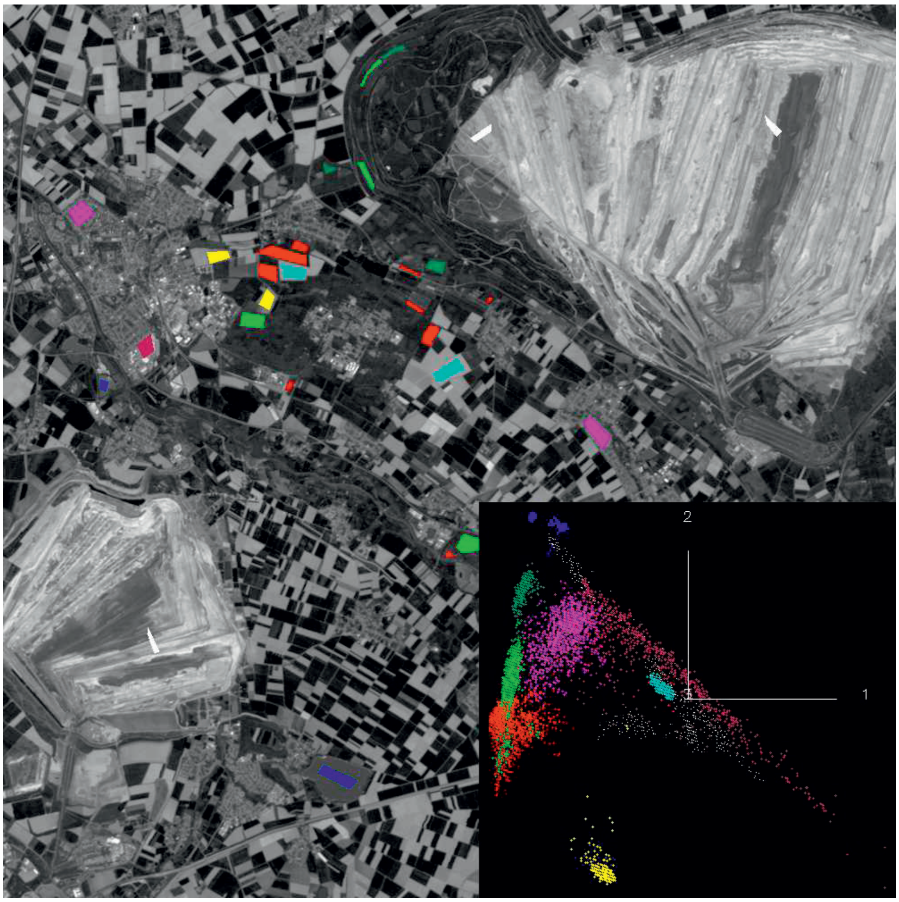


FIGURE 6.3
ROIs for supervised classification. The insert shows the training observations projected onto the plane of the first two principal axes.

The least upper bound is then determined by minimizing ϵ_u with respect to s. If $p(\boldsymbol{g} \mid 1)$ and $p(\boldsymbol{g} \mid 2)$ are multivariate normal distributions with equal covariance matrices $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2$, then it can be shown that the minimum in fact occurs at $s = 1/2$. Approximating the minimum as $s = 1/2$ also for the case where $\boldsymbol{\Sigma}_1 \neq \boldsymbol{\Sigma}_2$ leads to the (somewhat less tight) *Bhattacharyya bound* ϵ_B,

$$\epsilon \leq \epsilon_B = \sqrt{\Pr(1)\Pr(2)} \int \sqrt{p(\boldsymbol{g} \mid 1)p(\boldsymbol{g} \mid 2)}\; d\boldsymbol{g}. \tag{6.11}$$

This integral can be evaluated explicitly (Exercise 1). The result is

$$\epsilon_B = \sqrt{\Pr(1)\Pr(2)}\; e^{-B},$$

where B is the *Bhattacharyya distance*, given by

$$B = \frac{1}{8}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^\top \left[\frac{\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2}{2}\right]^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) + \frac{1}{2}\log\left(\frac{|\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2|/2}{\sqrt{|\boldsymbol{\Sigma}_1||\boldsymbol{\Sigma}_2|}}\right). \quad (6.12)$$

Large values of B imply small upper limits on the Bayes error and hence good separability. The first term in B is a squared average *Mahalanobis distance* (see Section 6.3) and expresses the class separability due to the dissimilarity of the class means.* The second term measures the difference between the covariance matrices of the two classes. It vanishes when $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2$.

The Bhattacharyya distance as a measure of separability has the disadvantage that it continues to grow even after the classes have become so well separated that any classification procedure could distinguish them perfectly. The *Jeffries–Matusita (J-M) distance* measures separability of two classes on a more convenient scale $[0 - 2]$ in terms of B:

$$J = 2(1 - e^{-B}). \quad (6.13)$$

As B continues to grow, the measure saturates at the value 2. The factor 2 comes from the fact that the Jeffries–Matusita distance can be derived independently as the average distance between two density functions; see Richards (2012) and Exercise 1.

We can calculate the J-M distance easily with the GEE Python API. After uploading the ASTER principal component image and the training area shape files to the code editor in the form of the (shared) FeatureCollection `train`, we access them with

```
import ee
ee.Initialize()
# first 4 principal components of ASTER image
image = ee.Image('projects/.../CRC5/AST_20070501_pca') \
                                        .select(0,1,2,3)
# training data
table = ee.FeatureCollection('projects/.../CRC5/train')
trainData = image.sampleRegions(table,['CLASS_ID'])
print(trainData.size().getInfo()  )
7173
```

The following function calculates the J-M separation for two classes:

```
def jmsep(class1, class2, image, table):
    # Jeffries-Matusita separability
    table1 = table.filter(
        ee.Filter.eq('CLASS_ID',str(class1-1)))
    m1 = image.reduceRegion(ee.Reducer.mean(),table1)\
              .toArray()
    s1 = image.toArray() \
```

*This term is proportional to the maximum value of the Fisher linear discriminant, as can be seen by substituting Equation (3.82) into Equation (3.81); see Exercise 13, Chapter 3.

```
                .reduceRegion(ee.Reducer.covariance(),table1)\
                .toArray()
          table2 = table.filter(
              ee.Filter.eq('CLASS_ID',str(class2-1)))
          m2 = image.reduceRegion(ee.Reducer.mean(),table2)\
                     .toArray()
          s2 = image.toArray() \
              .reduceRegion(ee.Reducer.covariance(),table2,15)\
                     .toArray()
          m12 = m1.subtract(m2)
          m12 = ee.Array([m12.toList()]) # makes 2D matrix
          s12i = s1.add(s2).divide(2).matrixInverse()
          # first term in Bhattacharyya distance
          B1 = m12.matrixMultiply(
                s12i.matrixMultiply(m12.matrixTranspose())) \
                  .divide(8)
          ds1 = s1.matrixDeterminant()
          ds2 = s2.matrixDeterminant()
          ds12 = s1.add(s2).matrixDeterminant()
          # second term
          B2 = ds12.divide(2).divide(ds1.multiply(ds2).sqrt())\
                    .log().divide(2)
          B = ee.Number(B1.add(B2).project([0]).toList().get(0))
          # J-M separability
          return ee.Number(1).subtract(ee.Number(1) \
                    .divide(B.exp())).multiply(2)
```

For example, for the classes “grain” and “grassland” (classes 7 and 8)

```
print jmsep(7,8,image,table).getInfo()
```

```
1.79160515129
```

Some more examples are shown in Table 6.1.

TABLE 6.1
The lowest 10 paired class separabilities for the first four principal components of the ASTER scene.

Class 1	Class 2	J-M Distance
Grain	Grassland	1.79
Settlement	Industry	1.82
Grassland	Herbivorous	1.88
Settlement	Herbivorous	1.96
Settlement	Grassland	1.98
Industry	Coniferous	1.99
Coniferous	Herbivorous	1.99
Sugar beet	Mining	1.99
Grain	Herbivorous	2.00
Industry	Herbivorous	2.00

6.3 Bayes maximum-likelihood classification

Consider once again Bayes' Theorem, expressed in the form of Equation (2.70),

$$\Pr(k \mid \boldsymbol{g}) = \frac{p(\boldsymbol{g} \mid k)\Pr(k)}{p(\boldsymbol{g})}, \tag{6.14}$$

where $\Pr(k)$, $k = 1 \dots K$, are prior probabilities, $p(\boldsymbol{g} \mid k)$ is a class-specific probability density function, and where $p(\boldsymbol{g})$ is given by

$$p(\boldsymbol{g}) = \sum_{j=1}^{K} p(\boldsymbol{g} \mid j)\Pr(j).$$

Since $p(\boldsymbol{g})$ is independent of k, we can write the decision rule, Equation (6.6), as

$$\boldsymbol{g} \text{ is in class } k \text{ provided } p(\boldsymbol{g} \mid k)\Pr(k) \ge p(\boldsymbol{g} \mid j)\Pr(j) \text{ for all } j = 1 \dots K. \tag{6.15}$$

Now suppose that the observations from class k are sampled from a multivariate normal distribution. Then the density functions are given by

$$p(\boldsymbol{g} \mid k) = \frac{1}{(2\pi)^{N/2}|\boldsymbol{\Sigma}_k|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{g} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g} - \boldsymbol{\mu}_k)\right). \tag{6.16}$$

Taking the logarithm of Equation (6.16) gives

$$\log\big(p(\boldsymbol{g} \mid k)\big) = -\frac{N}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}_k| - \frac{1}{2}(\boldsymbol{g} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g} - \boldsymbol{\mu}_k).$$

The first term may be ignored, as it too is independent of k. Together with Equation (6.15) and the definition of the *discriminant function*

$$d_k(\boldsymbol{g}) = \log(\Pr(k)) - \frac{1}{2}\log|\boldsymbol{\Sigma}_k| - \frac{1}{2}(\boldsymbol{g} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g} - \boldsymbol{\mu}_k), \tag{6.17}$$

we obtain the *Gaussian Bayes maximum-likelihood classifier*:

$$\boldsymbol{g} \text{ is in class } k \text{ provided } d_k(\boldsymbol{g}) \ge d_j(\boldsymbol{g}) \text{ for all } j = 1 \dots K. \tag{6.18}$$

There may be no information about the prior class probabilities $\Pr(k)$, in which case they can be set equal and ignored in the classification. Then the factor 1/2 in Equation (6.17) can be dropped as well and the discriminant becomes

$$d_k(\boldsymbol{g}) = -\log|\boldsymbol{\Sigma}_k| - (\boldsymbol{g} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g} - \boldsymbol{\mu}_k). \tag{6.19}$$

The second term in Equation (6.19) is the square of the Mahalanobis distance

$$\sqrt{(\boldsymbol{g}-\boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g}-\boldsymbol{\mu}_k)}.$$

The contours of constant multivariate probability density in Equation (6.16) are hyper-ellipsoids of constant Mahalanobis distance to the mean $\boldsymbol{\mu}_k$.

The moments $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$, which appear in the discriminant functions, may be estimated from the training data using the maximum-likelihood parameter estimates (see Section 2.4, Equations (2.74) and (2.75))

$$\begin{aligned}\hat{\boldsymbol{\mu}}_k &= \frac{1}{m_k}\sum_{\{\nu\mid \ell(\nu)=k\}} \boldsymbol{g}(\nu)\\ \hat{\boldsymbol{\Sigma}}_k &= \frac{1}{m_k}\sum_{\{\nu\mid \ell(\nu)=k\}} (\boldsymbol{g}(\nu)-\boldsymbol{\mu}_k)(\boldsymbol{g}(\nu)-\boldsymbol{\mu}_k)^\top,\end{aligned} \tag{6.20}$$

where m_k is the number of training pixels with class label k.

Having estimated the parameters from the training data, the generalization phase consists simply of applying the rule (6.18) to all of the pixels in the image. Because of the small number of parameters to be estimated, maximum likelihood classification is extremely fast. Its weakness lies in the restrictiveness of the assumption that all observations are drawn from multivariate normal probability distributions.[*] Computational efficiency eventually achieved at the cost of generality is a characteristic of *parametric classification models*, to which category the maximum-likelihood classifier belongs (Bishop, 1995).

Note that applying the rule of Equation (6.18) will place any observation into one of the K classes no matter how small its maximum discriminant function turns out to be. If it is thought that some classes may have been overlooked, or if no training data were available for one or two known classes, then it might be reasonable to assume that observations with small maximum discriminant functions belong to one of these inaccessible classes. Then they should perhaps not be classified at all. If this is desired, it may be achieved simply by setting a threshold on the maximum discriminant $d_k(\boldsymbol{g})$, marking the observations lying below the threshold as “unclassified.”

6.3.1 Naive Bayes on the GEE

The *naive Bayes classifier*, one of several supervised classification algorithms available on the GEE, essentially assumes that there are no correlations between the training features, i.e., that the class-specific covariance matrices $\boldsymbol{\Sigma}_k$ are diagonal. Since in our pre-processing of the ASTER image we carry out

[*]This can be mitigated somewhat by making use of Gaussian mixtures, which we will meet in Chapter 8 in the context of unsupervised classification.

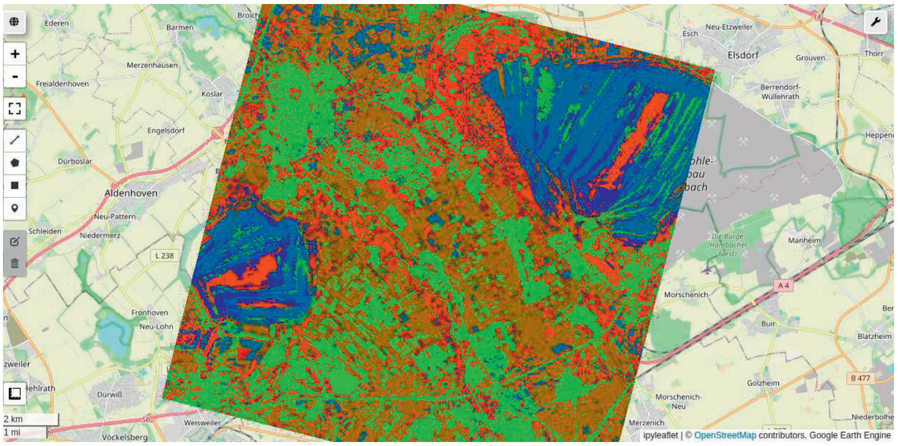


FIGURE 6.4
Naive Bayes classification of the first 4 principal components of the image in Figure 6.1.

a principal components transformation, this would seem to be not an unreasonable assumption. However the transformation diagonalizes the global covariance matrix of the image, not the covariance matrices of the class-specific observations. Nevertheless, let's continue with the training data of Section 6.2 and classify with naive Bayes on the GEE:

```
import geemap
# rename the class ids from strings to integers
trainData = image.sampleRegions(table,['CLASS_ID']) \
    .remap(['0','1','2','3','4','5','6','7','8','9'],
           [0,1,2,3,4,5,6,7,8,9],'CLASS_ID')
# train a naive Bayes classifier
classifier = ee.Classifier.smileNaiveBayes()
trained = classifier\
    .train(trainData,'CLASS_ID',image.bandNames())
# classify the image and display
classified = image.classify(trained)
map = geemap.Map()
map.centerObject(classified, 12)
map.addLayer(classified.byte(), {'min': 0.0, 'max': 10.0,
'palette': ['FF0000','#00FF00','#0000FF']},'Naive␣Bayes')
map
```

The output cell is shown as an RGB color image in Figure 6.4. The actual pixel grayscale values are the assigned class position in the array

```
['WATER', 'RAPESEED', 'SUGARBEET', 'SUBURBAN',
'INDUSTRIAL', 'CONIFEROUS', 'GRAIN', 'GRASSLAND',
'HERBIVOROUS', 'OPENCAST']
```

see Figure 6.5 below. Close inspection of the output in the accompanying Jupyter notebook reveals many obvious classification errors, especially in the opencast mines.

6.3.2 Scripts for supervised classification

The utility script `readshp.py` in the `auxil` package (Appendix C) extracts pixel data from shape files covering the training regions chosen for supervised classification. There, the labeled observations are stored in data matrix format in the $m \times N$ array variable `Gs` and the corresponding labels in the $m \times K$ array variable `Ls`. Here, m is the number of training or test observations, N is their dimensionality and K is the number of classes. The individual labels are in fact K-element arrays with zeroes everywhere except at the position of the class. This is referred to as *one-hot encoding*. For example, if there are 5 classes, class 2 corresponds to the label array $[0, 1, 0, 0, 0]$, adopting the convention that classes are numbered from 1 rather than 0. One-hot-coding will turn out to be convenient when we come to consider neural network classifiers.

All of the Python code for local (as opposed to GEE) supervised classification is bundled for convenience into the module `auxil.supervisedclass.py`. We begin with the code for the maximum likelihood method, as shown in Listing 6.1. The `train` method, starting from line 11, estimates the mean vectors and covariance matrices for each of the training classes. In `classify` (line 30) the new observations are classified with the discriminant of Equation (6.19), lines 38 to 41, whereby the Mahalanobis distance $(\boldsymbol{g} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g} - \boldsymbol{\mu}_k)$ is efficiently determined by Numpy's array processing.

For further post-processing of classification results we shall later make use of the posterior class membership probabilities $\Pr(k \mid \boldsymbol{g})$. However in this case, since the maximum-likelihood code does not calculate class probabilities, `None` is returned as a place-holder, line 43.

The front-end routine `classify.py`, documented in Appendix C, reads the image and training data, creates an instance of a classifier (in this case `Maxlike`) and then generates both a thematic map and test results file.

With regard to test observations in general, one approach to carrying out an unbiased assessment of the accuracy of supervised classification methods is to reserve a number of "ground truth" training regions containing areas of labeled data not used during the training phase. These are then classified in the evaluation phase. We will prefer a somewhat different philosophy, arguing that, if other representative training areas are indeed available for evaluation, then they should also be used to train the classifier. For evaluation purposes, some portion of the pixels in *all* of the training areas can be held back, but such test data should be selected from the pool of available labeled observations. This point of view assumes that all training/test areas are equally representative of their respective classes, but if that were not the case, then there would be no justification to use them at all. Accordingly, the

Listing 6.1: Maximum-likelihood classifier (excerpt from the Python module `auxil.supervisedclass.py`).

```
class Maxlike(object):
    '''Maximum Likelihood Classifier'''
    def __init__(self, gs, ls):
        N = gs.shape[1]
        self._K = ls.shape[1]
        self._Gs = gs
        self._ls = np.argmax(ls, 1)
        self._sigma = np.zeros((self._K, N, N))
        self._sigma_i = np.zeros((self._K, N, N))
        self._mu =  np.zeros((self._K, N))
    def train(self):
        try:
            for k in range(self._K):
                idx = np.where(self._ls == k)[0]
                # observations in class k
                gs_k = self._Gs[idx]
                # estimated mean for class k
                self._mu[k] = np.mean(gs_k, axis=0)
                # centered observations
                gs_k = (gs_k - self._mu[k])
                # estimated covariance matrix
                self._sigma[k] =  \
                      np.cov(gs_k, rowvar=False)
                self._sigma_i[k] = \
                      np.linalg.inv(self._sigma[k])
            return True
        except Exception as e:
            print('Error:␣%s' % e)
            return None
    def classify(self, gs):
        try:
            d = np.zeros((self._K, gs.shape[0]))
            for k in range(self._K):
                # centered observations
                gs = (gs - self._mu[k])
                # discriminant array
                sig_i = self._sigma_i[k]
                d[k] = -(np.dot(gs, sig_i)*gs) \
                            .sum(axis=1)
                d[k] -= np.log(np.linalg \
                            .det(self._sigma[k]))
            classes = np.argmax(d, axis=0)
            return (classes, None)
        except Exception as e:
            print('Error:␣%s' % e)
```

training observations returned by `readshp.py` to the front-end `classify.py` are partitioned randomly into training and test datasets in the ratio 2:1. Test classification results are saved to a file in a format consistent with that used by Bayes Maximum-Likelihood and all of the other the classification routines to be described in the remainder of this chapter. Accuracy evaluation using the test results will be discussed in Chapter 7, as well as a more efficient *cross-validation* train/test procedure.

To demonstrate, we invoke the script `classify.py` for the ASTER image and associated training shape files, setting the band position `-p` flag to the first five principal components and the algorithm `-a` flag to 1 in order to select the Bayes Maximum Likelihood classifier:

```
run scripts/classify -p [1,2,3,4,5] -a 1 \
 imagery/AST_20070501_pca.tif imagery/train.shp

reading training data...
training on 4774 pixel vectors...
```

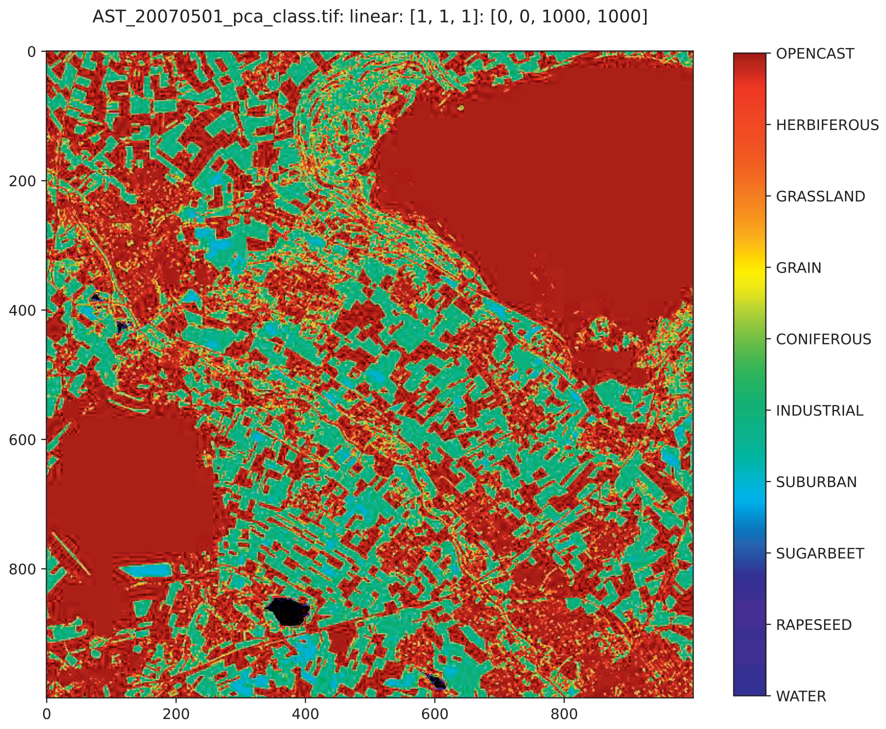


FIGURE 6.5
Bayes maximum-likelihood supervised classification of the first four principal components of the image in Figure 6.1.

```
classes: ['WASSER [BL', 'RAPS [YELL', ... 'TAGEBAU [W']
elapsed time 0.00220489501953
classifying...
elapsed time 3.31426310539

thematic map written to:.../AST_20070501_pca_class.tif
test results written to:.../AST_20070501_pca_MaxLike.tst
```

Note that both training and prediction times are very short, a characteristic, as already mentioned, of parametric models. The classified image is shown in Figure 6.5. Qualitative comparison with Figure 6.4 indicates a considerably poorer result than for Naive Bayes. The Naive Bayes classifier, by assuming conditional independence of the image band intensities, allows the parameters to be estimated separately for each feature. Although clearly an approximation, this is more scalable and robust to small sample sizes. The complete Bayes classifier requires estimating the full N-dimensional distribution, which can be challenging with limited training data as is here the case.

6.4 Gaussian kernel classification

Non-parametric classification models estimate the class-specific probability densities $p(\boldsymbol{g} \mid k)$, as in the preceding section, from a set of training data. However, unlike the maximum likelihood classifier, no strong prior assumptions about the nature of the densities are made. In the *Parzen window* approach to non-parametric classification (Duda et al., 2001), each training observation $\boldsymbol{g}(\nu)$, $\nu = 1 \dots m$, is used as the center of a local kernel function. The probability density for class k at a point $\boldsymbol{g}$ is taken to be the average of the kernel functions for the training data in that class, evaluated at $\boldsymbol{g}$. For example, using a Gaussian kernel, the probability density for the kth class is estimated as

$$p(\boldsymbol{g} \mid k) \approx \frac{1}{m_k} \sum_{\{\nu \mid \ell(\nu)=k\}} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\|\boldsymbol{g} - \boldsymbol{g}(\nu)\|^2}{2\sigma^2}\right). \tag{6.21}$$

The quantity σ is a smoothing parameter, which has been chosen in this case to be class-independent. Since the Gaussian functions are normalized, we have

$$\int_{-\infty}^{\infty} p(\boldsymbol{g} \mid k) d\boldsymbol{g} = \frac{1}{m_k} \sum_{\{\nu \mid \ell(\nu)=k\}} 1 = 1,$$

as required of a probability density. Under fairly general conditions, the right-hand side of Equation (6.21) can be shown to converge to $p(\boldsymbol{g} \mid k)$ as the number of training observations tends to infinity. If as before we set the prior

Listing 6.2: Gaussian kernel classification (excerpt from the Python module auxil.supervisedclass.py).

```
class Gausskernel(object):
    '''Gauss Kernel Classifier'''
    def __init__(self,Gs,ls):
        self._K = ls.shape[1]
        self._Gs = Gs
        self._N = Gs.shape[1]
        self._ls = np.argmax(ls,1)
        self._m = Gs.shape[0]
    def output(self,sigma,Hs,symm=True):
        pvs = np.zeros((Hs.shape[0],self._K))
        kappa = auxil1.kernelMatrix(
            Hs,self._Gs,gam=0.5/(sigma**2),
                                        k=1)[0]
        if symm:
            kappa[range(self._m),range(self._m)] = 0
        for j in range(self._K):
            kpa = np.copy(kappa)
            idx = np.where(self._ls!=j)[0]
            nj = self._m - idx.size
            kpa[:,idx] = 0
            pvs[:,j] = np.sum(kpa,1).ravel()/nj
        s = np.transpose(np.tile(np.sum(pvs,1),
                                              (self._K,1)))
        return pvs/s
    def theta(self,sigma):
        pvs = self.output(sigma,self._Gs,True)
        labels = np.argmax(pvs,1)
        idx = np.where(labels != self._ls)[0]
        n = idx.size
        error = float(n)/(self._m)
        print ('sigma: %f  error: %f'%(sigma,error))
        return error

    def train(self):
        result = minimize_scalar(
         self.theta,bracket=(0.001,0.1,1.0),tol=0.001)
        if result.success:
            self._sigma_min = result.x
            return True
        else:
            print (result.message)
            return None
```

probabilities in the decision rule, Equation (6.6), equal to one another, then an observation $\boldsymbol{g}$ will be assigned to class k when

$$p(\boldsymbol{g} \mid k) \ge p(\boldsymbol{g} \mid j), \quad j = 1 \dots K.$$

Training the Gaussian kernel classifier involves searching for an optimal value of the smoothing parameter σ. Too large a value will wash out the class dependency, and too small a value will lead to poor generalization on new data. Training can be effected very conveniently by minimizing the misclassification rate with respect to σ. When presenting an observation vector $\boldsymbol{g}(\nu)$ to the classifier during the training phase, the contribution to the probability density at the point $\boldsymbol{g}(\nu)$ from class k is, from Equation (6.21) and apart from a constant factor, given by

$$\begin{aligned} p(\boldsymbol{g}(\nu) \mid k) &= \frac{1}{m_k} \sum_{\{\nu' \mid \ell(\nu')=k\}} \exp\left(-\frac{\|\boldsymbol{g}(\nu) - \boldsymbol{g}(\nu')\|^2}{2\sigma^2}\right) \\ &= \frac{1}{m_k} \sum_{\{\nu' \mid \ell(\nu')=k\}} (\boldsymbol{\mathcal{K}})_{\nu\nu'}, \end{aligned} \tag{6.22}$$

where $\boldsymbol{\mathcal{K}}$ is an $m \times m$ Gaussian kernel matrix, see Equation (4.25). It is advisable to delete the contribution of $\boldsymbol{g}(\nu)$ itself to the sum in the above equation in order to avoid biasing the classification in favor of the training observation's own label, a bias which would otherwise arise due to the appearance of a zero in the exponent and a dominating contribution to $p(\boldsymbol{g}(\nu) \mid k)$; see Masters (1995). This amounts to zeroing the diagonal of $\boldsymbol{\mathcal{K}}$ before performing the sum in Equation (6.22).

The Python object class for Gaussian kernel classification is in Listing 6.2. The instance method `output(sigma,Hs,symm)` calculates the arrays of class probability densities for all pixel vectors in the data matrix `Hs` using the current value of the smoothing parameter `sigma`. The property `self._Gs` contains the training pixels, also in data matrix format. Their labels are stored in the m-dimensional array `self._ls`. In the training phase, the keyword `symm` is set to `True`, indicating that `Hs` and `self._Gs` are in fact identical. The diagonal of the symmetric kernel matrix `kappa` is set to zero (line 15). The sums in Equation (6.22) are calculated in the `for`-loop in lines 16 to 21. The seemingly redundant normalization of the probability vectors `pvs`, line 24, is necessary because, due to the fact that Equation (6.22) is a discrete approximation to the posterior probabilities, the vector components don't sum exactly to one.

The minimization of the misclassification rate with respect to σ takes place using Brent's parabolic interpolation method in the Scipy function `minimize-_scalar()`, line 35. The class function `theta(sigma)`, passed to this minimization routine, calculates the misclassification rate with a call to `output (sigma,self._Gs,True)`. After training, classification of the entire image proceeds with repeated calls to `output(sigma,Hs,False)` where `sigma` is fixed

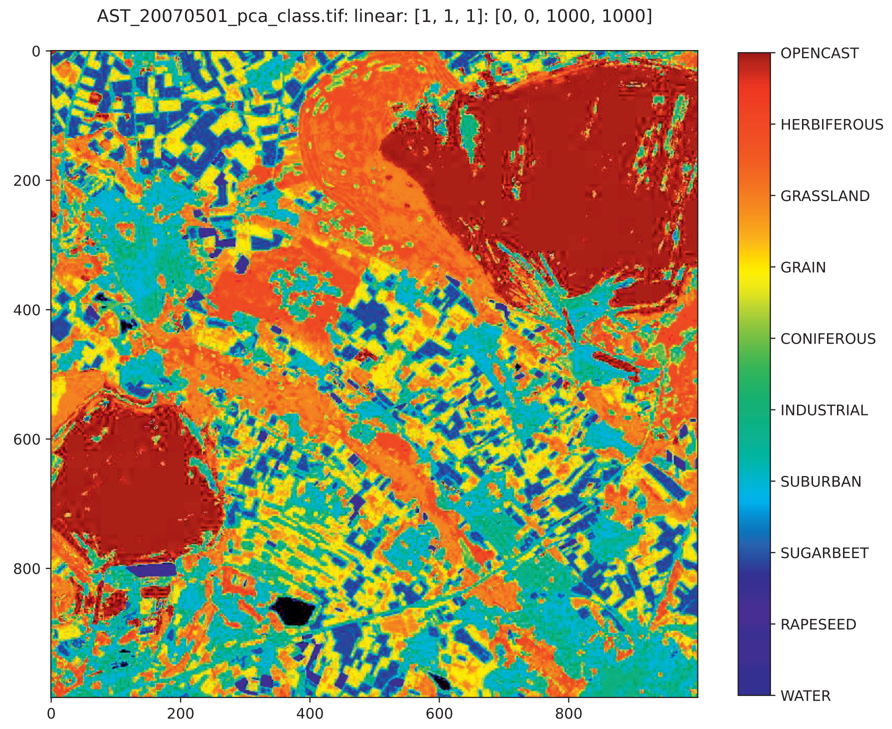


FIGURE 6.6
Gaussian kernel supervised classification of the first five principal components of the image in Figure 6.1.

to its optimum value `self._sigma_min` and where `Hs` is a batch of image pixel vectors in data matrix format. As with the maximum-likelihood classifier, the labeled observations are split into training and test pixels with the latter held back for later accuracy evaluation. To run the algorithm, we set the flag `-a` to 2 and also the `-P` flag to request output of the class probability image:

```
run scripts/classify -p [1,2,3,4,5] -a 2 -P \
 imageryAST_20070501_pca.tif imagery/train.shp

Training with Gausskernel
reading training data...
training on 4774 pixel vectors...
classes: ['WASSER [BL', ... 'LAUBWALD [', 'TAGEBAU [W']
sigma: 0.001000  error: 0.232928
sigma: 0.100000  error: 0.076246
...
sigma: 0.024095  error: 0.039589
```

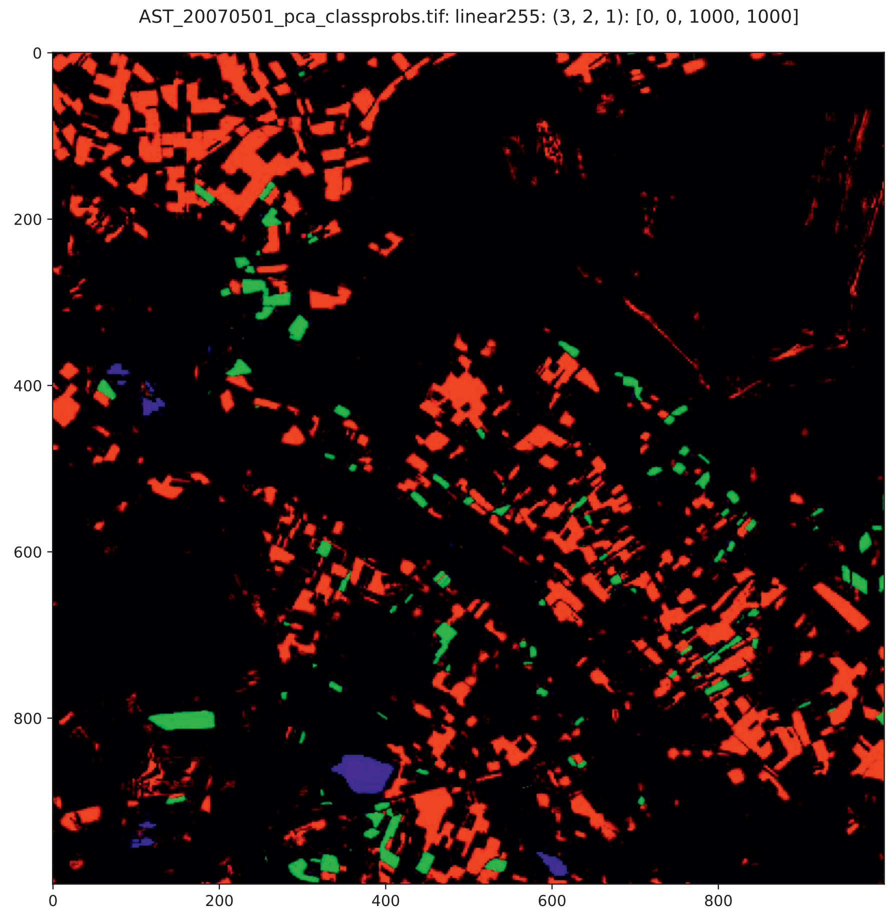

AST_20070501_pca_classprobs.tif: linear255: (3, 2, 1): [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800

FIGURE 6.7
Gaussian kernel probabilities image for water (blue, class 1) , rapeseed (green, class 2), and sugar beet (red, class 3).

```
sigma: 0.024136  error: 0.039799
sigma: 0.024071  error: 0.039589
elapsed time 36.6559169292
classifying...
row: 0
row: 100
...
row: 900
elapsed time 361.4562931060791
class probabilities written to:
 imagery/AST_20070501_pca_classprobs.tif
```

```
thematic map written to:
 imagery/AST_20070501_pca_class.tif
test results written to:
 imagery/AST_20070501_pca_Gausskernel.tst
```

The result is shown in Figure 6.6, where we see a considerable qualitative improvement over both the Naive and complete Bayes maximum-likelihood classifiers. An example of the probabilities image output from the algorithm is given in Figure 6.7. The Gaussian kernel classifier, like most other non-parametric methods, suffers from the drawback of requiring that all training data points be used in the generalization phase (a so-called *memory-based* classifier). Evaluation is very slow if the number of training points is large, which, on the other hand, should be the case if a reasonable approximation to the class-specific densities is to be achieved. The object class `Gausskernel` is in fact unacceptably slow for training datasets exceeding a few thousand pixels. Moreover, the number of training samples needed for a good approximation of the class probability densities grows exponentially with the dimensionality N of the data—the so-called *curse of dimensionality* (Bellman, 1961). Quite generally, the complexity of the calculation is determined by the amount of training data, not by the difficulty of the classification problem itself—an undesirable state of affairs.

6.5 Neural networks

Neural networks belong to the category of semi parametric models for probability density estimation, a category which lies somewhere between the parametric and non-parametric extremes (Bishop, 1995). They make no strong assumptions about the form of the probability distributions and can be adjusted flexibly to the complexity of the system that they are being used to model. They therefore provide an attractive compromise.

To motivate their use for classification, let us consider two classes $k = 1$ and $k = 2$ for two-dimensional observations $\boldsymbol{g} = (g_1, g_2)^\top$. We can write the maximum likelihood decision rule, Equation (6.18) with Equation (6.17), in terms of a new discriminant function

$$I(\boldsymbol{g}) = d_1(\boldsymbol{g}) - d_2(\boldsymbol{g})$$

and say that

$$\boldsymbol{g} \quad \text{is class} \begin{cases} 1 & \text{if } I(\boldsymbol{g}) \ge 0 \\ 2 & \text{if } I(\boldsymbol{g}) < 0. \end{cases}$$

The discriminant $I(\boldsymbol{g})$ is a rather complicated quadratic function of $\boldsymbol{g}$. The *simplest* discriminant that could conceivably decide between the two classes

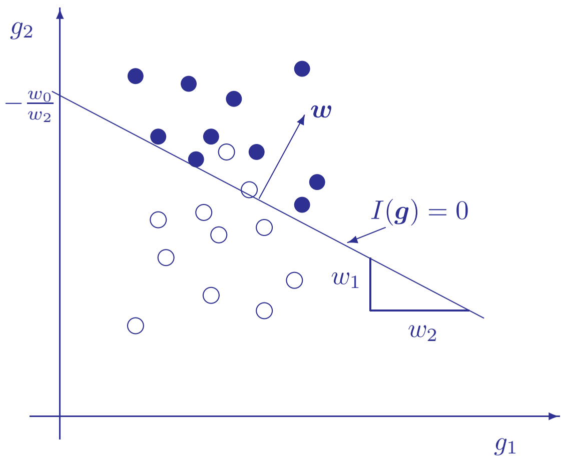


FIGURE 6.8
A linear discriminant for two classes. The vector $\boldsymbol{w} = (w_1, w_2)^\top$ is normal to the separating line in the direction of class $k = 1$, shown as black dots.

is a linear function of the form[*]

$$I(\boldsymbol{g}) = w_0 + w_1 g_1 + w_2 g_2, \tag{6.23}$$

where w_0, w_1 and w_2 are parameters. The decision boundary occurs for $I(\boldsymbol{g}) = 0$, i.e., for

$$g_2 = -\frac{w_1}{w_2} g_1 - \frac{w_0}{w_2},$$

as depicted in Figure 6.8

Extending discussion now to N-dimensional observations, we can work with the discriminant

$$I(\boldsymbol{g}) = w_0 + w_1 g_1 + \ldots + w_N g_N = \boldsymbol{w}^\top \boldsymbol{g} + w_0. \tag{6.24}$$

In this higher dimensional feature space, the decision boundary $I(\boldsymbol{g}) = 0$ generalizes to an *oriented hyperplane*. Equation (6.24) can be represented schematically as an *artificial neuron* or *perceptron*, as shown Figure 6.9, along with some additional jargon. Thus the "input signals" $g_1 \ldots g_N$ are multiplied with "synaptic weights" $w_1 \ldots w_N$ and the results are summed in a "neuron" to produce the "output signal" $I(\boldsymbol{g})$. The w_0 term is treated by introducing a "bias" input of unity, which is multiplied by w_0 and included in the summation.

[*]A linear decision boundary will arise in a maximum-likelihood classifier if the covariance matrices for the two classes are identical, see Exercise 2.

In keeping with the biological analogy, the output $I(\boldsymbol{g})$ may be modified by a so-called *sigmoid* (= S-shaped) "activation function," for example by the *logistic* function

$$f(\boldsymbol{g}) = \frac{1}{1+e^{-I(\boldsymbol{g})}},$$

which is often used for example to model population growth under limited resources.

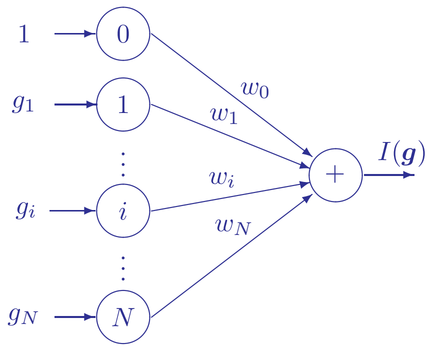


FIGURE 6.9
An artificial neuron representing Equation (6.24). The first input is always unity and is called the bias.

$I(\boldsymbol{g})$ is then referred to as the *activation* of the neuron inducing the output signal $f(\boldsymbol{g})$. This modification of the discriminant has the advantage that the output signal saturates at values zero or one for large negative or positive inputs, respectively.* However, Bishop (1995) suggests that there is also a good statistical justification for using it. Suppose the two classes are normally distributed with $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \boldsymbol{I}$. Then

$$p(\boldsymbol{g} \mid k) = \frac{1}{2\pi}\exp(\frac{-\|\boldsymbol{g}-\boldsymbol{\mu}_k\|^2}{2}),$$

for $k = 1, 2$, and we have with Bayes' Theorem,

$$\begin{aligned}
\Pr(1 \mid \boldsymbol{g}) &= \frac{p(\boldsymbol{g} \mid 1)\Pr(1)}{p(\boldsymbol{g} \mid 1)\Pr(1) + p(\boldsymbol{g} \mid 2)\Pr(2)} \\
&= \frac{1}{1 + p(\boldsymbol{g} \mid 2)\Pr(2)/(p(\boldsymbol{g} \mid 1)\Pr(1))} \\
&= \frac{1}{1 + \exp(-\frac{1}{2}[\|\boldsymbol{g}-\boldsymbol{\mu}_2\|^2 - \|\boldsymbol{g}-\boldsymbol{\mu}_1\|^2])(\Pr(2)/\Pr(1))}.
\end{aligned}$$

With the substitution

$$e^{-a} = \Pr(2)/\Pr(1)$$

we get

$$\begin{aligned}
\Pr(1 \mid \boldsymbol{g}) &= \frac{1}{1 + \exp(-\frac{1}{2}[\|\boldsymbol{g}-\boldsymbol{\mu}_2\|^2 - \|\boldsymbol{g}-\boldsymbol{\mu}_1\|^2] - a)} \\
&= \frac{1}{1 + \exp(-\boldsymbol{w}^\top \boldsymbol{g} - w_0)} \\
&= \frac{1}{1+e^{-I(\boldsymbol{g})}} = f(\boldsymbol{g}).
\end{aligned}$$

*For so-called *deep learning neural networks*, this can sometimes be a disadvantage and alternatives such as the regularized linear unit (ReLU) are preferred. Moreover, synaptic weight initialization becomes critical; see Géron (2023) and Section 6.5.4.

In the second equality above, we have made the additional substitutions

$$\begin{aligned}\boldsymbol{w} &= \boldsymbol{\mu}_1 - \boldsymbol{\mu}_2 \\ w_0 &= -\frac{1}{2}\|\boldsymbol{\mu}_1\|^2 + \frac{1}{2}\|\boldsymbol{\mu}_2\|^2 + a.\end{aligned}$$

Thus we expect that the output signal $f(\boldsymbol{g})$ of the neuron will not only discriminate between the two classes, *but also that it will approximate the posterior class membership probability* $\Pr(1 \mid \boldsymbol{g})$.

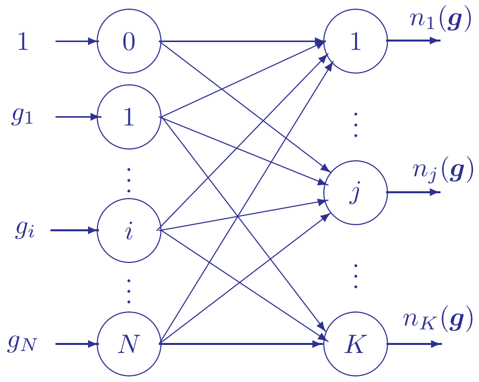


FIGURE 6.10
A single-layer neural network.

The extension of linear discriminants from two to K classes is straightforward, and leads to the *single-layer neural network* of Figure 6.10. There are K neurons (the circles on the right), each of which calculates its own discriminant

$$n_j(\boldsymbol{g}) = f(I_j(\boldsymbol{g})),\ j = 1\ldots K.$$

The observation $\boldsymbol{g}$ is assigned to the class whose neuron produces the maximum output signal, i.e.,

$$k = \arg\max_j n_j(\boldsymbol{g}).$$

Each neuron is associated with a synaptic weight vector $\boldsymbol{w}_j$, which we from now on will understand to include the bias weight w_0. Thus, for the jth neuron,

$$\boldsymbol{w}_j = (w_{0j}, w_{1j}\ldots w_{Nj})^\top,$$

and, for the whole network,

$$\boldsymbol{W} = (\boldsymbol{w}_1, \boldsymbol{w}_2 \ldots \boldsymbol{w}_K) = \begin{pmatrix} w_{01} & w_{02} & \cdots & w_{0K} \\ w_{11} & w_{12} & \cdots & w_{1K} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N1} & w_{N2} & \cdots & w_{NK} \end{pmatrix}, \tag{6.25}$$

which we shall call the *synaptic weight matrix* for the neuron layer.

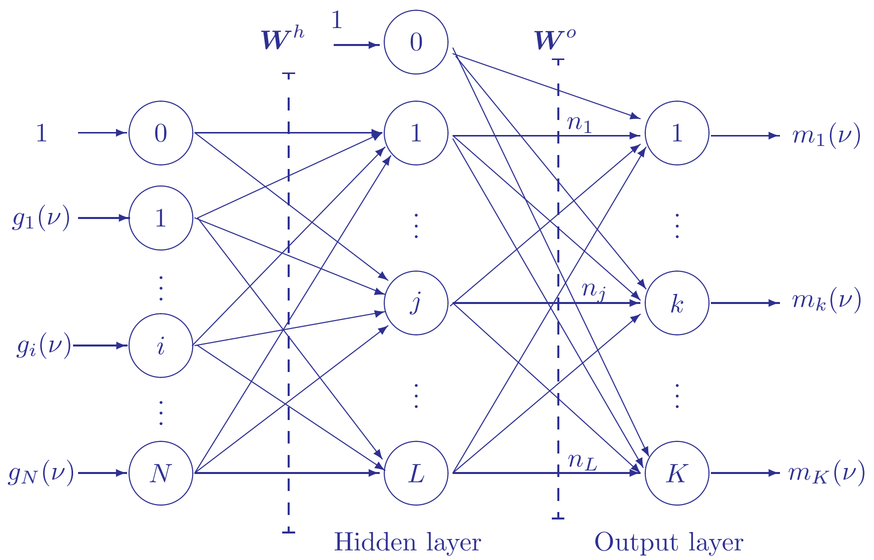


FIGURE 6.11
A two-layer feed-forward neural network with L hidden neurons for classification of N-dimensional data into K classes. The argument ν identifies a training example.

6.5.1 The neural network classifier

Single-layer networks turn out to be rather limited in the kind of classification tasks that they can handle. In fact, only so-called *linearly separable* problems, in which classes of training observations can be separated by hyperplanes, are fully solvable; see Exercise 3. On the other hand, networks with just one additional layer of processing neurons can approximate any given decision boundary arbitrarily closely (Bishop, 1995; Müller et al., 2001) provided that the first, or *hidden*, layer outputs are nonlinear. The input data are transformed by the hidden layer into a higher dimensional function space in which the problem becomes linearly separable. This is also the strategy used for designing *support vector machines* (Belousov et al., 2002), as we will see later.

Accordingly, we shall develop a classifier based on the two-layer, feed-forward architecture* shown in Figure 6.11. For the νth training pixel, the

*The adjective *feed-forward* merely serves to differentiate this network structure from other networks having *feedback* connections.

input to the network is the $(N+1)$-component (biased) observation vector

$$\boldsymbol{g}(\nu) = (1, g_1(\nu) \dots g_N(\nu))^\top.$$

This input is distributed simultaneously to all of the L neurons in the hidden layer of neurons. These in turn determine an $(L+1)$-component vector of intermediate outputs (adding in the bias input for the next layer)

$$\boldsymbol{n}(\nu) = (1, n_1(\nu) \dots n_L(\nu))^\top$$

in which $n_j(\nu)$ is shorthand for

$$f(I_j^h(\boldsymbol{g}(\nu))), \quad j = 1 \dots L.$$

In this expression, the activation I_j^h of the hidden neurons is given by

$$I_j^h(\boldsymbol{g}(\nu)) = \boldsymbol{w}_j^{h\top} \boldsymbol{g}(\nu),$$

where the vector $\boldsymbol{w}_j^h$ is the weight vector for the jth neuron in the hidden layer,

$$\boldsymbol{w}_j^h = (w_{0j}^h, w_{1j}^h \dots w_{Nj}^h)^\top.$$

In terms of a hidden weight matrix $\boldsymbol{W}^h$ having the form of Equation (6.25), namely

$$\boldsymbol{W}^h = (\boldsymbol{w}_1^h, \boldsymbol{w}_2^h \dots \boldsymbol{w}_L^h),$$

we can write all of this more compactly in vector notation as

$$\boldsymbol{n}(\nu) = \begin{pmatrix} 1 \\ f(\boldsymbol{W}^{h\top} \boldsymbol{g}(\nu)) \end{pmatrix}. \tag{6.26}$$

Here we just have to interpret the logistic function of a vector $\boldsymbol{v}$, namely $f(\boldsymbol{v})$, as a vector of logistic functions of the components of $\boldsymbol{v}$.

The vector $\boldsymbol{n}(\nu)$ is then fed in the same manner to the *output layer* with its associated output weight matrix

$$\boldsymbol{W}^o = (\boldsymbol{w}_1^o, \boldsymbol{w}_2^o \dots \boldsymbol{w}_K^o),$$

and the output signal $\boldsymbol{m}(\nu)$ is calculated, as in Equation (6.26), as

$$\boldsymbol{m}(\nu) = f(\boldsymbol{W}^{o\top} \boldsymbol{n}(\nu)). \tag{6.27}$$

However, this last equation is not quite satisfactory. According to our previous considerations, we would like to interpret the network outputs as class membership probabilities. That means that we must ensure that

$$0 \le m_k(\nu) \le 1, \quad k = 1 \dots K,$$

Listing 6.3: A feed-forward neural network class (excerpt from the Python module `auxil.supervisedclass.py`).

```
class Ffn(object):
    '''Base Class for Neural Net Classifiers'''
    def __init__(self,Gs,ls,Ls,epochs,validate):
    #   setup the network architecture
        self._L = Ls[0]
        self._m,self._N = Gs.shape
        self._K = ls.shape[1]
        self._epochs = epochs
    #   biased input as column vectors
        Gs = np.mat(Gs).T
        self._Gs = np.vstack((np.ones(self._m),Gs))
    #   biased output vector from hidden layer
        self._n = np.mat(np.zeros(self._L+1))
    #   labels as column vectors
        self._ls = np.mat(ls).T
        if validate:
    #       split into train and validate sets
            self._m = self._m//2
            self._Gsv = self._Gs[:,self._m:]
            self._Gs = self._Gs[:,:self._m]
            self._lsv = self._ls[:,self._m:]
            self._ls = self._ls[:,:self._m]
        else:
            self._Gsv = self._Gs
            self._lsv = self._ls
    #   weight matrices
        self._Wh=np.mat(np.random. \
                          random((self._N+1,self._L)))-0.5
        self._Wo=np.mat(np.random. \
                          random((self._L+1,self._K)))-0.5

    def forwardpass(self,G):
    #   forward pass through the network
        expnt = self._Wh.T*G
        self._n = np.vstack((np.ones(1),1.0/ \
                                    (1+np.exp(-expnt))))
    #   softmax activation
        I = self._Wo.T*self._n
        A = np.exp(I-max(I))
        return A/np.sum(A)
```

and, furthermore, that

$$\sum_{k=1}^{K} m_k(\nu) = 1.$$

The logistic function f satisfies the first condition, but there is no reason why the second condition should be met. It can be enforced, however, by using a modified logistic activation function for the output neurons, called *softmax* (Bridle, 1990). The softmax function is defined as

$$m_k(\nu) = \frac{e^{I_k^o(\boldsymbol{n}(\nu))}}{e^{I_1^o(\boldsymbol{n}(\nu))} + e^{I_2^o(\boldsymbol{n}(\nu))} + \ldots + e^{I_K^o(\boldsymbol{n}(\nu))}}, \tag{6.28}$$

where

$$I_k^o(\boldsymbol{n}(\nu)) = \boldsymbol{w}_k^{o\top}\boldsymbol{n}(\nu), \quad k = 1 \ldots K, \tag{6.29}$$

and clearly guarantees that the output signals sum to unity.

The Equations (6.26), (6.28), and (6.29) now provide a complete mathematical representation of the neural network classifier shown in Figure 6.10. It turns out to be a very useful classifier indeed. To quote Bishop (1995):

> ... [two-layer, feed-forward] networks can approximate arbitrarily well any functional continuous mapping from one finite dimensional space to another, provided the number [L] of hidden units is sufficiently large. ... An important corollary of this result is, that in the context of a classification problem, networks with sigmoidal non-linearities and two layers of weights can approximate any decision boundary to arbitrary accuracy. ... More generally, the capability of such networks to approximate general smooth functions allows them to model posterior probabilities of class membership.

The Python object class `Ffn`, an excerpt of which is given in Listing 6.3, mirrors the network architecture of Figure 6.10. It will form the basis for the implementation of the back propagation training algorithm developed below and also for the more efficient training algorithms described in Appendix B.

6.5.2 Cost functions

We have not yet considered the correct choice of synaptic weights, that is, how to go about training the neural network classifier. As mentioned in Section 6.3.2, the training data are most conveniently represented as the set of labeled pairs

$$\mathcal{T} = \{(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu)) \mid \nu = 1 \ldots m\},$$

where the label

$$\boldsymbol{\ell}(\nu) = (0 \ldots 0, 1, 0 \ldots 0)^\top$$

is a K-dimensional column vector of zeroes, except with the "1" at the kth position to indicate that $\boldsymbol{g}(\nu)$ belongs to class k.

Under certain assumptions about the distribution of the training data, the *quadratic cost function*

$$E(\boldsymbol{W}^h, \boldsymbol{W}^o) = \frac{1}{2}\sum_{\nu=1}^{m} \|\boldsymbol{\ell}(\nu) - \boldsymbol{m}(\nu)\|^2 \tag{6.30}$$

can be justified as a training criterion for feed-forward networks (Exercise 4). The network weights $\boldsymbol{W}^h$ and $\boldsymbol{W}^o$ must be adjusted so as to minimize E. This minimization will clearly tend to make the network produce the output signal

$$\boldsymbol{m} = (1, 0 \dots 0 \dots 0)^\top$$

whenever it is presented with a training observation $\boldsymbol{g}(\nu)$ from class $k = 1$, and similarly for the other classes.

This, of course, is what we wish it to do. However, a more appropriate cost function for classification problems can be obtained with a probabilistic argument, the maximum-likelihood criterion: Choose the synaptic weights so as to *maximize the probability of observing the training data.* The joint probability for observing the training example $(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu))$ is

$$\Pr(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu)) = \Pr(\boldsymbol{\ell}(\nu) \mid \boldsymbol{g}(\nu))\Pr(\boldsymbol{g}(\nu)), \tag{6.31}$$

where we have used Equation (2.63). The neural network, as was argued, approximates the posterior class membership probability $\Pr(\boldsymbol{\ell}(\nu) \mid \boldsymbol{g}(\nu))$. In fact, this probability can be expressed directly in terms of the network output signal in the form

$$\Pr(\boldsymbol{\ell}(\nu) \mid \boldsymbol{g}(\nu)) = \prod_{k=1}^{K} [\ m_k(\boldsymbol{g}(\nu))\ ]^{\ell_k(\nu)}. \tag{6.32}$$

In order to see this, consider the case $\boldsymbol{\ell} = (1, 0 \dots 0)^\top$. Then, according to Equation (6.32),

$$\Pr((1, 0 \dots 0)^\top \mid \boldsymbol{g}) = m_1(\boldsymbol{g})^1 \cdot m_2(\boldsymbol{g})^0 \cdots m_K(\boldsymbol{g})^0 = m_1(\boldsymbol{g}),$$

which is the probability that $\boldsymbol{g}$ is in class 1, as desired. Now, substituting Equation (6.32) into Equation (6.31), we therefore wish to maximize

$$\Pr(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu)) = \prod_{k=1}^{K}\ [\ m_k(\boldsymbol{g}(\nu))\ ]^{\ell_k(\nu)}\Pr(\boldsymbol{g}(\nu)).$$

Taking logarithms, dropping terms which are independent of the synaptic weights, summing over all of the training data and changing the sign, we see that this is equivalent to minimizing the *categorical cross-entropy* cost function

$$E(\boldsymbol{W}^h, \boldsymbol{W}^o) = -\sum_{\nu=1}^{m}\sum_{k=1}^{K} \ell_k(\nu)\log[m_k(\boldsymbol{g}(\nu))] \tag{6.33}$$

with respect to the synaptic weight parameters; compare with Equation (2.113).

6.5.3 Back propagation

A minimum of the cost function, Equation (6.33), can be found with various search algorithms, whereby *back propagation* is certainly the most well-known and extensively used method. Modern neural network libraries like TensorFlow/Keras or PyTorch implement back propagation very efficiently with so-called *automatic differentiation*, as well as by taking advantage of parallel architectures where available, e.g., GPUs. When integrating these libraries into an application, they tend to have a bit of a "black box" feel to them. For this reason it may be worthwhile to work through the mathematics of gradient descent back propagation with a single hidden layer in detail. In doing so, we will also make reference to *scaled conjugate gradient* and *Kalman filter* training as efficient alternatives to gradient descent methods, but reserve their derivation to an Appendix.* In the next section, we'll turn our attention to automatic methods with TensorFlow and Keras.

Our starting point is the so-called *local version* of the cross entropy, Equation (6.33):

$$E(\boldsymbol{W}^h, \boldsymbol{W}^o, \nu) = -\sum_{k=1}^{K} \ell_k(\nu) \log[m_k(\boldsymbol{g}(\nu))], \quad \nu = 1 \dots m.$$

This is just the cost function for a single training example. If we manage to make it smaller at each step of the calculation and cycle, either sequentially or randomly, through the available training pairs, then we are obviously minimizing the overall cost function as well. With the abbreviation $m_k(\nu) = m_k(\boldsymbol{g}(\nu))$, the local cost function can be written a little more compactly as

$$E(\nu) = -\sum_{k=1}^{K} \ell_k(\nu) \log[m_k(\nu)].$$

Here the dependence of the cost function on the synaptic weights is also implicit. More compactly still, it can be represented in vector form as an inner product:

$$E(\nu) = -\boldsymbol{\ell}(\nu)^\top \log[\boldsymbol{m}(\nu)]. \tag{6.34}$$

Our problem then is to minimize Equation (6.34) with respect to the synaptic weights, which are the $(N+1) \times L$ elements of the matrix $\boldsymbol{W}^h$ and the $(L+1) \times K$ elements of $\boldsymbol{W}^o$. Let us consider the following algorithm:

Algorithm (Stochastic gradient descent (SGD) or back propagation)

1. Initialize the synaptic weights with, e.g., normally distributed random numbers and set ν equal to a random integer in the interval $[1, m]$.

*Python scripts for supervised classification with a feed-forward neural network trained with any of these analytical algorithms are provided with the software.

2. Choose training pair $(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu))$ and determine the output response $\boldsymbol{m}(\nu)$ of the network.

3. For $k = 1 \dots K$ and $j = 0 \dots L$, replace w^o_{jk} with $w^o_{jk} - \eta \frac{\partial E(\nu)}{\partial w^o_{jk}}$.

4. For $j = 1 \dots L$ and $i = 0 \dots N$, replace w^h_{ij} with $w^h_{ij} - \eta \frac{\partial E(\nu)}{\partial w^h_{ij}}$.

5. If $\sum_\nu E(\nu)$ ceases to change significantly, stop, otherwise set ν equal to a new random integer in $[1, m]$ and go to step 2.

The algorithm jumps randomly through the training data, reducing the local cost function at each step. The reduction is accomplished by changing each synaptic weight w by an amount proportional to the negative slope $-\partial E(\nu)/\partial w$ of the local cost function with respect to that weight parameter, stopping when the overall cost function, Equation (6.33), can no longer be reduced. The constant of proportionality η is referred to as the *learning rate* for the network. This algorithm only makes use of the first derivatives of the cost function with respect to the synaptic weight parameters and belongs to the class of *gradient descent* methods.

To implement the algorithm, the partial derivatives of $E(\nu)$ with respect to the synaptic weights are required. Let us begin with the output neurons, which generate the softmax output signals

$$m_k(\nu) = \frac{e^{I^o_k(\nu)}}{e^{I^o_1(\nu)} + e^{I^o_2(\nu)} + \dots + e^{I^o_K(\nu)}}, \tag{6.35}$$

where

$$I^o_k(\nu) = \boldsymbol{w}^{o\top}_k \boldsymbol{n}(\nu).$$

We wish to determine (step 3 of the back propagation algorithm)

$$\frac{\partial E(\nu)}{\partial w^o_{jk}}, \quad j = 0 \dots L,\ k = 1 \dots K.$$

Recalling the rules for vector differentiation in Chapter 1 and applying the chain rule we get

$$\frac{\partial E(\nu)}{\partial \boldsymbol{w}^o_k} = \frac{\partial E(\nu)}{\partial I^o_k(\nu)} \frac{\partial I^o_k(\nu)}{\partial \boldsymbol{w}^o_k} = -\delta^o_k(\nu)\boldsymbol{n}(\nu), \quad k = 1 \dots K, \tag{6.36}$$

where we have introduced the quantity $\delta^o_k(\nu)$ given by

$$\delta^o_k(\nu) = -\frac{\partial E(\nu)}{\partial I^o_k(\nu)}. \tag{6.37}$$

This is the negative rate of change of the local cost function with respect to the activation of the kth output neuron.

Again, applying the chain rule and using Equations (6.34) and (6.35),

$$
\begin{aligned}
-\delta_k^o(\nu) = \frac{\partial E(\nu)}{\partial I_k^o(\nu)} &= \sum_{k'=1}^{K} \frac{\partial E(\nu)}{\partial m_{k'}(\nu)} \frac{\partial m_{k'}(\nu)}{\partial I_k^o(\nu)} \\
&= \sum_{k'=1}^{K} -\frac{\ell_{k'}(\nu)}{m_{k'}(\nu)} \left(\frac{e^{I_k^o(\nu)}\delta_{kk'}}{\sum_{k''=1}^{K} e^{I_{k''}^o(\nu)}} - \frac{e^{I_{k'}^o(\nu)} e^{I_k^o(\nu)}}{(\sum_{k''=1}^{K} e^{I_{k''}^o(\nu)})^2} \right).
\end{aligned}
$$

Here, $\delta_{kk'}$ is given by

$$
\delta_{kk'} = \begin{cases} 0 & \text{if } k \neq k' \\ 1 & \text{if } k = k'. \end{cases}
$$

Continuing, making use of Equation (6.35),

$$
\begin{aligned}
-\delta_k^o(\nu) &= \sum_{k'=1}^{K} -\frac{\ell_{k'}(\nu)}{m_{k'}(\nu)} m_k(\nu)(\delta_{kk'} - m_{k'}(\nu)) \\
&= -\ell_k(\nu) + m_k(\nu) \sum_{k'=1}^{K} \ell_{k'}(\nu).
\end{aligned}
$$

But this last sum over the K components of the label $\boldsymbol{\ell}(\nu)$ is just unity, and therefore we have

$$
-\delta_k^o(\nu) = -\ell_k(\nu) + m_k(\nu), \quad k = 1 \ldots K,
$$

which may be written as the K-component vector

$$
\boldsymbol{\delta}^o(\nu) = \boldsymbol{\ell}(\nu) - \boldsymbol{m}(\nu). \tag{6.38}
$$

From Equation (6.36), we can therefore express the third step in the back propagation algorithm in the form of the matrix equation (see Exercise 6)

$$
\boldsymbol{W}^o(\nu+1) = \boldsymbol{W}^o(\nu) + \eta\, \boldsymbol{n}(\nu)\boldsymbol{\delta}^o(\nu)^\top. \tag{6.39}
$$

Here $\boldsymbol{W}^o(\nu+1)$ indicates the synaptic weight matrix *after* the update for νth training pair. Note that the second term on the right-hand side of Equation (6.39) is an outer product, yielding a matrix of dimension $(L+1) \times K$ and so matching the dimension of $\boldsymbol{W}^o(\nu)$.

For the hidden weights, step 4 of the algorithm, we proceed similarly:

$$
\frac{\partial E(\nu)}{\partial \boldsymbol{w}_j^h} = \frac{\partial E(\nu)}{\partial I_j^h(\nu)} \frac{\partial I_j^h(\nu)}{\partial \boldsymbol{w}_j^h} = -\delta_j^h(\nu)\boldsymbol{g}(\nu), \quad j = 1 \ldots L, \tag{6.40}
$$

where $\delta_j^h(\nu)$ is the negative rate of change of the local cost function with respect to the activation of the jth hidden neuron:

$$
\delta_j^h(\nu) = -\frac{\partial E(\nu)}{\partial I_j^h(\nu)}.
$$

Applying the chain rule again:

$$\begin{aligned}
-\delta^h_j(\nu) &= \sum_{k=1}^{K}\frac{\partial E(\nu)}{\partial I^0_k(\nu)}\frac{\partial I^o_k(\nu)}{\partial I^h_j(\nu)} = -\sum_{k=1}^{K}\delta^o_k(\nu)\frac{\partial I^o_k(\nu)}{\partial I^h_j(\nu)}\\
&= -\sum_{k=1}^{K}\delta^o_k(\nu)\frac{\partial \boldsymbol{w}^{o\top}_k\boldsymbol{n}(\nu)}{\partial I^h_j(\nu)} = -\sum_{k=1}^{K}\delta^o_k(\nu)\boldsymbol{w}^{o\top}_k\frac{\partial \boldsymbol{n}(\nu)}{\partial I^h_j(\nu)}.
\end{aligned}$$

In the last partial derivative, since $I^h_j(\nu) = \boldsymbol{w}^{h\top}_j\boldsymbol{g}(\nu)$, only the output of the jth hidden neuron is a function of $I^h_j(\nu)$. Therefore,

$$\delta^h_j(\nu) = \sum_{k=1}^{K}\delta^o_k(\nu)w^o_{jk}\frac{\partial n_j(\nu)}{\partial I^h_j(\nu)}. \tag{6.41}$$

Recall that the hidden units in our simple network use the logistic activation function

$$n_j(I^h_j) = f(I^h_j) = \frac{1}{1+e^{-I^h_j}}.$$

This function has the derivative:

$$\frac{\partial n_j(x)}{\partial x} = n_j(x)(1-n_j(x)).$$

Therefore, we can write Equation (6.41) as

$$\delta^h_j(\nu) = \sum_{k=1}^{K}\delta^o_k(\nu)w^o_{jk}n_j(\nu)(1-n_j(\nu)), \quad j = 1\dots L,$$

or more compactly as the matrix equation

$$\begin{pmatrix}0\\ \boldsymbol{\delta}^h(\nu)\end{pmatrix} = \boldsymbol{n}(\nu)\cdot(\boldsymbol{1}-\boldsymbol{n}(\nu))\cdot\big(\boldsymbol{W}^o\boldsymbol{\delta}^o(\nu)\big). \tag{6.42}$$

The dot is intended to denote simple component-by-component (so-called *Hadamard*) multiplication. The equation must be written in this rather awkward way because the expression on the right-hand side has $L+1$ components. This also makes the fact that $1 - n_0(\nu) = 0$ explicit. Equation (6.42) is the origin of the term "back propagation," since it propagates the negative rate of change of the cost function with respect to the output activations $\boldsymbol{\delta}^o(\nu)$ backwards through the network to determine the negative rate of change with respect to the hidden activations $\boldsymbol{\delta}^h(\nu)$.

Finally, with Equation (6.40), we obtain the update rule for step 4 of the back propagation algorithm:

$$\boldsymbol{W}^h(\nu+1) = \boldsymbol{W}^h(\nu) + \eta\,\boldsymbol{g}(\nu)\boldsymbol{\delta}^h(\nu)^\top. \tag{6.43}$$

Listing 6.4: A class for a feed-forward neural network trained with back propagation (excerpt from the Python module `auxil.supervisedclass.py`).

```
class Ffnbp(Ffn):
    '''Ordinary Backpropagation Neural Net Classifier'''
    def __init__(self,Gs,ls,Ls,epochs=100,valid=False):
        Ffn.__init__(self,Gs,ls,Ls,epochs,valid)

    def train(self):
        eta = 0.01
        alpha = 0.9
        maxitr = self._epochs*self._m
        inc_o1 = 0.0
        inc_h1 = 0.0
        epoch = 0
        cost = []
        costv = []
        itr = 0
        try:
            while itr<maxitr:
                # select train example pair at random
                nu = np.random.randint(0,self._m)
                x = self._Gs[:,nu]
                ell = self._ls[:,nu]
                # send it through the network
                m = self.forwardpass(x)
                # determine the deltas
                d_o = ell - m
                d_h = np.multiply(np.multiply(self._n,\
                     (1-self._n)),(self._Wo*d_o))[1::]
                # update synaptic weights
                inc_o = eta*(self._n*d_o.T)
                inc_h = eta*(x*d_h.T)
                self._Wo += inc_o + alpha*inc_o1
                self._Wh += inc_h + alpha*inc_h1
                inc_o1 = inc_o
                inc_h1 = inc_h
                # record cost function
                if itr % self._m == 0:
                    cost.append(self.cost())
                    costv.append(self.costv())
                    epoch += 1
                itr += 1
        except Exception as e:
            print ('Error: %s'%e)
            return None
        return (np.array(cost),np.array(costv))
```

The choice of an appropriate learning rate η is problematic: small values imply slow convergence and large values produce oscillation. Some improvement can be achieved with an additional, purely heuristic parameter called *momentum*, which maintains a portion of the preceding weight increments in the current iteration. Equation (6.39) is replaced with

$$\boldsymbol{W}^o(\nu+1) = \boldsymbol{W}^o(\nu) + \Delta^o(\nu) + \alpha\Delta^o(\nu-1), \tag{6.44}$$

where $\Delta^o(\nu) = \eta\, \boldsymbol{n}(\nu)\boldsymbol{\delta}^{o\top}(\nu)$ and α is the momentum parameter. A similar expression replaces Equation (6.43). In Exercise 8 the reader is asked to show that, in extended regions of constant gradient in the cost function $\Delta^o(\nu+1) = \Delta^o(\nu)$ the momentum will increase the rate of convergence by the factor $1/(1-\alpha)$. Typical choices for the back propagation parameters are $\eta = 0.01$ and $\alpha = 0.9$.

Listing 6.4 shows part of the object class `Ffnbp(Ffn)` extending the class `Ffn(object)` of Listing 6.3 to implement the SGD back propagation algorithm. It lists the code for the method `train()`, which closely parallels the equations developed above. Figure 6.12 shows the local cross entropy cost function for training with gradient descent for 1000 *epochs*, that is, 1000 passes through the complete training set or $1000 \times 4774 \approx 5 \times 10^6$ training examples.

The disadvantage of our gradient descent back propagation implementation

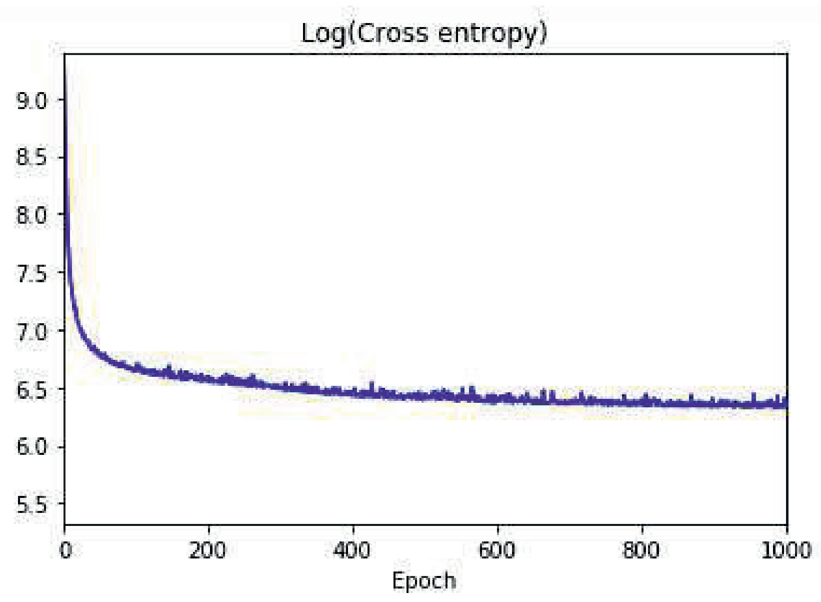


FIGURE 6.12
Local cross entropy as a function of training epoch for gradient descent back propagation.

(minimizing the local cost function Equation (6.34)) relative to the much more efficient scaled conjugate gradient training algorithm described in Appendix B can be seen in a direct comparison. First with the flag `-a` set to 3 for gradient descent back propagation, `-e 1000` for 1000 epochs and `-L [10]` for one hidden layer of 10 neurons:

```
Training with NNet(Backprop)
reading training data...
training on 4798 pixel vectors...
classes: ['WASSER␣[BL', ... 'TAGEBAU␣[W']
elapsed time 491.03439497947693
classifying...
row: 0
row: 100
row: 200
...
```

Whereas with scaled conjugate gradient (`-a 4`):

```
Training with NNet(Congrad)
reading training data...
training on 4798 pixel vectors...
classes: ['WASSER␣[BL', ... 'TAGEBAU␣[W']
elapsed time 5.62642502784729
classifying...
row: 0
row: 100
row: 200
...
```

we achieve an improvement in training time by a factor of about 90. The difference is partly due to the efficiency of the scaled conjugate gradient algorithm itself and also because the algorithm minimizes the global cost function Equation (6.33), rather than the local version we have preferred for demonstration purposes up until now.* The speed-up for Kalman filter training is about a factor of 15, due to its extremely fast convergence (less than 10 epochs, see Appendix B and the accompanying Jupyter notebook).

6.5.4 Deep learning networks

The astounding renaissance of interest in artificial intelligence that we are presently (anno 2025) experiencing is due to the—sometimes spectacular—success of so-called *deep learning neural networks* (DNNs) when applied to very large data sets. In essence one works with networks of varying architectures, all characterized by having many hidden layers and correspondingly

*The TensorFlow/Keras implementation of gradient descent discussed in the next Section also minimize a global cost function.

very many free parameters. Training times, which a decade ago would have been prohibitive, are now becoming acceptable due to modern parallel computing techniques such as GPU processing (CUDA) and dedicated and optimized software like TensorFlow or PyTorch. The remote sensing community has also "joined the bandwagon" so to speak, with a plethora of deep learning applications, both real and envisaged; see Zhu et al. (2017) for an early overview and Section 7.6 in Chapter 7. On the same note, the GEE development team is gradually interfacing DNN functionality with the GEE API.[*]

In Chapter 7, we will look at object classification of remote sensing imagery with *convolutional neural networks*. But in this Chapter, in our relatively modest domain of pixel-based image classification, the deep learning paradigm amounts to increasing the number of neurons and/or hidden layers in the feed forward network of Figure 6.10 in the hope that it can better learn the underlying probability distributions. As may be evident from the mathematics of Section 6.5.3, analytic expressions for propagating the rate of change of the output neuron weights backward to two or more hidden layers would become quite complicated, the more so if we were to use either of the more sophisticated training methods of Appendix B. The TensorFlow platform overcomes this complication by exposing a very clean framework for array propagation[†] taking advantage of the numerical technique of *automatic differentiation*; see Rall (1981) and Géron (2023), Appendix D. This greatly simplifies and accelerates the computations necessary for network training. Moreover, the requirement for a well-behaved activation function can be relaxed in favor of more efficient activations which are not everywhere differentiable. *Keras* is a deep learning API written in Python and capable of running on top of TensorFlow or PyTorch and we will use it here for a very elegant representation of a FFN with any number of hidden layers.

A feed forward neural network classifier with L dense hidden layers is scripted in the object class `Dnn_keras()`, Listing 6.5. We've taken advantage of the high-level class `tf.keras.Sequential()` for multilayer feed forward networks. The code, apart from the activation function chosen for the hidden layers, namely `ReLU` or rectified linear units, is self-explanatory. As for `RelU`, to cite Géron (2023):

> The $ReLU(z)$ function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make gradient descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.

[*]See https://developers.google.com/earth-engine/guides/machine-learning for a more up-to-date overview, including interfacing the GEE with large scale machine learning models hosted on Vertex AI.

[†]The "tensors" in TensorFlow and PyTorch are just multidimensional arrays. The deeper mathematical/physical significance of the term tensor, for instance as a coordinate system independent quantity, plays no role.

Listing 6.5: A class for a deep learning neural network (excerpt from the Python module `supervisedclass.py`).

```
class Dnn_keras(object):
    '''TensorFlow (Keras) Dnn classifier,'''
    def __init__(self,Gs,ls,Ls,epochs=100):
        # setup the network architecture
        self._Gs = Gs
        n_classes = ls.shape[1]
        self._labels = ls
        self._epochs = epochs
        self._dnn = tf.keras.Sequential()
        # hidden layers
        for L in Ls:
            self._dnn \
             .add(layers.Dense(L, 'relu'))
        # output layer
        self._dnn \
            .add(layers.Dense(n_classes, 'softmax'))
        # initialize
        self._dnn.compile(
           optimizer=tf.keras.optimizers.SGD(0.01),
           loss='categorical_crossentropy')

    def train(self):
        try:
            self._dnn.fit(self._Gs,self._labels,
                  epochs=self._epochs,verbose=0)
            return True
        except Exception as e:
            print ('Error:␣%s'%e )
            return None

    def classify(self,Gs):
        # predict new data
        Ms = self._dnn.predict(Gs)
        cls = np.argmax(Ms,1)+1
        return (cls,Ms)
```

We will now instantiate the `Dnn_keras()` class to a three hidden layer FFN and train it on our ASTER image, at the considerable risk of over-fitting; see the next Section. The flag `-a 6` below selects the DNN classifier:

```
run scripts/classify -p [1,2,3,4,5] -a 6 -e 1000 \
     -L [10,10,10] imagery/AST_20070501_pca.tif\
     imagery/train.shp

Training with Dnn(tensorflow)
```

```
reading training data...
training on 4798 pixel vectors...
classes: ['WASSER [BL', ... , 'TAGEBAU [W']
elapsed time 191.424
classifying...
...
elapsed time 19.486
thematic map written to:
               imagery/AST_20070501_pca_class.tif
```

Since there are now three hidden layers, the training time is over 30 times that for the scaled conjugate gradient algorithm. The efficiency would be greatly improved by having parallel computing hardware installed, e.g., GPU processors that TensorFlow can take advantage of. Whether of not the classification accuracy has benefited from the increased depth of the network will be discussed in the next Chapter.

6.5.5 Over fitting and generalization

A fundamental and much-discussed dilemma in the application of neural networks (and other learning algorithms) is that of *over fitting*. Reduced to its essentials, the question is: "How many hidden neurons are enough?" The number of neurons in the output layer of the network in Figure 6.11 is determined by the number of training classes. The number in the hidden layer is fully undetermined. If "too few" hidden neurons are chosen (and thus too few synaptic weights), there is danger that the classification will be suboptimal: there will be an insufficient number of adjustable parameters to resolve the class structure of the training data. If, on the other hand, "too many" hidden neurons are selected, and if the training data have a large noise variance, there will be a danger that the network will fit the data all too well, including their detailed random structure. Such detailed structure is a characteristic of the particular training sample chosen and not of the underlying class distributions that the network is supposed to learn. It is here that one speaks of over fitting. In either case, the capability of the network to generalize to unknown inputs will be impaired. One can find excellent discussions of this subject in Hertz et al. (1991), Chapter 6, and in Bishop (1995), Chapter 9, where regularization techniques are introduced to penalize over fitting. Alternatively, *growth* and *pruning* algorithms can be applied in which the network architecture is optimized during the training procedure. A popular growth algorithm is the *cascade correlation neural network* of Fahlman and LeBiere (1990).

We shall restrict ourselves here to a solution which presupposes an over-dimensioned network, that is, one with too many hidden weights, as well as the availability of a second dataset, which is usually referred to as the *validation* dataset. An option in the Python script `classify.py` documented in Appendix C allows the training data to be split in two, so that half the data are reserved for validation purposes. (If test data were held back, the training

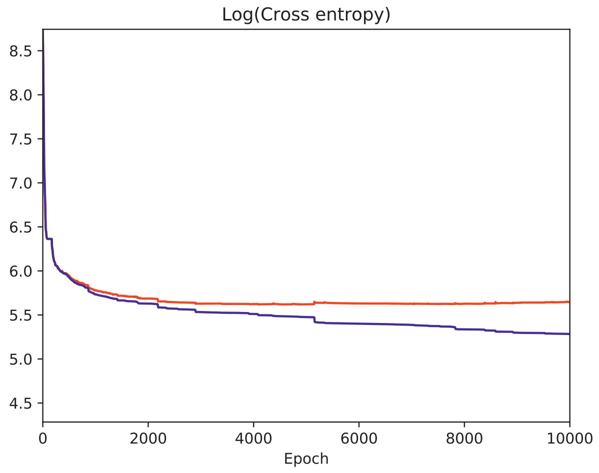


FIGURE 6.13
Training (blue) and validation (red) cost functions for 20 hidden neurons and 10,000 epochs.

dataset is then one third of its original size.) Both halves are still representative of the class distributions and are statistically independent. During the training phase, which is carried out only with the training data, cost functions calculated both with the training data as well as with the validation data are displayed. Over fitting is indicated by a continued decrease in the training cost function accompanied by a gradual increase or stagnation in the validation cost function. Figure 6.13 shows a fairly typical example, for 20 hidden neurons and 10,000 epochs (training time 6 minutes with the scaled conjugate gradient algorithm). The gradual saturation and increase in the validation cost function beginning at around 3000 epochs, together with the continued slow decrease in the training cost, is indicative of over fitting: the network is learning the detailed structure of the training data without improving, or at the cost of, its ability to generalize. The algorithm should thus be stopped when the upper curve ceases to decrease (so-called *early stopping*).

In Chapter 7, we shall see how to compare the generalization capability of neural networks with that of the maximum likelihood and Gaussian kernel classifiers which we developed previously as well as with the support vector machine (the topic of the next section) and the random forest classifiers (Chapter 7).

6.6 Support vector machines

Let us return to the simple linear discriminant function $I(\boldsymbol{g})$ for a two-class problem given by Equation (6.24), with the convention that the weight vector does not include the bias term, i.e.,

$$I(\boldsymbol{g}) = \boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0,$$

where

$$\boldsymbol{w} = (w_1 \dots w_N)^\top$$

and where the training observations are

$$\boldsymbol{g}(\nu) = (g_1(\nu) \dots g_N(\nu))^\top,$$

with corresponding (this time scalar) labels

$$\ell(\nu) \in \{0, 1\}, \quad \nu = 1 \dots m.$$

A quadratic cost function for training this discriminant on two classes would then be

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{\nu=1}^{m}(\boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0 - \ell(\nu))^2. \tag{6.45}$$

This expression is essentially the same as Equation (6.30), when it is written for the case of a single neuron. Training the neuron of Figure 6.9 to discriminate the two classes means finding the weight parameters which minimize the above cost function. This can be done, for example, by using a gradient descent method, or more efficiently with the *perceptron algorithm* as explained in Exercise 3(c). We shall consider in the following an alternative to such a cost function approach. The method we describe is reminiscent of the Gaussian kernel method of Section 6.4 in that the training observations are also used at the classification phase, but as we shall see, not all of them.

6.6.1 Linearly separable classes

It is convenient first of all to relabel the training observations as $\ell(\nu) \in \{-1, 1\}$, rather than $\ell(\nu) \in \{0, 1\}$. With this convention, the product

$$\ell(\nu)(\boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0), \quad \nu = 1 \dots m,$$

is called *margin* of the νth training pair $(\boldsymbol{g}(\nu), \ell(\nu))$ relative to the hyperplane

$$I(\boldsymbol{g}) = \boldsymbol{w}^\top \boldsymbol{g} + w_0 = 0.$$

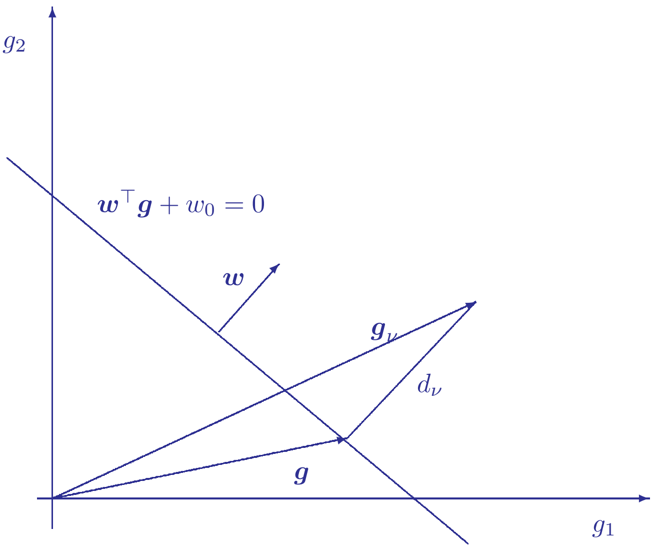


FIGURE 6.14
Distance d to the separating hyperplane.

The perpendicular distance d_ν of a point $\boldsymbol{g}(\nu)$ to the hyperplane is, see Figure 6.14, given by

$$d_\nu = \frac{1}{\|\boldsymbol{w}\|}(\boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0). \tag{6.46}$$

That is, from the figure,

$$d_\nu = \|\boldsymbol{g}(\nu) - \boldsymbol{g}\|,$$

and, since $\boldsymbol{w}$ is perpendicular to the hyperplane,

$$\boldsymbol{w}^\top (\boldsymbol{g}(\nu) - \boldsymbol{g}) = \|\boldsymbol{w}\| \|\boldsymbol{g}(\nu) - \boldsymbol{g}\| = \|\boldsymbol{w}\| d_\nu.$$

But the left-hand side of the above equation is just

$$\boldsymbol{w}^\top (\boldsymbol{g}(\nu) - \boldsymbol{g}) = \boldsymbol{w}^\top \boldsymbol{g}(\nu) - \boldsymbol{w}^\top \boldsymbol{g} = \boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0,$$

from which Equation (6.46) follows. The distance d_ν is understood to be positive for points above the hyperplane and negative for points below. The quantity

$$\gamma_\nu = \ell(\nu) d_\nu = \frac{1}{\|\boldsymbol{w}\|} \ell(\nu)(\boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0) \tag{6.47}$$

is called the *geometric margin* for the observation. Observations lying on the hyperplane have zero geometric margins, incorrectly classified observations have negative geometric margins, and correctly classified observations have positive geometric margins.

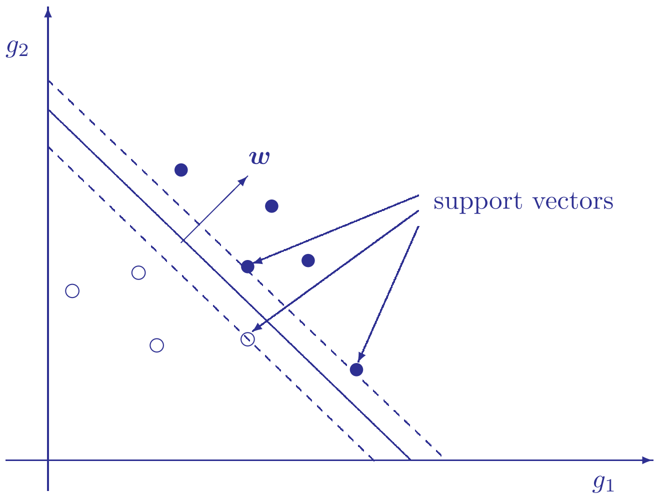


FIGURE 6.15
A maximal margin hyperplane for linearly separable training data and its support vectors (see text).

The *geometric margin of a hyperplane* (relative to a given training set), is defined as the smallest (geometric) margin over the observations in that set, and the *maximal margin hyperplane* is the hyperplane which maximizes the smallest margin, i.e., the hyperplane with parameters $\boldsymbol{w}, w_0$ given by

$$\arg\max_{\mathbf{w},w_0}\left(\frac{1}{\|\boldsymbol{w}\|}\min_{\nu}\left(\ell(\nu)(\boldsymbol{w}^\top\boldsymbol{g}(\nu)+w_0)\right)\right). \tag{6.48}$$

If the training data are linearly separable, then the resulting smallest margin will be positive since all observations are correctly classified. A maximal margin hyperplane is illustrated in Figure 6.15.

6.6.1.1 Primal formulation

The maxmin problem (6.48) can be reformulated as follows (Cristianini and Shawe-Taylor, 2000; Bishop, 2006). If we transform the parameters according to

$$\boldsymbol{w} \to \kappa\boldsymbol{w}, \quad w_0 \to \kappa w_0$$

for some constant κ, then the distance d_ν will remain unchanged, as is clear from Equation (6.46). Now let us choose κ such that

$$\ell(\nu)(\boldsymbol{w}^\top\boldsymbol{g}(\nu')+w_0)=1 \tag{6.49}$$

for whichever training observation $\boldsymbol{g}(\nu')$ happens to be closest to the hyperplane. This implies that the following constraints are met:

$$\ell(\nu)I(\boldsymbol{g}(\nu))=\ell(\nu)(\boldsymbol{w}^\top\boldsymbol{g}(\nu)+w_0)\ge 1, \quad \nu=1\dots m, \tag{6.50}$$

and the geometric margin of the hyperplane (the smallest geometric margin over the observations) is, with Equation (6.47), simply $\|\boldsymbol{w}\|^{-1}$. For observations for which equality holds in Equation (6.50), the constraints are called *active*, otherwise *inactive*. Clearly, for any choice of $\boldsymbol{w}, w_0$, there will always be at least one active constraint. So the problem expressed in Equation (6.48) is equivalent to maximizing $\|\boldsymbol{w}\|^{-1}$ or, expressed more conveniently, to solving

$$\arg\min_{\mathbf{w}} \frac{1}{2}\|\boldsymbol{w}\|^2 \tag{6.51}$$

subject to the constraints of Equation (6.50). These constraints define the *feasible region* for the minimization problem. Taken together, Equations (6.50) and (6.51) constitute the primal formulation of the original maxmin problem, Equation (6.48). The bias parameter w_0 is determined implicitly by the constraints, as we will see later.

6.6.1.2 Dual formulation

To solve Equation (6.51), we will apply the Lagrangian formalism for *inequality constraints*, a generalization of the method which we have used extensively up till now and first introduced in Section 1.6. Bishop (2006), Appendix E and Cristianini and Shawe-Taylor (2000), Chapter 5, provide good discussions. We introduce a Lagrange multiplier α_ν for each of the inequality constraints, to obtain in the Lagrange function

$$L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{\nu=1}^{m} \alpha_\nu \big(\ell(\nu)(\boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0) - 1\big), \tag{6.52}$$

where $\boldsymbol{\alpha} = (\alpha_1 \dots \alpha_m)^\top \geq \mathbf{0}$. Minimization over $\boldsymbol{w}$ and w_0 requires that the respective derivatives be set equal to zero:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_\nu \ell(\nu)\alpha_\nu \boldsymbol{g}(\nu) = \mathbf{0}, \tag{6.53}$$

$$\frac{\partial L}{\partial w_0} = \sum_\nu \ell(\nu)\alpha_\nu = 0. \tag{6.54}$$

Therefore, from Equation (6.53),

$$\boldsymbol{w} = \sum_\nu \ell(\nu)\alpha_\nu \boldsymbol{g}(\nu).$$

Substituting this back into Equation (6.52) and using Equation (6.54) gives

$$\begin{aligned} L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) &= \frac{1}{2}\sum_{\nu\nu'} \ell(\nu)\ell(\nu')\alpha_\nu\alpha_{\nu'}(\boldsymbol{g}(\nu)^\top \boldsymbol{g}(\nu')) \\ &\quad - \sum_{\nu\nu'} \ell(\nu)\ell(\nu')\alpha_\nu\alpha_{\nu'}(\boldsymbol{g}(\nu)^\top \boldsymbol{g}(\nu')) + \sum_\nu \alpha_\nu \\ &= \sum_\nu \alpha_\nu - \frac{1}{2}\sum_{\nu\nu'} \ell(\nu)\ell(\nu')\alpha_\nu\alpha_{\nu'}(\boldsymbol{g}(\nu)^\top \boldsymbol{g}(\nu')), \end{aligned} \tag{6.55}$$

in which $\boldsymbol{w}$ and w_0 no longer appear. We thus obtain the *dual formulation* of the minimization problem, Equations (6.50) and (6.51), namely maximize

$$\tilde{L}(\boldsymbol{\alpha}) = \sum_{\nu} \alpha_\nu - \frac{1}{2}\sum_{\nu\nu'} \ell(\nu)\ell(\nu')\alpha_\nu\alpha_{\nu'}(\boldsymbol{g}(\nu)^\top \boldsymbol{g}(\nu')) \tag{6.56}$$

with respect to the *dual variables* $\boldsymbol{\alpha} = (\alpha_1 \ldots \alpha_m)^\top$ subject to the constraints

$$\begin{aligned} \boldsymbol{\alpha} &\geq \mathbf{0}, \\ \sum_{\nu} \ell(\nu)\alpha_\nu &= 0. \end{aligned} \tag{6.57}$$

We must maximize because, according to *duality theory*, $\tilde{L}(\boldsymbol{\alpha})$ is a lower limit for $L(\boldsymbol{w}, w_0, \boldsymbol{\alpha})$ and, for our problem,

$$\max_{\boldsymbol{\alpha}} \tilde{L}(\boldsymbol{\alpha}) = \min_{\mathbf{w}, w_0} L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}),$$

see, e.g., Cristianini and Shawe-Taylor (2000), Chapter 5. We shall see how to do this in the sequel; however, let us for now suppose that we have found the solution $\boldsymbol{\alpha}^*$ which maximizes $\tilde{L}(\boldsymbol{\alpha})$. Then

$$\boldsymbol{w}^* = \sum_{\nu} \ell(\nu)\alpha^*_\nu \boldsymbol{g}(\nu)$$

determines the maximal margin hyperplane and it has geometric margin $\|\boldsymbol{w}^*\|^{-1}$. In order to classify a new observation $\boldsymbol{g}$, we simply evaluate the sign of

$$I^*(\boldsymbol{g}) = \boldsymbol{w}^{*\top}\boldsymbol{g} + w_0^* = \sum_{\nu} \ell(\nu)\alpha^*_\nu(\boldsymbol{g}(\nu)^\top \boldsymbol{g}) + w_0^*, \tag{6.58}$$

that is, we ascertain on which side of the hyperplane the observation lies. (We still need an expression for w_0^*. This is described below.) Note that both the training phase, i.e., the solution of the dual problem Equations (6.56) and (6.57), as well as the generalization phase, Equation (6.58), involve only inner products of the observations $\boldsymbol{g}$.

6.6.1.3 Quadratic programming and support vectors

The optimization problem represented by Equations (6.56) and (6.57) is a *quadratic programming problem*, the objective function $\tilde{L}(\boldsymbol{\alpha})$ being quadratic in the dual variables α_v. According to the *Karush–Kuhn–Tucker* (KKT) conditions for quadratic programs,* in addition to the constraints of Equations (6.57), the *complementarity condition*

$$\alpha_\nu\big(\ell(\nu)I(\boldsymbol{g}(\nu)) - 1\big) = 0, \quad \nu = 1 \ldots m, \tag{6.59}$$

*See again Cristianini and Shawe-Taylor (2000), Chapter 5.

must be satisfied. Taken together, Equations (6.50), (6.57), and (6.59) are *necessary and sufficient conditions* for a solution $\boldsymbol{\alpha} = \boldsymbol{\alpha}^*$. The complementarity condition says that each of the constraints in Equation (6.50) is either active, that is, $\ell(\nu)I(\boldsymbol{g}(\nu)) = 1$, or it is inactive, $\ell(\nu)I(\boldsymbol{g}(\nu)) > 1$, in which case, from Equation (6.59), $\alpha_\nu = 0$.

When classifying a new observation with Equation (6.58), either the training observation $\boldsymbol{g}(\nu)$ satisfies $\ell(\nu)I^*(\boldsymbol{g}(\nu)) = 1$, which is to say it has minimum margin (Equation (6.49)), or $\alpha_\nu = 0$ by virtue of the complementarity condition, meaning that it *plays no role in the classification*. The labeled training observations with minimum margin are called *support vectors* (see Figure 6.15). After solution of the quadratic programming problem, they are the only observations which can contribute to the classification of new data.

Let us call SV the set of support vectors. Then from the complementarity condition Equation (6.59) we have, for $\nu \in SV$,

$$\ell(\nu)\left(\sum_{\nu' \in SV} \ell(\nu')\alpha^*_{\nu'}(\boldsymbol{g}(\nu')^\top \boldsymbol{g}(\nu)) + w_0^*\right) = 1. \tag{6.60}$$

We can therefore write

$$\begin{aligned}\|\boldsymbol{w}^*\|^2 = \boldsymbol{w}^{*\top}\boldsymbol{w}^* &= \sum_{\nu,\nu'} \ell(\nu)\ell(\nu')\alpha_\nu\alpha_{\nu'}(\boldsymbol{g}(\nu)^\top \boldsymbol{g}(\nu')) \\ &= \sum_{\nu\in SV}\alpha^*_\nu\left(\ell(\nu)\sum_{\nu'\in SV}\ell(\nu')\alpha^*_{\nu'}(\boldsymbol{g}(\nu)^\top\boldsymbol{g}(\nu'))\right),\end{aligned}$$

and, from Equation (6.60),

$$\|\boldsymbol{w}^*\|^2 = \sum_{\nu\in SV}\alpha^*_\nu(1-\ell(\nu)w_0^*) = \sum_{\nu\in SV}\alpha^*_\nu, \tag{6.61}$$

where in the second equality we have made use of the second constraint in Equation (6.57). Thus the geometric margin of the maximal hyperplane is given in terms of the dual variables by

$$\|\boldsymbol{w}^*\|^{-1} = \left(\sum_{\nu\in SV}\alpha^*_\nu\right)^{-1/2}. \tag{6.62}$$

Equation (6.60) can be used to determine w_0^* once the quadratic program has been solved, simply by choosing an arbitrary $\nu \in SV$. Bishop (2006) suggests a numerically more stable procedure: Multiply Equation (6.60) by $\ell(\nu)$ and make use of the fact that $\ell(\nu)^2 = 1$. Then take the average of the equations for all support vectors and solve for w_0^* to get

$$w_0^* = \frac{1}{|SV|}\sum_{\nu\in SV}\left(\ell(\nu) - \sum_{\nu'\in SV}\alpha_{\nu'}\ell(\nu')(\boldsymbol{g}(\nu)^\top\boldsymbol{g}(\nu'))\right). \tag{6.63}$$

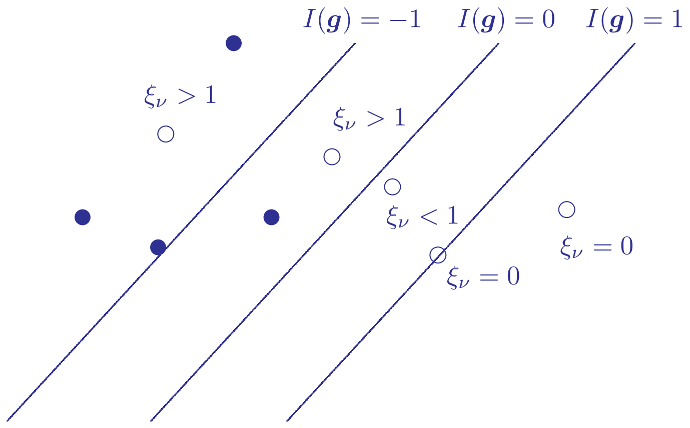


FIGURE 6.16
Slack variables. Observations with values less than 1 are correctly classified.

6.6.2 Overlapping classes

With the SVM formalism introduced so far, one can solve two-class classification problems which are linearly separable. Using kernel substitution, which will be treated shortly, nonlinearly separable problems can also be solved. In general, though, one is faced with labeled training data that overlap considerably, so that their complete separation would imply gross over fitting.

To overcome this problem, we introduce so-called *slack variables* ξ_ν associated with each training vector, such that

$$\xi_\nu = \begin{cases} |\ell(\nu) - I(\boldsymbol{g}(\nu))| & \text{if } \boldsymbol{g}(\nu) \text{ is on the wrong side of the margin boundary} \\ 0 & \text{otherwise.} \end{cases}$$

The situation is illustrated in Figure 6.16. The constraints of Equation (6.50) now become

$$\ell(\nu)I(\boldsymbol{g}(\nu)) \ge 1 - \xi_\nu, \quad \nu = 1 \dots m, \tag{6.64}$$

and are often referred to as *soft margin* constraints.

Our objective is again to maximize the margin, but to penalize points with large values of ξ. We therefore modify the objective function of Equation (6.51) by introducing a regularization term:

$$C\sum_{\nu=1}^{m} \xi_\nu + \frac{1}{2}\|\boldsymbol{w}\|^2. \tag{6.65}$$

The parameter C determines the degree of penalization. When minimizing this objective function, in addition to the above inequality constraints, we require that

$$\xi_\nu \ge 0, \quad \nu = 1 \dots m.$$

The Lagrange function is now given by

$$L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{\nu=1}^{m}\xi_\nu - \sum_{\nu}\alpha_\nu(\ell(\nu)(\boldsymbol{w}^\top \boldsymbol{g}(\nu) + w_0) - 1 + \xi_\nu) - \sum_{\nu=1}^{m}\mu_\nu\xi_\nu, \tag{6.66}$$

where $\mu_\nu \geq 0$, $\nu = 1\dots m$, are additional Lagrange multipliers. Setting the derivatives of the Lagrange function with respect to $\boldsymbol{w}$, w_0 and ξ_ν equal to zero, we get

$$\boldsymbol{w} = \sum_{\nu=1}^{m}\alpha_\nu\ell(\nu)\boldsymbol{g}(\nu) \tag{6.67}$$

$$\sum_{\nu=1}^{m}\alpha_\nu\ell(\nu) = 0 \tag{6.68}$$

$$\alpha_\nu = C - \mu_\nu, \tag{6.69}$$

which leads to the dual form

$$\tilde{L}(\boldsymbol{\alpha}) = \sum_{\nu}\alpha_\nu - \frac{1}{2}\sum_{\nu\nu'}\ell(\nu)\ell(\nu')\alpha_\nu\alpha_{\nu'}(\boldsymbol{g}(\nu)^\top\boldsymbol{g}(\nu')). \tag{6.70}$$

This is the same as for the separable case, Equation (6.56), except that the constraints are slightly different. From Equation (6.69), $\alpha_\nu \leq C$, so that

$$0 \leq \alpha_\nu \leq C, \quad \nu = 1\dots m. \tag{6.71}$$

Furthermore, the complementarity conditions now read

$$\begin{aligned} \alpha_\nu\big(\ell(\nu)I(\boldsymbol{g}(\nu)) - 1 + \xi_\nu\big) &= 0 \\ \mu_\nu\xi_\nu &= 0, \quad \nu = 1\dots m. \end{aligned} \tag{6.72}$$

Equations (6.64), (6.68), (6.71), and (6.72) are again necessary and sufficient conditions for a solution $\boldsymbol{\alpha} = \boldsymbol{\alpha}^*$.

As before, when classifying new data with Equation (6.58), only the support vectors, i.e., the training observations for which $\alpha_\nu > 0$, will play a role. If $\alpha_\nu > 0$, there are now two possibilities to be distinguished:

- $0 \leq \alpha_\nu < C$. Then we must have $\mu_\nu > 0$ (Equation (6.69)), implying from the second condition in Equation (6.72) that $\xi_\nu = 0$. The training vector thus lies exactly on the margin and corresponds to the support vector for the separable case.

- $\alpha_\nu = C$. The training vector lies inside the margin and is correctly classified ($\xi \leq 1$) or incorrectly classified ($\xi > 1$).

We can once again use Equation (6.63) to determine $\boldsymbol{w}_0^*$ after solving the quadratic program for $\boldsymbol{w}^*$, except that the set SV must be restricted to those observations for which $0 < \alpha_\nu < C$, i.e., to the support vectors lying on the margin.

6.6.3 Solution with sequential minimal optimization

To train the classifier, we must maximize Equation (6.70) subject to the boundary conditions given by Equations (6.68) and (6.71), which define the feasible region for the problem. We outline here a popular method for doing this called *sequential minimal optimization* (SMO); see Cristianini and Shawe-Taylor (2000). In this algorithm, pairs of Lagrange multipliers $(\alpha_\nu, \alpha_{\nu'})$ are chosen and varied so as to increase the objective function while still satisfying the constraints. (At least 2 multipliers must be considered in order to guarantee fulfillment of the equality constraint, Equation (6.68)). The SMO method has the advantage that the maximum can be found analytically at each step, thus leading to fast computation.

Algorithm (Sequential minimal optimization)

1. Set $\alpha_\nu^{old} = 0,\ \nu = 1 \dots m$ (clearly $\boldsymbol{\alpha}^{old}$ is in the feasible region).
2. Choose a pair of Lagrange multipliers, calling them without loss of generality α_1 and α_2. Let all of the other multipliers be fixed.
3. Maximize $\tilde{L}(\boldsymbol{\alpha})$ with respect to α_1 and α_2.
4. If the complementarity conditions are satisfied (within some tolerance) stop, else go to step 2.

In in step 2, heuristic rules must be used to choose appropriate pairs. Step 3 can be carried out analytically by maximizing the quadratic function

$$\tilde{L}(\alpha_1, \alpha_2) = \alpha_1 + \alpha_2 - \frac{1}{2}\big(\boldsymbol{g}(1)^\top \boldsymbol{g}(1)\big)\alpha_1^2 - \frac{1}{2}\big(\boldsymbol{g}(2)^\top \boldsymbol{g}(2)\big)\alpha_2^2 - \ell_1\ell_2\alpha_1\alpha_2 \\ - \ell_1\alpha_1 v_1 - \ell_2\alpha_2 v_2 + \text{Const},$$

where

$$v_i = \sum_{\nu=3}^{n} \ell(\nu)\alpha_\nu^{old}\big(\boldsymbol{g}(\nu)^\top \boldsymbol{g}(i)\big), \quad i = 1, 2.$$

The maximization is carried out in the feasible region for α_1, α_2. The equality constraint, Equation (6.68), requires

$$\ell_1\alpha_1 + \ell_2\alpha_2 = \ell_1\alpha_1^{old} + \ell_2\alpha_2^{old},$$

or equivalently

$$\alpha_2 = -\ell_1\ell_2\alpha_1 + \gamma,$$

where $\gamma = \ell_1\ell_2\alpha_1^{old} + \alpha_2^{old}$, so that (α_1, α_2) lies on a line with slope ± 1, depending on the value of $\ell_1\ell_2$. The inequality constraint, Equation (6.71), defines the endpoints of the line segment that need to be considered, namely $(\alpha_1 = 0, \alpha_2 = \max(0, \gamma))$ and $(\alpha_1 = C, \alpha_2 = \min(C, -\ell_1\ell_2 C + \gamma))$. Regarding step 4, the complementarity conditions are as follows; see Equations (6.72):

For all ν,

$$\begin{aligned}
&\text{if } \alpha_\nu = 0, \text{ then } \ell(\nu)I(\boldsymbol{g}(\nu)) - 1 \ge 0\\
&\text{if } \alpha_\nu > 0, \text{ then } \ell(\nu)I(\boldsymbol{g}(\nu)) - 1 = 0\\
&\text{if } \alpha_\nu = C, \text{ then } \ell(\nu)I(\boldsymbol{g}(\nu)) - 1 \le 0.
\end{aligned}$$

6.6.4 Multiclass SVMs

A distinct advantage of the SVM for remote sensing image classification over neural networks is the unambiguity of the solution. Since one maximizes a quadratic function, the maximum is global and will always be found. There is no possibility of becoming trapped in a local optimum. However, SVM classifiers have two disadvantages. They are designed for two-class problems and their outputs, unlike feed-forward neural networks, do not model posterior class membership probabilities in a natural way.

A common way to overcome the two-class restriction is to determine all possible two-class results and then use a voting scheme to decide on the class label (Wu et al., 2004). That is, for K classes, we train $K(K-1)/2$ SVM's on each of the possible pairs $(i,j) \in \mathcal{K} \otimes \mathcal{K}$. For a new observation $\boldsymbol{g}$ and the SVM for (i,j), let

$$\mu_{ij}(\boldsymbol{g}) = \Pr(\ell = i \mid \ell = i \text{ or } j,\ \boldsymbol{g}) = \frac{\Pr(\ell = i \mid \boldsymbol{g})}{\Pr(\ell = i \text{ or } j \mid \boldsymbol{g})}. \tag{6.73}$$

The last equality follows from the definition of conditional probability, Equation (2.63), since the joint probability (given $\boldsymbol{g}$) for i or j and i is just the probability for i. Now suppose that r_{ij} is some rough estimator for μ_{ij}, perhaps simply $r_{ij} = 1$ if $\mu_{ij} > 0.5$ and 0 otherwise. The voting rule is then

$$k = \arg\max_i \left(\sum_{j \ne i}^{K} [[r_{ij} > r_{ji}]] \right), \tag{6.74}$$

where $[[\cdots]]$ is the *indicator function*

$$[[x]] = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{if } x \text{ is false.} \end{cases}$$

With regard to the second restriction, a very simple estimate of the posterior class membership probability is the ratio of the number of votes to the total number of classifications,

$$\Pr(k \mid \boldsymbol{g}) \approx \frac{2}{K(K-1)} \sum_{j \ne i}^{K} [[r_{kj} > r_{jk}]]. \tag{6.75}$$

If r_{ij} is a more realistic estimate, we can proceed as follows (Price et al., 1995):

$$\begin{aligned}\sum_{j\ne i}\Pr(\ell=i \text{ or } j\mid \boldsymbol{g}) &= \sum_{j\ne i}(\Pr(\ell=i\mid \boldsymbol{g})+\Pr(\ell=j\mid \boldsymbol{g}))\\ &=(K-1)\Pr(\ell=i\mid \boldsymbol{g})+\sum_{j\ne i}\Pr(\ell=j\mid \boldsymbol{g})\\ &=(K-2)\Pr(\ell=i\mid \boldsymbol{g})+\sum_{j}\Pr(\ell=j\mid \boldsymbol{g})\\ &=(K-2)\Pr(\ell=i\mid \boldsymbol{g})+1.\end{aligned}\tag{6.76}$$

Combining this with Equation (6.73) gives, see Exercise 9,

$$\Pr(\ell=i\mid \boldsymbol{g})=\frac{1}{\sum_{j\ne i}\mu_{ij}^{-1}-(K-2)}\tag{6.77}$$

or, substituting the estimate r_{ij}, we estimate the posterior class membership probabilities as

$$\Pr(k\mid \boldsymbol{g})\approx\frac{1}{\sum_{j\ne k}r_{kj}^{-1}-(K-2)}.\tag{6.78}$$

Because of the substitution, the probabilities will not sum exactly to one, and must be normalized.

6.6.5 Kernel substitution

Of course the reason why we have replaced the relatively simple optimization problem, Equation (6.51), with the more involved dual formulation is due to the fact that the labeled training observations only enter into the dual formulation in the form of scalar products $\boldsymbol{g}(\nu)^\top\boldsymbol{g}(\nu')$. This allows us once again to use the elegant method of kernelization introduced in Chapter 4. Then we can apply support vector machines to situations in which the classes are not linearly separable.

To give a simple example (Müller et al., 2001), consider the classification problem illustrated in Figure 6.17. While the classes are clearly separable, they cannot be separated with a hyperplane. Now introduce the transformation

$$\boldsymbol{\phi}:\mathbb{R}^2\mapsto\mathbb{R}^3,$$

such that

$$\boldsymbol{\phi}(\boldsymbol{g})=\begin{pmatrix} g_1^2\\ \sqrt{2}g_1g_2\\ g_2^2\end{pmatrix}.$$

In this new feature space, the observations are transformed as shown in Figure 6.18, and can be separated by a two-dimensional hyperplane. We can thus

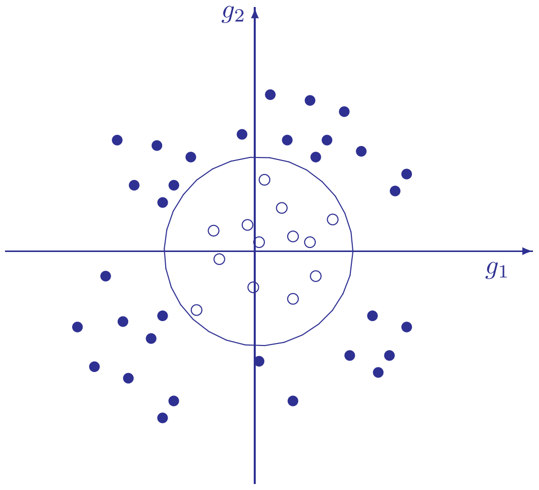


FIGURE 6.17
Two classes which are not linearly separable in the two-dimensional space of observations.

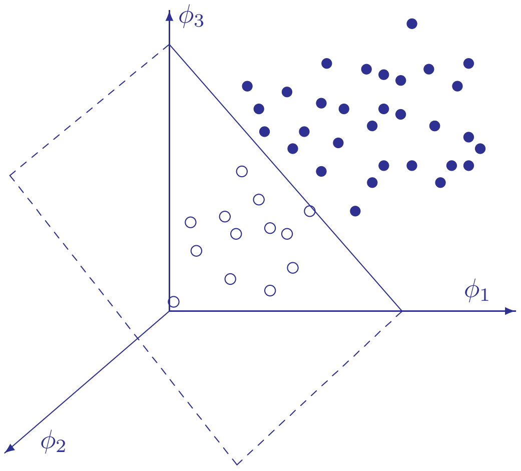


FIGURE 6.18
The classes of Figure 6.17 become linearly separable in a three-dimensional, nonlinear feature space.

Listing 6.6: A class for a support vector machine (excerpt from the Python module `supervisedclass.py`).

```
 1 class Svm(object):
 2     '''Suppot Vector Machine Classifier'''
 3     def __init__(self, Gs, ls, gamma=0.1, C=10):
 4         self._Gs = Gs
 5         self._ls = np.argmax(ls, axis=1)
 6         self._clf = SVC(gamma=gamma, C=C,
 7                kernel='rbf', probability=True)
 8 
 9     def train(self):
10         try:
11             self._clf.fit(self._Gs, self._ls)
12             return True
13         except Exception as e:
14             print('Error:␣%s' % e)
15             return None
16 
17     def classify(self, Gs):
18         classes = self._clf.predict(Gs) + 1
19         probs = self._clf.predict_proba(Gs)
20         return (classes, probs)
```

apply the support vector formalism unchanged, simply replacing $\boldsymbol{g}_v^\top \boldsymbol{g}_{v'}$ by $\boldsymbol{\phi}(\boldsymbol{g}(\nu))^\top \boldsymbol{\phi}(\boldsymbol{g}(\nu'))$. In this case, the kernel function is given by

$$k(\boldsymbol{g}(\nu), \boldsymbol{g}(\nu')) = (g_1^2, \sqrt{2}g_1g_2, g_2^2)_\nu \begin{pmatrix} g_1^2 \\ \sqrt{2}g_1g_2 \\ g_2^2 \end{pmatrix}_{\nu'} = (\boldsymbol{g}(\nu)^\top \boldsymbol{g}(\nu'))^2,$$

called the quadratic kernel.

A wrapper for the `sklearn.svm.SVC` class is shown in Listing 6.6. It implements a kernel support vector machine with `gamma` equal to 0.1 and regularization parameter `C` equal to 10. The kernels available in the `SLC` library are:

$$\begin{aligned} k_{\text{lin}}(\boldsymbol{g}_i, \boldsymbol{g}_j) &= \boldsymbol{g}_i^\top \boldsymbol{g}_j \\ k_{\text{poly}}(\boldsymbol{g}_i, \boldsymbol{g}_j) &= (\gamma \boldsymbol{g}_i^\top \boldsymbol{g}_j + r)^d \\ k_{\text{rbf}}(\boldsymbol{g}_i, \boldsymbol{g}_j) &= \exp(-\gamma \|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2) \\ k_{\text{sig}}(\boldsymbol{g}_i, \boldsymbol{g}_j) &= \tanh(\gamma \boldsymbol{g}_i^\top \boldsymbol{g}_j + r). \end{aligned}$$

The most commonly chosen kernel is the Gaussian kernel (k_{rbf}) and this is the one used in `supervisedclass.py`, line 7. The parameter γ essentially determines the training/generalization trade-off, with large values leading to over fitting (Shawe-Taylor and Cristianini, 2004). Whereas both the maximum likelihood and Gaussian kernel training procedures have essentially no

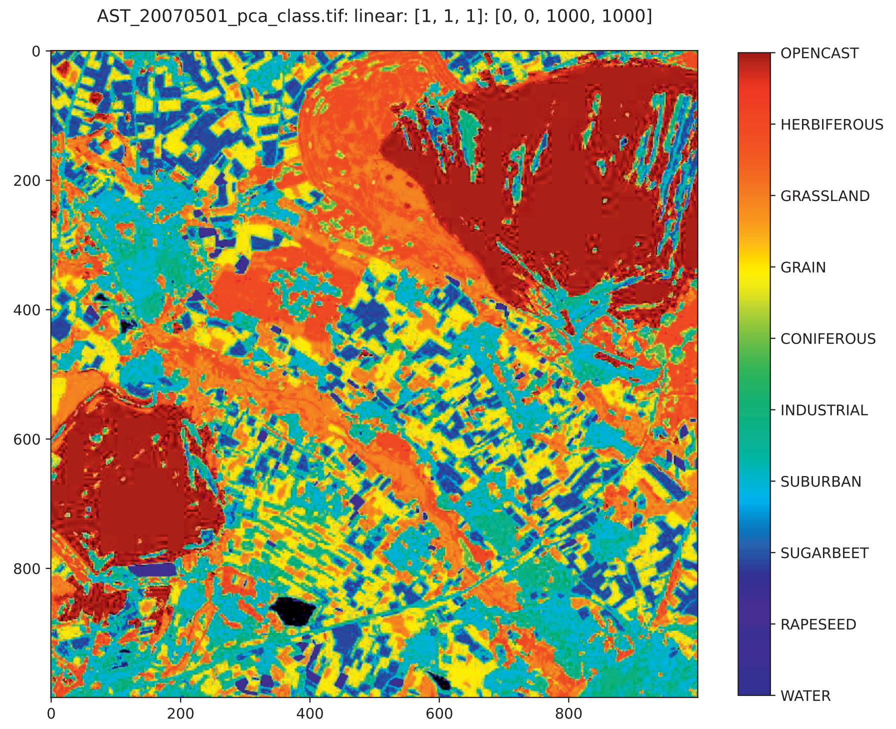


FIGURE 6.19
Classification with the kernel support vector machine classifier.

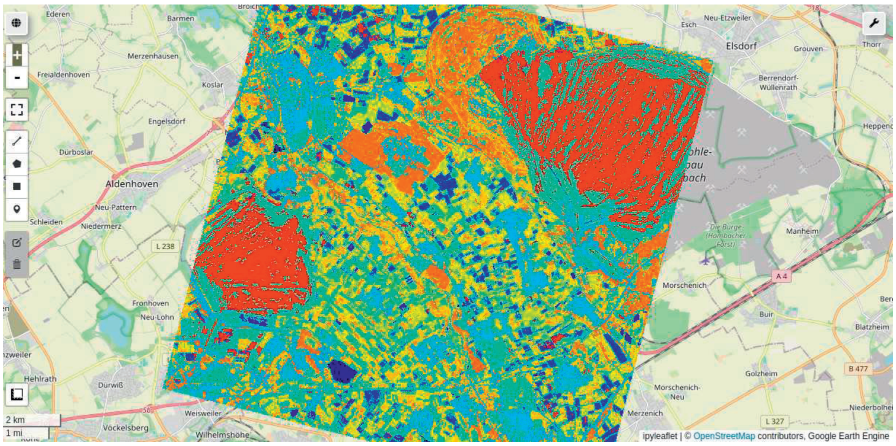


FIGURE 6.20
SVM classification on the GEE API.

externally adjustable parameters,[*] this is not the case for support vector machines. First, one must decide upon a kernel, and then choose associated kernel parameters as well as the soft margin penalization constant C. (The usual initial choice for the `gamma` parameter is the reciprocal of the number of features.) The classified ASTER image is shown in Figure 6.19. Again, test results are saved to a file in a format consistent with that used by the other classification routines described earlier; see Appendix C.

To conclude, we'll also train a SVM with the GEE API:

```
# train a SVM classifier
import geemap
classifier = ee.Classifier.libsvm(kernelType='RBF'
                                ,gamma=0.1,cost=10)
trainData = image.sampleRegions(table,['CLASS_ID']) \
    .remap(['0','1','2','3','4','5','6','7','8','9'],
           [0,1,2,3,4,5,6,7,8,9],'CLASS_ID')
trained = classifier.\
    train(trainData,'CLASS_ID',image.bandNames())
# classify the image and display
classified = image.classify(trained)
map = geemap.Map()
map.centerObject(classified, 12)
map.addLayer(classified.byte(), {'min': 0.0, 'max': 10.0,
       'palette': ['FF0000','#00FF00','#0000FF']}, 'SVM')
map
```

see Figure 6.20. The keyword `cost` corresponds to the soft margin parameter `C` in Listing 6.6. The parameters `C` and `gamma` are the same as for the `sklearn` program.[†]

6.7 Exercises

1. (a) Perform the integration in Equation (6.11) for the one-dimensional case:

$$p(g \mid k) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{1}{2\sigma_k^2}(g-\mu_k)^2\right), \quad k = 1, 2. \tag{6.79}$$

(*Hint:* Use the definite integral $\int_{-\infty}^{\infty} \exp(ag - bg^2)dg = \sqrt{\frac{\pi}{b}} \exp(a^2/4b)$.)
(b) The Jeffries–Matusita distance between two probability densities

[*]Basic neural network training with back propagation has three (learn rate, momentum, and the number of hidden neurons).

[†]This is merely for illustration: the class colors are arbitrary.

$p(\boldsymbol{g} \mid 1)$ and $p(\boldsymbol{g} \mid 2)$ is defined as

$$J = \int_{-\infty}^{\infty} \left(p(\boldsymbol{g} \mid 1)^{1/2} - p(\boldsymbol{g} \mid 2)^{1/2} \right)^2 d\boldsymbol{g}.$$

Show that this is equivalent to the definition in Equation (6.13).

(c) A measure of the separability of $p(\boldsymbol{g} \mid 1)$ and $p(\boldsymbol{g} \mid 2)$ can be written in terms of the Kullback–Leibler divergence, see Equation (2.117), as follows (Richards, 2012):

$$d_{12} = \mathrm{KL}\left(p(\boldsymbol{g} \mid 1), p(\boldsymbol{g} \mid 2)\right) + \mathrm{KL}\left(p(\boldsymbol{g} \mid 2), p(\boldsymbol{g} \mid 1)\right).$$

Explain why this is a satisfactory separability measure.

(d) Show that, for the one-dimensional distributions of Equation (6.79),

$$d_{12} = \frac{1}{2}\left(\frac{1}{\sigma_1^2} - \frac{1}{\sigma_2^2}\right)(\sigma_1^2 - \sigma_2^2) + \frac{1}{2}\left(\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}\right)(\mu_1 - \mu_2)^2.$$

2. (Ripley, 1996) Assuming that all K land cover classes have identical covariance matrices $\boldsymbol{\Sigma}_k = \boldsymbol{\Sigma}$:

(a) Show that the discriminant in Equation (6.17) can be replaced by the linear discriminant

$$d_k(\boldsymbol{g}) = \log(\Pr(k)) - \boldsymbol{\mu}_k^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{g} + \frac{1}{2}\boldsymbol{\mu}_k^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_k.$$

(b) Suppose that there are just two classes $k = 1$ and $k = 2$. Show that the maximum likelihood classifier will choose $k = 1$ if

$$h = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}^{-1} \left(\boldsymbol{g} - \frac{\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2}{2} \right) > \log\left(\frac{\Pr(2)}{\Pr(1)}\right).$$

(c) The quantity $d = \sqrt{(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)}$ is the Mahalanobis distance between the class means. Demonstrate that, if $\boldsymbol{g}$ belongs to class 1, then h is the realization of a normally distributed random variable H_1 with mean $d^2/2$ and variance d^2. What is the corresponding distribution if $\boldsymbol{g}$ belongs to class 2?

(d) Prove from the above considerations that the probability of misclassification is given by

$$\Pr(1) \cdot \Phi\left(-\frac{1}{2}d + \frac{1}{d}\log\left(\frac{\Pr(2)}{\Pr(1)}\right)\right) + \Pr(2) \cdot \Phi\left(-\frac{1}{2}d - \frac{1}{d}\log\left(\frac{\Pr(2)}{\Pr(1)}\right)\right),$$

where Φ is the standard normal distribution function.

(e) What is the minimum possible probability of misclassification?

3. (Linear separability) With reference to Figures 6.8 and 6.9:

   (a) Show that the vector $\boldsymbol{w}$ is perpendicular to the hyperplane

$$I(\boldsymbol{g}) = \boldsymbol{w}^\top \boldsymbol{g} + w_0 = 0.$$

   (b) Suppose that there are just three training observations or points $\boldsymbol{g}_i$, $i = 1, 2, 3$, in a two-dimensional feature space and that these do not lie on a straight line. Since there are only two possible classes, the three points can be labeled in $2^3 = 8$ possible ways. Each possibility is obviously *linearly separable*. That is to say, one can find an oriented hyperplane which will correctly classify the three points, i.e., all class 1 points (if any) lie on one side and all class 2 points (if any) lie on the other side, with the vector $\boldsymbol{w}$ pointing to the class 1 side. The three points are said to be *shattered* by the set of hyperplanes. The maximum number of points that can be shattered is called the *Vapnik–Chervonenskis* (VC) dimension of the hyperplane classifier. What is the VC dimension in this case?

   (c) A training set is linearly separable if a hyperplane can be found for which the smallest margin is positive. The following *perceptron algorithm* is guaranteed to find such a hyperplane, that is, to train the neuron of Figure 6.9 to classify any linearly separable training set of n observations (Cristianini and Shawe-Taylor, 2000):

   i. Set $\boldsymbol{w} = \boldsymbol{0}$, $b = 0$, and $R = \max_\nu \|\boldsymbol{g}(\nu)\|$.

   ii. Set $m = 0$.

   iii. For $i = 1$ to n do: if $\gamma_\nu \le 0$, then set $\boldsymbol{w} = \boldsymbol{w} + \ell(\nu)\boldsymbol{g}(\nu)$, $b = b + \ell(\nu)R^2$ and $m = 1$.

   iv. If $m = 1$ go to ii, else stop.

   The algorithm stops with a separating hyperplane $\boldsymbol{w}^\top \boldsymbol{g} + b = 0$. Implement this algorithm in Python and test it with linearly separable training data, see, e.g., Exercise 13 in Chapter 3 and Figure 3.13.

4. (Bishop, 1995) Neural networks can also be used to approximate continuous vector functions $\boldsymbol{h}(\boldsymbol{g})$ of their inputs $\boldsymbol{g}$. Suppose that, for a given observation $\boldsymbol{g}(\nu)$, the corresponding training value $\boldsymbol{\ell}(\nu)$ is not known exactly, but that its components $\ell_k(\nu)$ are normally and independently distributed about the (unknown) functions $h_k(\boldsymbol{g}(\nu))$, i.e.,

$$p(\ell_k(\nu) \mid \boldsymbol{g}(\nu)) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(h_k(\boldsymbol{g}(\nu)) - \ell_k(\nu))^2}{2\sigma^2}\right), \quad k = 1 \dots K. \tag{6.80}$$

   Show that the appropriate cost function to train the synaptic weights so as to best approximate $\boldsymbol{h}$ with the network outputs $\boldsymbol{m}$ is the quadratic

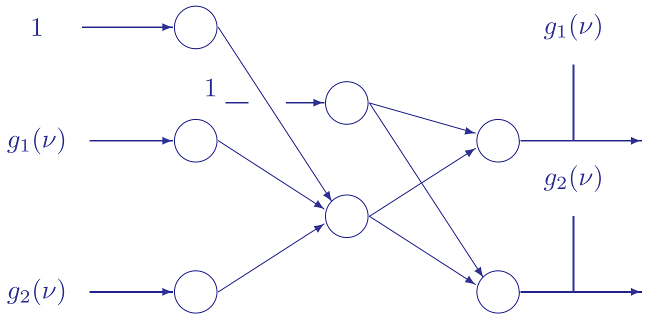


FIGURE 6.21
Principal components analysis with a neural network.

cost function, Equation (6.30). *Hint:* The probability density for a particular training pair $(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu))$ can be written as (see Equation (2.65))

$$p(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu)) = p(\boldsymbol{\ell}(\nu) \mid \boldsymbol{g}(\nu))p(\boldsymbol{g}(\nu)).$$

The likelihood function for n training examples chosen independently from the same distribution is, accordingly,

$$\prod_{\nu=1}^{n} p(\boldsymbol{\ell}(\nu) \mid \boldsymbol{g}(\nu))p(\boldsymbol{g}(\nu)).$$

Argue that maximizing this likelihood with respect to the synaptic weights is equivalent to minimizing the cost function

$$E = -\sum_{\nu=1}^{n} \log p(\boldsymbol{\ell}(\nu) \mid \boldsymbol{g}(\nu))$$

and then show that, with Equation (6.80), this reduces to Equation (6.30).

5. As an example of continuous function approximation, the neural network of Figure 6.11 can be used to perform principal components analysis on sequential data. The method applied involves so-called *self-supervised* classification, or *auto encoding*. The idea is illustrated in Figure 6.21 for two-dimensional observations. The training data are presented to a network with a single hidden neuron. The output is constrained to be identical to the input (self-supervision). Since the output from the hidden layer is one-dimensional, the constraint requires that as much information as possible about the input signal be coded by the hidden neuron. This is the case if the data are projected along the first principal axis. As more and more data are presented, the synaptic weight vector

(w_1^h, w_2^h) will therefore point more and more in the direction of that axis. Use the Python class `Ffnbp()` defined in the `supervisedclass.py` module to implement the network in Figure 6.21 in and test it with simulated data. (According to the previous exercise, the cross-entropy cost function used in `Ffnbp()` is not fully appropriate, but suffices nevertheless.)

6. Demonstrate Equation (6.39).

7. (Network symmetries) The hyperbolic tangent is defined as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

(a) Show that the logistic function $f(x) = 1/(1+e^{-x})$ can be expressed in the form

$$f(x) = \frac{1}{2}\tanh\left(\frac{x}{2}\right) + \frac{1}{2}.$$

(b) Noting that $\tanh(x)$ is an odd function, i.e., $\tanh(x) = -\tanh(-x)$, argue that, for the feed-forward network of Figure 6.11, there are (at least) 2^L identical local minima in the cost function of Equation (6.33).

8. When using a momentum parameter α to train a feed-forward network, the synaptic weights are updated according to

$$w(\nu+1) = w(\nu) + \Delta(\nu) + \alpha\Delta(\nu-1).$$

Show that, if $\Delta(\nu) = \Delta$, independently of ν, then as $\nu \to \infty$, the update rule tends to

$$w(\nu+1) = w(\nu) + \frac{1}{\alpha-1}\Delta.$$

9. Show that Equation (6.77) follows from Equations (6.73) and (6.76).

7

Supervised Classification Part 2

Continuing in the present Chapter on the subject of supervised classification, we will begin with a discussion of post classification processing methods to improve results on the basis of contextual information, after which we turn our attention to statistical procedures for evaluating classification accuracy and for making quantitative comparisons between different classifiers. As examples of *ensembles* of classifiers, we then examine the *adaptive boosting* technique, applying it in particular to improve the generalization accuracy of neural networks, and discuss as well the *random forest* classifier, an ensemble of *binary decision trees*. The remainder of the Chapter examines more specialized forms of supervised image classification, namely as applied to polarimetric SAR imagery, to data with hyper-spectral resolution, and to intermediate and high resolution multispectral imagery using *convolutional neural networks*, introducing the concepts of *transfer learning* and *semantic segmentation*.

7.1 Postprocessing

Intermediate resolution remote sensing satellite platforms used for land cover/land use classification, e.g., LANDSAT, SPOT, RapidEye, Sentinel-2, or ASTER, have ground sample distances (GSDs), ranging between a few to a few tens of meters. These are typically smaller than the landscape objects being classified (agricultural fields, forests, urban areas, etc.). The imagery that they generate is therefore characterized by a high degree of spatial correlation. In Chapter 8 we will see examples of how spatial or contextual information might be incorporated into unsupervised classification. The supervised classification case is somewhat different, since reference is always being made to a—generally quite small—subset of labeled training data. Two approaches can be distinguished for inclusion of contextual information into supervised classification: moving window (or filtering) methods and segmentation (or region growing) methods. Both approaches can be applied either during classification or as a post processing step. We will restrict ourselves here to post classification filtering, in particular mentioning briefly the majority filtering function offered in the standard image processing environments, and then discussing in some detail a modification of a probabilistic technique described in Richards

DOI: 10.1201/9781003503286-7

(2012). For an overview (and an example of), the use of segmentation for contextual classification, see Stuckens et al. (2000) and references therein.

7.1.1 Majority filtering

Majority postclassification filtering employs a moving window, with each central pixel assigned to the majority class of the pixels within the window. This clearly will have the effect of reducing the "salt-and-pepper" appearance typical of the thematic maps generated by pixel-oriented classifiers, and, to quote Stuckens et al. (2000), "it also results in larger classification units that might adhere more to the human perception of land cover." Majority filtering merely examines the labels of neighborhood pixels. The classifiers we have discussed in the last chapter generate, in addition to class labels, class membership probability vectors for each observation. Therefore, neighboring pixels offer considerably more information than that exploited in majority filtering, information which can also be included in the relabeling process.

7.1.2 Probabilistic label relaxation

Recalling Figure 4.14, the 4-neighborhood $\mathcal{N}_i$ of an image pixel with intensity vector $\boldsymbol{g}_i$ consists of the pixels above, below, to the left, and to the right of the pixel. The *a posteriori* class membership probabilities of the central pixel, as approximated by one of the classifiers of Chapter 6, are given by

$$\Pr(k \mid \boldsymbol{g}_i), \quad k = 1 \dots K, \quad \text{where} \sum_{k=1}^{K} \Pr(k \mid \boldsymbol{g}_i) = 1,$$

which, for notational convenience, we represent in the following as the K-component column vector $\boldsymbol{P}_i$ having components

$$P_i(k) = \Pr(k \mid \boldsymbol{g}_i), \quad k = 1 \dots K. \tag{7.1}$$

According to the standard decision rule, Equation (6.6), the maximum component of $\boldsymbol{P}_i$ determines the class membership of the ith pixel.

In analogy to majority filtering, we might expect that a possible misclassification of the pixel can be corrected by examining the membership probabilities in its neighborhood. If that is so, the neighboring pixels will have in some way to modify $\boldsymbol{P}_i$ such that its maximum component is more likely to correspond to the true class. We now describe a purely heuristic but nevertheless intuitively satisfying procedure to do just that, the so-called *probabilistic label relaxation* (PLR) method (Richards, 2012).

Let us postulate a multiplicative *neighborhood function* $Q_i(k)$ for the ith pixel which corrects $P_i(k)$ in the above sense, that is,

$$P'_i(k) = P_i(k) \frac{Q_i(k)}{\sum_j P_i(j) Q_i(j)}, \quad k = 1 \dots K. \tag{7.2}$$

The denominator ensures that the corrected values sum to unity and so still constitute a probability vector. In an obvious vector notation, we can write

$$\boldsymbol{P}_i' = \boldsymbol{P}_i \cdot \frac{\boldsymbol{Q}_i}{\boldsymbol{P}_i^\top \boldsymbol{Q}_i}, \tag{7.3}$$

where the dot signifies ordinary component-by-component multiplication.

The vector $\boldsymbol{Q}_i$ must somehow reflect the contextual information of the neighborhood. In order to define it, a *compatibility measure*

$$P_{ij}(k \mid m), \quad j \in \mathcal{N}_i$$

is introduced, namely, the conditional probability that pixel i has class label k, given that a neighboring pixel $j \in \mathcal{N}_i$ belongs to class m. A "small piece of evidence" (Richards, 2012) that i should be classified to k would then be

$$P_{ij}(k \mid m)P_j(m), \quad j \in \mathcal{N}_i.$$

This is the conditional probability that pixel i is in class k if neighboring pixel j is in class m multiplied by the probability that pixel j actually is in class m. We obtain the component $Q_i(k)$ of the neighborhood function by summing over all pieces of evidence and then averaging over the neighborhood:

$$\begin{aligned} Q_i(k) &= \frac{1}{4}\sum_{j\in\mathcal{N}_i}\sum_{m=1}^{K} P_{ij}(k \mid m)P_j(m) \\ &= \sum_{m=1}^{K} P_{i\mathcal{N}_i}(k \mid m)P_{\mathcal{N}_i}(m). \end{aligned} \tag{7.4}$$

Here $P_{\mathcal{N}_i}(m)$ is an average over all four neighborhood pixels:

$$P_{\mathcal{N}_i}(m) = \frac{1}{4}\sum_{j\in\mathcal{N}_i} P_j(m),$$

and $P_{i\mathcal{N}_i}(k \mid m)$ also corresponds to the *average compatibility* of pixel i with its entire neighborhood. We can write Equation (7.4) in matrix notation in the form

$$\boldsymbol{Q}_i = \boldsymbol{P}_{i\mathcal{N}_i}\boldsymbol{P}_{\mathcal{N}_i}$$

and Equation (7.3) finally as

$$\boldsymbol{P}_i' = \boldsymbol{P}_i \cdot \frac{\boldsymbol{P}_{i\mathcal{N}_i}\boldsymbol{P}_{\mathcal{N}_i}}{\boldsymbol{P}_i^\top \boldsymbol{P}_{i\mathcal{N}_i}\boldsymbol{P}_{\mathcal{N}_i}}. \tag{7.5}$$

For supervised classification, the matrix of average compatibilities $\boldsymbol{P}_{i\mathcal{N}_i}$ is not *a priori* available. However, it may easily be estimated directly from the initially classified image by assuming that it is independent of pixel location.

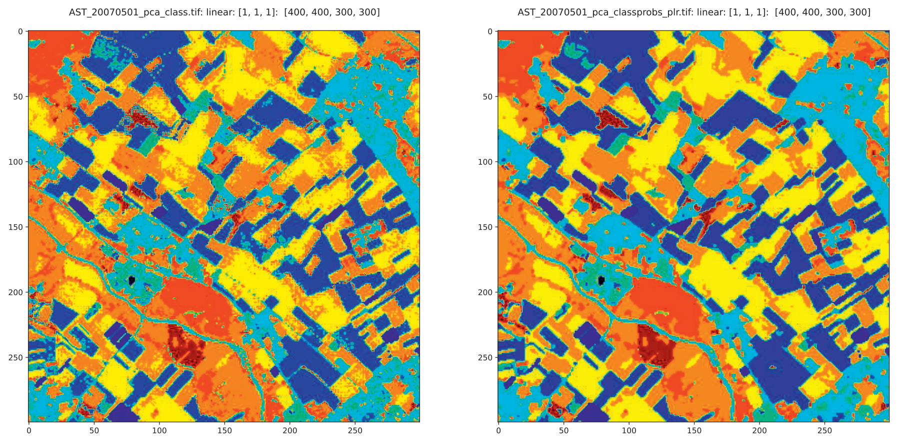

AST_20070501_pca_class.tif: linear: [1, 1, 1]: [400, 400, 300, 300]
AST_20070501_pca_classprobs_plr.tif: linear: [1, 1, 1]: [400, 400, 300, 300]
0
50
100
150
200
250

FIGURE 7.1
An example of post classification processing. Left: original classification of a portion of the ASTER image of Figure 6.1 with a neural network. Right: after three iterations of PLR.

First, a random central pixel i is chosen and its class label $\ell_i = k$ determined. Then, again randomly, a pixel $j \in \mathcal{N}_i$ is chosen and its class label $\ell_j = m$ is also determined. Thereupon the matrix element $P_{i\mathcal{N}_i}(k \mid m)$ (which was initialized to 0) is incremented by 1. This is repeated many times, and finally the rows of the matrix are normalized. All of which constitutes the first step of the following algorithm:

Algorithm (Probabilistic Label Relaxation)

1. Carry out a supervised classification and determine the $K \times K$ compatibility matrix $\boldsymbol{P}_{i\mathcal{N}_i}$.
2. For each pixel i, determine the average neighborhood vector $\boldsymbol{P}_{\mathcal{N}_i}$ and replace $\boldsymbol{P}_i$ with $\boldsymbol{P}'_i$ as in Equation (7.5). Reclassify pixel i according to $\ell_i = \arg\max_k P'_i(k)$.
3. If only a few reclassifications took place, stop; otherwise go to step 2.

The stopping condition in the algorithm is obviously rather vague. Experience shows that the best results are obtained after 3 to 4 iterations; see Richards (2012). Too many iterations lead to a widening of the effective neighborhood of a pixel to such an extent that fully irrelevant spatial information falsifies the final product.

A Python script for probabilistic label relaxation is documented in Appendix C. Figure 7.1 shows a neural network classification result before and after PLR:

```
run scripts/plr imagery/AST_20070501_pca_classprobs.tif

=====================
       PLR
=====================
infile:  imagery/AST_20070501_pca_classprobs.tif
iterations:  3
estimating compatibility matrix...
label relaxation...
iteration 1
iteration 2
iteration 3
result written to:
imagery/ AST_20070501_pca_classprobs_plr.tif
elapsed time: 99.3968970776
--done------------------------
```

The spatial coherence of the classes is improved. The PLR method can also be applied to any unsupervised classification algorithm that generates posterior class membership probabilities. An example is the Gaussian mixture clustering algorithm that will be met in Chapter 8.

7.2 Evaluation and comparison of classification accuracy

Assuming that sufficient labeled data are available for some to be set aside for test purposes, test data can be used to make an unbiased estimate of the *misclassification rate* of a trained classifier, i.e., the fraction of new data that will be incorrectly classified. This quantity provides a reasonable yardstick not only for evaluating the overall accuracy of supervised classifiers, but also for comparison of alternatives, for example, to compare the performance of a neural network with a maximum-likelihood classifier on the same set of data.

7.2.1 Accuracy assessment

The classification of a single test datum is a random experiment, the possible outcomes of which constitute the sample space $\{\bar{A}, A\}$, where $\bar{A}$ = *misclassified*, A = *correctly classified*. Let us define a real-valued function X on this set, i.e., a random variable

$$X(\bar{A}) = 1, \quad X(A) = 0, \tag{7.6}$$

with mass function

$$\Pr(X = 1) = \theta, \quad \Pr(X = 0) = 1 - \theta.$$

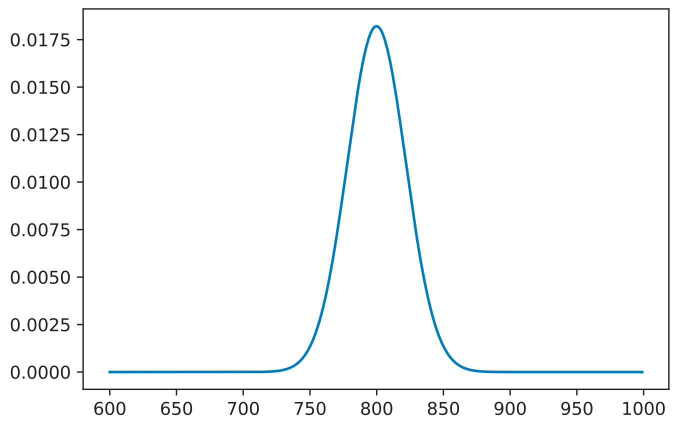


FIGURE 7.2
The binomial distribution for $n = 2000$ and $\theta = 0.4$ closely approximates a normal distribution.

The mean value of X is then

$$\langle X\rangle = 1\theta + 0(1-\theta) = \theta \tag{7.7}$$

and its variance is (see Equation (2.13))

$$\mathrm{var}(X) = \langle X^2\rangle - \langle X\rangle^2 = 1^2\theta + 0^2(1-\theta) - \theta^2 = \theta(1-\theta). \tag{7.8}$$

For the classification of n test data, which are represented by the i.i.d. sample $X_1 \dots X_n$, the random variable

$$Y = X_1 + X_2 + \dots X_n$$

corresponds to the total number of misclassifications. The random variable describing the misclassification rate is therefore Y/n having mean value

$$\langle \frac{1}{n}Y\rangle = \frac{1}{n}(\langle X_1\rangle + \dots + \langle X_n\rangle) = \frac{1}{n}\cdot n\theta = \theta. \tag{7.9}$$

From the independence of the X_i, $i = 1\dots n$, the variance of Y is given by

$$\mathrm{var}(Y) = \mathrm{var}(X_1) + \dots + \mathrm{var}(X_n) = n\theta(1-\theta), \tag{7.10}$$

so the variance of the misclassification rate is

$$\sigma^2 = \mathrm{var}\left(\frac{Y}{n}\right) = \frac{1}{n^2}\mathrm{var}(Y) = \frac{\theta(1-\theta)}{n}. \tag{7.11}$$

For y observed misclassifications, we estimate θ as $\hat{\theta} = y/n$. Then the estimated variance is given by

$$\hat{\sigma}^2 = \frac{\hat{\theta}(1-\hat{\theta})}{n} = \frac{\frac{y}{n}\left(1-\frac{y}{n}\right)}{n} = \frac{y(n-y)}{n^3},$$

and the estimated standard deviation by

$$\hat{\sigma} = \sqrt{\frac{y(n-y)}{n^3}}. \tag{7.12}$$

As was pointed out in Section 2.1.1, Y is (n, θ)-binomially distributed. However, for a sufficiently large number n of test data, the binomial distribution is well approximated by the normal distribution. This is illustrated by the following Python code snippet, which generates the mass function for the binomial distribution, Equation (2.3) for $n = 2000$. The result is shown in Figure 7.2.

```
import numpy as np
from scipy.stats import binom
import matplotlib.pyplot as plt

theta = 0.4
n = 2000
x = np.arange(600, 1000)
# pmf = probability mass function
plt.plot(x, binom.pmf(x, n, theta))
```

Mean and standard deviation are thus generally sufficient to characterize the distribution of misclassification rates completely. To obtain an interval estimation for θ, recalling the discussion in Section 2.2.2, we make use of the fact that the random variable $(Y/n - \theta)/\sigma$ is approximately standard normally distributed. Then

$$\Pr\left(-s < \frac{Y/n - \theta}{\sigma} \le s\right) = 2\Phi(s) - 1, \tag{7.13}$$

so that the random interval $(Y/n - s\sigma, Y/n + s\sigma)$ covers the unknown misclassification rate θ with probability $2\Phi(s) - 1$. However, note that from Equation (7.11), σ is itself a function of θ. It is easy to show (Exercise 1) that, for $1 - \alpha = 0.95 = 2\Phi(s) - 1$ (which gives $s = 1.96$ from the normal distribution table), a 95% confidence interval for θ is

$$\left(\frac{y + 1.92 - 1.96 \cdot \sqrt{0.96 + \frac{y(n-y)}{n}}}{3.84 + n}, \frac{y + 1.92 + 1.96 \cdot \sqrt{0.96 + \frac{y(n-y)}{n}}}{3.84 + n}\right). \tag{7.14}$$

This interval should be routinely stated for any supervised classification of land use/land cover.

Detailed test results for supervised classification are usually presented in the form of a *contingency table*, or *confusion matrix*, which for K classes is defined as

$$\boldsymbol{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1K} \\ c_{21} & c_{22} & \cdots & c_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ c_{K1} & c_{K2} & \cdots & c_{KK} \end{pmatrix}. \tag{7.15}$$

The matrix element c_{ij} is the number of test pixels with class label j which are classified as i. Note that the estimated misclassification rate is

$$\hat{\theta} = \frac{y}{n} = \frac{n - \sum_{i=1}^{K} c_{ii}}{n} = \frac{n - \mathrm{tr}(\boldsymbol{C})}{n}$$

and only takes into account the diagonal elements of the confusion matrix. The so-called *Kappa-coefficient*, on the other hand, makes use of all the matrix elements. It corrects the classification rate for the possibility of chance correct classifications and is defined as follows (Cohen, 1960):

$$\kappa = \frac{\Pr(\text{correct classification}) - \Pr(\text{chance classification})}{1 - \Pr(\text{chance classification})}.$$

An expression for κ can be obtained in terms of the row and column sums in the matrix $\boldsymbol{C}$, which we write as

$$c_{i\bullet} = \sum_{j=1}^{K} c_{ij} \quad \text{and} \quad c_{\bullet i} = \sum_{j=1}^{K} c_{ji},$$

respectively. For n randomly labeled test pixels, the proportion of entries in the ith row is $c_{i\bullet}/n$ and in the ith column $c_{\bullet i}/n$. The probability of a chance correct classification (chance coincidence of row and column index) is therefore approximately given by the sum over i of the product of these two proportions:

$$\sum_{i=1}^{K} \frac{c_{i\bullet}\, c_{\bullet i}}{n^2}.$$

Hence, an estimate for the Kappa coefficient is

$$\hat{\kappa} = \frac{\sum_i c_{ii}/n - \sum_i c_{i\bullet} c_{\bullet i}/n^2}{1 - \sum_i c_{i\bullet} c_{\bullet i}/n^2}. \tag{7.16}$$

Again, the Kappa coefficient alone tells us little about the quality of the classifier. We require its uncertainty. This can be calculated in the large sample limit $n \to \infty$ to be (Bishop et al., 1975)

$$\hat{\sigma}_\kappa^2 = \frac{1}{n}\left(\frac{\theta_1(1-\theta_1)}{(1-\theta_2)^2} + \frac{2(1-\theta_1)(2\theta_1\theta_2 - \theta_3)}{(1-\theta_2)^3} + \frac{(1-\theta_1)^2(\theta_4 - 4\theta_2^2)}{(1-\theta_2)^4} \right), \tag{7.17}$$

where

$$\theta_1 = \frac{1}{n}\sum_{i=1}^{K} c_{ii}, \quad \theta_2 = \frac{1}{n^2}\sum_{i=1}^{K} c_{i\bullet}c_{\bullet i}, \quad \theta_3 = \frac{1}{n^2}\sum_{i=1}^{K} c_{ii}(c_{i\bullet}+c_{\bullet i}),$$

$$\theta_4 = \frac{1}{n^3}\sum_{i,j=1}^{K} c_{ij}(c_{j\bullet}+c_{\bullet i})^2.$$

The Python script `ct.py` (Appendix C) prints out misclassification rate $\hat{\theta}$, standard deviation $\hat{\sigma}$, the confidence interval of Equation (7.14), Kappa coefficient $\hat{\kappa}$, standard deviation $\hat{\sigma}_\kappa$ and a contingency table $\boldsymbol{C}$ from the test result files generated by any of the classifiers described in Chapter 6. Here is a sample output for one of the FFN classifiers:

```
 1 run scripts/ct imagery/AST_20070501_pca_NNet(Congrad).tst
 2
 3 =========================
 4 classification statistics
 5 =========================
 6 NNet(Congrad)test results for imagery/AST_20070501_pca.tif
 7 Tue Apr 16 17:57:57 2024
 8 Classification image: imagery/AST_20070501_pca_class.tif
 9 Class probabilities image:
10                    imagery/AST_20070501_pca_classprobs.tif
11
12 Misclassification rate: 0.088832
13 Standard deviation: 0.005851
14 Conf. interval (95 percent): [0.078021 , 0.100978]
15 Kappa coefficient: 0.899539
16 Standard deviation: 0.006611
17 Contingency Table
18 [[ 155.        0.        0.     ...   155.        1.    ]
19  [   0.      167.        0.     ...   167.        1.    ]
20  [   0.        0.      244.     ...   244.        1.    ]
21  [   0.        0.        0.     ...   313.        0.92  ]
22  [   0.        1.        0.     ...   224.        0.96  ]
23  [   0.        0.        0.     ...   125.        1.    ]
24  [   0.        0.        0.     ...   431.        0.935]
25  [   0.        0.        0.     ...   174.        0.833]
26  [   0.        0.        0.     ...   360.        0.972]
27  [   0.        0.        0.     ...   171.        0.994]
28  [ 155.      168.      244.     ...  2364.        0.    ]
29  [   1.        0.994     1.     ...     0.        0.    ]]
```

The matrix $\boldsymbol{C}$, Equation (7.15), is in the upper left 10×10 block in the table. Row 11 (line 28 in the above listing) gives the column sums $c_{\bullet i}$ and column 11 (second last) gives the row sums $c_{i\bullet}$. Row 12 (line 29) contains the ratios $c_{ii}/c_{\bullet i}$, $i = 1\ldots10$, which are referred to as the *producer accuracies*. Column 12 contains the *user accuracies* $c_{ii}/c_{i\cdot}$, $i = 1 \ldots 10$. The producer accuracy is the probability that an observation with label i will be classified as such. The user accuracy is the probability that the true class of an observation is i given that the classifier has labeled it as i. The rate of *correct classification*, which is more commonly quoted in the literature, is of course one minus the misclassification rate. A detailed discussion of confusion matrices for assessment of classification accuracy is provided in Congalton and Green (1999).

7.2.2 Accuracy assessment on the GEE

Not surprisingly, the Google Earth Engine API includes similar functions for assessing the generalization capability of trained classifiers. To illustrate, we return to the support vector machine from Chapter 6. We'll again choose the ASTER PCA image of Figure 6.1 and the associated training regions, both of which which were previously uploaded to the GEE code editor:

```
import ee
ee.Initialize()
# first 4 principal components of ASTER image
image = ee.Image('projects/␣.../CRC5/T_20070501_pca') \
                    .select(0,1,2,3)
```

The following code samples the image for the training observations and splits them into train and test subsets in the ratio two to one:

```
# sample the image with the polygons to a feature
# collection, rename the class id columns from strings
# to integers, add a column of random numbers in [0,1]
trainTestData = image.sampleRegions(collection=table,
                            properties=['CLASS_ID'],
                            scale=15) \
 .remap(list(map(str,range(10))),list(range(10)),
                                       'CLASS_ID') \
 .randomColumn('rand',seed=12345)
# filter on the random column to split into training and
# test feature collections in the ration of 2:1
trainData=trainTestData.filter(ee.Filter.lt('rand',0.67))
testData=trainTestData.filter(ee.Filter.gte('rand',0.67))
print('train␣pixels:␣%i'%trainData.size().getInfo())
print('test␣pixels:␣␣%i'%testData.size().getInfo())

train pixels: 4801
test pixels:  2372
```

Then the train and test steps are carried out with the SVM classifier:

```
# train a (default) SVM classifier on training data
classifier = ee.Classifier.libsvm()
trained = classifier.train(trainData,'CLASS_ID',
                               image.bandNames())
```

```
# test the trained classifier with the test data
tested = testData.classify(trained, 'classification')
```

Finally, we generate a `Confusion Matrix()` instance and use it to evaluate classification accuracy and kappa value:

```
# generate a confusion matrix with the
# classified test data
cm = tested.errorMatrix('CLASS_ID','classification')
# and from it determine the accuracy and kappa
print('accuracy: %f'%cm.accuracy().getInfo())
print('kappa:    %f'%cm.kappa().getInfo())

accuracy: 0.946880
kappa:    0.939896
```

7.2.3 Cross-validation on parallel architectures

Representative ground reference data at or sufficiently near the time of image acquisition are generally difficult and/or expensive to come by; see Section 6.2. In this regard, the simple 2:1 train:test split used above is rather wasteful of the available labeled training pixels. Moreover, the variability due to the training data is not taken properly into account, since the data are sampled just once from their underlying distributions. In the case of neural networks, we have also so far ignored the variability of the training procedure itself with respect to the random initialization of the synaptic weights. Different initializations may lead to different local minima in the cost function and correspondingly different misclassification rates. Only if these aspects are considered to be negligible should the simple procedures discussed above be applied. Including them properly may constitute a very computationally intensive task (Ripley, 1996).

An alternative approach, one which at least makes more efficient use of the training data, is to apply *n-fold cross-validation*: A small fraction (one nth of the labeled pixels) is held back for testing, and the remaining data are used to train the classifier. This is repeated n times for n complementary test data subsets and then the results, e.g., misclassification rates, are averaged. In this way a larger fraction of the labeled data, namely $(n-1)/n$, is used for training. Moreover, all of the data are used for both training/testing and each observation is used for testing exactly once. For neural network classifiers, the effect of synaptic weight initialization is also reflected in the variance of the test results. The drawback here, of course, is the necessity to repeat the train/test procedure n times rather than carrying it through only once. This is a problem especially for classifiers like neural networks or support vector machines with computationally expensive training algorithms. The cross-validation steps can, however, be performed in parallel given appropriate computer resources. Fortunately these are now generally available, not only in the form of multicore processors, GPU hardware, etc., but also cloud computing services.

We shall illustrate n-fold cross-validation using the IPython `ipyparallel` package to run parallel IPython engines on the available cores of the host CPU (`http://ipyparallel.readthedocs`, which implements parallel processing, is shown in Listing 7.1. Before running this script within a Jupyter notebook the IPython parallel engines can be started by entering the command

```
ipcluster start -n <i> --daemonize
```

Here `<i>` is the number of engines desired, normally the number of available CPU cores/threads and typically 2, 4, or 8, and the option `daemonize` pushes them to the background so that you can proceed to the next cell. Lines 4 through 8 in the script construct a list called `traintest` consisting of ten input tuples which will be passed to the function `crossvalidate()` (defined in lines 22–42). Each tuple corresponds to a different train/test split of the available training pairs `(Gs, ls)` in the ratio of nine to one. In line 11 a client is created that acts as a proxy to the IPython engines. Then a so-called *view* `v = c[:]` into the array of available engines is defined, encompassing, in this case, all of the engines. Next, in line 14, the view's `map_sync()` method is invoked to distribute the train/test tuples to parallel versions of `crossvalidate()` running on the available engines. Each version returns the misclassification rate determined with the chosen classifier's `test()` method (line 40). These are collected into a list and returned to the variable `result` when all processes have completed. Finally, in lines 19 and 20, the mean and standard deviation are calculated from the 10 train/test combinations and printed. For example, with an four-core, eight thread CPU (no GPU) and eight IPython engines training a deep neural network with two hidden layers over 1000 epochs:

```
run scripts/crossvalidate -p [1,2,3,4,5] -a 6 -L [10,10]
 -e 1000  imagery/AST_20070501_pca.tif imagery/train.shp
10-fold cross validation, algorithm: Dnn(Tensorflow)
reading training data...
7162 training pixel vectors were read in
attempting parallel calculation ...
available engines [0, 1, 2, 3, 4, 5, 6, 7]
execution time: 877.8633694648743
misclassification rate: 0.044403
standard deviation:     0.005660
```

Whereas running the same test without a cluster increases the computation time on the same hardware by about a factor of three:

```
10-fold cross validation, algorithm: Dnn(Tensorflow)
reading training data...
7162 training pixel vectors were read in
attempting parallel calculation ...
failed, running sequentially ...
```

Listing 7.1: Cross-validation with IPython engines (excerpt from the script `crossvalidate.py`).

```
    #  cross-validation
    start = time.time()
    traintest = []
    for i in range(10):
        sl = slice(i*m//10,(i+1)*m//10)
        traintest.append(
            (np.delete(Gs,sl,0),np.delete(ls,sl,0), \
            Gs[sl,:],ls[sl,:],L,epochs,trainalg) )
    try:
        print('attempting parallel calculation ...')
        c = Client()
        print('available engines %s'%str(c.ids))
        v = c[:]
        result = v.map_sync(crossvalidate,traintest)
    except Exception as e:
        print('%s \nfailed, running sequentially ...'%e)
        result = list(map(crossvalidate,traintest))
    print('execution time: %s' %str(time.time()-start))
    print('misclassification rate: %f' %np.mean(result))
    print('standard deviation:      %f' %np.std(result))

def crossvalidate(slice):
    Gstrn,lstrn,Gstst,lstst,L,epochs,trainalg = slice
    import auxil.supervisedclass as sc
    if  trainalg == 1:
        classifier = sc.Maxlike(Gstrn,lstrn)
    elif trainalg == 2:
        classifier = sc.Gausskernel(Gstrn,lstrn)
    elif trainalg == 3:
        classifier = sc.Ffnbp(Gstrn,lstrn,L,epochs)
    elif trainalg == 4:
        classifier = sc.Ffncg(Gstrn,lstrn,L,epochs)
    elif trainalg == 5:
        classifier = sc.Ffnekf(Gstrn,lstrn,L,epochs)
    elif trainalg == 6:
        classifier = sc.Dnn_keras(Gstrn,lstrn,L,epochs)
    elif trainalg == 7:
        classifier = sc.Svm(Gstrn,lstrn)
    if classifier.train() is not None:
        return classifier.test(Gstst,lstst)
    else:
        return None
```

```
execution time: 2511.2689793109894
misclassification rate: 0.045377
standard deviation:     0.007194
```

7.2.4 Model comparison

A good value for a misclassification rate is $\theta \approx 0.05$. In order to claim that two rates produced by two different classifiers differ from one another significantly, a rule of thumb is that they should lie at least two standard deviations apart. A commonly used heuristic (van Niel et al., 2005) is to choose a minimum of $n \approx 30 \cdot N \cdot K$ training samples in all, where, as always, N is the data dimensionality and K is the number of classes. For the ASTER PCA image classification examples using $N = 4$ principal components and $K = 10$ classes, this gives $n \approx 30 \cdot 4 \cdot 10 = 1200$. If one third are to be reserved for testing, the number should be increased to $1200 \cdot 3/2 = 1800$. But suppose we wish to claim, for instance, that misclassification rates 0.05 and 0.06 are significantly different. Then, according to our thumb rule, their standard deviations should be no greater than 0.005. From Equation (7.11), this means $0.05(1-0.05)/n \approx 0.005^2$, or $n \approx 2000$ observations are needed for testing alone. Since we are dealing with pixel data, this number of test observations (assuming sufficient training areas are available) is still realistic.

In order to decide whether classifier A is better than classifier B in a more precise manner, a hypothesis test must be formulated. The individual misclassifications Y_A and Y_B are, as we have seen, approximately normally distributed. If they were also independent, then the test statistic

$$S = \frac{Y_A - Y_B}{\sqrt{\text{var}(Y_A) + \text{var}(Y_B)}}$$

would be standard normally distributed under the null hypothesis $\theta_A = \theta_B$. In fact, the independence of the misclassification rates is not given, since they are determined with the same set of test data.

There exist computationally expensive alternatives. The buzzwords here are cross-validation, as discussed in the preceding section, and *bootstrapping*; see Weiss and Kulikowski (1991), Chapter 2, for an excellent introduction. As an example, suppose that a series of p trials is carried out. In the ith trial, the training data are split randomly into training and test sets in the ratio 2:1 as before. Both classifiers are then trained and tested on these sets to give misclassifications represented by the random variables Y_A^i and Y_B^i. If the differences $Y_i = Y_A^i - Y_B^i$, $i = 1 \dots p$, are independent and normally distributed, then a Student-t statistic can be constructed to test the null hypothesis that the mean numbers of misclassifications are equal:

$$T = \frac{\bar{Y}}{\sqrt{S/p}},$$

where

$$\bar{Y} = \frac{1}{p}\sum_{i=1}^{p} Y_i, \quad S = \frac{1}{p-1}\sum_{i=1}^{p}(Y_i - \bar{Y})^2.$$

The statistic T is Student-t distributed with $p-1$ degrees of freedom; see Equation (2.80).

There are some objections to this approach, the most obvious again being the need to repeat the training/test cycle many times (typically $p \approx 30$). Moreover, as Dietterich (1998) points out in a comparative investigation of several such test procedures, Y_i is not normally distributed since Y_A^i and Y_B^i are not independent. He considers a *nonparametric* hypothesis test which avoids these problems and which we shall adopt here; see also Ripley (1996).

After training of the two classifiers which are to be compared, the following events for classification of the test data can be distinguished:

$$\bar{A}B,\ A\bar{B},\ \bar{A}\bar{B},\ \text{and}\ AB.$$

The event $\bar{A}B$ is *test observation is misclassified by A and correctly classified by B*, while $A\bar{B}$ is the event *test observation is correctly classified by A and misclassified by B* and so on. As before, we define random variables:

$$X_{\bar{A}B},\ X_{A\bar{B}},\ X_{\bar{A}\bar{B}}\ \text{and}\ X_{AB}$$

where

$$X_{\bar{A}B}(\bar{A}B) = 1, \quad X_{\bar{A}B}(A\bar{B}) = X_{\bar{A}B}(\bar{A}\bar{B}) = X_{\bar{A}B}(AB) = 0,$$

with mass function

$$\Pr(X_{\bar{A}B} = 1) = \theta_{\bar{A}B}, \quad \Pr(X_{\bar{A}B} = 0) = 1 - \theta_{\bar{A}B}.$$

Corresponding definitions are made for $X_{A\bar{B}}$, $X_{\bar{A}\bar{B}}$ and X_{AB}.

Now, in comparing the two classifiers, we are interested in the events $\bar{A}B$ and $A\bar{B}$. If the number of the former is significantly smaller than the number of the latter, then A is better than B and vice versa. Events $\bar{A}\bar{B}$ in which *both* methods perform poorly are excluded.

For n test observations, the random variables

$$\begin{aligned} Y_{\bar{A}B} &= X_{\bar{A}B_1} + \ldots X_{\bar{A}B_n} \quad \text{and} \\ Y_{A\bar{B}} &= X_{A\bar{B}_1} + \ldots X_{A\bar{B}_n} \end{aligned}$$

are the frequencies of the respective events. We then have

$$\begin{aligned} \langle Y_{\bar{A}B}\rangle &= n\theta_{\bar{A}B}, \quad \mathrm{var}(Y_{\bar{A}B}) = n\theta_{\bar{A}B}(1-\theta_{\bar{A}B}) \\ \langle Y_{A\bar{B}}\rangle &= n\theta_{A\bar{B}}, \quad \mathrm{var}(Y_{A\bar{B}}) = n\theta_{A\bar{B}}(1-\theta_{A\bar{B}}). \end{aligned}$$

We expect that $\theta_{\bar{A}B} \ll 1$, that is, $\mathrm{var}(Y_{\bar{A}B}) \approx n\theta_{\bar{A}B} = \langle Y_{\bar{A}B}\rangle$. The same holds for $Y_{A\bar{B}}$. It follows that the random variables

$$\frac{Y_{\bar{A}B} - \langle Y_{\bar{A}B}\rangle}{\sqrt{\langle Y_{\bar{A}B}\rangle}} \quad \text{and} \quad \frac{\mathrm{Y_{A\bar{B}}} - \langle \mathrm{Y_{A\bar{B}}}\rangle}{\sqrt{\langle \mathrm{Y_{A\bar{B}}}\rangle}}$$

are approximately standard normally distributed.

Under the null hypothesis (equivalence of the two classifiers), the expectation values of $Y_{\bar{A}B}$ and $Y_{A\bar{B}}$ satisfy

$$\langle Y_{\bar{A}B}\rangle = \langle Y_{A\bar{B}}\rangle =: \langle Y\rangle.$$

We form the *McNemar test statistic*

$$S = \frac{(Y_{\bar{A}B} - \langle Y\rangle)^2}{\langle Y\rangle} + \frac{(Y_{A\bar{B}} - \langle Y\rangle)^2}{\langle Y\rangle}, \tag{7.18}$$

which is chi-square distributed with one degree of freedom; see Section 2.1.5 and, e.g., Siegel (1965). Let $y_{\bar{A}B}$ and $y_{A\bar{B}}$ be the number of events actually measured. Then the mean $\langle Y\rangle$ is estimated as

$$\langle\hat{Y}\rangle = \frac{y_{\bar{A}B} + y_{A\bar{B}}}{2}$$

and a realization of the test statistic S is therefore

$$s = \frac{(y_{\bar{A}B} - \frac{y_{\bar{A}B}+y_{A\bar{B}}}{2})^2}{\frac{y_{\bar{A}B}+y_{A\bar{B}}}{2}} + \frac{(y_{A\bar{B}} - \frac{y_{\bar{A}B}+y_{A\bar{B}}}{2})^2}{\frac{y_{\bar{A}B}+y_{A\bar{B}}}{2}}.$$

With a little algebra, this expression can be simplified to

$$s = \frac{(y_{\bar{A}B} - y_{A\bar{B}})^2}{y_{\bar{A}B} + y_{A\bar{B}}}. \tag{7.19}$$

A correction is usually made to Equation (7.19), writing it in the form

$$s = \frac{(|y_{\bar{A}B} - y_{A\bar{B}}| - 1)^2}{y_{\bar{A}B} + y_{A\bar{B}}}, \tag{7.20}$$

which takes into approximate account the fact that the statistic is discrete, while the chi-square distribution is continuous. From the percentiles of the chi-square distribution, the critical region for rejection of the null hypothesis of equal misclassification rates at the 5% significance level is $s \geq 3.841$. The Python script `mcnemar.py` (Appendix C) compares two classifiers on the basis of their test result files, printing out $y_{\bar{A}B}$, $y_{A\bar{B}}$, s and the P-value $1 - P_{\chi^2;1}(s)$. Here is an example comparing the SVM and a neural network classifiers for the ASTER scene: The neural network with two hidden layers was trained for 500 epochs:

```
%run scripts/mcnemar \
imagery/AST_20070501_pca_SVM.tst \
imagery/AST_20070501_pca_Dnn(tensorflow).tst

=========================
     McNemar test
=========================
first classifier:
SVMtest results for imagery/AST_20070501_pca.tif
Sun Apr 21 17:54:10 2024
Classification image: imagery/AST_20070501_pca_class.tif
Class probabilities image: None

second classifier:
Dnn(tensorflow)test results for imagery/AST_20070501_
  pca.tif
Sun Apr 21 17:52:31 2024
Classification image: imagery/AST_20070501_pca_class.tif
Class probabilities image: None

test observations: 2364
classes: 10
first classifier: 98
second classifier: 86
McNemar statistic: 0.782609
P-value: 0.376344
```

In this case, although the neural network seems “better,” the null hypothesis cannot be rejected at the 5% significance level, whereas comparing the Gauss kernel classifier with the neural network, we conclude that the Gauss kernel is indeed better at 5% significance:

```
--------------------------
run scripts/mcnemar \
imagery/AST_20070501_pca_Gausskernel.tst \
imagery/AST_20070501_pca_Dnn(tensorflow).tst
          ...
test observations: 2364
classes: 10
first classifier: 54
second classifier: 91
McNemar statistic: 9.441379
P-value: 0.002121
```

7.3 Ensembles

Further enhancement of classifier accuracy is sometimes possible by combining several classifiers into an *ensemble* or *committee* and then applying some kind of voting scheme to generalize to new data. An excellent introduction to ensemble-based systems is given by Polikar (2006). The basic idea is to generate several classifiers and pool them in such a way as to improve on the performance of any single one. This implies that the pooled classifiers make errors on *different* observations, implying further that each classifier be as unique as possible, particularly with respect to misclassified instances (Polikar, 2006). One way of achieving this uniqueness is to use different training sets for each classifier, for example, by re-sampling the training data with replacement, a procedure referred to as "bootstrap aggregation" or *bagging* (Breiman, 1996); see Section 7.3.2 below and Exercise 2.

Representative of ensemble methods, we consider first a powerful technique called *adaptive boosting* or *AdaBoost* for short (Freund and Shapire, 1996), and then an example of a *random forest* classifier.

7.3.1 Adaptive boosting

Adaboost involves training a sequence of classifiers, placing increasing emphasis on hard-to-classify data, and then combining the sequence so as to reduce the overall training error. AdaBoost was originally suggested for combining binary classifiers, i.e., for two-class problems. However, Freund and Shapire (1997) proposed two multiclass extensions, the more commonly used of which is *AdaBoost.M1*. In the following, we shall apply AdaBoost.M1 to an ensemble of neural network classifiers. For other examples of adaptive boosting of neural networks, see Schwenk and Bengio (2000) and Murphey et al. (2001).

In order to motivate the adaptive boosting idea, consider the training of the feed-forward neural network classifier of Chapter 6 when there are just two classes to choose between. Making use of stochastic training, we train the network by minimizing the local cost function, Equation (6.34), on randomly selected labeled examples. To begin with the training data are sampled uniformly, as is done, for example, in the back propagation training algorithm of Listing 6.4. We can represent such a sampling scheme with the uniform, discrete probability distribution

$$p_1(\nu) = 1/m, \quad \nu = 1 \dots m,$$

over the m training examples. Let U_1 be the set of incorrectly classified examples after completion of the training procedure. Then the classification error is given by

$$\epsilon_1 = \sum_{\nu \in U_1} p_1(\nu).$$

Let us now find a new sampling distribution $p_2(\nu)$ such that the trained classifier would achieve an error of 50% if trained with respect to that distribution. In other words, it would perform as well as uninformed random guessing. The intention is, through the new distribution, to achieve a new classifier–training set combination which is as different as possible from the one just used. We obtain the new distribution $p_2(\nu)$ by reducing the probability for correctly classified examples by a factor $\beta_1 < 1$ so that the accuracy obtained is 1/2, that is,

$$\frac{1}{2} = \sum_{\nu \in U_1} p_2(\nu) = \sum_{\nu \notin U_1} p_2(\nu) = \frac{1}{Z} \sum_{\nu \notin U_1} \beta_1 p_1(\nu) = \frac{1}{Z} \beta_1 (1 - \epsilon_1). \tag{7.21}$$

The denominator Z is a normalization which ensures that $\sum_\nu p_2(\nu) = 1$ so that $p_2(\nu)$ is indeed a probability distribution,

$$Z = \sum_{\nu \in U_1} p_1(\nu) + \beta_1 \sum_{\nu \notin U_1} p_1(\nu) = \epsilon_1 + \beta_1(1 - \epsilon_1). \tag{7.22}$$

Combining Equations (7.21) and (7.22) gives

$$\beta_1 = \frac{\epsilon_1}{1 - \epsilon_1}. \tag{7.23}$$

If we now train the network with respect to $p_2(\nu)$, we will get (it is to be hoped) a different set U_2 of incorrectly classified training examples and, correspondingly, a different classification error

$$\epsilon_2 = \sum_{\nu \in U_2} p_2(\nu).$$

This leads to a new reduction factor β_2 and the procedure is repeated. The sequence must, of course, terminate at i classifiers when $\epsilon_{i+1} > 1/2$, as then the incorrectly classified examples can no longer be emphasized since $\beta_{i+1} > 1$.

At the generalization phase, the "importance" of each classifier is set to some function of β_i, the smaller β_i, the more important the classifier. As we shall see below, an appropriate weight is $\log(1/\beta_i)$. Thus, if C_k is the set of networks which classify feature vector $\boldsymbol{g}$ as k, then that class receives the "vote"

$$V_k = \sum_{i \in C_k} \log(1/\beta_i), \quad k = 1, 2,$$

after which $\boldsymbol{g}$ is assigned to the class with the maximum vote.

To place things on a more precise footing, we will define the *hypothesis* generated by the neural network classifier for input observation $\boldsymbol{g}(\nu)$ as

$$h(\boldsymbol{g}(\nu)) = h(\nu) = \arg\max_k (m_k(\boldsymbol{g}(\nu))), \quad \nu = 1 \dots m. \tag{7.24}$$

This is just the index of the output neuron whose signal is largest. For a two-class problem, $h(\nu) \in \{1, 2\}$. Suppose that $k(\nu)$ is the label of observation $\boldsymbol{g}(\nu)$. Following Freund and Shapire (1997), define the indicator

$$[[h(\nu) \neq k(\nu)]] = \begin{cases} 1 & \text{if the hypothesis } h(\nu) \text{ is incorrect} \\ 0 & \text{if it is correct.} \end{cases} \tag{7.25}$$

With this notation, we can give an exact formulation of the adaptive boosting algorithm for a sequence of neural networks applied to two-class problems.

Then we can prove a theorem on the upper bound of the overall training error for that sequence. Here, first of all, is the algorithm:

Algorithm (AdaBoost)

1. Define an initial uniform probability distribution $p_1(\nu) = 1/m$, $\nu = 1 \dots m$, and the number N_c of classifiers in the sequence. Define initial weights $w_1(\nu) = p_1(\nu)$, $\nu = 1 \dots m$.

2. For $i = 1 \dots N_c$ do the following:

   (a) Set $p_i(\nu) = w_i(\nu)/\sum_{\nu'=1}^m w_i(\nu')$, $\nu = 1 \dots m$.

   (b) Train a network with the sampling distribution $p_i(\nu)$ to get back the hypotheses $h_i(\nu)$, $\nu = 1 \dots m$.

   (c) Calculate the error $\epsilon_i = \sum_{\nu=1}^m p_i(\nu)[[h_i(\nu) \neq k(\nu)]]$.

   (d) Set $\beta_i = \epsilon_i/(1-\epsilon_i)$.

   (e) Determine a new weights w_{i+1} according to

   $$w_{i+1}(\nu) = w_i(\nu)\beta_i^{1-[[h_i(\nu)\neq k(\nu)]]},\ \nu = 1 \dots m.$$

3. Given an unlabeled observation $\boldsymbol{g}$, obtain the total vote received by each class,

   $$V_k = \sum_{\{i|h_i(\boldsymbol{g})=k\}} \log(1/\beta_i), \quad k = 1, 2,$$

   and assign $\boldsymbol{g}$ to the class with maximum vote.

The training error in the AdaBoost algorithm is the fraction of training examples that will be incorrectly classified when put into the voting procedure in step 3 above. We have the following theorem (the proof is given in Appendix A):

THEOREM 7.1
The training error ϵ for the algorithm AdaBoost is bounded above according to

$$\epsilon \leq 2^{N_c} \prod_{i=1}^{N_c} \sqrt{\epsilon_i(1-\epsilon_i)}. \tag{7.26}$$

This theorem tells us that, provided each classifier in the sequence can return an error $\epsilon_i < 1/2$, the training error will approach zero exponentially. It can be shown (Freund and Shapire, 1997) that the result is also valid for the multi class case $K > 2$. The boosting algorithm is then referred to as AdaBoost.M1.

A Python script `adaboost.py` for boosting neural networks trained with the fast Kalman filter algorithm of Appendix B is documented in Appendix C. An

Listing 7.2: Adaptive boosting of a neural network (excerpt from the script `adaboost.py`).

```
    errtrn = []
    errtst = []
    # initial probability distribution
    p = np.ones(mtrn)/mtrn
    # loop through the network instance
    start = time.time()
    instance = 1
    while instance<instances:
        trial = 1
        while trial < 6:
            print('running␣instance:␣%i␣␣trial:␣%i' \
                                        %(instance,trial))
            # instantiate a ffn and train it
            ffn = Ffnekfab(Xstrn,Lstrn,p,L,epochs)
            ffn.train()
            # determine beta
            labels,_ = ffn.classify(Xstrn)
            labels -= 1
            idxi = np.where(labels != labels_train)[0]
            idxc = np.where(labels == labels_train)[0]
            epsilon = np.sum(p[idxi])
            beta = epsilon/(1-epsilon)
            if beta < 1.0:
                # continue
                ffns.append(ffn)
                alphas.append(np.log(1.0/beta))
                # update distribution
                p[idxc] = p[idxc]*beta
                p = p/np.sum(p)
                # train error
                labels,_=seq_class(ffns,Xstrn,alphas,K)
                tmp=np.where(labels!=labels_train,1,0)
                errtrn.append(np.sum(tmp)/float(mtrn))
                # test error
                labels,_=seq_class(ffns,Xstst,alphas,K)
                tmp = np.where(labels!=labels_test,1,0)
                errtst.append(np.sum(tmp)/float(mtst))
                print('train␣error:␣%f␣test␣error:␣%f'\
                     %(errtrn[-1],errtst[-1]))
                # this instance is done
                trial = 6
                instance += 1
            else:
                trial += 1
```

ENVI/IDL version is given in Canty (2014). In the Python script, an excerpt from which is shown in Listing 7.2, a sequence of neural networks, $i = 1, 2, \ldots$, is trained on samples chosen with respect to distributions $p_1(\nu), p_2(\nu) \ldots$, the sequence terminating at i' when $\epsilon_{i'+1} \geq 1/2$ or when a maximum sequence length is reached. In order to take into account the fact that a network may become trapped in a local minimum of the cost function, training is restarted with a new random synaptic weight configuration if the current training error ϵ_i exceeds 1/2. The maximum number of restarts for a given network is five, after which the boosting terminates. The classifiers in the sequence are implemented in the object-oriented framework introduced in Chapter 6 as instances of a neural network object class. In summary, the algorithm is as follows:

Algorithm (Adaptive boosting of a sequence of neural network classifiers)

1. Set $p_1(\nu) = 1/m$, $\nu = 1 \ldots m$, where m is the number of observations in the set of labeled training data. Choose maximum sequence length N_{max}. Set $i = 1$.

2. Set $r = 0$.

3. Create a new neural network instance FFN(i) with random synaptic weights. Train FFN(i) with sampling distribution $p_i(\nu)$. Let U_i be the set of incorrectly classified training observations after completion of the training procedure.

4. Calculate $\epsilon_i = \sum_{\nu \in U_i} p_i(\nu)$. If $\epsilon_i < 1/2$, then continue, else if $r < 5$, then set $r = r + 1$, destroy the instance FFN(i), and go to 3, else stop.

5. Set $\beta_i = \epsilon_i/(1 - \epsilon_i)$ and update the distribution:

$$p_{i+1}(\nu) = \frac{p_i(\nu)}{Z_i} \times \begin{cases} \beta_i & \text{if } \nu \notin U_i \\ 1 & \text{otherwise} \end{cases}, \quad \nu = 1 \ldots m,$$

   where $Z_i = \sum_{\nu \in U_i} p_i(\nu) + \beta_i \sum_{\nu \notin U_i} p_i(\nu)$.

6. Set $i = i + 1$. If $i > N_{\max}$, then stop, else go to 2.

During the training phase, the program prints the training and generalization errors of the boosted sequence:

```
run scripts/adaboost -p [1,2,3,4,5]  -L [10] \
-n 75 imagery/AST_20070501_pca.tif  imagery/train.shp

Training with ADABOOST.M1 and 5 epochs per ffn
reading training data...
running instance: 1  trial: 1
train error: 0.034181 test error: 0.033841
running instance: 2  trial: 1
```

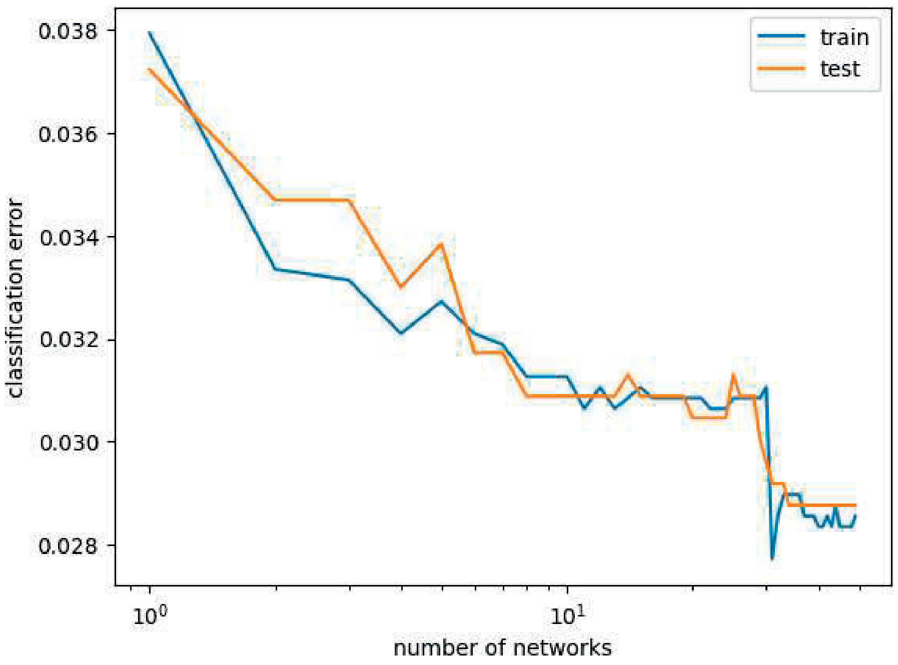


FIGURE 7.3
Adaptive boost training of an ensemble of neural networks.

```
train error: 0.031263 test error: 0.031303
running instance: 3  trial: 1
train error: 0.031055 test error: 0.029611
running instance: 4  trial: 1
train error: 0.030221 test error: 0.030034
running instance: 5  trial: 1
train error: 0.030221 test error: 0.030880
...
...
running instance: 43  trial: 1
running instance: 43  trial: 2
train error: 0.027303 test error: 0.029611
running instance: 44  trial: 1
train error: 0.027303 test error: 0.029611
running instance: 45  trial: 1
running instance: 45  trial: 2
train error: 0.027303 test error: 0.029188
running instance: 46  trial: 1
train error: 0.027095 test error: 0.029611
running instance: 47  trial: 1
train error: 0.027095 test error: 0.029611
running instance: 48  trial: 1
running instance: 48  trial: 2
running instance: 48  trial: 3
train error: 0.026886 test error: 0.029188
running instance: 49  trial: 1
train error: 0.026469 test error: 0.030034
elapsed time 1235.8035917282104
```

An example is shown in Figure 7.3 for the training/test datasets of the ASTER scene. Again five principal components were used for training, achieving a classification error on the test data of about 0.029. For a detailed comparison with other classifiers, see Canty (2009).

7.3.2 Binary decision trees and random forests

Classification and regression with *decision trees*, binary or otherwise, is a wide field; an introduction can be found in Géron (2023). In this text, we are dealing specifically with multispectral satellite imagery, and so we'll continue to use use the notation chosen thus far for classification algorithms. Training is carried out with respect to a set of representative multispectral pixel intensity vectors $\boldsymbol{g}$ and their respective class labels ℓ. With reference to the simple decision tree of Figure 7.4, the root node $\mathcal{T}_n,\ n = 1$, consists of the training dataset

$$\mathcal{T}_1 = \{\boldsymbol{g}(\nu), \ell(\nu)\}, \quad \nu = 1 \dots m_1, \tag{7.27}$$

where m_1 is the number of training observations, $\boldsymbol{g}(\nu)$ is the νth observation vector, and

$$\ell(\nu) \in \mathcal{K} = \{1 \dots K\} \tag{7.28}$$

is its class label, here simply an integer. Training consists of growing the binary tree from the root node. The initial objective is to split $\mathcal{T}_1$ into two subsets $\mathcal{T}_2^L$ and $\mathcal{T}_2^R$ according to a prescription which compares one (and only one) component of $\boldsymbol{g}$ to a threshold:

$$\begin{aligned} \mathcal{T}_2^L(\phi) &= \{(\boldsymbol{g}, \ell) \mid g_i \le t_1\} \\ \mathcal{T}_2^R(\phi) &= \mathcal{T}_1 \setminus \mathcal{T}_2^L(\phi). \end{aligned} \tag{7.29}$$

Here ϕ denotes the the candidate split $\phi = (i, t_1)$, i.e., the choice of the ith component of the observation vectors $\boldsymbol{g}(\nu)$ and a corresponding threshold t_1. The proportion of observations of class ℓ in the dataset $\mathcal{T}_1$ is given by

$$p_{1\ell} = \frac{1}{m_1} \sum_{\boldsymbol{\ell}' \in \mathcal{T}_1} \delta_{\ell'\ell}. \tag{7.30}$$

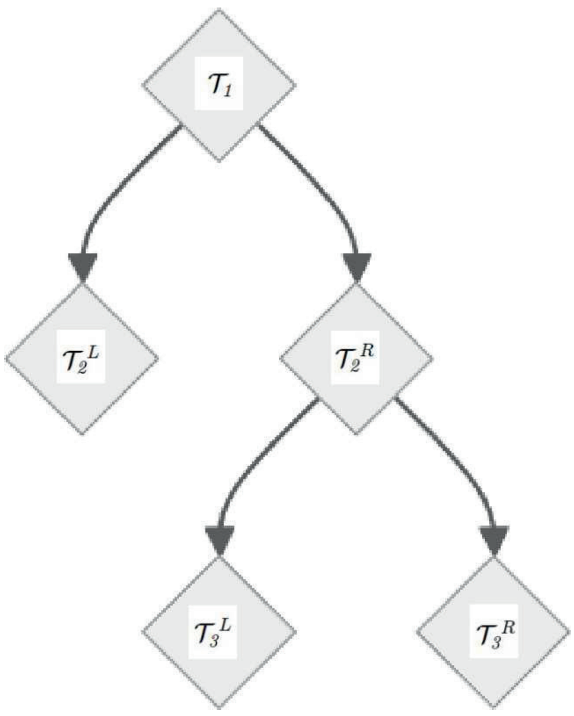


FIGURE 7.4
A binary decision tree with a root node, one internal node and three leaf nodes.

where $\delta_{\ell'\ell}$ is the Kroneker delta. A commonly used criterion for choosing ϕ is the so-called *Gini impurity*[*] which, for the root node $\mathcal{T}_1$, is given by

$$G(\mathcal{T}_1) = \sum_{\ell=1}^{K} p_{1\ell}(1 - p_{1\ell}). \tag{7.31}$$

If $\mathcal{T}_1$ were to consist of one class only then, trivially, the impurity is $G(\mathcal{T}_1) = 0$. This is of course not the case. In order to find the best branching from the root node, we determine the average impurity of the left- and right-hand subsets weighted according to the number of observations m_1^L and m_1^R in each subset

$$\hat{G}(\mathcal{T}_1, \phi) = \frac{m_1^L}{m_1} G(\mathcal{T}_2^L(\phi)) + \frac{m_1^R}{m_1} G(\mathcal{T}_2^R(\phi)), \tag{7.32}$$

[*]After the Italian statistician Corrado Gini (1884–1965).

and seek that split ϕ which minimizes it.[*] This determines the optimal component g_i of the observation vector and the threshold t_1 to which it is compared in Equation 7.29. As indicated in Figure 7.4 the procedure is recursed (in the Figure only on the subset $\mathcal{T}_2^R$), continuing until a stopping criterion is reached such as maximum tree depth or minimum subset population. Each leaf or terminal node of the tree is then labeled by the class with the maximum number of observations in that node. This algorithm is referred to as CART (Classification and Regression Tree).

In the prediction phase, unlabeled observations $\boldsymbol{g}$ are passed through the tree and classified according to the leaf node they reach. In a more or less balanced tree, the prediction time is very fast even for large training sets. See Géron (2023) for a more detailed discussion of the hyper parameters (regularization) used to control tree growth and avoid over fitting.

The definitive introduction of *random forests* into the arsenal of supervised classifiers was made in a paper by Leo Breiman in 2001, in which he describes a method for building an ensemble or 'forest" of uncorrelated decision tree classifiers trained using the above CART procedure. In its most general form, a random forest is

DEFINITION 7.1 *(Breiman, 2001) ... a classifier consisting of a collection of tree-structured classifiers* $h(\boldsymbol{g}, \Phi_j),\ \ j = 1, \ldots,$ *where the* Φ_j *are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input* $\boldsymbol{g}$.

For the jth tree, a random vector Φ_j is generated, independently of the past random vectors $\Phi_1 \ldots \Phi_{j-1}$ but with the same distribution. A tree is grown using the training set and Φ_j, resulting in a classifier $h(\boldsymbol{g}, \Phi_j)$ for input vector $\boldsymbol{g}$. While the predictions of a single tree are highly sensitive to noise in its training set, the average of many uncorrelated trees is not, thus reducing the variance.

To be specific, we can choose the binary decision tree just described as classifier and apply bagging to the ensemble. That is, train each classifier in the forest with a randomly chosen resampling with replacement vector Φ_j and then choose the most frequent prediction in the ensemble as the class of the input observation $\boldsymbol{g}$ (plurality vote). The algorithm is very suited to parallel computing, since each tree can be trained independently of the others in the ensemble. The same goes for the prediction phase.

As was the case for the support vector machine classifier of Chapter 6, convenient Python classes for decision trees and bagging are available in `sklearn`, namely `sklearn.tree.DecisionTreeClassifier()` and `sklearn.ensemble.BaggingClassifier()`. Listing 7.3 is a wrapper class `RF()` for random forest

[*]For each feature, the data points are sorted based on the feature values and splits between each pair of consecutive values are evaluated.

Listing 7.3: A class for random forest classification (excerpt from the Python module `supervisedclass.py`).

```python
class RF(object):
    '''Random Forest Classifier'''
    def __init__(self, Gs, ls, mnl=50, nest=500):
        self._Gs = Gs
        self._ls = np.argmax(ls, axis=1)
        self._clf = BaggingClassifier(
            DecisionTreeClassifier(max_leaf_nodes=mnl),
            n_estimators=nest, n_jobs=-1)

    def train(self):
        try:
            self._clf.fit(self._Gs, self._ls)
            return True
        except Exception as e:
            print('Error: %s' % e)
            return None

    def classify(self, Gs):
        classes = self._clf.predict(Gs) + 1
        probs = self._clf.predict_proba(Gs)
        return (classes, probs)
```

supervised classification, which is as usual called from the script `classify.py` to apply it to multispectral images.

There is a plethora of regularization parameters in both the decision tree as well as the bagging classifiers*; however, for illustration it is sufficient to accept most of the defaults (but see the Exercise 3). The random forest algorithm is selected with the command line parameter `-a 8`:

```
run scripts/classify -p [1,2,3,4,5] -a 8 \
imagery/AST_20070501_pca.tif imagery/train.shp

Training with RF
reading training data...
...
thematic map written to: imagery/AST_20070501_pca_class.
  tif
test results written to: imagery/AST_20070501_pca_RF.tst
done
```

After which we can compare the test results, for instance, with the Gauss kernel method of Chapter 6:

*https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble

```
%run scripts/mcnemar \
      imagery/AST_20070501_pca_Gausskernel.tst \
      imagery/AST_20070501_pca_RF.tst

=========================
     McNemar test
=========================
first classifier:
Gausskerneltest results for imagery/AST_20070501_pca.tif
Sun Apr 21 18:02:04 2024
Classification image: imagery/AST_20070501_pca_class.tif
Class probabilities image: None

second classifier:
RFtest results for imagery/AST_20070501_pca.tif
Mon Apr 22 17:15:55 2024
Classification image: imagery/AST_20070501_pca_class.tif
Class probabilities image: None

test observations: 2364
classes: 10
first classifier: 65
second classifier: 84
McNemar statistic: 2.422819
P-value: 0.119580
```

So at the 5% significance level it cannot be claimed that the Gauss kernel classifier performs better than the random forest with the chosen hyper parameters.

7.4 Classification of polarimetric SAR imagery

We saw in Chapter 6, Section 6.3, how to derive a Bayes maximum-likelihood classifier for normally distributed optical/infrared pixels. In the case of the fully polarimetric m-look SAR data discussed in Chapter 5, the image observations are expressed in complex covariance matrix form

$$\bar{\boldsymbol{c}} = \frac{1}{m}\boldsymbol{x},$$

where

$$\boldsymbol{x} = \sum_{\nu=1}^{m} \boldsymbol{s}(\nu)\boldsymbol{s}(\nu)^\dagger, \quad \boldsymbol{s} = (s_{hh}, \sqrt{2}s_{hv}, s_{vv})^T.$$

The corresponding random matrix $\boldsymbol{X}$, as was pointed out, will follow a complex Wishart distribution, Equation (2.62),

$$p_{W_c}(\boldsymbol{x}) = \frac{|\boldsymbol{x}|^{(m-N)} \exp(-\mathrm{tr}(\boldsymbol{\Sigma}^{-1}\boldsymbol{x}))}{\pi^{N(N-1)/2}|\boldsymbol{\Sigma}|^m \prod_{i=1}^N \Gamma(m+1-i)}, \tag{7.33}$$

where N is the dimension of the covariance matrix: $N = 3$ for quad, $N = 2$ for dual and $N = 1$ for single polarimetric SAR images.* Following exactly the same argument as in Chapter 6 (see also Exercise 4 and Lee et al. (1994)), we can derive a maximum-likelihood discriminant function for observations $\bar{\boldsymbol{c}}$, namely,

$$d_k(\bar{\boldsymbol{c}}) = \log(\Pr(k)) - m\left(\log|\boldsymbol{\Sigma}_k| + \mathrm{tr}(\boldsymbol{\Sigma}_k^{-1}\bar{\boldsymbol{c}})\right). \tag{7.34}$$

In the training phase, the class-specific complex covariance matrices $\boldsymbol{\Sigma}_k$, $k = 1 \dots K$, are estimated using pixels within selected training areas of the SAR image.

TABLE 7.1
Kappa values for SAR classification.

Polarization	Filter	Kappa	Sigma
Dual	None	0.468	0.006
Dual	MMSE	0.558	0.006
Quad	None	0.619	0.005
Quad	MMSE	0.666	0.005

From Equation (7.34), it is evident that the prior class membership probabilities, $\Pr(k)$, will play a smaller role in the classification when the number of looks is large. In fact, if we set all prior probabilities equal, then the discriminant is simply

$$d_k(\bar{\boldsymbol{c}}) = -\log|\boldsymbol{\Sigma}_k| - \mathrm{tr}(\boldsymbol{\Sigma}_k^{-1}\bar{\boldsymbol{c}}), \tag{7.35}$$

independent of the number of looks m. This independence holds provided that m is a global parameter for the entire image, which for look-averaged imagery is the case. However, if an adaptive filter, such as the MMSE filter of Chapter 5, is applied prior to classification, then m may vary somewhat from one land cover class to the next.

An example (calculated with an ENVI/IDL script; see Canty (2014)) is shown in Figure 7.5, where a quad polarimetric SAR image obtained with the EMISAR airborne sensor (Conradsen et al., 2003) is classified with and without prior adaptive filtering using the MMSE filter of Chapter 5.

*For $N = 1$, Equation (7.33) reduces to the gamma distribution.

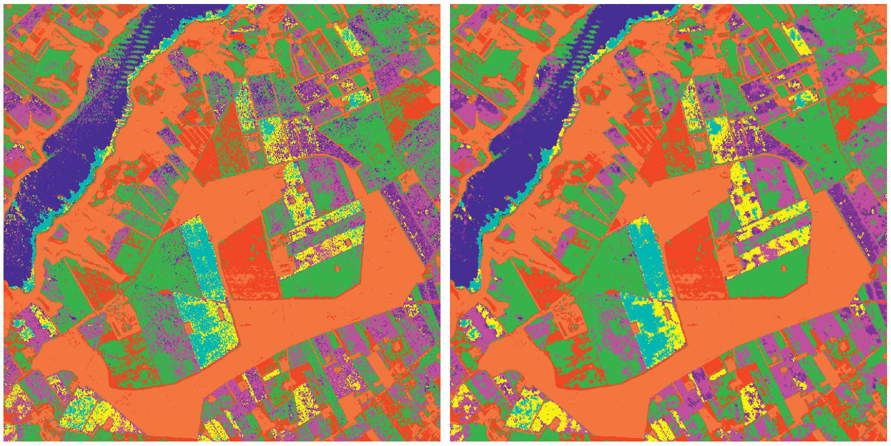


FIGURE 7.5
Maximum-likelihood classification of an EMISAR L-band quad polarimetric SAR image acquired over a test agricultural area in Denmark, left: without prior adaptive filtering, right: with prior adaptive filtering. The land use categories are: winter wheat (red), rye (green), water (blue), spring barley (yellow), oats (cyan), beets (magenta), peas (purple), coniferous forest (coral).

Qualitatively, the classification is seen to be improved by prior filtering. This is confirmed quantitatively in Table 7.1, which lists the Kappa coefficients calculated with the `ct.py` script described in Section 7.2.1. The table also shows reduced Kappa values for dual polarimetry,[*] confirming that classification accuracy improves significantly with the number of polarimetric channels.

7.5 Hyperspectral image analysis

Hyperspectral—as opposed to multispectral—images combine both high or moderate spatial resolution with high spectral resolution. Typical remote sensing imaging spectrometers generate in excess of two hundred spectral channels. Figure 7.6 shows part of a so-called *image cube* for the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) sensor taken over a region of the California coast. Figure 7.7 displays the spectrum of a single pixel in the image. Sensors of this kind produce much more complex data and provide

[*]In which case the observations are 2×2 matrices; see Equation (5.32).

correspondingly much more information about the reflecting surfaces examined than their multispectral counterparts.

The classification methods discussed in Chapter 6 and in the present chapter must in general be modified considerably in order to cope with the volume of data provided in a hyperspectral image. For example, the covariance matrix of an AVIRIS scene has dimension 224×224, so that the modeling of image or class probability distributions is much more difficult. Here one speaks of "ill-posed" classification problems, meaning that the available training data may be insufficient to estimate the model parameters adequately, and some sort of dimensionality reduction is needed (Richards, 2012). We will restrict discussion in the following to the concept of spectral unmixing, a method often used in lieu of "conventional" classification, and one of the most common techniques for hyperspectral image analysis.

7.5.1 Spectral mixture modeling

In multispectral image classification the fact that, at the scale of observation, a pixel often contains a mixture of land cover categories is generally treated as a second-order effect and more or less ignored. When working with hyperspectral imaging spectrometers, it is possible to treat the problem of the "mixed pixel" quantitatively. While one can still speak of classification of land surfaces, the training data now consist of external spectral libraries or, in some instances, reference spectra derived from within the images themselves. Comparison with external spectral libraries requires detailed attention to atmospheric correction. The end product is not the discrete labeling of the pixels that we have become familiar with, but consists rather of image planes or maps showing, at each pixel location, the proportion of surface material contributing to the observed reflectance.

The basic premise of mixture modeling is that, within a given scene, the surface is dominated by a small number of common materials that have characteristic spectral properties. These are referred to as the *end-members* and it is assumed that the spectral variability captured by the remote sensing system can be modeled by mixtures of end-members.

Suppose that there are K end-members and N spectral bands. Denote the spectrum of the ith end-member by the column vector

$$\boldsymbol{m}^i = (m^i_1, m^i_2 \dots m^i_N)^\top$$

and the matrix of end-member spectra $\boldsymbol{M}$ by

$$\boldsymbol{M} = (\boldsymbol{m}^1 \dots \boldsymbol{m}^K) = \begin{pmatrix} m^1_1 & \dots & m^K_1 \\ \vdots & \ddots & \vdots \\ m^1_N & \dots & m^K_N \end{pmatrix},$$

with one column for each end-member. For hyperspectral imagery we always have $K \ll N$, unlike the situation for multispectral data where $K \approx N$.

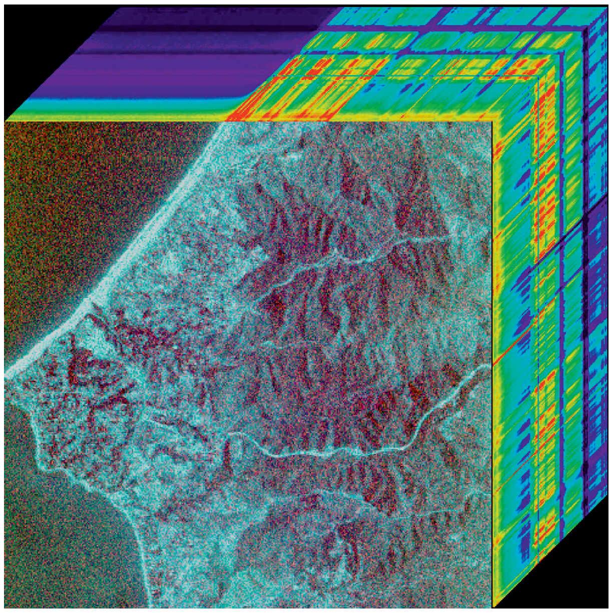


FIGURE 7.6
AVIRIS hyperspectral image cube over the Santa Monica Mountains acquired on April 7, 1997 at a GSD of 20 m.

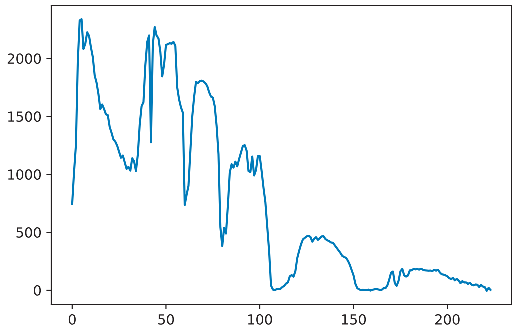


FIGURE 7.7
AVIRIS spectrum at one pixel location in Figure 7.6. There are 224 spectral bands covering the 0.4–1.5 μm wavelength interval.

The measured spectrum, represented by random vector $\boldsymbol{G}$, may be modeled as a linear combination of end-members plus a residual term $\boldsymbol{R}$ which is understood to be the variation in $\boldsymbol{G}$ not explained by the mixture model:

$$\boldsymbol{G} = \alpha_1 \boldsymbol{m}^1 + \ldots + \alpha_K \boldsymbol{m}^K + \boldsymbol{R} = \boldsymbol{M}\boldsymbol{\alpha} + \boldsymbol{R}. \tag{7.36}$$

The vector $\boldsymbol{\alpha} = (\alpha_1 \ldots \alpha_K)^\top$ contains non-negative mixing coefficients which are to be determined. Let us assume that the residual $\boldsymbol{R}$ is a normally distributed, zero mean random vector with covariance matrix

$$\boldsymbol{\Sigma}_R = \begin{pmatrix} \sigma_1^2 & 0 & \ldots & 0 \\ 0 & \sigma_2^2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \sigma_N^2 \end{pmatrix}.$$

The standardized residual is $\boldsymbol{\Sigma}_R^{-1/2}\boldsymbol{R}$ (why?) and the square of the standardized residual is

$$(\boldsymbol{\Sigma}_R^{-1/2}\boldsymbol{R})^\top(\boldsymbol{\Sigma}_R^{-1/2}\boldsymbol{R}) = \boldsymbol{R}^\top\boldsymbol{\Sigma}_R^{-1}\boldsymbol{R}. \tag{7.37}$$

The mixing coefficients $\boldsymbol{\alpha}$ may then be determined by minimizing this quantity with respect to the α_i under the condition that they sum to unity,

$$\sum_{i=1}^K \alpha_i = 1, \tag{7.38}$$

and are all non-negative,

$$\alpha_i \geq 0, \quad i = 1 \ldots K. \tag{7.39}$$

Ignoring the requirement of Equation (7.39) for the time being, a Lagrange function for minimization of Equation (7.37) under the constraint of Equation (7.38) is

$$\begin{aligned} L &= \boldsymbol{R}^\top\boldsymbol{\Sigma}_R^{-1}\boldsymbol{R} + 2\lambda(\sum_{i=1}^K \alpha_i - 1) \\ &= (\boldsymbol{G} - \boldsymbol{M}\boldsymbol{\alpha})^\top\boldsymbol{\Sigma}_R^{-1}(\boldsymbol{G} - \boldsymbol{M}\boldsymbol{\alpha}) + 2\lambda(\sum_{i=1}^K \alpha_i - 1), \end{aligned}$$

the last equality following from Equation (7.36). Solving the set of equations

$$\frac{\partial L}{\partial \boldsymbol{\alpha}} = 0, \quad \frac{\partial L}{\partial \lambda} = 0,$$

and replacing $\boldsymbol{G}$ by its realization $\boldsymbol{g}$, we obtain the estimates for the mixing coefficients (Exercise 5)

$$\begin{aligned} \hat{\boldsymbol{\alpha}} &= (\boldsymbol{M}^\top\boldsymbol{\Sigma}_R^{-1}\boldsymbol{M})^{-1}(\boldsymbol{M}^\top\boldsymbol{\Sigma}_R^{-1}\boldsymbol{g} - \lambda\boldsymbol{1}_K) \\ \hat{\boldsymbol{\alpha}}^\top\boldsymbol{1}_K &= 1, \end{aligned} \tag{7.40}$$

where $\mathbf{1}_K$ is a column vector of K ones. The first equation determines the mixing coefficients in terms of known quantities and λ. The second equation can be used to eliminate λ. Neglecting the constraint in Equation (7.39) is common practice. It can, however, be dealt with using appropriate numerical methods (Nielsen, 2001).

7.5.2 Unconstrained linear unmixing

If we work, for example, with MNF-transformed data (see Section 3.4), then we can assume that $\boldsymbol{\Sigma}_R = \boldsymbol{I}$.[*] If furthermore we ignore both of the constraints on $\boldsymbol{\alpha}$, Equations (7.38) and (7.39), which amounts to the assumption that the end-member spectra $\boldsymbol{M}$ are capable of explaining the observations completely apart from random noise, then Equation (7.40) reduces to the ordinary least squares estimate for $\boldsymbol{\alpha}$,

$$\hat{\boldsymbol{\alpha}} = [(\boldsymbol{M}^\top \boldsymbol{M})^{-1}\boldsymbol{M}^\top]\boldsymbol{g}. \tag{7.41}$$

The expression in square brackets is the pseudo inverse of the matrix $\boldsymbol{M}$ and the covariance matrix for $\boldsymbol{\alpha}$ is $\sigma^2(\boldsymbol{M}^\top \boldsymbol{M})^{-1}$, as is explained in Section 2.6.3. More sophisticated approaches, which are applicable when not all of the end-members are known, are discussed in the Exercises.

7.5.3 Intrinsic end-members and pixel purity

When a spectral library for all of the K end-members in $\boldsymbol{M}$ is available, the mixture coefficients can be calculated directly using the above methods. The primary product of the spectral mixture analysis consists of fraction images which show the spatial distribution and abundance of the end-member components in the scene. If such external data are unavailable, there are various strategies for determining end-members from the hyperspectral image itself.

For example, as a first step one can reduce the dimensionality of the data. This may be accomplished with the minimum noise fraction transformation described in Section 3.4.2:

```
run scripts/mnf.py imagery/97_radianz_registriert_voll

------------MNF ---------------
Mon Apr 29 11:57:23 2024
Input imagery/97_radianz_registriert_voll
Signal to noise ratios:
[8.176397   5.2275453  4.3195086  2.9181838  2.5331788
2.2690113
```

[*]This is at least the case for the MNF transformation of Section 3.4.1. If the PCA/MNF algorithm is used, Section 3.4.2, then the components of $\boldsymbol{R}$ must first be divided by the corresponding eigenvalues of the transformation.

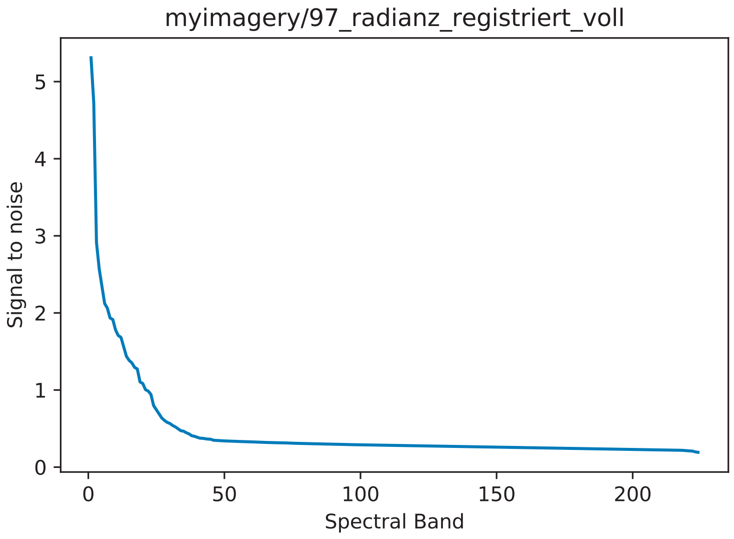


FIGURE 7.8
The signal to noise ratios for the MNF transformation of the image in Figure 7.6.

```
  2.195611    2.0360427   1.7489603   1.7093334   1.6091063
 1.5357301
  1.3793161   1.2976725   1.2197104   1.1656473   1.109391
 1.0812812
  ...
  ...
 MNFs written to: imagery/97_radianz_registriert_voll_mnf
 elapsed time: 7.130625486373901
```

By examining the signal-to-noise ratios of the transformation and retaining only the components with values exceeding some threshold, the number of dimensions can be reduced substantially; see Figure 7.8.

The so-called *pixel purity index* (PPI) may then be used to find the most spectrally pure, or extreme, pixels in the reduced feature space. The most spectrally pure pixels typically correspond to end-members. These pixels must be on the corners, edges or faces of the data cloud. The PPI is computed by repeatedly projecting n-dimensional scatter plots onto a random unit vector. The extreme pixels in each projection are noted and the number of times each pixel is marked as extreme is recorded. A threshold value is used to define how many pixels are marked as extreme at the ends of the projected vector. This value should be 2 to 3 times the variance in the data, which is 1 when using the MNF-transformed bands. A minimum of about 5000 iterations is usually required to produce useful results.

When the iterations are completed, a PPI image is created in which the intensity of each pixel corresponds to the number of times that pixel was recorded as extreme. Bright pixels are generally end-members. The end-members, projected back onto image coordinates, also hint at locations and sites that could be visited for ground truth measurements, should that be feasible. This sort of "data-driven" analysis has both advantages and disadvantages. To quote Mustard and Sunshine (1999):

> This method is repeatable and has distinct advantages for objective analysis of a data set to assess the general dimensionality and to define end-members. The primary disadvantage of this method is that it is fundamentally a statistical approach dependent on the specific spectral variance of the scene and its components. Thus the resulting end-members are mathematical constructs and may not be physically realistic.

7.5.4 Anomaly detection: The RX algorithm

The highly resolved spectral information provided by hyperspectral imagery has led to its frequent use in so-called *target detection*, the discovery of small-scale features of interest, most often of military or law enforcement relevance. Target detection typically involves two steps (Kwon and Nasrabadi, 2005): First, localized spectral anomalies are pinpointed by an unsupervised filtering procedure. Second, the significance of each identified anomaly, i.e., whether or not it is a target, is ascertained. The latter step usually involves comparison with known spectral signatures. In the following we outline a well-known procedure for carrying out the first step, the *RX anomaly detector* proposed by Reed and Yu (1990). It has been extensively used in the context of target detection and is equally applicable to multispectral imagery. Theiler and Matsekh (2009) discuss the algorithm in the context of anomalous change detection.

By referring to the likelihood ratio test introduced in Section 2.5, a derivation of the RX algorithm is straightforward. Consider a local neighborhood, e.g., a rectangular window within a multi- or hyperspectral image, and distinguish the central pixel as a possible anomaly. Let $\boldsymbol{g}(\nu)$, $\nu = 1 \dots m$, denote the observed pixel vectors in the background neighborhood and $\boldsymbol{g}(m+1)$ the central pixel. We construct the following null and alternative hypotheses:

$$\begin{aligned} &H_0 : \boldsymbol{g}(\nu) \sim \mathcal{N}(\boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b),\ \ \nu = 1 \dots m+1, \\ &H_1 : \boldsymbol{g}(\nu) \sim \mathcal{N}(\boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b),\ \ \nu = 1 \dots m, \quad \boldsymbol{g}(m+1) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}_b). \end{aligned} \tag{7.42}$$

Thus the null, or no-anomaly, hypothesis is that all of the observations in the window are uniformly sampled from a multivariate normal distribution with mean $\boldsymbol{\mu}_b$ and covariance matrix $\boldsymbol{\Sigma}_b$. The alternative hypothesis states that the central pixel is characterized by the *same* covariance matrix but by a different

Listing 7.4: Anomaly detection (excerpt from the script `rx.py`).

```
    gdal.AllRegister()
    infile = args[0]
    path = os.path.dirname(infile)
    basename = os.path.basename(infile)
    root, ext = os.path.splitext(basename)
    outfile = path+'/'+root+'_rx'+ext
    print('------------␣RX␣---------------')
    print(time.asctime())
    print('Input␣%s'%infile)
    start = time.time()
#   input image, convert to ENVI format
    inDataset = gdal.Open(infile,GA_ReadOnly)
    cols = inDataset.RasterXSize
    rows = inDataset.RasterYSize
    projection = inDataset.GetProjection()
    geotransform = inDataset.GetGeoTransform()
    driver = gdal.GetDriverByName('ENVI')
    enviDataset = driver.CreateCopy('imagery/entmp',
                                     inDataset)
    inDataset = None
    enviDataset = None
#   RX-algorithm
    img = envi.open('imagery/entmp.hdr')
    arr = img.load()
    rx = RX(background=calc_stats(arr))
    res = rx(arr)
#   output
    driver = gdal.GetDriverByName('GTiff')
    outDataset = driver.Create(outfile,cols,rows,1,\
                                  GDT_Float32)
    if geotransform is not None:
        outDataset.SetGeoTransform(geotransform)
    if projection is not None:
        outDataset.SetProjection(projection)
    outBand = outDataset.GetRasterBand(1)
    outBand.WriteArray(np.asarray(res,np.float32),0,0)
    outBand.FlushCache()
```

mean vector $\boldsymbol{\mu}$. In the notation of Definition 2.7, the maximized likelihoods are given by

$$\max_{\theta\in\omega_0} L(\theta) = \prod_{\nu=1}^{m+1} \exp\left(-\frac{1}{2}(\boldsymbol{g}(\nu)-\hat{\boldsymbol{\mu}}_b)^\top \hat{\boldsymbol{\Sigma}}_b^{-1}(\boldsymbol{g}(\nu)-\hat{\boldsymbol{\mu}}_b)\right)$$

and by

$$\max_{\theta\in\omega} L(\theta) = \prod_{\nu=1}^{m} \exp\left(-\frac{1}{2}(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_b)^\top \hat{\boldsymbol{\Sigma}}_b^{-1}(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_b)\right)$$
$$\cdot \exp\left(-\frac{1}{2}(\boldsymbol{g}(m+1) - \hat{\boldsymbol{\mu}})^\top \hat{\boldsymbol{\Sigma}}_b^{-1}(\boldsymbol{g}(m+1) - \hat{\boldsymbol{\mu}})\right),$$

where $\hat{\boldsymbol{\mu}}_b$ and $\hat{\boldsymbol{\Sigma}}_b$ are the maximum-likelihood estimates of the mean and covariance matrix of the background. But $\hat{\boldsymbol{\mu}} = \boldsymbol{g}(m+1)$ (there is only one observation), so the last exponential factor in the above equation is unity. Therefore, the likelihood ratio test has the critical region

$$Q = \frac{\max_{\theta\in\omega_0} L(\theta)}{\max_{\theta\in\omega} L(\theta)} = \exp\left(-\frac{1}{2}(\boldsymbol{g}(m+1) - \hat{\boldsymbol{\mu}}_b)^\top \hat{\boldsymbol{\Sigma}}_b^{-1}(\boldsymbol{g}(m+1) - \hat{\boldsymbol{\mu}}_b)\right) \le k.$$

Thus we reject the null hypothesis (that no anomaly is present) for central observation $\boldsymbol{g} = \boldsymbol{g}(m+1)$ when the squared Mahalanobis distance

$$d = (\boldsymbol{g} - \hat{\boldsymbol{\mu}}_b)^\top \hat{\boldsymbol{\Sigma}}_b^{-1}(\boldsymbol{g} - \hat{\boldsymbol{\mu}}_b) \tag{7.43}$$

exceeds some threshold. This distance serves as the RX anomaly detector. In practice, quite good results can be obtained with a global, rather than local, estimate of the background statistical parameters $\hat{\boldsymbol{\mu}}_b$ and $\hat{\boldsymbol{\Sigma}}_b$.

A simple Python implementation using the `Spy` package* and allowing only for global background statistics is shown in Listing 7.4; see also Appendix C. In lines 12–19, the image filename is entered and the image copied to ENVI standard format in order to be read into a `Spy ImageFile` object. This occurs in lines 23 and 24. In line 25, the RX anomaly detector is invoked to set the global background statistics, and the result (the image of squared Mahalanobis distances) is returned in line 26. An example is shown in Figure 7.9 at the end of this section.

7.5.5 Anomaly detection: The kernel RX algorithm

An improvement in anomaly detection might be expected if possible nonlinearities in the data are included in the model. A kernelized variant of the RX algorithm was suggested by Kwon and Nasrabadi (2005). To quote from their introduction:

> The conventional RX distance measure does not take into account the higher order relationships between the spectral bands at different wavelengths. The nonlinear relationships between different spectral bands within the target or clutter spectral signature need to be exploited in order to better distinguish between

*`http://www.spectralpython.net/index.html`.

> the two hypotheses. Furthermore the Gaussian assumption in the RX-algorithm for the distributions [under] the two hypotheses H_0 and H_1 in general is not valid.

Their derivation of the kernel RX algorithm is reproduced in the following, merely adapting it to our notation.

To begin with, we write the mapping of the terms in Equation (7.43) from the linear input space to a nonlinear feature space in the form

$$\begin{aligned} \boldsymbol{g} &\to \boldsymbol{\phi}(\boldsymbol{g}) \equiv \boldsymbol{\phi} \\ \hat{\boldsymbol{\mu}}_b &\to (\hat{\boldsymbol{\mu}}_\phi)_b \equiv \boldsymbol{\mu}_\phi \\ \hat{\boldsymbol{\Sigma}}_b &\to (\hat{\boldsymbol{\Sigma}}_\phi)_b \equiv \boldsymbol{\Sigma}_\phi. \end{aligned}$$

The definitions on the right serve to simplify the notation. The centered data matrix of m observations in the feature space is, see Equation (4.30),

$$\tilde{\boldsymbol{\Phi}} = \begin{pmatrix} \tilde{\boldsymbol{\phi}}(1)^\top \\ \vdots \\ \tilde{\boldsymbol{\phi}}(m)^\top \end{pmatrix},$$

where

$$\tilde{\boldsymbol{\phi}}(\nu) = \boldsymbol{\phi}(\nu) - \boldsymbol{\mu}_\phi,\ \nu = 1\dots m, \quad \boldsymbol{\mu}_\phi = \frac{1}{m}\sum_{\nu=1}^{m}\boldsymbol{\phi}(\nu), \tag{7.44}$$

and the $m \times m$ centered kernel matrix for the m observations is

$$\tilde{\mathcal{K}} = \tilde{\boldsymbol{\Phi}}\tilde{\boldsymbol{\Phi}}^\top,$$

which can be calculated with Equation (4.31). The Mahalanobis distance measure, Equation (7.43), expressed in the nonlinear feature space, is given by

$$d_\phi(\boldsymbol{g}) = (\boldsymbol{\phi} - \boldsymbol{\mu}_\phi)^\top \boldsymbol{\Sigma}_\phi^{-1}(\boldsymbol{\phi} - \boldsymbol{\mu}_\phi), \tag{7.45}$$

and we wish to express it purely in terms of kernel functions. Let's begin with the covariance matrix $\boldsymbol{\Sigma}_\phi$. It is given by the outer product

$$\boldsymbol{\Sigma}_\phi = \frac{1}{m}\sum_{\nu=1}^{m}(\boldsymbol{\phi}(\nu) - \boldsymbol{\mu}_\phi)(\boldsymbol{\phi}(\nu) - \boldsymbol{\mu}_\phi)^\top = \frac{1}{m}\tilde{\boldsymbol{\Phi}}^\top\tilde{\boldsymbol{\Phi}}. \tag{7.46}$$

Let $\boldsymbol{w}_\phi^i$, $i = 1\dots r$, be its first r eigenvectors in order of decreasing eigenvalue, and define

$$\boldsymbol{W}_\phi = (\boldsymbol{w}_\phi^1, \dots \boldsymbol{w}_\phi^r).$$

In general, we must assume that the eigenvalues of $\boldsymbol{\Sigma}_\phi$ beyond the rth one are effectively zero. Then the eigendecomposition of $\boldsymbol{\Sigma}_\phi$ is given by, see Equation (1.49),

$$\boldsymbol{\Sigma}_\phi = \boldsymbol{W}_\phi \boldsymbol{\Lambda} \boldsymbol{W}_\phi{}^\top,$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix of the first r eigenvalues of $\boldsymbol{\Sigma}_\phi$. We replace $\boldsymbol{\Sigma}_\phi^{-1}$ in Equation (7.45) by the pseudo inverse, Equation (1.50),

$$\boldsymbol{\Sigma}_\phi^+ = \boldsymbol{W}_\phi \boldsymbol{\Lambda}^{-1} \boldsymbol{W}_\phi{}^\top. \tag{7.47}$$

Recall that Equation (3.49) in Chapter 3 expressed the eigenvectors $\boldsymbol{w}_i$ of the covariance matrix in terms of the dual vectors $\boldsymbol{\alpha}_i$, which are eigenvectors of the centered Gram matrix. In the nonlinear space, we do the same, writing Equation (3.49) in the form (see also Equation (1.13))

$$\boldsymbol{w}_\phi^i = \sum_{\nu=1}^m (\boldsymbol{\alpha}_i)_\nu \tilde{\boldsymbol{\phi}}(\nu) = (\tilde{\boldsymbol{\phi}}(1), \dots \tilde{\boldsymbol{\phi}}(m))\boldsymbol{\alpha}_i = \tilde{\boldsymbol{\Phi}}^\top \boldsymbol{\alpha}_i, \quad i = 1 \dots r, \tag{7.48}$$

where the $\boldsymbol{\alpha}_i$ are now eigenvectors of the centered kernel matrix, that is,

$$\tilde{\mathcal{K}} \boldsymbol{\alpha}_i = \lambda_i \boldsymbol{\alpha}_i.$$

Now let us write Equation (7.48) in matrix form,

$$\boldsymbol{W}_\phi = \tilde{\boldsymbol{\Phi}}^\top \boldsymbol{\alpha}, \tag{7.49}$$

where

$$\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \dots \boldsymbol{\alpha}_r).$$

Substituting Equation (7.49) into Equation (7.47), we obtain

$$\boldsymbol{\Sigma}_\phi^+ = \tilde{\boldsymbol{\Phi}}^\top \boldsymbol{\alpha} \boldsymbol{\Lambda}^{-1} \boldsymbol{\alpha}^\top \tilde{\boldsymbol{\Phi}}$$

and accordingly, with Equation (7.45), the distance measure

$$d_\phi(\boldsymbol{g}) = (\boldsymbol{\phi} - \boldsymbol{\mu}_\phi)^\top \tilde{\boldsymbol{\Phi}}^\top \boldsymbol{\alpha} \boldsymbol{\Lambda}^{-1} \boldsymbol{\alpha}^\top \tilde{\boldsymbol{\Phi}} (\boldsymbol{\phi} - \boldsymbol{\mu}_\phi). \tag{7.50}$$

The eigenvalues of $\boldsymbol{\Sigma}_\phi$, which appear along the diagonal of $\boldsymbol{\Lambda}$, are in fact the same as those of the centered kernel matrix $\tilde{\mathcal{K}}$ except for a factor $1/m$. That is,

$$\boldsymbol{\Sigma}_\phi \boldsymbol{w}_i = \boldsymbol{\Sigma}_\phi \tilde{\boldsymbol{\Phi}}^\top \boldsymbol{\alpha}_i = \frac{1}{m} \tilde{\boldsymbol{\Phi}}^\top \tilde{\boldsymbol{\Phi}} \tilde{\boldsymbol{\Phi}}^\top \boldsymbol{\alpha}_1 = \frac{1}{m} \tilde{\boldsymbol{\Phi}}^\top \tilde{\mathcal{K}} \boldsymbol{\alpha}_1 = \frac{1}{m} \tilde{\boldsymbol{\Phi}}^\top \lambda_i \boldsymbol{\alpha}_1 = \frac{\lambda_i}{m} \boldsymbol{w}_i.$$

Hence, the eigen-decomposition of $\tilde{\mathcal{K}}$ is

$$\tilde{\mathcal{K}} = m \boldsymbol{\alpha} \boldsymbol{\Lambda} \boldsymbol{\alpha}^\top$$

and its pseudo inverse is

$$\tilde{\mathcal{K}}^+ = \frac{1}{m} \boldsymbol{\alpha} \boldsymbol{\Lambda}^{-1} \boldsymbol{\alpha}^\top. \tag{7.51}$$

We can therefore write the nonlinear anomaly detector, Equation (7.50), in the form

$$d_\phi(\boldsymbol{g}) = m(\boldsymbol{\phi} - \boldsymbol{\mu}_\phi)^\top \tilde{\boldsymbol{\Phi}}^\top \tilde{\mathcal{K}}^+ \tilde{\boldsymbol{\Phi}} (\boldsymbol{\phi} - \boldsymbol{\mu}_\phi).$$

The factor m is irrelevant and can be dropped. To complete the kernelization of this expression, consider the inner product

$$\begin{aligned}\boldsymbol{\phi}^\top \tilde{\boldsymbol{\Phi}}^\top &= \boldsymbol{\phi}^\top \big(\tilde{\boldsymbol{\phi}}(1)\dots\tilde{\boldsymbol{\phi}}(m)\big)\\ &= \boldsymbol{\phi}^\top \big((\boldsymbol{\phi}(1)\dots\boldsymbol{\phi}(m)) - \boldsymbol{\mu}_\phi\big)\\ &= \big((\boldsymbol{\phi}^\top\boldsymbol{\phi}(1))\dots(\boldsymbol{\phi}^\top\boldsymbol{\phi}(m))\big) - \frac{1}{m}\sum_{\nu=1}^m(\boldsymbol{\phi}^\top\boldsymbol{\phi}(\nu)),\end{aligned}$$

or, in terms of the symmetric kernel function $k(\boldsymbol{g},\boldsymbol{g}') = \boldsymbol{\phi}(\boldsymbol{g})^\top\boldsymbol{\phi}(\boldsymbol{g}')$,

$$\boldsymbol{\phi}^\top \tilde{\boldsymbol{\Phi}}^\top = \big(k(\boldsymbol{g}(1),\boldsymbol{g})\dots k(\boldsymbol{g}(m),\boldsymbol{g})\big) - \frac{1}{m}\sum_{\nu=1}^m k(\boldsymbol{g}(\nu),\boldsymbol{g}) =: \boldsymbol{\mathcal{K}}_g.$$

In a similar way (Exercise 7), one can show that

$$\begin{aligned}\boldsymbol{\mu}_\phi^\top\tilde{\boldsymbol{\Phi}}^\top = \frac{1}{m}\sum_{\nu=1}^m\big(k(\boldsymbol{g}(\nu),\boldsymbol{g}(1))\dots k(\boldsymbol{g}(\nu),\boldsymbol{g}(m))\big) - \frac{1}{m^2}\sum_{\nu,\nu'=1}^m k(\boldsymbol{g}(\nu),\boldsymbol{g}(\nu'))\\ =: \boldsymbol{\mathcal{K}}_\mu.\end{aligned} \tag{7.52}$$

The first term in $\boldsymbol{\mathcal{K}}_\mu$ is the row vector of the column averages of the uncentered kernel matrix, and the second term is the overall average of the uncentered kernel matrix elements. Combining, we have finally

$$d_\phi(\boldsymbol{g}) = (\boldsymbol{\mathcal{K}}_g - \boldsymbol{\mathcal{K}}_\mu)\tilde{\boldsymbol{\mathcal{K}}}^+(\boldsymbol{\mathcal{K}}_g - \boldsymbol{\mathcal{K}}_\mu)^\top \tag{7.53}$$

as our kerneled RX anomaly detector.

An excerpt from the Python script `krx.py` for kernel RX (Appendix C) is shown in Listing 7.5. After the usual preliminaries, sample observation vectors are placed in the array `G` (line 10) and the kernel matrix and its centered version are determined in lines 16 and 17. In order to determine the number r of non-zero eigenvalues, the centered kernel matrix is diagonalized and its eigenvectors and eigenvalues sorted in decreasing order in lines 20–23. Then r is determined as the number of eigenvalues exceeding the machine tolerance, lines 24 and 25. The pseudoinverse of the centered kernel matrix, Equation (7.51), is calculated in line 28. The row vector $\boldsymbol{\mathcal{K}}_\mu$ (Python/Numpy array `Ku`), which depends only on the kernel matrix, is first determined, lines 31 and 32. Then, in the `for` loop, the anomaly image (squared Mahalanobis distances, Equation (7.53)) is calculated row by row. Figure 7.9 shows an example comparing the RX and kernel RX detectors. Qualitatively, the kernelized method performs better.

Listing 7.5: Kernelized anomaly detection (excerpt from the script `krx.py`).

```
    #  image data matrix
    GG = np.zeros((rows*cols,bands))
    for b in range(bands):
        band = inDataset.GetRasterBand(b+1)
        GG[:,b] = band.ReadAsArray(0,0,cols,rows)\
                              .astype(float).ravel()
    inDataset = None
    #  random training data matrix
    idx = np.random.randint(0,rows*cols,size=m)
    G = GG[idx,:]
    #  KRX-algorithm
    print('------------ KRX ---------------')
    print(time.asctime())
    print('Input %s'%infile)
    start = time.time()
    K,gma=auxil.kernelMatrix(G, k=1)
    Kc = auxil.center(K)
    print('GMA: %f'%gma)
    #  pseudoinvert centered kernel matrix
    lam, alpha = np.linalg.eigh(Kc)
    idx = range(m)[::-1]
    lam = lam[idx]
    alpha = alpha[:,idx]
    tol = max(lam)*m*np.finfo(float).eps
    r = np.where(lam>tol)[0].shape[0]
    alpha = alpha[:,:r]
    lam = lam[:r]
    Kci = alpha*np.diag(1./lam)*alpha.T
    #  row-by-row anomaly image
    res = np.zeros((rows,cols))
    Ku = np.sum(K,0)/m - np.sum(K)/m**2
    Ku = np.mat(np.ones(cols)).T*Ku
    for i in range(rows):
        if i % 100 == 0:
            print('row: %i'%i )
        GGi = GG[i*cols:(i+1)*cols,:]
        Kg,_=auxil.kernelMatrix(GGi,G,gam=gma,k=1)
        a = np.sum(Kg,1)
        a = a*np.mat(np.ones(m))
        Kg = Kg - a/m
        Kgu = Kg - Ku
        d = np.sum(np.multiply(Kgu,Kgu*Kci),1)
        res[i,:] = d.ravel()
```

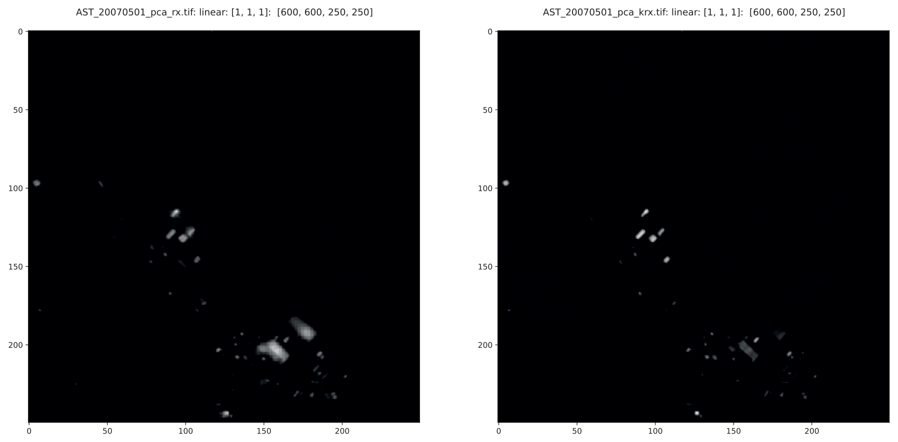

AST_20070501_pca_rx.tif: linear: [1, 1, 1]: [600, 600, 250, 250]
AST_20070501_pca_krx.tif: linear: [1, 1, 1]: [600, 600, 250, 250]

FIGURE 7.9
Anomaly detection: Left, spatial subset of the RX anomaly image calculated from all 9 bands of the ASTER image of Figure 6.1 using the script in Listing 7.4. Right, kernel RX anomaly image calculated with the script `krx.py`, Listing 7.5.

7.6 Convolutional neural networks

In Section 7.1 at the beginning of the present chapter, we pointed out that intermediate and high-resolution remote sensing imagery is characterized by a considerable degree of spatial correlation, and then we examined ways to include this information into the post processing of pixel-wise supervised classification approaches. A fairly obvious alternative approach is to attempt to identify characteristic landscape features directly from the observed multispectral data, and this, in the context of image analysis, is what convolutional neural networks (CNNs) are designed to do. CNNs consist of multiple layers that automatically and adaptively learn spatial hierarchies of features from input images. These layers perform convolutions, ReLU activations, pooling, and are often followed by the fully connected layers we are already familiar with. The convolutional layers apply filters to the input images to create feature maps that highlight specific features. Activation layers introduce nonlinearity, allowing the network to learn complex patterns. Pooling layers reduce the dimensionality of the data, simplifying the information without losing important features. Finally, fully connected layers aggregate the learned features to make predictions or classifications.

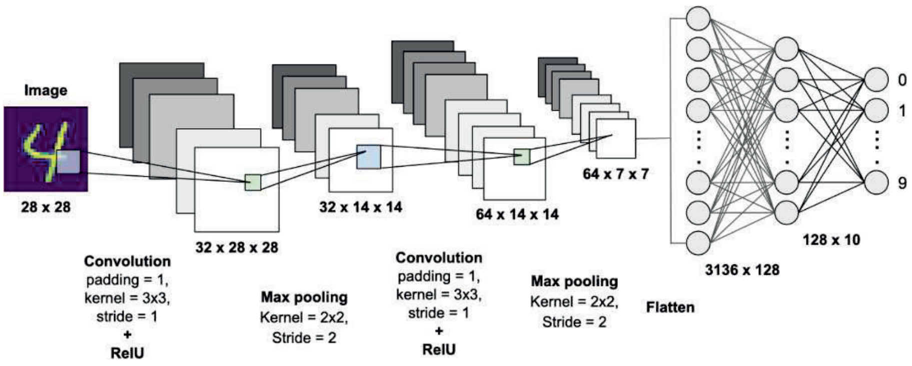


FIGURE 7.10
A convolutional neural network for handwritten digit recognition. The inputs are 28 × 28 pixel monochromatic images. (Reproduced with permission: https://krutpatel.medium.com/).

An illustration of this approach is given in Figure 7.10 showing a very typical CNN architecture for classification of the MNIST (LeCun, 2010) database of handwritten digits. As explained above, and evident from the Figure, two of the essential building blocks of the CNN are the convolutional layer and the pooling layer, and these behave in a fundamentally different way than the fully connected neuron layers we've seen so far. There are plenty of good explanations of CNNs in the literature and on the web, an excellent and thorough one can be found in Chapter 14 of Géron (2023). For our purposes, a walk through the structure of Figure 7.10 with the help of the Keras sequential model will be sufficient to cover the basics.

```
import tensorflow.keras as keras
```

The first convolutional layer in the Figure 7.10 consists of 32 *image maps*, arrays of 28 × 28 neurons. Each neuron has a *kernel* or *receptive field* of 3 × 3 pixels in the input image (which here has only one color channel). In Keras:

```
model = keras.Sequential()
model.add(keras.layers.Conv2D(filters=32,
                kernel_size=3,
                padding='same',
                activation='relu',
                input_shape=(28,28,1)))
```

All of the neurons in a given feature map share the same(!) set of weight parameters or *filter*, in this case of size $3 \times 3 \times 1 + 1$ for the receptive field plus bias. This drastically reduces the number of model parameters when compared to a fully connected network. Moreover, they all share in this example the same nonlinear ReLU output activation. The *stride* shown in the Figure 7.10 indicates the shift in receptive field seen by neighboring neurons and is here, by implication, one, since the feature maps have the same dimensions as the input array. The *padding* strategy chosen means that we needn't worry about edge effects.

Next comes a *pooling layer* with the same number of feature maps as the preceding convolutional layer. Here the receptive neuron field is 2×2, and the stride is 2 resulting in 32 pooled image maps of dimension 14×14. The inputs to the pooling layer neurons have no weights, the neurons merely serve to subsample the convolution outputs. Max pooling, used here, simply takes the maximum input from the receptive field as the neuron output passed along to the next layer. In the syntax of the Keras sequential model:

```
model.add(keras.layers.MaxPooling2D(pool_size=2))
```

The following convolutional layer is programmed to have 64 feature maps of size 14×14 (stride=1) and again a receptive field of size 3×3:

```
model.add(keras.layers.Conv2D(filters=64,
                kernel_size=3,
                padding='same',
                activation='relu'))
```

This implies that each feature map in the second convolutional layer has a filter of $3 \times 3 \times 32 + 1$ adjustable weights, or $64 \times (3 \times 3 \times 32 + 1) = 18,496$ in all. During training, when all intermediate values in the network are required for back propagation, CNN memory requirements can thus still be quite large.

The next pooling layer reduces the size to $7 \times 7 \times 64$, after which the outputs are flattened to a linear 3136 array:

```
model.add(keras.layers.MaxPooling2D(pool_size=2))
model.add(keras.layers.Flatten())
```

which in turn serves as input to a standard two-layer fully connected network with one output for each of the training characters, the digits 0 through 9:

```
model.add(keras.layers.Dense(128,activation='relu'))
model.add(keras.layers.Dense(10,activation='softmax'))
```

and we're done.

A drawback of applying a CNN like the one just described to remote sensing imagery is the necessity for very large amounts of labeled training data together with very high-dimensional networks and correspondingly many adjustable parameters. Parallel processing is therefore generally a prerequisite for reasonable training times.* In the following, we will examine two

*The reader with access to Google Drive and Colab can experiment with the above development in a GPU runtime and also follow the rest of this Section from the links given in the accompanying Jupyter Notebook.

approaches to remote sensing image segmentation with convolutional networks: so-called *transfer learning*, i.e., specialization of networks pretrained in another domain, and *fully convolutional networks* which learn and regenerate two-dimensional landscape features directly.

7.6.1 Transfer learning

Suppose we have trained the network of Figure 7.10 to recognize hand-written digits, but want to change the domain to images of fashion articles (shoes, sweaters, handbags and the like). Rather than starting from scratch, we might freeze the weights of the two convolutional layers and re-train only the final, flattened feed-forward part of the network on the new domain. The idea here is to try to take advantage the CNN's learned ability to recognize low level image structures and then to re-classify those structures with the FFN to the categories of interest. This, if successful, will reduce costs in computation time and memory.

Our domain is of course not fashion but satellite imagery. We will now demonstrate transfer learning with a EuroSAT image dataset* consisting of 27,000 labeled and geo-referenced Sentinel-2 image patches.

```
import tensorflow_datasets as tfds
(test_set , valid_set , train_set), info =
 tfds.load('eurosat/rgb',
 split = ['train[:10%]', 'train[10%:25]', 'train[25%:]'],
 as_supervised=True,
 with_info=True)
```

The code above downloads the RGB version of the data (consisting of Sentinel-2 bands b2, b3, b4 only) and splits them into test, validate and train sets in the ratio 10%:15%:75%. Examining the `info` attribute of the dataset, we see that the image patches are of size $64 \times 64 \times 3$ and that there are 10 class labels in all,

```
info.features
FeaturesDict({
 'filename': Text(shape=(), dtype=string),
 'image': Image(shape=(64, 64, 3), dtype=uint8),
 'label': ClassLabel(shape=(),dtype=int64,
   num_classes=10),
})

info.features['label'].names
['AnnualCrop',
 'Forest',
```

*https://www.tensorflow.org/datasets/catalog/eurosat

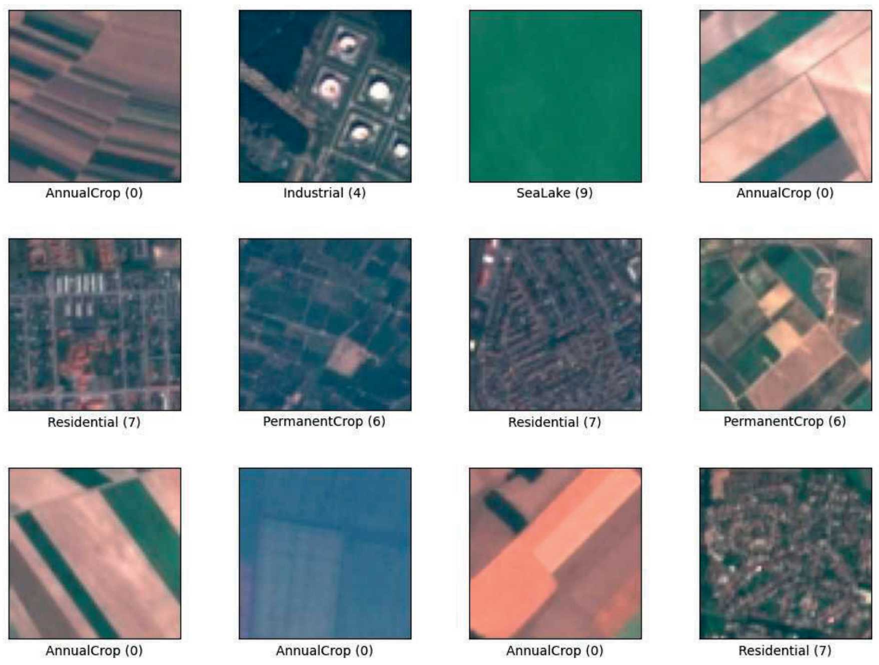


FIGURE 7.11
Training patches from the EuroSAT image dataset.

```
    'HerbaceousVegetation',
    'Highway',
    'Industrial',
    'Pasture',
    'PermanentCrop',
    'Residential',
    'River',
    'SeaLake']
```

A few random training patches can be viewed with

```
fig=tfds.show_examples(train_set, info, rows=3, cols=4)
```

and are shown in Figure 7.11.

Now for the base model. We choose the `Xception` application (Chollet, 2017) available from the TensorFlow website.* This is a so-called *depth wise separable* convolutional architecture and is explained in detail in the reference. Like our toy example, the final, or top, layer consists of a fully connected FFN with provisions for freezing the preceding CNN parameters.

*https://www.tensorflow.org/api_docs/python/tf/keras/applications/Xception

```
# download Xception (base) model without top layer
base_model = keras.applications.xception.
  Xception(weights='imagenet',include_top=False,
  input_shape=(224,224,3))

# add a pooling layer with input from the base model
# (averages over all output feature maps of the base
# model)
avg=keras.layers.GlobalAveragePooling2D()
  (base_model.output)

# add an output softmax layer with 10 neurons
# and softmax activation
output = keras.layers.Dense(10, activation='softmax')
  (avg)

# define the model
model=keras.Model(inputs=base_model.input,
  outputs=output)
```

The following points should be noted regarding the preceding listing:

- We built our toy network with the Keras sequential API but here we've used the more flexible *functional* API.
- The weights of the downloaded base model were determined with the IMAGENET dataset (https://image-net.org/).
- The original top layer is not included in the download but is replaced by a pooling layer averaging over the CNN outputs.
- A new top layer is added, a single array of 10 neurons fully connected to the preceding pooling layer with softmax activation matching the 10 land use classes of the EuroSAT training dataset.

The `Xception` model requires an input of 224×224 patches, so some preprocessing of the EuroSAT Sentinel-2 imagery is needed:

```
def preprocess(image,label):
   resized_image = tf.image.resize(image, [224,224])
   final_image = keras.applications.xception. \
                     preprocess_input(resized_image)
   return final_image, label

batch_size = 32

# Randomly shuffle the elements of the training dataset.
train_set = train_set.shuffle(1000)

# map the preprocessing function over the three datasets
```

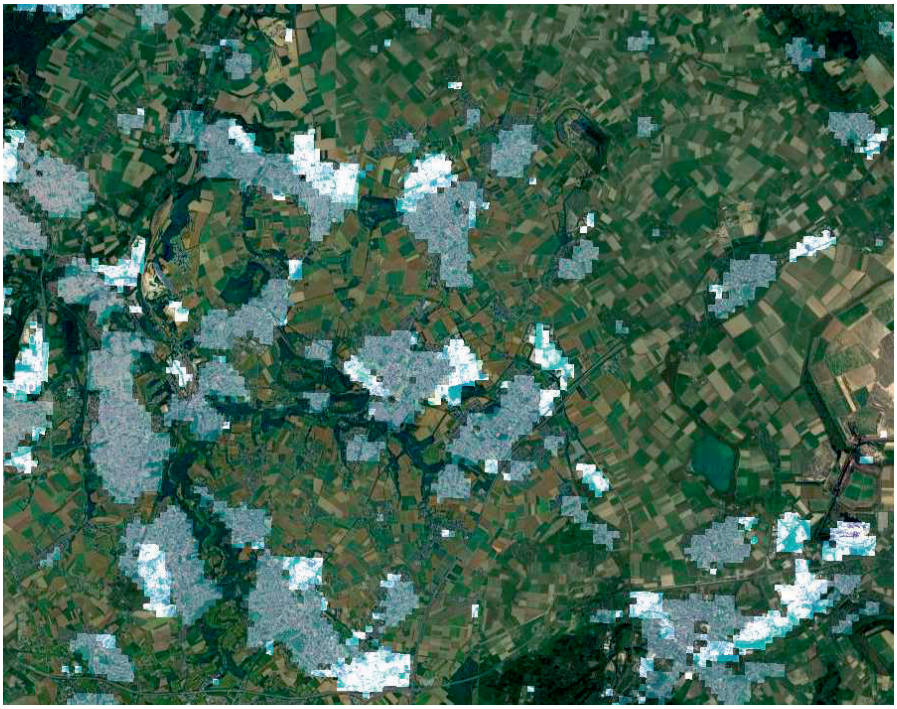


FIGURE 7.12
Land cover class predictions for a Sentinel-2 acquisition from May 8, 2018 north of Aachen, Germany with a 64×64 window moving in 8-pixel increments horizontally and vertically. Dark gray: residential, blue-white: industrial.

```
train_set = train_set.map(preprocess).batch(batch_size).
  \ prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).
  \ prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).
  \ prefetch(1)
```

For a GPU runtime, the prefetch method above makes the CPU prepare the next batch while the GPU is processing the current one. The next step is to define an optimization method, a loss function, and compile the model:

```
# scaled gradient descent optimizer
optimizer = keras.optimizers.SGD(learning_rate=0.2,
                                 momentum=0.9)

# cross entropy loss (for no one-hot labels encoding)
loss = "sparse_categorical_crossentropy"
```

```
# only display accuracy=overall correct/total examples
metrics = ['accuracy']
model.compile(loss=loss,optimizer=optimizer,
                                    metrics=metrics)
```

Now train the model, top layer weights only (and with a GPU runtime!):

```
history = model.fit(train_set, epochs=10,
                    validation_data=valid_set)

Epoch 1/10 ... accuracy: 0.8631 val_accuracy: 0.9064

Epoch 2/10 ... accuracy: 0.9133 val_accuracy: 0.9212

Epoch 3/10 ... accuracy: 0.9252 val_accuracy: 0.9244
...
Epoch 10/10 ... accuracy: 0.9513  val_accuracy: 0.9348
```

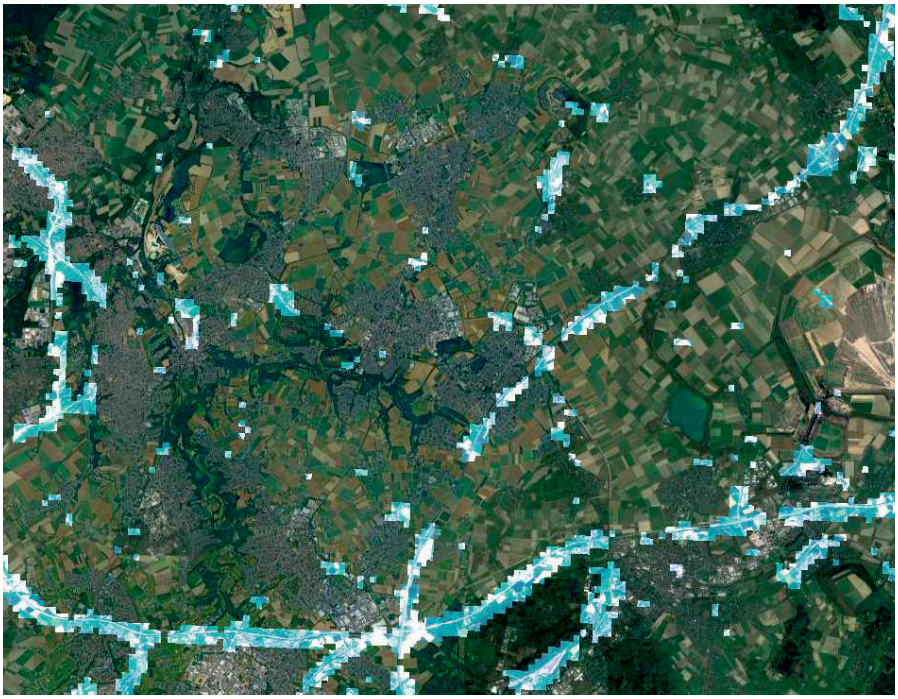


FIGURE 7.13
Land cover class predictions for a Sentinel-2 acquisition from May 8, 2018 north of Aachen, Germany with a 64×64 window moving in 8-pixel increments horizontally and vertically. Blue-white: highway.

Finally, fine tune the entire model with all of its weights but only for 5 epochs:

```
for layer in base_model.layers:
    layer.trainable = True

# not sufficient to make the layers trainable,
# we must also recompile
optimizer = keras.optimizers.SGD(learning_rate=0.01,
                                 momentum=0.9)

model.compile(loss="sparse_categorical_crossentropy",
            optimizer=optimizer, metrics=['accuracy'])

history = model.fit(train_set,epochs=5,
            validation_data=valid_set)

Epoch 1/5 ... accuracy: 0.9430  val_accuracy: 0.9748

Epoch 2/5 ... accuracy: 0.9817  val_accuracy: 0.9822

Epoch 3/5 ... accuracy: 0.9902  val_accuracy: 0.9746

Epoch 4/5 ... accuracy: 0.9931  val_accuracy: 0.9758

Epoch 5/5 ... accuracy: 0.9946  val_accuracy: 0.9785
```

Testing the final result, we appear to have have achieved a 97.6% accuracy!

```
model.evaluate(test_set)

85/85 - 12s 142ms/step loss: 0.0852 accuracy: 0.9759
```

It remains to save the model, giving it the `keras` extension:

```
model.save('/content/drive/.../eurosat_model.keras')
```

The python script `S2cnnclassify.py` (included in the software) predicts ground cover classes from the trained model when presented with the RGB bands of a Sentinel-2 image. To mitigate the coarse resolution caused by the 64×64 image patches, the script moves a 64×64 window in smaller increments across the input image. Examples are shown in Figures 7.12 and 7.13 for an acquisition over a heavily populated area north of Aachen, Germany. Industrial parks on the peripheries of larger towns and villages are clearly defined (Figure 7.12).

The highway classification (Figure 7.13) is somewhat less convincing due to the 64×64 moving window. An Autobahn or other major road will often overlap considerably with neighboring agricultural or other target land cover classes.

7.6.2 Semantic segmentation

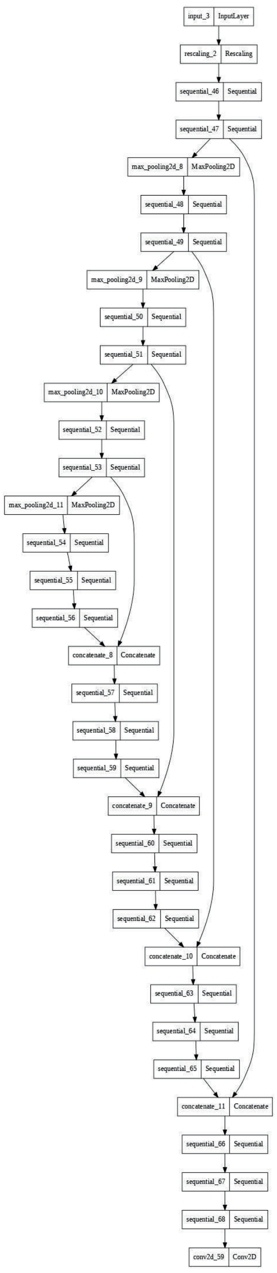


FIGURE 7.14
The UNet model.

So-called *fully convolutional* neural networks (FCNs) are commonly used for semantic image segmentation, by which is meant the assignment of every pixel in an image to one of two or more categories. Here we examine a popular FCN architecture, called UNet, to perform a specific semantic segmentation task, namely urban building recognition. The UNet convolutional neural network architecture is a highly effective model primarily used for semantic segmentation tasks in the field of medical imaging. It is often represented in the form of the letter "U" which gives it its name. In our case, we are concerned with the identification, within an arbitrarily complex remote sensing image, of houses, schools, commercial edifices, etc. Sirko et al. (2021) give a recent application of the UNet architecture for large-scale semantic classification of building footprints over the entire African continent. We'll program a "vanilla" UNet FCN based on the architecture described in de Jong et al. (2019). It consists of an encoder and decoder section with lateral connections between the two, see Figure 7.14. (Rotated through 90 degrees, and with a little good will, one can see the U-form.) The encoder section is a series of 5 pairs of Keras `Conv2D` convolutional layers, with successively doubling numbers of filters (64, 128, ...), connected by `MaxPooling2D` layers which successively halve the image dimensions (512, 256, ...). The decoder section reverses the process with the help of up sampling `Conv2DTranspose` layers, ultimately reconstructing the input image signal at the network output. The input to each up sampling layer consists of the output from the preceding layer, merged (concatenated) with the output of the corresponding Conv2d layer from the encoding section (the four cross connections in Figure 7.14). The idea is to restore higher resolution details lost during the image compression (encoding) phase while decoding takes place. The model takes as input a four channel RGB+NIR image in `np.uint8` format and outputs a 1-channel image in `np.float32` format. The output of the last

Conv2D layer is passed through a sigmoid activation function. This means that the pixel intensity can be interpreted as class membership probability, i.e., the probability of belonging to a building footprint.

Since the convolutional layers are all identical except for the number of filters, and since each is followed by a batch normalization layer, it improves readability to use some shortcut functions. These can be conveniently programmed with the tensorflow.keras sequential API:

```
# shortcuts
def conv2d(filters):
    return keras.models.Sequential([
        keras.layers.Conv2D(filters,3,padding = "same"),
        keras.layers.BatchNormalization(),
        keras.layers.Activation('relu')])
def conv2dtranspose(filters):
    return keras.models.Sequential([
        keras.layers.Conv2DTranspose(filters, 3,
                                   strides = 2,padding = "same"),
        keras.layers.BatchNormalization(),
        keras.layers.Activation('relu')])
# all MaxPooling layers have identical parameters
def maxpooling2d():
    return keras.layers.MaxPooling2D(pool_size = 2,
                                   strides = 2,padding = "same")
```

Here then is the full model, now written in the Keras functional API:

```
def unet_model(num_channels=4, image_size=512):
    inputs=keras.layers.Input(shape=(image_size,
                                     image_size, num_channels))
    rescaled= keras.layers.Rescaling(1./255)(inputs)
        #  encoder
    conv11=conv2d(64)(rescaled)
    conv12=conv2d(64)(conv11)
    max_pool1=maxpooling2d()(conv12)
    conv21=conv2d(128)(max_pool1)
    conv22=conv2d(128)(conv21)
    max_pool2=maxpooling2d()(conv22)
    conv31=conv2d(256)(max_pool2)
    conv32=conv2d(256)(conv31)
    max_pool3=maxpooling2d()(conv32)
    conv41=conv2d(512)(max_pool3)
    conv42=conv2d(512)(conv41)
    max_pool4=maxpooling2d()(conv42)
    conv51=conv2d(1024)(max_pool4)
    conv52=conv2d(1024)(conv51)
        #  decoder
    uconv51=conv2dtranspose(512)(conv52)
    merge_dec5=keras.layers.concatenate([conv42,uconv51],
                                         axis=3)
    conv_dec_41=conv2d(512)(merge_dec5)
    conv_dec_42=conv2d(512)(conv_dec_41)
    uconv41=conv2dtranspose(256)(conv_dec_42)
    merge_dec4=keras.layers.concatenate([conv32,uconv41],
                                         axis=3)
    conv_dec_31=conv2d(256)(merge_dec4)
    conv_dec_32=conv2d(256)(conv_dec_31)
    uconv31=conv2dtranspose(128)(conv_dec_32)
    merge_dec3=keras.layers.concatenate([conv22,uconv31],
                                            axis=3)
    conv_dec_21=conv2d(128)(merge_dec3)
    conv_dec_22=conv2d(128)(conv_dec_21)
    uconv21=conv2dtranspose(64)(conv_dec_22)
    merge_dec2=keras.layers.concatenate([conv12,uconv21],
                                           axis=3)
    conv_dec_11=conv2d(64)(merge_dec2)
    conv_dec_12=conv2d(64)(conv_dec_11)
        #  output
    conv_dec_12=conv2d(8)(conv_dec_12)
    output=keras.layers.Conv2D(1, 1,
         activation='sigmoid')(conv_dec_12)
    return keras.Model(inputs=inputs,outputs = output)
```

To train our model, we will make use of the dataset for the Inria Aerial Imaging Labeling Benchmark, published in 2016. The dataset consists of 30cm spatial resolution RGB images and their corresponding semantic labels over built up regions in Europe and USA. To quote from the Inria download site:

> [The images] cover dissimilar urban settlements, ranging from densely populated areas (e.g., San Francisco's financial district) to alpine towns (e.g., Lienz in Austrian Tyrol).

A presentation of the most successful semantic classifiers determined in the early part of the benchmark competition is given in Huang et al. (2018). However, we haven't made use of the Inria RGB images themselves, preferring rather to upload only the semantic labels to Google Earth Engine and then use them to generate 1-m resolution four-band (RGB+NIR) training image patches from the NAIP imagery in the GEE high-resolution image archive *earthengine/datasets/catalog/USDA_NAIP_DOQQ*.

The high resolution USDA NAIP images in the GEE archive are confined to the continental USA. There are 180 image/label pairs in all in the Inria competition dataset 72 of which are over two USA cities, the others are European scenes and therefore not relevant in the present case. There are an additional three American cities (108 images) in the reserved evaluation dataset and these were also included in the training dataset.

With the generated NAIP image patches, together with the Inria labels we can train the UNet semantic classifier and then apply it to any region

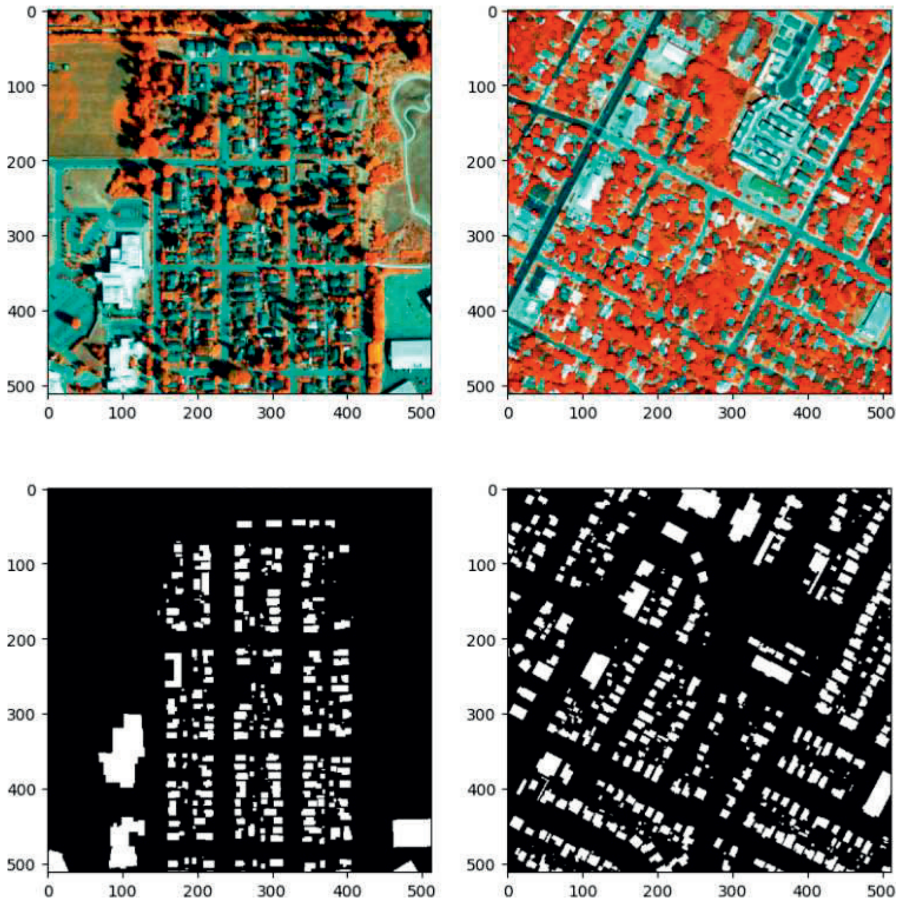


FIGURE 7.15
Two NAIP images and their corresponding Inria labels.

covered by the NAIP Program, i.e., anywhere within the continental USA. Since the GEE environment makes no provision for a TensorFlow/GPU runtime, the corresponding label images were uploaded to GEE assets, and then the matching NAIP images (and the labels) were exported to Google Drive in GeoTIFF format at a scale of 1m. The label uploading was done image-for-image from the GEE asset menu. Taking advantage of the fact that the NAIP imagery is acquired repeatedly over several years, training examples were generated from images acquired at different times. Figure 7.15 shows two examples of image—semantic label pairs.

Figure 7.16 is a plot of the train and validation accuracy achieved for a training run of 36 epochs on Colab using a GPU runtime, and Figure 7.17 compares a test image with its Inria label and the model prediction. The Colab

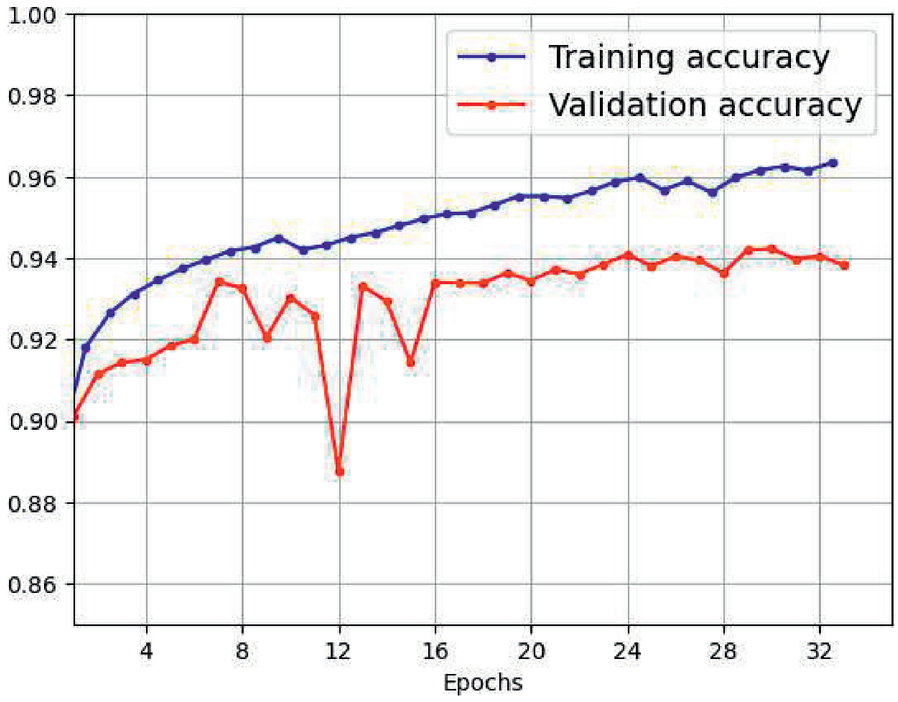


FIGURE 7.16
Training and validation accuracies achieved with the UNet model.

notebook accompanying this chapter walks the reader through the complete training procedure and includes an interactive widget interface to perform segmentation with the trained model anywhere in the USA. Figure 7.18 is an example.

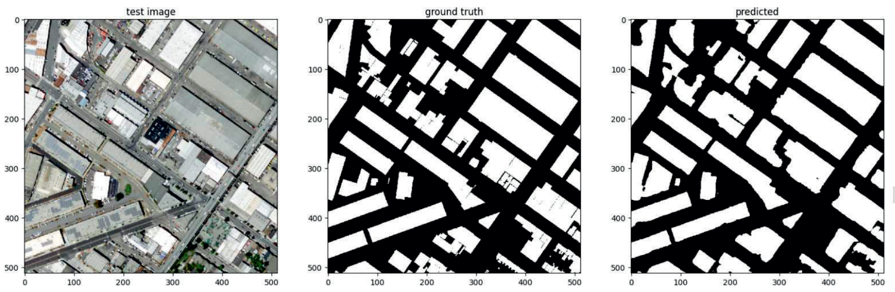


FIGURE 7.17
Comparison of ground truth and prediction for an image-label pair from the test dataset.

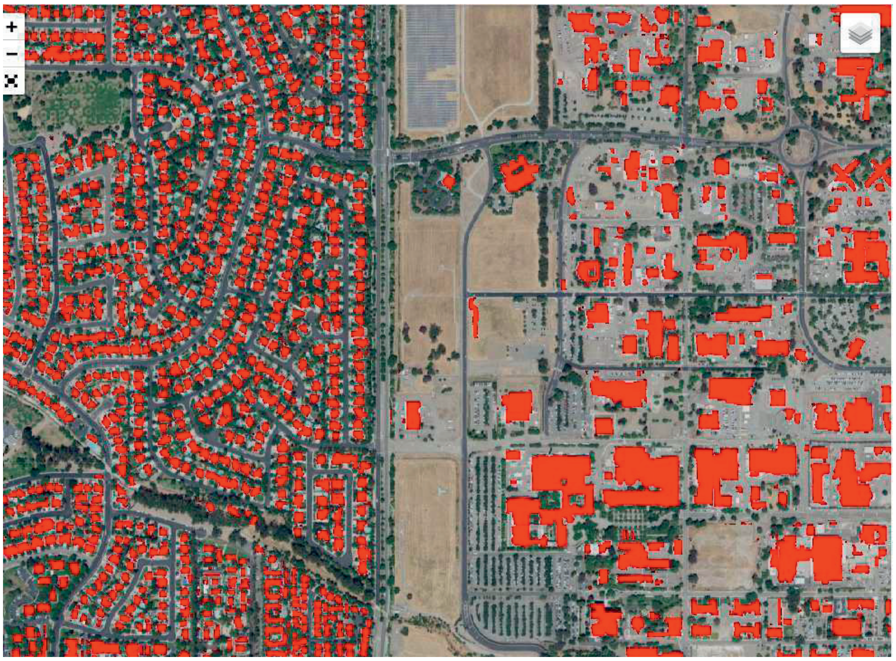


FIGURE 7.18
Classified buildings in Lawrence Livermore National Laboratory, California and an adjacent Livermore suburb.

From Figure 7.16, we've achieved an accuracy of around 94%. This is confirmed by evaluating the test dataset:

```
model.evaluate(test_dataset)

135/135 [==] - 83s 386ms/step - loss: 0.1769
                              - binary_accuracy: 0.9420
```

The main comparison metric used in the Inria benchmark competition is intersection over union (IoU), referring to the sets of labeled and classified pixels. A value of one implies perfect reproduction of the label image, a value zero means no correspondence whatsoever. The cell below calculates the mean IoU for all of the test image/label pairs (they are pipelined in batches of 4).

```
# IoU
t = 0.4
i = 0
sumIoU = 0
for test_example, test_label in test_dataset:
    for j in range(4):
        label = np.reshape(test_label[j], (512, 512))
```

```
                example = np.reshape(test_example[j],
                                               (1, 512, 512, 4))
                pred = model.predict(example)
                pred = np.reshape(np.where(pred > t, 1 ,0),
                                               (512, 512))
                I = label*pred
                U = label + pred - I
                sU = np.sum(U)
                if sU > 0: # buildings in subscene?
                    sumIoU += np.sum(I)/sU
                    i += 1
sumIoU/i
```

```
0.6701648623879097
```

This result exceeds the best results obtained in the initial benchmarking of the Inria dataset using a multilayer perceptron (MLP) architecture (IoU $\approx$ 0.60)*. The initial winners of the competition achieved IoUs of the order 0.7 with variations of the UNet model that we are using here, although later submissions reported as much as 0.8. All of the competition exercises were performed at 30cm spatial resolution, whereas we are training at 1m resolution so that a direct comparison may be misleading. In addition, unlike the original Inria competition image/label dataset, the NAIP train and test images are not temporally very well-defined and coincide in time only approximately with the Inria labels.

7.7 Exercises

1. Derive the confidence limits on the misclassification rate θ given by Equation (7.14).

2. In *bootstrap aggregation* or *bagging* (Breiman, 1996; Polikar, 2006) an ensemble of classifiers is derived from a single classifier by training it repeatedly on *bootstrapped replicas* of the training dataset, i.e., on random samples drawn from the training set *with replacement*. In each replica, some training examples may appear more than once, and some may be missing altogether. A simple majority voting scheme is then used to classify new data with the ensemble. Breiman (1996) shows that improved classification accuracy is to be expected over that obtained by the original classifier/training dataset, especially for relatively unstable classifiers. Unstable classifiers are ones whose decision boundaries tend

*https://inria.hal.science/hal-01468452/document to be sensitive to small perturbations of the training data, and neural networks belong to this category.

Write a Python script to implement bagging using the Python object class for Kalman filter training `Ffnekf` described in Appendix B.3 as the neural network classifier. Use the program `adaboost.py` (Appendix C) as a reference. Your implementation should do the following:

- Get the image to be classified, the training data, the number P of classifiers in the ensemble, and the size $n < m$ of the bootstrapped replicas, where m is the size of the training set.
- Create P instances of the neural network object class and train each of them on a different bootstrapped sample.
- Classify the entire image by constructing a $c \times P$ array of class labels for each row of the image using the P trained classifiers, where c is the number of pixels in each row. Then pass the array to a function `majorityVote()` which implements the voting scheme, and iterate over all of the rows of the image.
- Write the classified image to disk.

3. The random forest classifier is initialized as shown in Listing 7.3. There the `DecisionTreeClassifier()` object, a binary tree classifier, is wrapped in a `BaggingClassifier()` object to form the random forest. Only the maximum number of leaf nodes in each binary tree (default 50) and the number of estimators, or trees in the forest (default 500), can be specified in this implementation.[*] Experiment with these two parameters to try to improve the classification accuracy of the random forest classification result obtained in Section 7.3.2. This involves modifying the default parameters in the script `classify.py`:

```
elif trainalg == 8:
    classifier = sc.RF(Xstrn, Lstrn, 50, 500)
```

4. Derive the discriminant Equation (7.34) from the complex Wishart distribution, Equation (7.33).

5. Demonstrate that the mixing coefficients of Equation (7.40) minimize the standardized residual, Equation (7.37), under constraint Equation (7.38).

6. In searching for a single spectral signature of interest $\boldsymbol{d}$, we can write Equation (7.36) in the form

$$\boldsymbol{G} = \boldsymbol{d}\alpha_d + \boldsymbol{U}\boldsymbol{\beta} + \boldsymbol{R}, \tag{7.54}$$

[*]In fact there are 13 different parameters in the `sklearn.tree.DecisionTreeClassifier()` that can be chosen by the user!

where $\boldsymbol{U}$ is the $N \times (K-1)$ matrix of unwanted (or perhaps unknown) spectra and $\boldsymbol{\beta}$ is the vector of their mixing coefficients. The terms *partial unmixing* or *matched filtering* are often used to characterize methods to eliminate or reduce the effect of $\boldsymbol{U}$.

(a) *Orthogonal subspace projection* (OSP) (Harsanyi and Chang, 1994). Show that the matrix

$$\boldsymbol{P} = \boldsymbol{I} - \boldsymbol{U}(\boldsymbol{U}^\top \boldsymbol{U})^{-1}\boldsymbol{U}^\top, \tag{7.55}$$

where $\boldsymbol{I}$ is the $N \times N$ identity matrix, "projects out" the unwanted components. (Hilger and Nielsen (2000) give an example of the use of this transformation for the suppression of cloud cover from a multispectral image.)

(b) Write a Python function `osp(G,U)` which takes a data matrix $\boldsymbol{\mathcal{G}}$ and the matrix of undesired spectra $\boldsymbol{U}$ as input and returns the OSP projection $\boldsymbol{P}\boldsymbol{\mathcal{G}}^\top$.

(c) *Constrained energy minimization* (CEM) (Harsanyi, 1993; Nielsen, 2001). OSP still requires knowledge of all of the end-member spectra. In fact, Settle (1996) shows that it is fully equivalent to linear unmixing. If the spectra $\boldsymbol{U}$ are unknown, CEM reduces the influence of the undesired spectra by finding a projection direction $\boldsymbol{w}$ for which

$$\boldsymbol{w}^\top \boldsymbol{d} = 1$$

while at the same time minimizing the "mean of the output energy" $\langle(\boldsymbol{w}^\top \boldsymbol{G})^2\rangle$. Assuming that the mean of the projection $\boldsymbol{w}^\top \boldsymbol{G}$ is approximately zero, show that the desired projection direction is

$$\boldsymbol{w} = \frac{\boldsymbol{\Sigma}_R^{-1}\boldsymbol{d}}{\boldsymbol{d}^\top \boldsymbol{\Sigma}_R^{-1}\boldsymbol{d}}. \tag{7.56}$$

(d) *Spectral angle mapping* (SAM) (Kruse et al., 1993). In order to establish a measure of closeness to a desired spectrum, one can compare the angle θ between the desired spectrum $\boldsymbol{d}$ and each pixel vector $\boldsymbol{g}$. Show that this is equivalent to CEM with a diagonal covariance matrix $\boldsymbol{\Sigma}_R = \sigma^2\boldsymbol{I}$, i.e., to CEM without any allowance for covariance between the spectral bands.

7. Derive Equation (7.52).

8. To cite the Earth Engine Data Catalogue:

> The National Agriculture Imagery Program (NAIP) acquires aerial imagery during the agricultural growing seasons in the continental U.S. NAIP projects are contracted each year based upon available funding and the imagery acquisition cycle. Beginning in 2003, NAIP was acquired on a 5-year cycle. 2008 was a transition year, and a three-year cycle began in 2009.

Overall dataset availability is at the time of writing from June, 2002 to December, 2022. With the help of the Colab notebook for semantic segmentation accompanying Section 7.6.2, examine the possibilities of following suburban housing availability over time.

8 Unsupervised Classification

Supervised classification of remote sensing imagery, the subject of the previous two chapters, involves the use of a training dataset consisting of labeled pixels representative of each land cover category of interest in an image. We saw how to use these data to generalize to a complete labeling, or thematic map, for an entire scene. The choice of training areas which adequately represent the spectral characteristics of each category is very important for supervised classification, as the quality of the training set has a profound effect on the validity of the result. Finding and verifying training areas can be laborious, since the analyst must select representative pixels for each of the classes by visual examination of the image and by information extraction from additional sources such as ground reference data (ground truth), aerial photos, or existing maps.

Unlike supervised classification, unsupervised classification, or *clustering* as it is often called, requires no reference information at all. Instead, the attempt is made to find an underlying class structure automatically by organizing the data into groups sharing similar (e.g., spectrally homogeneous) characteristics. Often, one only needs to specify beforehand the number K of classes present. Unsupervised classification plays an especially important role when very little *a priori* information about the data is available. A primary objective of using clustering algorithms for multispectral remote sensing data is to obtain useful information for the selection of training regions in a subsequent supervised classification.

We can view the basic problem of unsupervised classification at the individual pixel level as the partitioning of a set of samples (the image pixel intensity vectors $\boldsymbol{g}(\nu)$, $\nu = 1 \dots m$) into K disjoint subsets, called classes or clusters. The members of each class are to be in some sense more similar to one another than to the members of the other classes. If one wishes to take the spatial context of the image into account, then the problem also becomes one of labeling a regular lattice. Here the random field concept introduced in Chapter 4 can be used to advantage.

Clearly a criterion is needed which will determine the quality of any given partitioning, along with a means to determine the partitioning which optimizes it. The *sum of squares* cost function will provide us with a sufficient basis to justify some of the most popular algorithms for clustering of multispectral imagery, so we begin with its derivation. For a broad overview of clustering techniques for multispectral images, see Tran et al. (2005).

DOI: 10.1201/9781003503286-8

Segmentation and clustering methods for SAR imagery are reviewed in Oliver and Quegan (2004).

8.1 Simple cost functions

Let the (N-dimensional) observations that are to be partitioned by a clustering algorithm comprise the set

$$\{\boldsymbol{g}(\nu) \mid \nu = 1 \dots m\},$$

which we can conveniently represent as the $m \times N$ data matrix

$$\boldsymbol{\mathcal{G}} = \begin{pmatrix} \boldsymbol{g}(1)^\top \\ \vdots \\ \boldsymbol{g}(m)^\top \end{pmatrix}.$$

A given partitioning may be written in the form

$$C = [C_1, \dots C_k, \dots C_K],$$

where C_k is the set of indices

$$C_k = \{\ \nu \mid \boldsymbol{g}(\nu) \text{ is in class } k\}.$$

Our strategy will be to maximize the posterior probability $\Pr(C \mid \boldsymbol{\mathcal{G}})$ for observing the partitioning C given the data $\boldsymbol{\mathcal{G}}$. From Bayes' Theorem we can write

$$\Pr(C \mid \boldsymbol{\mathcal{G}}) = \frac{p(\boldsymbol{\mathcal{G}} \mid C)\Pr(C)}{p(\boldsymbol{\mathcal{G}})}. \tag{8.1}$$

$\Pr(C)$ is the prior probability for C. The quantity $p(\boldsymbol{\mathcal{G}} \mid C)$ is the probability density function for the observations when the partitioning is C, also referred to as the likelihood of the partitioning C given the data $\boldsymbol{\mathcal{G}}$, while $p(\boldsymbol{\mathcal{G}})$ is a normalization independent of C.

Following Fraley (1996) we first of all make the strong assumption that the observations are chosen independently from K multivariate normally distributed populations corresponding, for instance, to the K land cover categories present in a satellite image. Under this assumption, the $\boldsymbol{g}(\nu)$ are realizations of random vectors

$$\boldsymbol{G}_k \sim N(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \quad k = 1 \dots K,$$

with multivariate normal probability densities, which we denote $p(\boldsymbol{g} \mid k)$. The likelihood is the product of the individual probability densities given the partitioning, i.e.,

$$\begin{aligned} L(C) = p(\boldsymbol{\mathcal{G}} \mid C) &= \prod_{k=1}^{K} \prod_{\nu \in C_k} p(\boldsymbol{g}(\nu) \mid k) \\ &= \prod_{k=1}^{K} \prod_{\nu \in C_k} (2\pi)^{-N/2} |\boldsymbol{\Sigma}_k|^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k)\right). \end{aligned}$$

Forming the product in this way is justified by the independence of the observations. Taking the logarithm gives the log-likelihood

$$\mathcal{L}(C) = \sum_{k=1}^{K} \sum_{\nu \in C_k} \left(-\frac{N}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}_k| - \frac{1}{2}(\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k)\right). \tag{8.2}$$

Maximizing the log-likelihood is obviously equivalent to maximizing the likelihood. From Equation (8.1), we can then write

$$\log \Pr(C \mid \boldsymbol{\mathcal{G}}) = \mathcal{L}(C) + \log \Pr(C) - \log p(\boldsymbol{\mathcal{G}}). \tag{8.3}$$

Since the last term is independent of C, maximizing $\Pr(C \mid \boldsymbol{\mathcal{G}})$ with respect to C is equivalent to maximizing $\mathcal{L}(C) + \log \Pr(C)$. If the even stronger assumption is now made that all K classes exhibit identical covariance matrices given by

$$\boldsymbol{\Sigma}_k = \sigma^2 \boldsymbol{I}, \quad k = 1 \dots K, \tag{8.4}$$

where $\boldsymbol{I}$ is the identity matrix, then $\mathcal{L}(C)$ is maximized when the last sum in Equation (8.2), namely the expression

$$\sum_{k=1}^{K} \sum_{\nu \in C_k} (\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k)^\top (\frac{1}{2\sigma^2}\boldsymbol{I})(\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k) = \sum_{k=1}^{K} \sum_{\nu \in C_k} \frac{\|\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k\|^2}{2\sigma^2},$$

is minimized. Finally, with Equation (8.3), $\Pr(C \mid \boldsymbol{\mathcal{G}})$ itself is maximized by minimizing the *cost function*

$$E(C) = \sum_{k=1}^{K} \sum_{\nu \in C_k} \frac{\|\boldsymbol{g}(\nu) - \boldsymbol{\mu}_k\|^2}{2\sigma^2} - \log \Pr(C). \tag{8.5}$$

Now let us introduce a "hard" class dependency in the form of a matrix $\boldsymbol{U}$ with elements

$$u_{k\nu} = \begin{cases} 1 & \text{if } \nu \in C_k \\ 0 & \text{otherwise.} \end{cases} \tag{8.6}$$

These matrix elements are required to satisfy the conditions

$$\sum_{k=1}^{K} u_{k\nu} = 1, \quad \nu = 1 \dots m, \tag{8.7}$$

meaning that each pixel $\boldsymbol{g}(\nu)$, $\nu = 1 \ldots m$, belongs to precisely one class, and

$$\sum_{\nu=1}^{m} u_{k\nu} = m_k > 0, \quad k = 1 \ldots K, \tag{8.8}$$

meaning that no class C_k is empty. The sum in Equation (8.8) is the number m_k of pixels in the kth class. Maximum-likelihood estimates for the mean of the kth cluster can then be written in the form

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{m_k} \sum_{\nu \in C_k} \boldsymbol{g}(\nu) = \frac{\sum_{\nu=1}^{m} u_{k\nu} \boldsymbol{g}(\nu)}{\sum_{\nu=1}^{m} u_{k\nu}}, \quad k = 1 \ldots K, \tag{8.9}$$

and for the covariance matrix as

$$\hat{\boldsymbol{\Sigma}}_k = \frac{\sum_{\nu=1}^{m} u_{k\nu} (\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)^\top}{\sum_{\nu=1}^{m} u_{k\nu}}, \quad k = 1 \ldots K; \tag{8.10}$$

see Section 2.4, Equations (2.74) and (2.75). The cost function, Equation (8.5), can also be expressed in terms of the class dependencies $u_{k\nu}$ as

$$E(C) = \sum_{k=1}^{K} \sum_{\nu=1}^{m} u_{k\nu} \frac{\|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2}{2\sigma^2} - \log \Pr(C). \tag{8.11}$$

The parameter σ^2 can be thought of as the average within-cluster or image noise variance. If we have no prior information on the class structure, we can simply say that all partitionings C are *a priori* equally likely. Then the last term in Equation (8.11) is independent of C and, dropping it and the multiplicative constant $1/2\sigma^2$, we get the *sum of squares* cost function

$$E(C) = \sum_{k=1}^{K} \sum_{\nu=1}^{m} u_{k\nu} \|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2. \tag{8.12}$$

8.2 Algorithms that minimize the simple cost functions

The problem to find the partitioning which minimizes the cost functions, Equation (8.11) or Equation (8.12), is unfortunately impossible to solve. The number of conceivable partitions, while obviously finite, is in any real situation astronomical. For example, for $m = 1000$ pixels and just $K = 2$ possible classes, there are $2^{1000-1} - 1 \approx 10^{300}$ possibilities (Duda and Hart, 1973). Direct enumeration is therefore not feasible. The line of attack most frequently taken is to start with some initial clustering and its associated cost function and then attempt to minimize the latter iteratively. This will always find a local minimum, but there is no guarantee that a global minimum for the cost function will be reached and one can never know if the best solution has been found. Nevertheless, the approach is used because the computational burden is acceptable.

We shall follow the iterative approach in this section, beginning with the well-known K-means algorithm (including a kernelized version), followed by consideration of a variant due to Palubinskas (1998) which uses the cost function of Equation (8.11) and for which the number of clusters is determined automatically. Then we discuss a common example of bottom-up or agglomerative hierarchical clustering and conclude with a "fuzzy" version of the K-means algorithm.

8.2.1 K-means clustering

The K-means clustering algorithm (sometimes referred to as basic ISODATA (Duda and Hart, 1973) or *migrating means* (Richards, 2012)) is based on the sum of squares cost function, Equation (8.12). After some random initialization of the cluster centers $\hat{\boldsymbol{\mu}}_k$ and setting the class dependency matrix $\boldsymbol{U} = \boldsymbol{0}$, the distance measure corresponding to a minimization of Equation (8.12), namely

$$d(\boldsymbol{g}(\nu), k) = \|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2, \tag{8.13}$$

is used to cluster the pixel vectors. Specifically, set $u_{k\nu} = 1$, where

$$k = \arg\min_k d(\boldsymbol{g}(\nu), k) \tag{8.14}$$

for $\nu = 1 \dots m$. Then Equation (8.9) is invoked to recalculate the cluster centers. This procedure is iterated until the class labels cease to change. A popular extension, referred to as ISODATA (Iterative Self-Organizing Data Analysis), involves splitting and merging of the clusters in repeated passes, albeit at the cost of setting additional parameters.

8.2.1.1 K-means with Scipy

The Scipy Python package includes a ready-made K-means function which can be used directly for clustering multispectral images, as illustrated in Listing 8.1. After reading in the chosen spatial/spectral image subsets, the K-means algorithm is called in line 30, returning the cluster centers and *distortions* (sums of the squared differences between the observations and the corresponding centroid), in this case in an anonymous variable since they are not used. The image observations are then labeled with the `vq()` function (line 31, this function also returns the distortions) and written to disk. Here we run the algorithm on the first four principal components of the ASTER image of Figure 6.1 with 8 clusters (default) and display the result:

```
run scripts/kmeans -p [1,2,3,4] \
                 imagery/AST_20070501_pca.tif
```

Listing 8.1: K-means clustering (excerpt from the script `kmeans.py`).

```python
    inDataset = gdal.Open(infile,GA_ReadOnly)
    cols = inDataset.RasterXSize
    rows = inDataset.RasterYSize
    bands = inDataset.RasterCount
    if dims:
        x0,y0,cols,rows = dims
    else:
        x0 = 0
        y0 = 0
    if pos is not None:
        bands = len(pos)
    else:
        pos = range(1,bands+1)
    path = os.path.dirname(infile)
    basename = os.path.basename(infile)
    root, ext = os.path.splitext(basename)
    outfile = path+'/'+root+'_kmeans'+ext
    print('------------␣k-means␣------------')
    print(time.asctime())
    print('Input␣%s'%infile)
    print('Number␣of␣clusters␣%i'%K)
    start = time.time()
    G = np.zeros((rows*cols,bands))
    k = 0
    for b in pos:
        band = inDataset.GetRasterBand(b)
        G[:,k] = band.ReadAsArray(x0,y0,cols,rows)\
                              .astype(float).ravel()
        k += 1
    centers, _ = kmeans(G,K)
    labels, _ = vq(G,centers)
    driver = gdal.GetDriverByName('GTiff')
    outDataset = driver.Create(outfile,
                    cols,rows,1,GDT_Byte)
    outBand = outDataset.GetRasterBand(1)
    outBand.WriteArray(np.reshape(labels+1,
                            (rows,cols)),0,0)
    outBand.FlushCache()
    outDataset = None
    inDataset = None
```

```
------------ k-means ------------
Wed Jun 12 11:30:43 2024
Input imagery/AST_20070501_pca.tif
Number of clusters 8
```

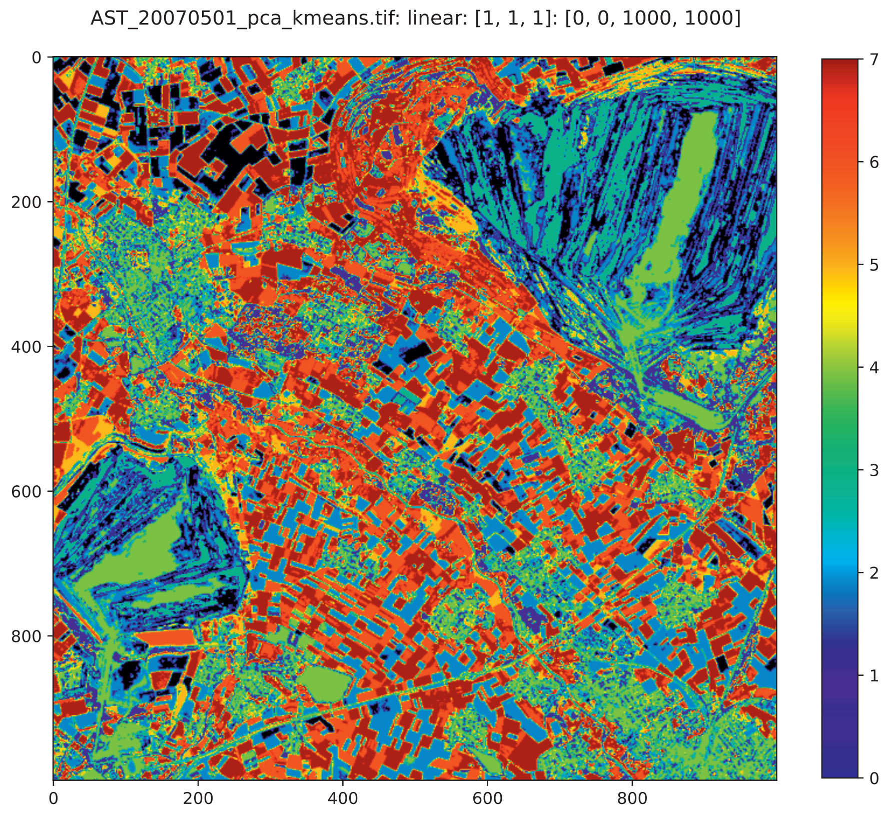

AST_20070501_pca_kmeans.tif: linear: [1, 1, 1]: [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800
0
1
2
3
4
5
6
7

FIGURE 8.1
Unsupervised K-means classification of a spectral subset of the image of Figure 6.1 with 8 clusters.

```
Kmeans result written to: imagery/AST_20070501_pca_
kmeans.tif
elapsed time: 25.16997218132019

%run scripts/dispms -f imagery/AST_20070501_pca_
%kmeans.tif -c
```

The classified image is shown in Figure 8.1.

8.2.1.2 K-means with GEE

The Google Earth Engine API offers several clustering algorithms, including K-means:

```
import ee
ee.Initialize()
```

```
image = ee.Image('projects/.../AST_20070501_pca')\
                              .select(0,1,2,3)
region = image.geometry()
training = image.sample(region=region,scale=15,
                               numPixels=100000)
clusterer = ee.Clusterer.wekaKMeans(8)
trained = clusterer.train(training)
clustered = image.cluster(trained)
```

The result can be displayed in the Jupyter notebook accompanying this chapter.

8.2.2 Kernel K-means clustering

To kernelize the K-means algorithm (Shawe-Taylor and Cristianini, 2004), one requires the dual formulation for Equation (8.13). With Equation (8.8), the matrix

$$\boldsymbol{M} = [\mathrm{Diag}(\boldsymbol{U}\mathbf{1}_m)]^{-1},$$

where $\mathbf{1}_m$ is an m-component vector of ones, can be seen to be a $K \times K$ diagonal matrix with inverse class populations along the diagonal,

$$\boldsymbol{M} = \begin{pmatrix} 1/m_1 & 0 & \dots & 0 \\ 0 & 1/m_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1/m_K \end{pmatrix}.$$

Therefore, from the rule for matrix multiplication, Equation (1.14), we can express the class means given by Equation (8.9) as the columns of the $N \times K$ matrix

$$(\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2 \dots \hat{\boldsymbol{\mu}}_K) = \boldsymbol{\mathcal{G}}^\top \boldsymbol{U}^\top \boldsymbol{M}.$$

We then obtain the dual formulation of Equation (8.13) as

$$\begin{aligned} d(\boldsymbol{g}(\nu), k) &= \|\boldsymbol{g}(\nu)\|^2 - 2\boldsymbol{g}(\nu)^\top \hat{\boldsymbol{\mu}}_k + \|\hat{\boldsymbol{\mu}}_k\|^2 \\ &= \|\boldsymbol{g}(\nu)\|^2 - 2\boldsymbol{g}(\nu)^\top [\boldsymbol{\mathcal{G}}^\top \boldsymbol{U}^\top \boldsymbol{M}]_{\cdot k} + [\boldsymbol{M}\boldsymbol{U}\boldsymbol{\mathcal{G}}\boldsymbol{\mathcal{G}}^\top \boldsymbol{U}^\top \boldsymbol{M}]_{kk} \\ &= \|\boldsymbol{g}(\nu)\|^2 - 2[(\boldsymbol{g}(\nu)^\top \boldsymbol{\mathcal{G}}^\top)\boldsymbol{U}^\top \boldsymbol{M}]_k + [\boldsymbol{M}\boldsymbol{U}\boldsymbol{\mathcal{G}}\boldsymbol{\mathcal{G}}^\top \boldsymbol{U}^\top \boldsymbol{M}]_{kk}\ . \end{aligned} \tag{8.15}$$

In the second line above, $[\ ]_{\cdot k}$ denotes the kth column. In the last line the observations appear only as inner products: $\boldsymbol{g}(\nu)^\top \boldsymbol{g}(\nu)$ (first term), in the Gram matrix $\boldsymbol{\mathcal{G}}\boldsymbol{\mathcal{G}}^\top$ (last term) and in $\boldsymbol{g}(\nu)^\top \boldsymbol{\mathcal{G}}^\top$ (second term). This last expression is just the νth row of the Gram matrix, i.e.,

$$\boldsymbol{g}(\nu)^\top \boldsymbol{\mathcal{G}}^\top = [\boldsymbol{\mathcal{G}}\boldsymbol{\mathcal{G}}^\top]_{\nu\cdot}.$$

For kernel K-means, where we work in an (implicit) nonlinear feature space $\boldsymbol{\phi}(\boldsymbol{g})$, we substitute $\boldsymbol{\mathcal{G}}\boldsymbol{\mathcal{G}}^\top \to \boldsymbol{\mathcal{K}}$ to get

$$d(\boldsymbol{\phi}(\boldsymbol{g}(\nu)), k) = [\boldsymbol{\mathcal{K}}]_{\nu\nu} - 2[\boldsymbol{\mathcal{K}}_{\nu\cdot}\boldsymbol{U}^\top \boldsymbol{M}]_k + [\boldsymbol{M}\boldsymbol{U}\boldsymbol{\mathcal{K}}\boldsymbol{U}^\top \boldsymbol{M}]_{kk}, \tag{8.16}$$

Listing 8.2: Kernel K-means clustering (excerpt from the script `kkmeans.py`).

```
 1  #   iteration
 2      change = True
 3      itr = 0
 4      onesm = np.mat(np.ones(m,dtype=float))
 5      while change and (itr < 100):
 6          change = False
 7          U = np.zeros((K,m))
 8          for i in range(m):
 9              U[labels[i],i] = 1
10          M =  np.diag(1.0/(np.sum(U,axis=1)+1.0))
11          MU = np.mat(np.dot(M,U))
12          Z = (onesm.T)*np.diag(MU*KK*(MU.T))-2*KK*(MU.T)
13          Z = np.array(Z)
14          labels1 = (np.argmin(Z,axis=1) % K).ravel()
15          if np.sum(labels1 != labels):
16              change = True
17          labels = labels1
18          itr += 1
19      print('iterations:␣%i'%itr)
```

where $\boldsymbol{\mathcal{K}}$ is the kernel matrix and $\boldsymbol{\mathcal{K}}_{\nu\cdot}$ is its νth row. Since the first term in Equation (8.16) doesn't depend on the class index k, the clustering rule for kernel K-means is to assign observation $\boldsymbol{g}(\nu)$ to class k, where

$$k = \arg\min_k \left([\boldsymbol{M}\boldsymbol{U}\boldsymbol{\mathcal{K}}\boldsymbol{U}^\top \boldsymbol{M}]_{kk} - 2[\boldsymbol{\mathcal{K}}_{\nu\cdot}\boldsymbol{U}^\top \boldsymbol{M}]_k \right). \tag{8.17}$$

A Python script for kernel K-means clustering is documented in Appendix C. As in the case of kernel PCA (Chapter 4), memory restrictions require that one work with only a relatively small training sample of m pixel vectors. A portion of the Python code is shown in Listing 8.2.

The variable `Z` in line 12 is a $K \times m$ array of current values of the expression on the right-hand side of Equation (8.17), with one row for each of the training observations and one column for each class label. In line 14, the class labels corresponding to the minimum distance to the mean are extracted into the variable `labels1`. This avoids an expensive `FOR`-loop over the training data. The iteration terminates when the class labels cease to change.

After convergence, unsupervised classification of the remaining pixels can again be achieved with Equation (8.17), merely replacing the row vector $\boldsymbol{\mathcal{K}}_{\nu\cdot}$ by

$$\big(\kappa(\boldsymbol{g},\boldsymbol{g}(1)), \kappa(\boldsymbol{g},\boldsymbol{g}(2)) \dots \kappa(\boldsymbol{g},\boldsymbol{g}(m))\big),$$

where $\boldsymbol{g}$ is a pixel to be classified. The processing of the entire image thus requires evaluation of the kernel for every image pixel with every training pixel. Therefore, it is best to classify the image row by row, as was the case

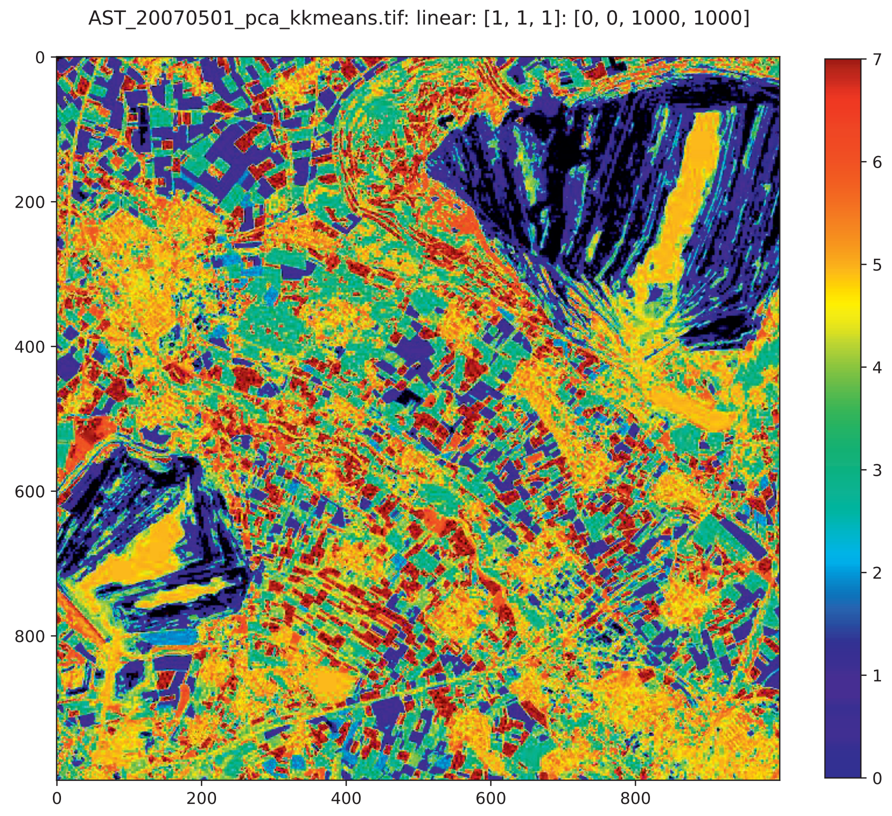

AST_20070501_pca_kkmeans.tif: linear: [1, 1, 1]: [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800
0
1
2
3
4
5
6
7

FIGURE 8.2
Unsupervised kernel K-means classification of a spectral subset of the image of Figure 6.1 with eight clusters.

for kernel PCA and kernel RX. Figure 8.2 shows an unsupervised classification of the image of Figure 6.1 with kernel K-means clustering:

```
run scripts/kkmeans -p [1,2,3,4] -n 1 -k 8 \
                imagery/AST_20070501_pca.tif

=========================
    kernel k-means
=========================
infile: imagery/AST_20070501_pca.tif
samples: 1000
clusters: 8
kernel matrix...
gamma: 0.000249
iterations: 22
```

```
classifying...

result written to: imagery/AST_20070501_pca_kkmeans.tif
elapsed time: 3.812978744506836
```

8.2.3 Extended K-means clustering

Starting this time from the cost function Equation (8.11), denote by $p_k = \Pr(C_k)$ the prior probability for cluster C_k. The entropy H associated with this prior distribution is given by

$$H = -\sum_{k=1}^{K} p_k \log p_k, \tag{8.18}$$

see Equation (2.113). It was shown in Section 2.7 that distributions with high entropy are those for which the p_k are all similar. So in this case, high entropy corresponds to the pixels being distributed evenly over all available clusters, whereas low entropy means that most of the data are concentrated in very few clusters. Following Palubinskas (1998), we choose a prior distribution $\Pr(C)$ in the cost function of Equation (8.11) for which few clusters (low entropy) are more probable than many clusters (high entropy), namely

$$\Pr(C) = A\exp(-\alpha_E H) = A\exp\big(\alpha_E \sum_{k=1}^{K} p_k \log p_k\big),$$

where A and α_E are parameters. The cost function can then be written in the form

$$E(C) = \sum_{k=1}^{K}\sum_{\nu=1}^{m} u_{k\nu}\frac{\|\boldsymbol{g}(\nu)-\hat{\boldsymbol{\mu}}_k\|^2}{2\sigma^2} - \alpha_E\sum_{k=1}^{K} p_k\log p_k, \tag{8.19}$$

dropping the $\log A$ term, which is independent of the clustering. With

$$p_k = \frac{m_k}{m} = \frac{1}{m}\sum_{\nu=1}^{m} u_{k\nu}, \tag{8.20}$$

Equation (8.19) is equivalent to the cost function

$$E(C) = \sum_{k=1}^{K}\sum_{\nu=1}^{m} u_{k\nu}\left[\frac{\|\boldsymbol{g}(\nu)-\hat{\boldsymbol{\mu}}_k\|^2}{2\sigma^2} - \frac{\alpha_E}{m}\log p_k\right]. \tag{8.21}$$

An estimate for the parameter α_E in Equation (8.21) may be obtained as follows (Palubinskas, 1998). With the approximation

$$\sum_{\nu=1}^{m} u_{k\nu}\|\boldsymbol{g}(\nu)-\hat{\boldsymbol{\mu}}_k\|^2 \approx m_k\sigma^2 = mp_k\sigma^2,$$

we can write Equation (8.21) as

$$E(C) \approx \sum_{k=1}^{K} \left[\frac{mp_k}{2} - \alpha_E p_k \log p_k \right].$$

Equating the likelihood and prior terms in this expression to give $E(C) = 0$ and taking $p_k \approx 1/\tilde{K}$, where $\tilde{K}$ is some *a priori* expected number of clusters, then gives

$$\alpha_E \approx -\frac{m}{2\log(1/\tilde{K})}. \tag{8.22}$$

The parameter σ^2 in Equation (8.21) (the within-cluster variance) can be estimated from the data, see below.

The *extended K-means* (EKM) algorithm is then as follows. First an initial configuration $\boldsymbol{U}$ with a very large number of clusters K is chosen (for single-band data this might conveniently be the $K \le 256$ gray values that an image with 8-bit quantization can possibly have) and initial values

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{m_k}\sum_{\nu=1}^{m} u_{k\nu}\boldsymbol{g}(\nu),\ \ p_k = \frac{m_k}{m}, \quad k = 1\dots K, \tag{8.23}$$

are determined. Then the data are re-clustered according to the distance measure which minimizes Equation (8.21):

$$k = \arg\min_k \left(\frac{\|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2}{2\sigma^2} - \frac{\alpha_E}{m}\log p_k \right), \quad \nu = 1\dots m. \tag{8.24}$$

The prior term tends to put more observations into fewer clusters. Any cluster for which, in the course of the iteration, m_k equals zero (or is less than some threshold) is simply dropped from the calculation so that the final number of clusters is determined by the data. The condition that no class be empty, Equation (8.8), is thus relaxed. The explicit choice of the number of clusters K is replaced by the necessity of choosing a value for the "meta-parameter" α_E (or $\tilde{K}$ if Equation (8.22) is used). This has the advantage that one can use a single parameter for a wide variety of images and let the algorithm itself decide on the actual value of K in any given instance. Iteration of Equations (8.23) and (8.24) continues until the cluster labels cease to change.

An implementation of the extended K-means algorithm in Python for gray-scale, or one-band, images is shown in Listing 8.3. The variance σ^2 is estimated (naively, see Exercise 5) by calculating the variance of the difference of the image with a copy of itself shifted by one pixel, program line 5. An *a priori* number of classes $\tilde{K}$ determines the meta-parameter α_E, line 6. The initial number K of clusters is chosen as the number of nonempty bins in the 256-bin histogram of the linearly stretched image, while the initial cluster means m_k and prior probabilities p_k are the corresponding bin numbers, respectively, the bin contents divided by the number of pixels, lines 7 to 11. The iteration

Listing 8.3: Extended K-means clustering (excerpt from the script `ekmeans.py`).

```
 1     m = rows*cols
 2     band = inDataset.GetRasterBand(b)
 3     G =band.ReadAsArray(x0,y0,cols,rows)
 4     labels = np.zeros(m)
 5     sigma2 = np.std(G - np.roll(G,(0,1)))**2
 6     alphaE = -1/(2*np.log(1.0/K))
 7     hist, _ = np.histogram(G,bins = 256)
 8     indices = np.where(hist>0)[0]
 9     K = indices.size
10     means = np.array(range(256))[indices]
11     priors = hist[indices]/float(m)
12     delta = 100.0
13     itr = 0
14     G = G.ravel()
15     while (delta>1.0) and (itr<100):
16         print('Clusters:␣%i␣delta:␣%f'%(K,delta))
17         indices = np.where(priors>0.01)[0]
18         K = indices.size
19         ds = np.zeros((K,m))
20         means = means[indices]
21         priors = priors[indices]
22         means1 = means
23         priors1 = priors
24         means = means*0.0
25         priors = priors*0.0
26         for j in range(K):
27             ds[j,:] = (G-means1[j])**2/(2*sigma2) \
28                       - alphaE*np.log(priors1[j])
29         min_ds = np.min(ds,axis=0)
30         for j in range(K):
31             indices = np.where(ds[j,:] == min_ds)[0]
32             if indices.size>0:
33                 mj = indices.size
34                 priors[j] = mj/float(m)
35                 means[j] = np.sum(G[indices])/mj
36                 labels[indices] = j
37         delta = np.max(np.abs(means-means1))
38         itr += 1
```

of Equations (8.23) and (8.24) is carried out in the `while` loop, lines 15 to 38. Clusters with prior probabilities less than 0.01 are discarded in line 17. The variable `indices` in line 31 locates the pixels, if any, which belong to cluster j for $j = 1 \ldots K$. For convenience, the termination condition is that there be no significant change in the cluster means, line 37.

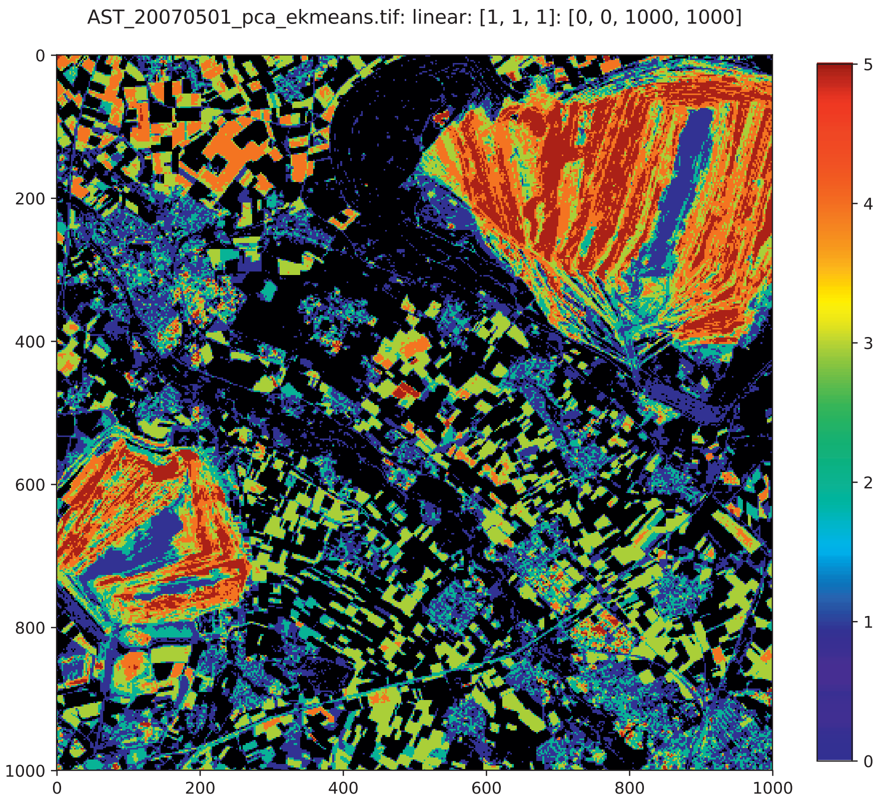

AST_20070501_pca_ekmeans.tif: linear: [1, 1, 1]: [0, 0, 1000, 1000]
0
200
400
600
800
1000
0
200
400
600
800
1000
5
4
3
2
1
0

FIGURE 8.3
Unsupervised extended K-means classification of the first principal component of the image of Figure 6.1.

We run the algorithm here on the first principal component of the ASTER scene:

```
run scripts/ekmeans -b 1 imagery/AST_20070501_pca.tif

------- extended k-means ---------
Sun Aug 26 16:42:58 2018
Input: imagery/AST_20070501_pca.tif
Band: 1
Meta-clusters: 8
Clusters: 256 delta: 100.000000
Clusters: 20 delta: 27.591092
Clusters: 7 delta: 7.004228
Clusters: 7 delta: 6.956754
Clusters: 7 delta: 4.108909
```

```
Clusters: 7 delta: 2.233817
Clusters: 6 delta: 1.584403
Clusters: 6 delta: 1.412826
Clusters: 6 delta: 1.281860
Clusters: 6 delta: 1.132235
Extended K-means result written to:
      imagery/AST_20070501_pca_ekmeans.tif
elapsed time: 0.746820926666
```

In this case, the algorithm prefers six clusters; see Figure 8.3.

8.2.4 Agglomerative hierarchical clustering

The unsupervised classification algorithm that we consider next is, as for ordinary K-means clustering, based on the cost function of Equation (8.12). Let us first write it in a more convenient form:

$$E(C) = \sum_{k=1}^{K} E_k, \tag{8.25}$$

where E_k is given by

$$E_k = \sum_{\nu \in C_k} \|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2. \tag{8.26}$$

The algorithm is initialized by assigning each observation to its own class. At this stage, the cost function is zero, since $C_\nu = \{\nu\}$ and $\boldsymbol{g}(\nu) = \hat{\boldsymbol{\mu}}_\nu$. Every agglomeration of clusters to form a smaller number will increase $E(C)$. We therefore seek a prescription for choosing two clusters for combination that increases $E(C)$ *by the smallest amount possible* (Duda and Hart, 1973).

Suppose that, at some stage of the algorithm, clusters k with m_k members and j with m_j members are merged, where $k < j$, and the new cluster is labeled k. Then the mean of the merged cluster is the weighted average of the original cluster means,

$$\hat{\boldsymbol{\mu}}_k \;\to\; \frac{m_k\hat{\boldsymbol{\mu}}_k + m_j\hat{\boldsymbol{\mu}}_j}{m_k + m_j} = \bar{\boldsymbol{\mu}}.$$

Thus, after the agglomeration, E_k increases to

$$E_k = \sum_{\nu \in C_k \cup C_j} \|\boldsymbol{g}(\nu) - \bar{\boldsymbol{\mu}}\|^2$$

and E_j disappears. The net change in $E(C)$ is therefore, after some algebra (Exercise 7),

$$\begin{aligned}\Delta(k,j) &= \sum_{\nu \in C_k \cup C_j} \|\boldsymbol{g}(\nu) - \bar{\boldsymbol{\mu}}\|^2 - \sum_{\nu \in C_k} \|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2 - \sum_{\nu \in C_j} \|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_j\|^2 \\ &= \frac{m_k m_j}{m_k + m_j} \cdot \|\hat{\boldsymbol{\mu}}_k - \bar{\boldsymbol{\mu}}_j\|^2.\end{aligned} \tag{8.27}$$

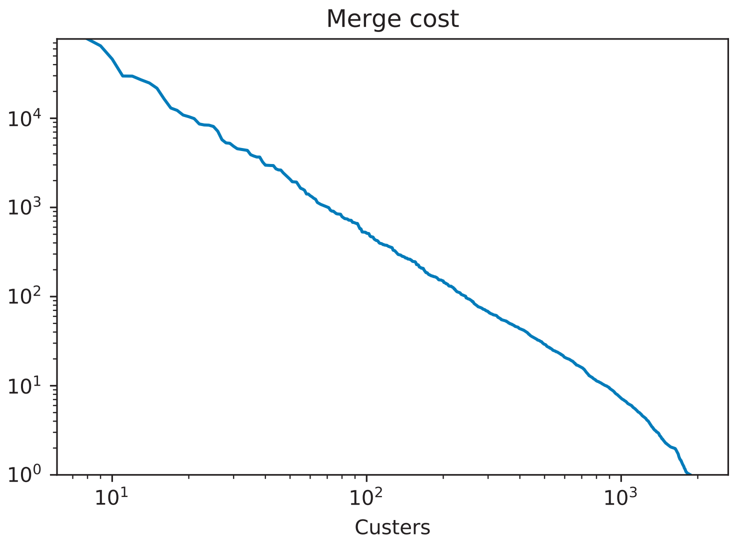


FIGURE 8.4
Agglomerative hierarchical clustering of the first four principal components of the ASTER scene, Figure 6.1.

The minimum increase in $E(C)$ is achieved by combining those two clusters k and j which minimize the above expression. Given two alternative candidate cluster pairs with similar combined memberships $m_k + m_j$ and whose means have similar separations $\|\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}}_j\|$, this prescription obviously favors combining that pair having the larger discrepancy between m_k and m_j. Thus similar-sized clusters are preserved and smaller clusters are absorbed by larger ones.

Let $\langle k, j\rangle$ represent the cluster formed by combination of the clusters k and j. Then the increase in cost incurred by combining this cluster with some other cluster r can be determined from Equation (8.27) as (Fraley, 1996)

$$\Delta(\langle k, j\rangle, r) = \frac{(m_k + m_r)\Delta(k, r) + (m_j + m_r)\Delta(j, r) - m_r\Delta(k, j)}{m_k + m_j + m_r}. \quad (8.28)$$

To see this, note that with Equation (8.27), the right-hand side of Equation (8.28) is, apart from the factor $1/(m_k + m_j + m_r)$, equivalent to

$$m_k m_r\|\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}}_r\|^2 + m_j m_r\|\hat{\boldsymbol{\mu}}_j - \hat{\boldsymbol{\mu}}_r\|^2 - \frac{m_r m_k m_j}{m_k + m_j}\|\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}}_j\|^2$$

and the left-hand side similarly is equivalent to

$$m_r(m_k + m_j)\left\|\frac{m_k\hat{\boldsymbol{\mu}}_k + m_j\hat{\boldsymbol{\mu}}_j}{m_k + m_j} - \hat{\boldsymbol{\mu}}_r\right\|^2.$$

The identity can be established by comparing coefficients of the vector products. For example, the coefficient of $\|\hat{\boldsymbol{\mu}}_k\|^2$ is $m_r m_k^2/(m_k+m_j)$ on both sides.

Once the quantities $\Delta(k,j)$ have been initialized from Equation (8.27), that is,

$$\Delta(k,j) = \frac{1}{2}\|\boldsymbol{g}(k) - \boldsymbol{g}(j)\|^2$$

for all possible combinations of observations, $k = 2\ldots m,\ j = 1\ldots k-1$, the recursive Equation (8.28) can be used to calculate the cost function efficiently for any further merging without reference to the original data. The algorithm terminates when the desired number of clusters has been reached or continues until a single cluster has been formed. Assuming that the data consist of $\tilde{K}$ compact and well-separated clusters, the slope of $E(C)$ vs. the number of clusters K should decrease (become more negative) for $K \le \tilde{K}$. This provides, at least in principle, a means to decide on the optimal number of clusters.[*] A Python script `hcl.py` for agglomerative hierarchic clustering is documented in Appendix C. It subsamples the image before beginning the merge process. Figure 8.4 shows the output cell with a log-log plot of the merge cost as a function of the number of clusters for a sample size of 2000 pixels:

```
run scripts/hcl -p [1,2,3,4] -k 8 -s 2000 \
            imagery/AST_20070501_pca.tif

Wed Oct 10 11:59:07 2018
Input: imagery/AST_20070501_pca.tif
Clusters: 8
Samples: 2000
classifying...

result written to: imagery/AST_20070501_pca_hcl.tif
elapsed time: 29.1420390606
```

Note the increase in slope below about 10 clusters.

8.2.5 Fuzzy K-means clustering

Introducing the parameter $q > 1$, we can write the cluster means and the sum of squares cost function equivalently as (Dunn, 1973)

$$\hat{\boldsymbol{\mu}}_k = \frac{\sum_{i=\nu}^m u_{k\nu}^q \boldsymbol{g}(\nu)}{\sum_{\nu=1}^m u_{k\nu}^q}, \quad k = 1\ldots K, \tag{8.29}$$

$$E(C) = \sum_{\nu=1}^m \sum_{k=1}^K u_{k\nu}^q \|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2. \tag{8.30}$$

[*]For multispectral imagery, however, well-separated clusters seldom occur.

Since $u_{k\nu} \in \{0,1\}$, these equations are identical to Equations (8.9) and (8.12). The transition from "hard" to "fuzzy" clustering is effected by label relaxation, that is, by replacing the class memberships by continuous variables

$$0 \le u_{k\nu} \le 1, \quad k = 1 \dots K, \ \nu = 1 \dots m, \tag{8.31}$$

but retaining the requirements in Equations (8.7) and (8.8). The parameter q now has an effect and determines the "degree of fuzziness." It is often chosen as $q = 2$. The matrix $\boldsymbol{U}$ is referred to as a *fuzzy class membership* matrix. The classification problem is to find that value of $\boldsymbol{U}$ which minimizes Equation (8.30). Minimization can be achieved by finding values for the $u_{k\nu}$ which solve the minimization problems

$$E_\nu = \sum_{k=1}^{K} u_{k\nu}^q \|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2 \ \to \ \min, \quad \nu = 1 \dots m,$$

under the constraints imposed by Equation (8.7). Accordingly, we define a Lagrange function which takes the constraints into account, i.e.,

$$L_\nu = E_\nu - \lambda \left(\sum_{k=1}^{K} u_{k\nu} - 1 \right),$$

and solve the unconstrained problem $L_\nu \ \to \ \min$ by setting the derivative with respect to $u_{k\nu}$ equal to zero,

$$\frac{\partial L_\nu}{\partial u_{k\nu}} = q(u_{k\nu})^{q-1} \|\boldsymbol{g}_\nu - \hat{\boldsymbol{\mu}}_k\|^2 - \lambda = 0, \quad k = 1 \dots K.$$

This gives an expression for fuzzy memberships in terms of λ,

$$u_{k\nu} = \left(\frac{\lambda}{q} \right)^{1/(q-1)} \left(\frac{1}{\|\boldsymbol{g}_\nu - \hat{\boldsymbol{\mu}}_k\|^2} \right)^{1/(q-1)}. \tag{8.32}$$

The Lagrange multiplier λ, in turn, is determined by the constraint

$$1 = \sum_{k=1}^{K} u_{k\nu} = \left(\frac{\lambda}{q} \right)^{1/(q-1)} \sum_{k=1}^{K} \left(\frac{1}{\|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2} \right)^{1/(q-1)}.$$

If we solve for λ and substitute it into Equation (8.32), we obtain

$$u_{k\nu} = \frac{\left(\frac{1}{\|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k\|^2} \right)^{1/(q-1)}}{\sum_{k'=1}^{K} \left(\frac{1}{\|\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_{k'}\|^2} \right)^{1/(q-1)}}, \quad k = 1 \dots K, \ \nu = 1 \dots m. \tag{8.33}$$

The *fuzzy K-means* (FKM) algorithm consists of a simple iteration of Equations (8.29) and (8.33). The iteration terminates when the cluster centers $\hat{\boldsymbol{\mu}}_k$, or alternatively the matrix elements $u_{k\nu}$, cease to change significantly. After convergence, each pixel vector is labeled with the index of the class with the highest membership probability,

$$\ell_\nu = \arg\max_k u_{k\nu}, \quad \nu = 1 \dots m.$$

This algorithm should give similar results to the K-means algorithm, Section 8.2.1. However, because every pixel "belongs" to some extent to all of the classes, one expects the algorithm to be less likely to become trapped in a local minimum of the cost function.

We might interpret the fuzzy memberships as "reflecting the relative proportions of each category within the spatially and spectrally integrated multispectral vector of the pixel" (Schowengerdt, 2006). This interpretation will be more plausible when, in the next section, we replace the fuzzy memberships $u_{k\nu}$ by posterior class membership probabilities. In this spirit, we will dispense with a Python implementation of FKM and proceed directly to a Gaussian mixture model.

8.3 Gaussian mixture clustering

The unsupervised classification algorithms of the preceding section are based on the cost functions of Equation (8.11) or Equation (8.12). These favor the formation of clusters having similar extent, so-called *Voronoi partitions*, due to the assumption made in Equation (8.4). The use of multivariate Gaussian probability densities to model the classes allows for ellipsoidal clusters of arbitrary extent and is considerably more flexible. Rather than deriving an algorithm from first principles (see Exercise 9), we can obtain one directly from the FKM algorithm. We merely replace Equation (8.33) for the class memberships $u_{k\nu}$ by the posterior probability $\Pr(k \mid \boldsymbol{g}(\nu))$ for class k given the observation $\boldsymbol{g}(\nu)$ (Gath and Geva, 1989):

$$u_{k\nu} \to \Pr(k \mid \boldsymbol{g}(\nu)).$$

Then, using Bayes' Theorem, we can write

$$u_{k\nu} \propto p(\boldsymbol{g}(\nu) \mid k)\Pr(k),$$

where $p(\boldsymbol{g}(\nu) \mid k)$ will be chosen to be a multivariate normal density function. Its estimated mean $\hat{\boldsymbol{\mu}}_k$ and covariance matrix $\hat{\boldsymbol{\Sigma}}_k$ are given by Equations (8.9) and (8.10), respectively. Thus, apart from a normalization factor, we have

$$\begin{aligned} u_{k\nu} &= p(\boldsymbol{g}(\nu) \mid k)\Pr(k) \\ &= \frac{1}{\sqrt{|\hat{\boldsymbol{\Sigma}}_k|}} \exp\left[-\frac{1}{2}(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)^\top \hat{\boldsymbol{\Sigma}}_k^{-1}(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)\right] p_k. \end{aligned} \tag{8.34}$$

In the above expression, the prior probability $\Pr(k)$ is replaced by

$$p_k = \frac{m_k}{m} = \frac{1}{m}\sum_{\nu=1}^{m} u_{k\nu}. \tag{8.35}$$

The algorithm consists of an iteration of Equations (8.9), (8.10), (8.34), and (8.35) with the same termination condition as for the fuzzy K-means algorithm. After each iteration, the columns of $\boldsymbol{U}$ are normalized according to Equation (8.7).

Unlike the FKM classifier, the memberships $u_{k\nu}$ are now functions of the directionally sensitive Mahalanobis distance

$$d = \sqrt{(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)^\top \hat{\boldsymbol{\Sigma}}_k^{-1} (\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)}\ .$$

Because of the exponential dependence of the memberships on d^2 in Equation (8.34), the computation is very sensitive to initialization conditions and can even become unstable. To avoid this problem, one can first obtain initial values for $\boldsymbol{U}$ by preceding the calculation with the fuzzy K-means algorithm (see Gath and Geva (1989), who referred to their algorithm as *fuzzy maximum likelihood expectation* (FMLE) clustering). Explicitly, then, the algorithm is as follows:

Algorithm (FMLE clustering)

1. Determine starting values for the initial memberships $u_{k\nu}$, e.g., by randomization or FKM.
2. Determine the cluster centers $\hat{\boldsymbol{\mu}}_k$ with Equation (8.9), the covariance matrices $\hat{\boldsymbol{\Sigma}}_k$ with Equation (8.10), and the priors p_k with Equation (8.35).
3. Calculate with Equation (8.34) the new class membership probabilities $u_{k\nu}$. Normalize the columns of $\boldsymbol{U}$.
4. If $\boldsymbol{U}$ has not changed significantly, stop, else go to 2.

8.3.1 Expectation maximization

Since the above algorithm has nothing to do with the simple cost functions of Section 8.1, we might ask why it should converge at all. The FMLE algorithm is in fact exactly equivalent to the application of the *Expectation Maximization* (EM) algorithm (Redner and Walker, 1984) to a Gaussian mixture model of the clustering problem. In that formulation, the probability density $p(\boldsymbol{g})$ for observing a value $\boldsymbol{g}$ in a given clustering configuration is modeled as a superposition of class-specific, multivariate normal probability density functions $p(\boldsymbol{g} \mid k)$ with mixing coefficients p_k:

$$p(\boldsymbol{g}) = \sum_{k=1}^{K} p(\boldsymbol{g} \mid k) p_k. \tag{8.36}$$

This expression has the same form as Equation (2.69), with the mixing coefficients p_k playing the role of the prior probabilities. To obtain the parameters of the model, the likelihood

$$\prod_{\nu=1}^{m} p(\boldsymbol{g}(\nu)) = \prod_{\nu=1}^{m} \left[\sum_{k=1}^{K} p(\boldsymbol{g}(\nu) \mid k) p_k \right] \tag{8.37}$$

is maximized. The maximization takes place in alternating *expectation* and *maximization* steps. In step 3 of the FMLE algorithm, the class membership probabilities $u_{k\nu} = p(k \mid \boldsymbol{g}(\nu))$ are computed. This corresponds to the expectation step of the EM algorithm in which the terms $p(\boldsymbol{g} \mid k)p_k$ of the mixture model are recalculated. In step 2 of the FMLE algorithm, the parameters for $p(\boldsymbol{g} \mid k)$, namely $\hat{\boldsymbol{\mu}}_k$ and $\hat{\boldsymbol{\Sigma}}_k$, and the mixture coefficients p_k are estimated for $k = 1 \dots K$. This corresponds to the EM maximization step; see also Exercise 9.

The likelihood must in fact increase on each iteration and therefore the algorithm will indeed converge to a (local) maximum in the likelihood. To demonstrate this, we follow Bishop (2006) and write Equation (8.37) in the more general form

$$p(\boldsymbol{\mathcal{G}} \mid \theta) = \prod_{\nu=1}^{m} \left[\sum_{k=1}^{K} p(\boldsymbol{g}(\nu) \mid \theta_k) p_k \right], \tag{8.38}$$

where $\boldsymbol{\mathcal{G}}$ is the observed dataset, θ_k represents the parameters $\mu_k, \boldsymbol{\Sigma}_k$ of the kth Gaussian distribution and θ is the set of all parameters of the model, $\theta = \{\mu_k, \boldsymbol{\Sigma}_k, p_k \mid k = 1 \dots K\}$.

Now suppose that the class labels were known. That is, in the notation of Chapter 6, suppose we were given

$$\ell = \{\boldsymbol{\ell}_1 \dots \boldsymbol{\ell}_m\}, \quad \boldsymbol{\ell}_\nu = (0, \dots 1, \dots 0)^\top,$$

where, if observation ν is in class k, $(\boldsymbol{\ell}_\nu)_k = \ell_{\nu k} = 1$ and the other components are 0. Then it would only be necessary to maximize the likelihood function

$$p(\boldsymbol{\mathcal{G}}, \ell \mid \theta) = \prod_{\nu=1}^{m} \left[\sum_{k=1}^{K} \ell_{\nu k} p(\boldsymbol{g}(\nu) \mid \theta_k) p_k \right] = \prod_{\nu=1}^{m} \prod_{k=1}^{K} p(\boldsymbol{g}(\nu) \mid \theta_k)^{\ell_{\nu k}} p_k^{\ell_{\nu k}}, \tag{8.39}$$

which is straightforward, since it can be done separately for each class. Thus, taking the logarithm of Equation (8.39), we get the log-likelihood

$$\log p(\boldsymbol{\mathcal{G}}, \ell \mid \theta) = \sum_{\nu} \sum_{k} \log \left[p(\boldsymbol{g}(\nu) \mid \theta_k) p_k \right] \ell_{\nu k} = \sum_{k} \sum_{\nu \in C_k} \log \left[p(\boldsymbol{g}(\nu) \mid \theta_k) p_k \right].$$

However, the variables ℓ are unknown; they are referred to as *latent* variables (Bishop, 2006). Let us postulate an unknown mass function $q(\ell)$ for the latent variables. Then the likelihood function $p(\boldsymbol{\mathcal{G}} \mid \theta)$, Equation (8.38), can be expressed as an average of $p(\boldsymbol{\mathcal{G}}, \ell \mid \theta)$ over $q(\ell)$,

$$p(\boldsymbol{\mathcal{G}} \mid \theta) = \sum_{\ell} p(\boldsymbol{\mathcal{G}} \mid \ell, \theta) q(\ell) = \sum_{\ell} p(\boldsymbol{\mathcal{G}}, \ell \mid \theta). \tag{8.40}$$

The second equality above follows from the definition of conditional probability, Equation (2.65). The log-likelihood may now be separated into two terms,

$$\log p(\boldsymbol{\mathcal{G}} \mid \theta) = \mathcal{L}(q, \theta) + \mathrm{KL}(q, p), \tag{8.41}$$

where

$$\mathcal{L}(q, \theta) = \sum_{\ell} q(\ell) \log \left(\frac{p(\boldsymbol{\mathcal{G}}, \ell \mid \theta)}{q(\ell)} \right) \tag{8.42}$$

and

$$\mathrm{KL}(q, p) = -\sum_{\ell} q(\ell) \log \left(\frac{p(\ell \mid \boldsymbol{\mathcal{G}}, \theta)}{q(\ell)} \right). \tag{8.43}$$

This decomposition can be seen immediately by writing

$$p(\boldsymbol{\mathcal{G}}, \ell \mid \theta) = p(\ell \mid \boldsymbol{\mathcal{G}}, \theta) p(\boldsymbol{\mathcal{G}} \mid \theta),$$

which again follows from the definition of conditional probability, and expanding Equation (8.42):

$$\begin{aligned} \mathcal{L}(q, \theta) &= \sum_{\ell} q(\ell)[\log p(\ell \mid \boldsymbol{\mathcal{G}}, \theta) + \log p(\boldsymbol{\mathcal{G}} \mid \theta) - \log q(\ell)] \\ &= -\mathrm{KL}(q, p) + \sum_{\ell} q(\ell) \log p(\boldsymbol{\mathcal{G}} \mid \theta) = -\mathrm{KL}(q, p) + \log p(\boldsymbol{\mathcal{G}} \mid \theta). \end{aligned}$$

This is just Equation (8.41).

From Section 2.7.1 we recognize $\mathrm{KL}(q, p)$ as the Kullback–Leibler divergence between $q(\ell)$ and the posterior density $p(\ell \mid \boldsymbol{\mathcal{G}}, \theta)$. Since $\mathrm{KL}(q, p) \geq 0$, with equality only when the two densities are equal, it follows that

$$\log p(\boldsymbol{\mathcal{G}} \mid \theta) \geq \mathcal{L}(q, \theta),$$

that is, $\mathcal{L}(q, \theta)$ is a lower bound for $\log p(\boldsymbol{\mathcal{G}} \mid \theta)$. The EM procedure is then as follows:

- **E-step:** Let the current best set of parameters be θ'. The lower bound on the log-likelihood is maximized by estimating $q(\ell)$ as $p(\ell \mid \boldsymbol{\mathcal{G}}, \theta')$, causing the $\mathrm{KL}(q, p)$ term to vanish and not affecting the log-likelihood since the likelihood does not depend on $q(\ell)$; see Equation (8.40). At this stage $\log p(\boldsymbol{\mathcal{G}} \mid \theta') = \mathcal{L}(q, \theta')$.

- **M-step:** Now $q(\ell)$ is held fixed and $p(\mathcal{G}, \ell \mid \theta)$ is maximized wrt θ (which, as we saw above, is straightforward), giving a new set of parameters θ'', and causing the lower bound $\mathcal{L}(q, \theta'')$ again to increase (unless it is already at a maximum). This will necessarily cause the log-likelihood to increase, since $q(\ell)$ is not the same as $p(\ell \mid \mathcal{G}, \theta'')$ and therefore the Kullback–Leibler divergence will be positive. Now the E-step can be repeated. Iteration of the two steps will therefore always cause the log-likelihood to increase until a maximum is reached.

At any stage of the iteration of the FMLE algorithm, the current estimate for the log-likelihood for the dataset can be shown to be given by (Bishop, 1995)

$$\sum_{\nu=1}^{m} \log p(\boldsymbol{g}(\nu)) = \sum_{k=1}^{K} \sum_{\nu=1}^{m} [u_{k\nu}]^o \log u_{k\nu}. \tag{8.44}$$

Here $[u_{k\nu}]^o$ is the "old" value of the posterior class membership probability, i.e., the value determined on the previous step.

8.3.2 Simulated annealing

Notwithstanding the initialization procedure, the FMLE (or EM) algorithm, like all iterative methods, may be trapped in a local optimum. A remedial scheme is to apply a technique called *simulated annealing*. The membership probabilities in the early iterations are given a strong random component and only gradually are the calculated class values allowed to influence the estimation of the class means and covariance matrices. The rate of reduction of randomness may be determined by a temperature parameter T, e.g.,

$$u_{k\nu} \to u_{k\nu}(1 - r^{1/T}) \tag{8.45}$$

on each iteration, where $r \in [0, 1]$ is a uniformly distributed random number and where T is initialized to some maximum value T_0 (Hilger, 2001). The temperature T is reduced at each iteration by a factor $c < 1$ according to $T \to cT$. As T approaches zero, $u_{ki\nu}$ will be determined more and more by the probability distribution parameters in Equation (8.34) alone.

8.3.3 Partition density

Since the sum of squares cost function $E(C)$ in Equation (8.12) is no longer relevant, we choose with Gath and Geva (1989) the *partition density* as a possible criterion for selecting the best number of clusters. To obtain it, first of all define the *fuzzy hyper-volume* as

$$FHV = \sum_{k=1}^{K} \sqrt{|\hat{\boldsymbol{\Sigma}}_k|}. \tag{8.46}$$

Listing 8.4: Gaussian mixture clustering (excerpt from the script `em.py`).

```
    dU = 1.0
    itr = 0
    T = T0
    print ('running␣EM␣on␣%i␣pixel␣vectors'%m)
    while ((dU > 0.001) or (itr < 10)) and (itr < 500):
        Uold = U+0.0
        ms = np.sum(U,axis=1)
        # prior probabilities
        Ps = np.asarray(ms/m).ravel()
        # cluster means
        Ms = np.asarray((np.mat(U)*np.mat(G)).T)
        # loop over the cluster index
        for k in range(K):
            Ms[:,k] = Ms[:,k]/ms[k]
            W = np.tile(Ms[:,k].ravel(),(m,1))
            Ds = G - W
            # covariance matrix
            for i in range(N):
                W[:,i] = np.sqrt(U[k,:]) \
                    .ravel()*Ds[:,i].ravel()
            C = np.mat(W).T*np.mat(W)/ms[k]
            Cs[k,:,:] = C
            sqrtdetC = np.sqrt(np.linalg.det(C))
            Cinv = np.linalg.inv(C)
            qf = np.asarray(np.sum(np.multiply(Ds \
                        ,np.mat(Ds)*Cinv),1)).ravel()
            # class hypervolume and partition density
            fhv[k] = sqrtdetC
            idx = np.where(qf < 1.0)
            pdens[k] = np.sum(U[k,idx])/fhv[k]
            # new memberships
            U[k,unfrozen] = np.exp(-qf[unfrozen]/2.0)\
                   *(Ps[k]/sqrtdetC)
            # random membership for annealing
            if T > 0.0:
                Ur = 1.0 - np.random\
                     .random(len(unfrozen))**(1.0/T)
                U[k,unfrozen] = U[k,unfrozen]*Ur
```

This expression is proportional to the volume in feature space occupied by the ellipsoidal clusters generated by the algorithm. For instance, for a two-dimensional cluster with a normal (elliptical) probability density we have, in its principal axis coordinate system,

$$\sqrt{|\hat{\Sigma}|} = \sqrt{\begin{vmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{vmatrix}} = \sigma_1\sigma_2 \approx \text{ area (volume) of the ellipse.}$$

Next, let s be the sum of all membership probabilities for observations which lie within unit Mahalanobis distance of a cluster center:

$$s = \sum_{\nu \in \mathcal{N}} \sum_{k=1}^{K} u_{\nu k}, \quad \mathcal{N} = \{\nu \mid (\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)^\top \hat{\boldsymbol{\Sigma}}_k^{-1} (\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k) < 1\}.$$

Then the partition density is defined as

$$P_D = s/FHV. \tag{8.47}$$

Assuming that the data consist of $\tilde{K}$ well-separated clusters of approximately multivariate normally distributed pixels, the partition density should exhibit a maximum at $K = \tilde{K}$.

8.3.4 Implementation notes

A Python script `em.py` for FMLE (or EM) clustering, is documented in Appendix C. The need for continual re-estimation of $\boldsymbol{\Sigma}_k$ is computationally expensive, so the algorithm is not suited for clustering large and/or high-dimensional datasets. The necessity to invert the covariance matrices can also occasionally lead to instability. Listing 8.4 shows an excerpt from the script. In the listing, `N` is the dimension of the observations, `K` is the number of classes, and `U` is the class membership matrix. By means of the index array variable `unfrozen`, the observations partaking in the clustering process can be additionally specified. In program line 32, only the membership probabilities $u_{j\nu}$ for “unfrozen” observations are recalculated, while the remaining observations are “frozen” to their current values; see, e.g., Bruzzone and Prieto (2000). This will be made use of in Section 8.4.1 below. Since the cluster memberships are multivariate probability densities, the script optionally generates a probability image; see Chapters 6 and 7. This image can be post-processed, for example, with the probabilistic label relaxation filter discussed in Section 7.1.2. We will postpone giving an example of EM clustering until we have discussed the inclusion of spatial information into the algorithm.

8.4 Including spatial information

All of the clustering algorithms described so far make use exclusively of the spectral properties of the individual observations (pixel vectors). Spatial relationships within an image, such as scale, coherent regions or textures, are not taken into account. In the following, we look at two refinements which deal with spatial context. They will be illustrated in connection with Gaussian mixture classification but could equally well be applied to other clustering approaches.

Listing 8.5: Multiresolution clustering (excerpt from the script `em.py`).

```
    #  cluster at minimum scale
    try:
        U,Ms,Cs,Ps,pdens = em(G,U,T0,beta,rows,cols)
    except:
        print('em␣failed')
        return
    #  sort clusters wrt partition density
    idx = np.argsort(pdens)
    idx = idx[::-1]
    U = U[idx,:]
    #  clustering at increasing scales
    for i in range(max_scale-min_scale):
        # expand U and renormalize
        U = np.reshape(U,(K,rows,cols))
        rows = rows*2
        cols = cols*2
        U = nd.zoom(U,(1,2,2))
        U = np.reshape(U,(K,rows*cols))
        idx = np.where(U<0.0)
        U[idx] = 0.0
        den = np.sum(U,axis=0)
        for j in range(K):
            U[j,:] = U[j,:]/den
        # expand the image
        for i in range(bands):
            DWTbands[i].invert()
        G = [DWTbands[i].get_quadrant(
              0,float=True).ravel()
                 for i in range(bands)]
        G = np.transpose(np.array(G))
        # cluster
        unfrozen = np.where(np.max(U,axis=0) < 0.90)
        try:
            U,Ms,Cs,Ps,pdens=em(G,U,0.0,beta,rows,cols,
                                         unfrozen=unfrozen)
        except:
            print('em␣failed')
            return
```

8.4.1 Multiresolution clustering

Prior to performing Gaussian mixture clustering with the EM algorithm, an image pyramid for the chosen scene may be created using, for example, the discrete wavelet transform (DWT) filter bank described in Chapter 4. Clustering will then begin at the coarsest resolution and proceed to finer resolutions, with the membership probabilities passed on to each successive scale (Hilger, 2001). In our implementation, the membership probabilities from preceding scales which, after up-sampling to the next scale, exceed some threshold (e.g., 0.9) are frozen in the manner discussed in Section 8.3.4. The depth of the pyramid can be chosen by the user.

The relevant program code is shown in Listing 8.5. The variable `DWTbands` is a list of `DWTArray` objects, one for each image band, see Section 4.3.2. The initial clustering carried out at the lowest resolution (the call to the procedure `em()` in line 3) is gradually refined in the `for`-loop beginning at line 12, whereby pixel vectors determined to have sufficiently high membership probabilities (≥ 0.9, line 32) at the lower resolutions are not reclassified. In line 8, the clusters are sorted according to decreasing partition density.[*] Apart from improving the rate of convergence of the calculation, scaling has the intended effect that the "spatial awareness" extracted at coarse resolution is passed up to the finer scales.

Listing 8.6: Spatial clustering (excerpt from the script `em.py`).

```
        # spatial membership
        if beta > 0:
            # normalize class probabilities
            a = np.sum(U,axis=0)
            idx = np.where(a == 0)[0]
            a[idx] = 1.0
            for k in range(K):
                U[k,:] = U[k,:]/a
            for k in range(K):
                U_N = 1.0 - nd.convolve(
                    np.reshape(U[k,:],(rows,cols)),Nb)
                V[k,:] = np.exp(-beta*U_N).ravel()
            # combine spectral/spatial
            U[:,unfrozen] = U[:,unfrozen]*V[:,unfrozen]
```

8.4.2 Spatial clustering

As described in Chapter 4, class labels for multispectral images can be represented by realizations of a Markov random field, for which the label of a given pixel may be influenced only by the class labels of other pixels in its immediate neighborhood. According to Gibbs–Markov equivalence, Theorem 4.3, the

[*]This will be particularly useful in Chapter 9 when the algorithm is used to cluster multispectral change images, since the no-change pixels usually form the most dense cluster.

probability density for any complete labeling ℓ of the image is given by

$$p(\ell) = \frac{1}{Z}\exp(-\beta U(\ell)), \tag{8.48}$$

where Z is a normalization and the energy function $U(\ell)$ is given by a sum over clique potentials,

$$U(\ell) = \sum_{c\in\mathcal{C}} V_c(\ell), \tag{8.49}$$

relative to a neighborhood system $\mathcal{N}$. If we restrict discussion to 4-neighborhoods, the only possible cliques are singletons and pairs of vertically or horizontally adjacent pixels, see Figure 4.15(a). If, furthermore, the potential for singleton cliques is set to zero and the random field is isotropic (independent of clique orientation), then we can write Equation (8.49) in the form

$$U(\ell) = \sum_{\nu\in\mathcal{I}} \sum_{\nu'\in\mathcal{N}_\nu} V_2(\ell_\nu, \ell_{\nu'}), \tag{8.50}$$

where $\mathcal{I}$ is the complete image lattice, $\mathcal{N}_\nu$ is the neighborhood of pixel ν and $V_2(\ell_\nu, \ell_{\nu'})$ is the clique potential for two neighboring sites. Let us now choose

$$V_2(\ell_\nu, \ell_{\nu'}) = \frac{1}{4}(1 - u_{\ell_\nu\nu'}), \tag{8.51}$$

where $u_{\ell_\nu\nu'}$ is an element of the cluster membership probability matrix $\boldsymbol{U}$. This says that, when the probability $u_{\ell_\nu\nu'}$ is large that neighboring site ν' has the same label ℓ_ν as site ν, then the clique potential is small and the configuration is favored. Combining Equations (8.50) and (8.51),

$$U(\ell) = \sum_{\nu\in\mathcal{I}} \frac{1}{4}\left(4 - \sum_{\nu'\in\mathcal{N}_\nu} u_{\ell_\nu\nu'}\right) = \sum_{\nu\in\mathcal{I}}(1 - u_{\ell_\nu\mathcal{N}_\nu}). \tag{8.52}$$

Here $u_{\ell_\nu\mathcal{N}_\nu}$ is the averaged membership probability for ℓ_ν within the neighborhood,

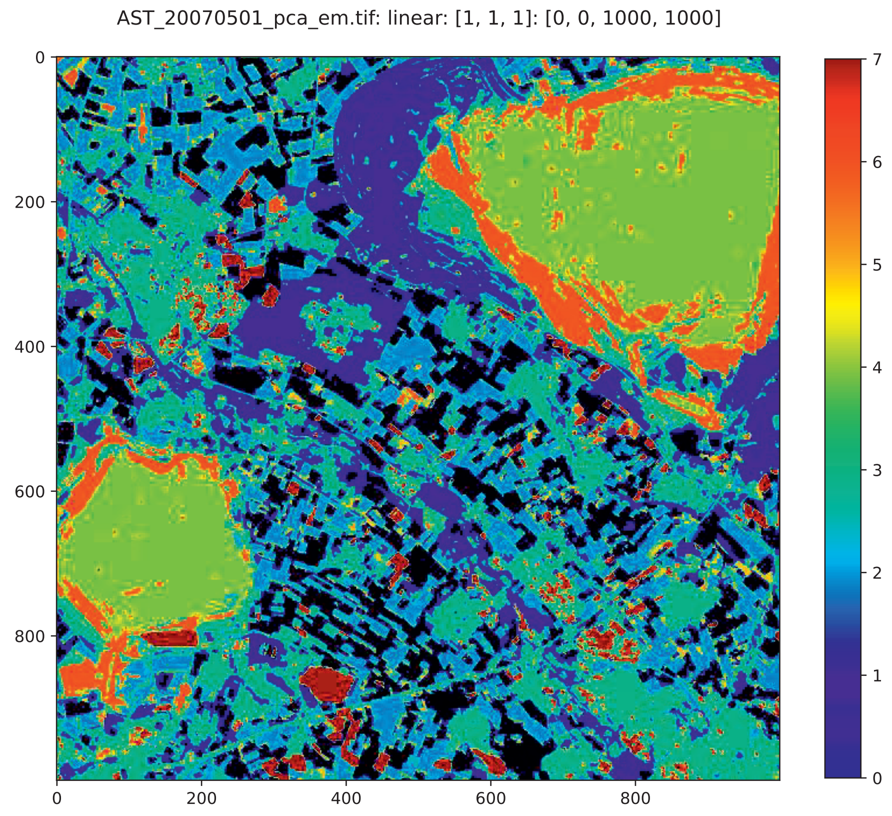
$$u_{\ell_\nu\mathcal{N}_\nu} = \frac{1}{4}\sum_{\nu'\in\mathcal{N}_\nu} u_{\ell_\nu\nu'}\ . \tag{8.53}$$

Substituting Equation (8.52) into Equation (8.48), we obtain

$$p(\ell) = \frac{1}{Z}\prod_{\nu\in\mathcal{I}} \exp(-\beta(1 - u_{\ell_\nu\mathcal{N}_\nu})). \tag{8.54}$$

Equation (8.54) is reminiscent of the likelihood functions that we have been using for pixel-based clustering and suggests the following heuristic ansatz (Hilger, 2001): Along with the *spectral* class membership probabilities $u_{k\nu}$ that characterize the FMLE algorithm and which are given by Equation (8.34), introduce a *spatial* class membership probability $v_{k\nu}$,

$$v_{k\nu} \propto \exp(-\beta(1 - u_{k\mathcal{N}_\nu})). \tag{8.55}$$



AST_20070501_pca_em.tif: linear: [1, 1, 1]: [0, 0, 1000, 1000]
0
200
400
600
800
0
1
2
3
4
5
6
7

FIGURE 8.5
Gaussian mixture clustering of the first four principal components of the ASTER scene of Figure 6.1, eight clusters.

A combined *spectral-spatial* class membership probability for the νth observation is then determined by replacing $u_{k\nu}$ with

$$\frac{u_{k\nu}v_{k\nu}}{\sum_{k'=1}^{K} u_{k'\nu}v_{k'\nu}}, \tag{8.56}$$

apart from which the algorithm proceeds as before. Note that, from Equations (8.34) and (8.55), the combined membership probability above is now proportional to

$$\frac{1}{\sqrt{|\hat{\boldsymbol{\Sigma}}_k|}} \cdot \frac{m_k}{m} \cdot \exp\Big(-\frac{1}{2}(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k)^\top \boldsymbol{\mathcal{C}}_k^{-1}(\boldsymbol{g}(\nu) - \hat{\boldsymbol{\mu}}_k) - \beta(1 - u_{k\mathcal{N}_\nu})\Big).$$

This way of folding spatial information with the spectral similarity measure (here the Mahalanobis distance) is referred to in Tran et al. (2005) as the *addition form*.

The quantity $u_{k\mathcal{N}_\nu}$ appearing in Equation (8.55) can be determined for an entire image simply by convolving the two-dimensional kernel

$$\begin{matrix} 0 & \frac{1}{4} & 0 \\ \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & \frac{1}{4} & 0 \end{matrix}$$

with the array $\boldsymbol{U}$, after reforming the latter to the correct image dimensions; see Listing 8.6. (The kernel is stored in the variable `Nb`.)

Figure 8.5 shows an example, again using the ASTER image. The classification was determined with an image pyramid of depth 2 (i.e., three levels), annealing temperature $T_0 = 0.5$ and with $\beta = 0.5$; see the accompanying Jupyter notebook.

8.5 A benchmark

If we include fuzzy K-means, we now have introduced no less than six pixel-oriented clustering methods. So which one should we use? The degree of success of unsupervised image classification is notoriously difficult to quantify; see, e.g., Duda and Canty (2002). This is because there is, by definition, no prior information with regard to what one might expect to be a “reasonable” result. Ultimately, judgment is qualitative, almost a question of aesthetics.

Nevertheless, in order to compare the algorithms we have looked at so far more objectively, we will generate a “toy” benchmark image with the code:

```
from osgeo.gdalconst import GDT_Float32

image = np.zeros((800,800,3))
b = 2.0
image[99:699 ,299:499 ,:] = b
image[299:499 ,99:699 ,:] = b
image[299:499 ,299:499 ,:] = 2*b
n1 = np.random.randn(800,800)
n2 = np.random.randn(800,800)
n3 = np.random.randn(800,800)
image[:,:,0] += n1
image[:,:,1] += n2+n1
image[:,:,2] += n3+n1/2+n2/2
driver = gdal.GetDriverByName('GTiff')
outDataset = driver.Create('imagery/toy.tif',
                           800,800,3,GDT_Float32)
for k in range(3):
    outBand = outDataset.GetRasterBand(k+1)
    outBand.WriteArray(image[:,:,k],0,0)
    outBand.FlushCache()
outDataset = None
```

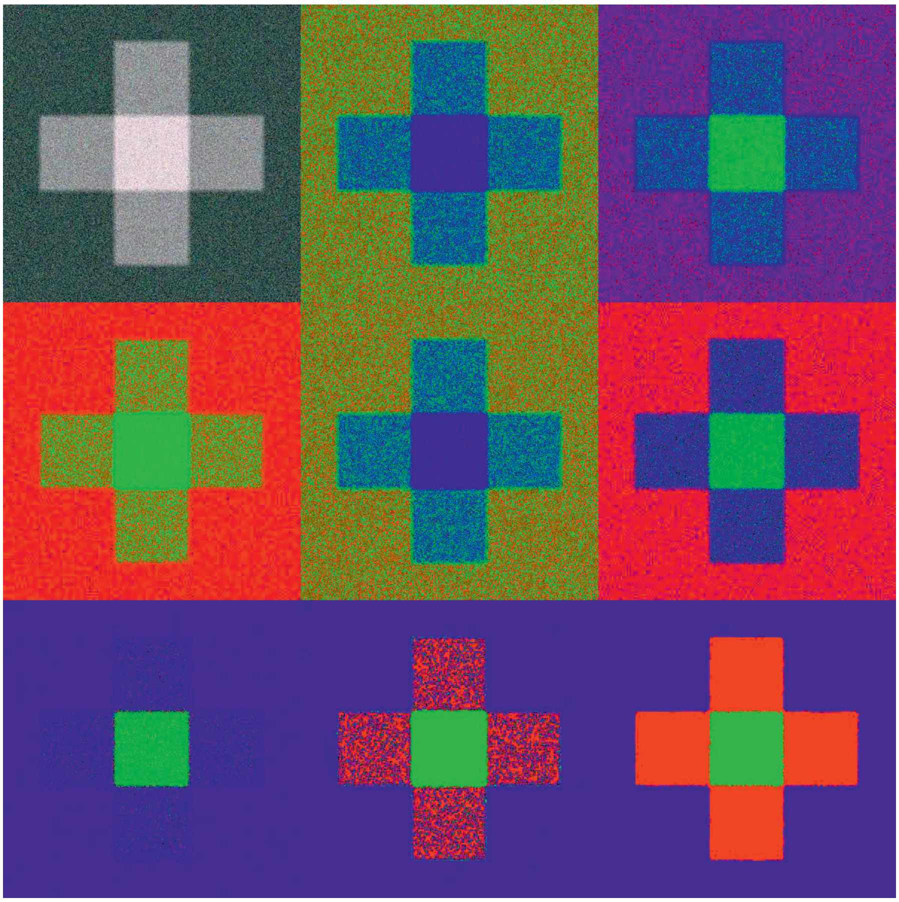


FIGURE 8.6
Unsupervised classification of a toy image. Row-wise, left to right, top to bottom: The toy image, K-means, kernel K-means, extended K-means (on first principal component), fuzzy K-means, agglomerative hierarchical, Gaussian mixture with depth 0 and $\beta = 1.0$, Gaussian mixture with depth 2 and $\beta = 0.0$, Gaussian mixture with depth 2 and $\beta = 1.0$.

The image is shown in the upper left-hand corner of Figure 8.6. It consists of three "spectral bands" and three classes, namely the dark background, the four points of the cross and the cross center. The cluster sizes differ considerably (approximately in the ratio 12:4:1) with a large overlap, and the bands are strongly correlated. The program in Listing 3.4 estimates the image noise covariance matrix:

```
run scripts/ex3_2 imagery/toy.tif

Noise covariance, file imagery/toy.tif
```

```
[[1.00600974 1.00667639 0.50545603]
 [1.00667639 2.00463715 1.00493322]
 [0.50545603 1.00493322 1.50670931]]
```

Some results are shown in Figure 8.6. Neither the K-means variants nor the Gaussian mixture algorithm without scaling identify all three classes. Agglomerative hierarchical clustering succeeds reasonably well. The "best" classification is obtained with the Gaussian mixture model when both multiresolution and spatial clustering are employed. The success of Gaussian mixture classification is not particularly surprising since the toy image classes are multivariate normally distributed. However, if the other methods exhibit inferior performance on normal distributions, then they might be expected to be similarly inferior when presented with real, non-Gaussian data.

8.6 The Kohonen self-organizing map

The *Kohonen self-organizing map* (SOM), a simple example of which is sketched in Figure 8.7, belongs to a class of neural networks which are trained by

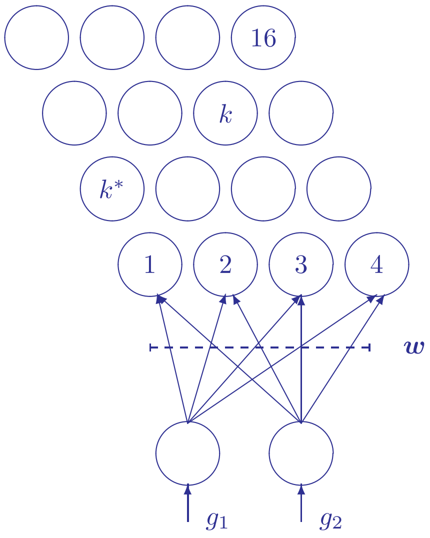


FIGURE 8.7
The Kohonen self-organizing map in two dimensions with a two-dimensional input. The inputs are connected to all 16 neurons (only the first four connections are shown).

competitive learning (Hertz et al., 1991; Kohonen, 1989). It is very useful as a visualization tool for exploring the class structure of multispectral imagery. The layer of neurons shown in the figure can have any geometry, but usually a one-, two-, or three-dimensional array is chosen. The input signal is the observation vector $\boldsymbol{g} = (g_1, g_2 \dots g_N)^\top$, where, in the figure, $N = 2$. Each input to a neuron is associated with a *synaptic weight* so that, for K neurons, the synaptic weights can be represented as an $(N \times K)$ matrix

$$\boldsymbol{w} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1K} \\ w_{21} & w_{22} & \cdots & w_{2K} \\ \vdots & \vdots & \vdots & \vdots \\ w_{N1} & w_{N2} & \cdots & w_{NK} \end{pmatrix}. \tag{8.57}$$

The components of the vector $\boldsymbol{w}_k = (w_{1k}, w_{2k} \dots w_{Nk})^\top$ are the synaptic weights of the kth neuron. The set of observations $\{\boldsymbol{g}(\nu) \mid \nu = 1 \dots m\}$, where $m > K$, comprise the training data for the network. The synaptic weight vectors are to be adjusted so as to reflect in some way the class structure of the training data in an N-dimensional feature space.

The training procedure is as follows. First of all, the neurons' weights are initialized from a random subset of K training observations, setting

$$\boldsymbol{w}_k = \boldsymbol{g}(k), \quad k = 1 \dots K.$$

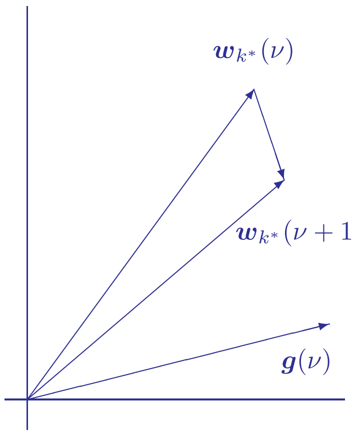


FIGURE 8.8
Movement of a synaptic weight vector in the direction of a training vector.

Then, when a new training vector $\boldsymbol{g}(\nu)$ is presented to the input of the network, the neuron whose weight vector $\boldsymbol{w}_k$ lies nearest to $\boldsymbol{g}(\nu)$ is designated to be the "winner." Distances are given by $\|\boldsymbol{g}(\nu) - \boldsymbol{w}_k\|$. Suppose that the winner's index is k^*. Its weight vector is then shifted a small amount in the direction of the training vector:

$$\boldsymbol{w}_{k^*}(\nu + 1) = \boldsymbol{w}_{k^*}(\nu) + \eta(\boldsymbol{g}(\nu) - \boldsymbol{w}_{k^*}(\nu)), \tag{8.58}$$

where $\boldsymbol{w}_{k^*}(\nu + 1)$ is the weight vector after presentation of the νth training vector; see Figure 8.8. The parameter η is called the *learning rate* of the network. The intention is to repeat this learning procedure until the synaptic weight vectors reflect the class structure of the training data, thereby achieving what is often referred to as a *vector quantization* of the feature space (Hertz et al., 1991). In order for this method to converge, it is necessary to allow the learning rate to decrease gradually during the training process. A convenient function for this is

$$\eta(\nu) = \eta_{max}\left(\frac{\eta_{min}}{\eta_{max}}\right)^{\nu/m}.$$

However, the SOM algorithm goes a step further and attempts to map the *topology* of the feature space onto the network as well. This is achieved by defining a neighborhood function for the winner neuron on the network of neurons. Usually a Gaussian of the form

$$\mathcal{N}(k^*,k) = \exp(-d^2(k^*,k)/2\sigma^2)$$

is chosen, where $d^2(k^*,k)$ is the square of the distance between neurons k^* and k in the network array. For example, for a two-dimensional array of $n \times n$ neurons $d^2(k^*,k)$ can by calculated in integer arithmetic as

$$\begin{aligned} d^2(k^*,k) =& [(k^*-1) \bmod n - (k-1) \bmod n]^2 \\ &+ [(k^*-1)/n - (k-1)/n]^2. \end{aligned} \tag{8.59}$$

During the learning phase, not only the weight vectors of the winner neuron, but also those of the neurons in its neighborhood, are moved in the direction of the training vectors by an amount proportional to the value of the neighborhood function:

$$\boldsymbol{w}_k(\nu+1) = \boldsymbol{w}_k(\nu) + \eta(\nu)\mathcal{N}(k^*,k)(\boldsymbol{g}(\nu) - \boldsymbol{w}_k(\nu)), \quad k = 1\dots K. \tag{8.60}$$

Finally, the extent of the neighborhood is allowed to shrink steadily as well:

$$\sigma(\nu) = \sigma_{max}\left(\frac{\sigma_{min}}{\sigma_{max}}\right)^{\nu/m}.$$

Typically, $\sigma_{max} \approx n/2$ and $\sigma_{min} \approx 1/2$. The neighborhood is initially the entire network, but toward the end of training it becomes very localized.

For clustering of multispectral satellite imagery a cubic network geometry is useful (Groß and Seibert, 1993). After training on some representative sample of pixel vectors, the entire image is classified by associating each pixel vector with the neuron having the closest synaptic weight vector. Then the pixel is colored by mapping the position of that neuron in the cube to coordinates in RGB color space. Thus, the N-dimensional feature space is "projected" onto the three-dimensional RGB color cube and pixels that are close together in feature space are given similar colors. A Python script for the Kohonen self-organizing map is documented in Appendix C. Running it on all nine bands of the Jülich ASTER image:

```
run scripts/som -c 6  imagery/AST_20070501.tif

--------SOM ------------
```

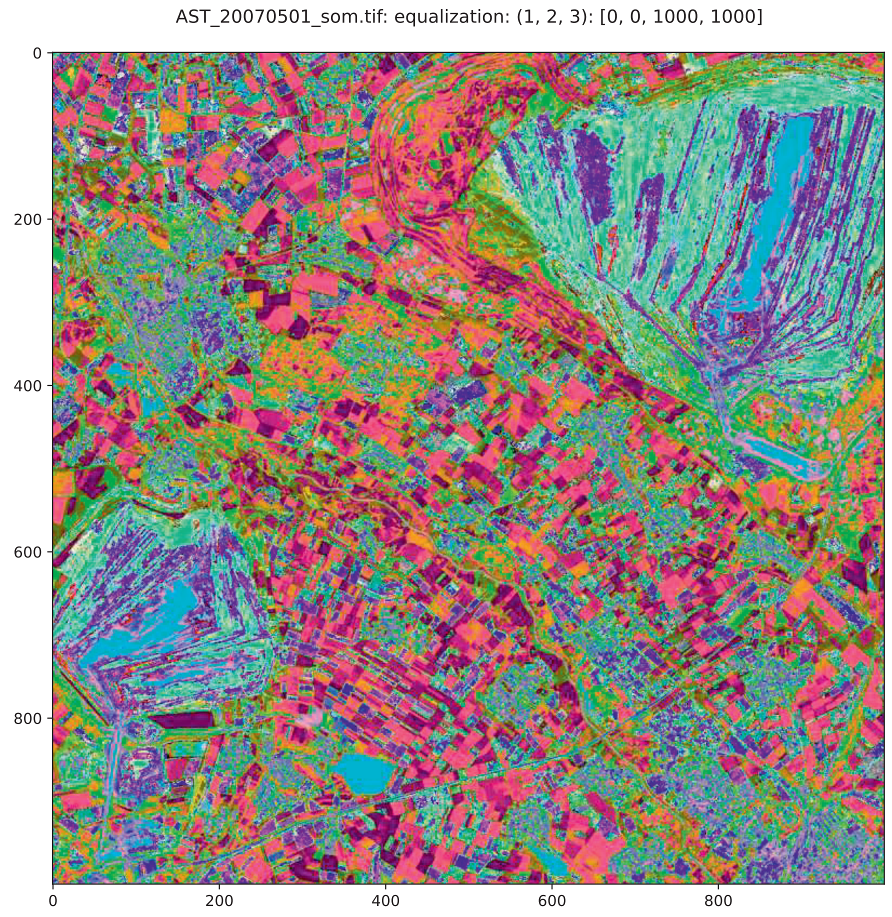

AST_20070501_som.tif: equalization: (1, 2, 3): [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800

FIGURE 8.9
Kohonen self-organizing map of the nine VNIR and sharpened SWIR spectral bands of the ASTER scene of Figure 1.1. The network is a cube having dimensions $6 \times 6 \times 6$.

```
Sun Jun 16 15:16:59 2024
Input imagery/AST_20070501.tif
Color cube dimension 6
training...
elapsed time: 8.092823266983032
clustering...
elapsed time: 15.008501052856445
SOM written to: imagery/AST_20070501_som.tif
```

gives the Kohonen SOM shown in Figure 8.9.

8.7 Image segmentation and the mean shift

The term image segmentation refers, in its broadest sense, to the process of partitioning a digital image into multiple regions. Certainly all of the unsupervised classification algorithms that we have considered in the present Chapter fall within this category. However, here we will use the term in a more restrictive way, namely as referring to the partitioning of pixels not only by spectral similarity (essentially what has been discussed so far) but also by spatial proximity. This approach to segmentation plays a major role in low-level computer vision and in autonomous image processing in general. Popular methods include the use of edge detectors, watershed segmentation, region growing algorithms and morphology; see Gonzalez and Woods (2017), Chapter 10, for a good overview.

Segmentation also offers the possibility of a refinement of clustering on the basis of characteristics of the individual segments not necessarily related to pixel intensities, such as shape, compactness, proximity to other segments, etc. One speaks generally of *object-based* classification. We conclude this Chapter with the description of an especially popular, nonparametric algorithm called the *mean shift* (Fukunaga and Hostetler, 1975; Comaniciu and Meer, 2002).

The mean shift algorithm is non-parametric in the sense that no assumptions are made regarding the probability density of the observations being clustered. The mean shift partitions pixels in an N-dimensional multispectral feature space by associating each pixel with a local maximum in the estimated probability density, called a *mode*. For each pixel, the associated mode is determined by defining a (hyper-)sphere of radius $r_{spectral}$ centered at the pixel and calculating the mean of the pixels that lie within the sphere. Then the sphere's center is shifted to that mean. This continues until convergence, i.e., until the mean shift is less than some threshold. At each iteration, the sphere moves to a region of higher probability density until a stationary mode is reached. The pixel is assigned that mode.

It will be apparent from the above that mean shift clustering *per se* will not lead to image segmentation in the restricted sense that we are discussing here. However, simply by extending the feature space to include the spatial position of the pixels, an elegant segmentation algorithm emerges. We only need distinguish, additionally to $r_{spectral}$, a spatial radius $r_{spatial}$ for determining the mean shift. After an appropriate normalization of the spectral and spatial distances, for example, by re-scaling the pixel intensities according to

$$\boldsymbol{g}(\nu) \to \boldsymbol{g}(\nu)\frac{r_{spatial}}{r_{spectral}}, \quad \nu = 1 \dots m,$$

the mean shift procedure is carried out in the concatenated, $N+2$-dimensional, spectral-spatial feature space using a hyper-sphere of radius $r = r_{spatial}$.

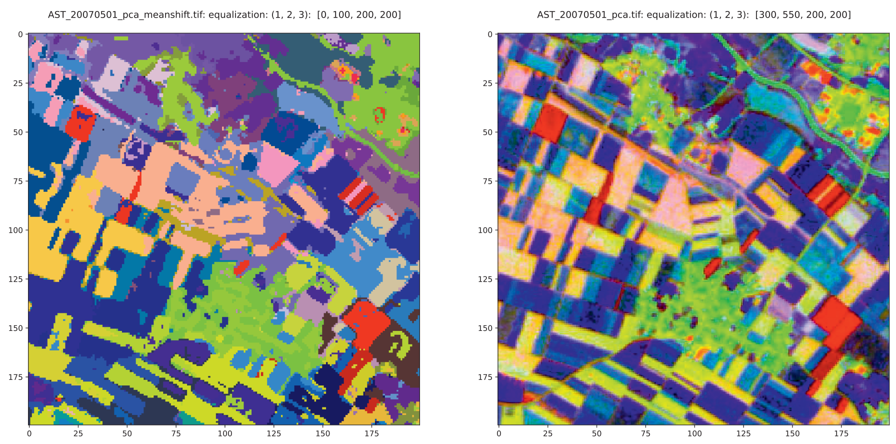

AST_20070501_pca_meanshift.tif: equalization: (1, 2, 3): [0, 100, 200, 200]
AST_20070501_pca.tif: equalization: (1, 2, 3): [300, 550, 200, 200]
0
25
50
75
100
125
150
175

FIGURE 8.10
Left: mean shift segmentation of the first four principal components (byte-stretched) of the ASTER scene of Figure 6.1 (spatial subset of the RGB composite of the first three bands of the segmented image). Right: RGB composite of the first three principal components.

Specifically, let $\boldsymbol{y}_\nu = \begin{pmatrix} \boldsymbol{g}(\nu) \\ \boldsymbol{x}_\nu \end{pmatrix}$, $\nu = 1 \dots m$, represent the image pixels in the combined feature space, let $\boldsymbol{z}_\nu$, $\nu = 1 \dots m$, denote the mean shift filtered pixels after the segmentation has concluded, and define $S(\boldsymbol{w})$ as the set of pixels within a radius r of a point $\boldsymbol{w}$ (cardinality $|S(\boldsymbol{w})|$). Then the algorithm is as follows:

Algorithm (Mean shift segmentation)

```
For each $\nu = 1 \dots m$, do the following:
    1. Set $k = 1$ and $\boldsymbol{w}_k = \boldsymbol{y}_\nu$.
    2. Repeat:
        $\boldsymbol{w}_{k+1} = \frac{1}{|S(\boldsymbol{w}_k)|} \sum_{\boldsymbol{y}_\nu \in S(\boldsymbol{w}_k)} \boldsymbol{y}_\nu$
        $k = k + 1$
       until convergence to $\boldsymbol{w} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{x} \end{pmatrix}$.
    3. Assign $\boldsymbol{z}_\nu = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{x}_\nu \end{pmatrix}$.
```

The last step assigns the filtered spectral component $\boldsymbol{f}$ (the spectral mode) to the original pixel location $\boldsymbol{x}_\nu$. Segmentation simply involves identifying all pixels with the same mode as a segment.

A direct implementation of this algorithm for multi-spectral image segmentation would be prohibitively slow, since mean shifts are to be computed for all of the pixels. In the Python script `meanshift.py` (Appendix C), two approximations are employed to speed things up. First, all pixels within radius r of convergence point $\boldsymbol{w}$ are assumed also to converge to that mode and are excluded from further processing. Second, all pixels lying sufficiently close (within a distance $r/3$) to the path traversed from the initial vector $\boldsymbol{y}_\nu$ to the mode $\boldsymbol{w}$ are similarly assigned to that mode and excluded from the calculation. Additionally, a user-defined minimum segment size can be specified. Segments with smaller extent than this minimum are assigned to the nearest segment in the combined spatial/spectral feature space. An example of the mean shift is shown in Figure 8.10. It was calculated with $r_{spatial} = 30$, $r_{spectral} = 15$ and a minimum segment size of 10 pixels:

```
run scripts/meanshift -p [1,2,3,4] -d [300,450,400,400] \
 -s 30 -r 15 -m 10 imagery/AST_20070501_pca.tif

 =========================
    mean shift
=========================
infile: imagery/AST_20070501_pca.tif
filtering pixels...
result written to: imagery/AST_20070501_pca_meanshift.tif
elapsed time: 149.2563214302063
```

8.8 Exercises

1. (a) Show that the sum of squares cost function, Equation (8.12), can be expressed equivalently as

$$E(C) = \frac{1}{2}\sum_{k=1}^{K} \bar{s}_k, \tag{8.61}$$

where $\bar{s}_k$ is the average squared distance between points in the kth cluster,

$$\bar{s}_k = \frac{1}{m_k}\sum_{i=1}^{m}\sum_{i'=1}^{m} u_{ki}u_{ki'}\|\boldsymbol{g}_i - \boldsymbol{g}_{i'}\|^2. \tag{8.62}$$

(b) Recall that, in the notation of Section 8.2.2,

$$\mathcal{\boldsymbol{G}}^\top \boldsymbol{U}^\top \boldsymbol{M} = (\hat{\boldsymbol{\mu}}_1 \dots \hat{\boldsymbol{\mu}}_k).$$

It follows that $(\mathcal{\boldsymbol{G}}^\top \boldsymbol{U}^\top \boldsymbol{M})\boldsymbol{U}$ is an $N \times m$ matrix whose νth column is the mean vector associated with the νth observation. Use this to demonstrate that the sum of squares cost function can also be written in the form

$$E(C) = \mathrm{tr}\big(\boldsymbol{\mathcal{G}}\boldsymbol{\mathcal{G}}^\top\big) - \mathrm{tr}\big(\boldsymbol{U}^\top \boldsymbol{M}\boldsymbol{U}\boldsymbol{\mathcal{G}}\boldsymbol{\mathcal{G}}^\top\big). \tag{8.63}$$

2. (a) The Python script `kkmeans.py` makes use of the Gaussian kernel only. Modify it to include the option of using polynomial kernels; see Chapter 4, Exercise 8.

   (b) The code

```
from osgeo.gdalconst import GDT_Float32
import numpy as np
import gdal

image = np.zeros((400,400,2))
n = np.random.randn(400,400)
n1 = 8*np.random.rand(400,400)-4
image[:,:,0] = n1+8
image[:,:,1] = n1**2+0.3*np.random.randn(400,400)+8
image[:200,:,0] = np.random.randn(200,400)/2+8
image[:200,:,1] = np.random.randn(200,400)+14
driver = gdal.GetDriverByName('GTIFF')
outDataset = driver.Create('imagery/toy.tif',\
                            400,400,3,GDT_Float32)
for k in range(2):
    outBand= outDataset.GetRasterBand(k+1)
    outBand.WriteArray(image[:,:,k],0,0)
    outBand.FlushCache()
outDataset = None
```

   (see the accompanying Jupyter notebook) generates a two-band toy image with the two not linearly separable clusters shown in Figure 8.11. Experiment with the kernel K-means script to achieve a good clustering of the image. Compare your result with the Python K-means implementation in Listing 8.1.

3. Kernel K-means clustering is closely related to *spectral clustering* (see, e.g., Dhillon et al. (2005)). Let $\boldsymbol{\mathcal{K}}$ be an $m \times m$ Gaussian kernel matrix. Define the diagonal *degree* matrix

$$\boldsymbol{D} = \mathrm{Diag}(d_1 \dots d_m),$$

   where $d_i = \sum_{j=1}^m (\boldsymbol{\mathcal{K}})_{ij}$, and the symmetric *Laplacian* matrix

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{\mathcal{K}}. \tag{8.64}$$

   Minimization of the kernelized version of the sum of squares cost function, namely

$$E(C) = \mathrm{tr}\big(\boldsymbol{\mathcal{K}}\big) - \mathrm{tr}\big(\boldsymbol{U}^\top \boldsymbol{M}\boldsymbol{U}\boldsymbol{\mathcal{K}}\big),$$

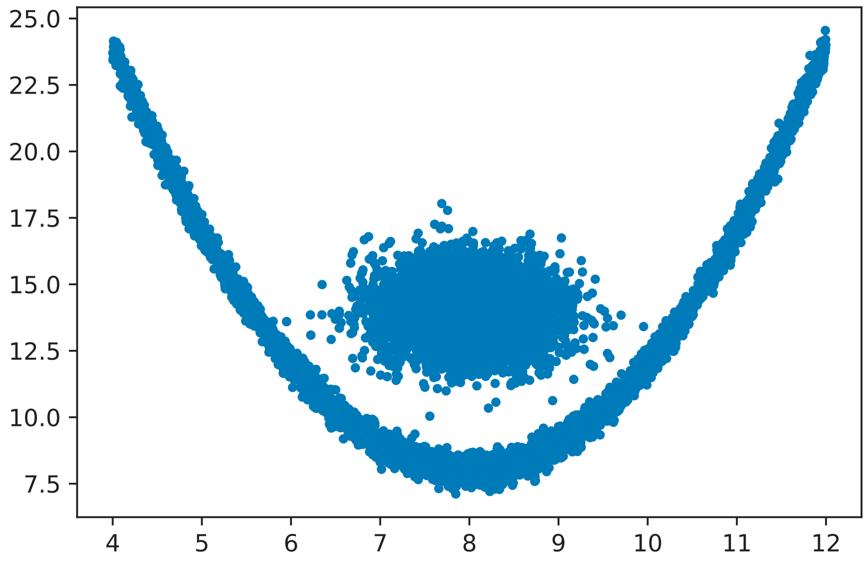


FIGURE 8.11
Two clusters which are not linearly separable.

see Equation (8.63), can be shown to be equivalent to ordinary (linear) clustering of the m training observations in the space of the first K eigenvectors of $\boldsymbol{L}$ corresponding to the K smallest eigenvalues (von Luxburg, 2006; Shawe-Taylor and Cristianini, 2004).

(a) Show that $\boldsymbol{L}$ is positive semi-definite and that its smallest eigenvalue is zero with corresponding eigenvector $\mathbf{1}_m$.

(b) Write a Python function `spectral_cluster(G,K)` to implement the following algorithm: (The input observations $\boldsymbol{g}(\nu)$, $\nu = 1 \dots m$, are passed to the function in the form of a data matrix $\boldsymbol{\mathcal{G}}$ along with K, the number of clusters.)

Algorithm (Unnormalized spectral clustering)

1. Determine the Laplacian matrix $\boldsymbol{L}$ as given by Equation (8.64).
2. Compute the $m \times K$ matrix $\boldsymbol{V} = (\boldsymbol{v}_1 \dots \boldsymbol{v}_K)$, the columns of which are the eigenvectors of $\boldsymbol{L}$ corresponding to the K smallest eigenvalues. Let $\boldsymbol{y}(\nu)$ be the K-component vector consisting of the νth row of $\boldsymbol{V}$, $\nu = 1 \dots m$.
3. Partition the vectors $\boldsymbol{y}(\nu)$ with the K-means algorithm into clusters $C_1 \dots C_K$.
4. Output the cluster labels for $\boldsymbol{g}(\nu)$ as those for $\boldsymbol{y}(\nu)$.

(c) Thinking of $\boldsymbol{\mathcal{G}}$ in the algorithm as a training dataset sampled from a multispectral image suggest how the clustering result might be generalized to the whole image.

4. Write a Python routine to give a first-order estimate of the entropy of an image spectral band.[*] Use the definition in Equation (8.18) and interpret p_i as the probability that a pixel in the image has intensity i, $i = 0 \dots 255$. (*Hint:* Some intensities may have zero probabilities. The logarithm of zero is undefined, but the product $p \cdot \log p \to 0$ as $p \to 0$.)

5. Modify the program in Listing 8.3 (EKM algorithm) to give a more robust estimation of the variance σ^2 by masking out edge pixels with a thresholded Sobel filter; see Section 5.2.1.

6. Using the modified program from Exercise 5 as a starting point, write a script to perform extended K-means clustering on image data with more than one spectral band. *Hint:* You might consider running the agglomerative hierarchic clustering algorithm with $K \approx 200$ to reduce the dimensionality and then apply the EKM algorithm to the result.

7. Derive the expression in Equation (8.27) for the net change in the sum of squares cost function when clusters k and j are merged.

8. Recall the definition of fuzzy hyper-volume in Section 8.3.3, namely

$$FHV = \sum_{k=1}^{K} \sqrt{|\hat{\boldsymbol{\Sigma}}_k|},$$

where $\hat{\boldsymbol{\Sigma}}_k$ is the estimated covariance matrix for the kth cluster, Equation (8.10). This quantity might itself be used as a cost function, since it measures the compactness of the clusters corresponding to some partitioning C. Suppose that the observations $\boldsymbol{g}(\nu)$ are re-scaled by some arbitrary non-singular transformation matrix $\boldsymbol{T}$, i.e., $\boldsymbol{g}'(\nu) = \boldsymbol{T}\boldsymbol{g}(\nu)$. Show that any algorithm which achieves partitioning C on the basis of the original observations $\boldsymbol{g}(\nu)$ by minimizing FHV will produce the same partitioning with the transformed observations $\boldsymbol{g}'(\nu)$.

9. (EM algorithm) Consider a set of one-dimensional observations $\{g(\nu) \mid \nu = 1 \dots m\}$ which are to be clustered into K classes according to a Gaussian mixture model of the form

$$p(g) = \sum_{k=1}^{K} p(g \mid k)\Pr(k),$$

[*] *First order* means that one assumes that all pixel intensities are statistically independent.

where the $\Pr(k)$ are weighting factors, $\sum_{k=1}^{K}\Pr(k)=1$, and

$$p(g \mid k) = \frac{1}{\sqrt{2\pi}\sigma_k}\exp\left(-\frac{(g-\mu_k)^2}{2\sigma_k^2}\right).$$

The log-likelihood for the observations is

$$\mathcal{L} = \log\prod_{\nu=1}^{m} p(g(\nu)) = \sum_{\nu=1}^{m}\log\left(\sum_{k=1}^{K} p(g(\nu) \mid k)\Pr(k)\right).$$

(a) Show, with the help of Bayes' Theorem, that this expression is maximized by the following values for the model parameters μ_k, σ_k and $\Pr(k)$:

$$\begin{aligned}\mu_k &= \frac{\sum_\nu \Pr(k \mid g(\nu))g(\nu)}{\sum_\nu \Pr(k \mid g(\nu))}\\ \sigma_k^2 &= \frac{\sum_\nu \Pr(k \mid g(\nu))(g(\nu)-\mu_k)^2}{\sum_\nu \Pr(k \mid g(\nu))}\\ \Pr(k) &= \frac{1}{m}\sum_\nu \Pr(k \mid g(\nu)).\end{aligned}$$

(b) The EM algorithm consists of iterating these three equations (in the order given), together with

$$\Pr(k \mid g(\nu)) \propto p(g(\nu) \mid k)\Pr(k), \quad \sum_k \Pr(k \mid g(\nu)) = 1,$$

until convergence. Explain why this is identical to the FMLE algorithm for one-dimensional data.

10. Give an integer arithmetic expression for $d^2(k^*, k)$ for a cubic geometry self-organizing map.

11. The traveling salesperson problem (TSP) is to find the shortest closed route between n points on a map (cities) which visits each point exactly once. Program an approximate solution to the TSP using a one-dimensional self-organizing map in the form of a closed loop.

12. (a) The mean shift segmentation algorithm of Section 8.7 is said to be "edge-preserving." Explain why this is so.

    (b) A uniform kernel was used for simplicity to determine the mean shift, the hyper-sphere S in the algorithm, which we can represent as

$$S(\boldsymbol{w}-\boldsymbol{y}_\nu) = \begin{cases}1 & \text{for } \|\boldsymbol{w}-\boldsymbol{y}_\nu\| \le r\\ 0 & \text{otherwise}\end{cases}.$$

We could alternatively have used the radial basis (Gaussian) kernel

$$k(\boldsymbol{w}-\boldsymbol{y}_\nu) = \exp\left(-\frac{\|\boldsymbol{w}-\boldsymbol{y}_\nu\|^2}{2r^2}\right).$$

The density estimate at the point $\boldsymbol{w}$ is (see Section 6.4)

$$p(\boldsymbol{w}) = \frac{1}{m(2\pi r^2)^{(N+2)/2}} \sum_{\nu=1}^{m} \exp\left(-\frac{\|\boldsymbol{w}-\boldsymbol{y}_\nu\|^2}{2r^2}\right)$$

and the estimated gradient at $\boldsymbol{w}$, i.e., the direction of maximum change in $p(\boldsymbol{w})$, is

$$\frac{\partial p(\boldsymbol{w})}{\partial \boldsymbol{w}}.$$

Show that the mean shift is always in the direction of the gradient. (This demonstrates that the mean shift algorithm will find the modes of the distribution without actually estimating its density, and can be shown to be a general result (Comaniciu and Meer, 2002).)

(c) The `OpenCV` Python package exports the function `PyrMeanShiftFiltering()`, which performs mean shift filtering of RGB color images. An image pyramid is first created and the filtering procedure is run on the smallest (top) layer first. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ by more than `sr` from the lower-resolution layer of the pyramid. In the code snippet:

```
import cv2 as cv
src = cv.fromarray(src)
dst = cv.CreateMat(rows, cols, cv.CV_8UC3)
cv.PyrMeanShiftFiltering(src, sp, sr, dst, maxLevel)
```

the variables are:

`src`: the source 8-bit, 3-channel image in a `numpy` array
`sp`: the spatial window radius
`sr`: the color window radius.
`dst`: the destination image, same format and same size as the source.
`maxLevel`: the maximum pyramid level, beginning at 0.

Write a Python script to perform mean shift segmentation of a three-band multispectral image with a two-level pyramid (`maxLevel=1`).

9

Change Detection

To quote a well-known, if now rather dated, review article on change detection (Singh, 1989),

> The basic premise in using remote sensing data for change detection is that changes in land cover must result in changes in radiance values ... [which] must be large with respect to radiance changes from other factors.

When comparing multispectral images of a given scene taken at different times, it is therefore desirable to correct the pixel intensities as much as possible for uninteresting differences such as those due to solar illumination, atmospheric conditions, viewing angle, terrain effects or sensor calibration. In the case of SAR imagery, solar illumination or cloud cover plays no role, but other considerations are similarly important. If comparison is on a pixel-by-pixel basis, then the images must also be co-registered to high accuracy in order to avoid spurious signals resulting from misaligned pixels. Some of the

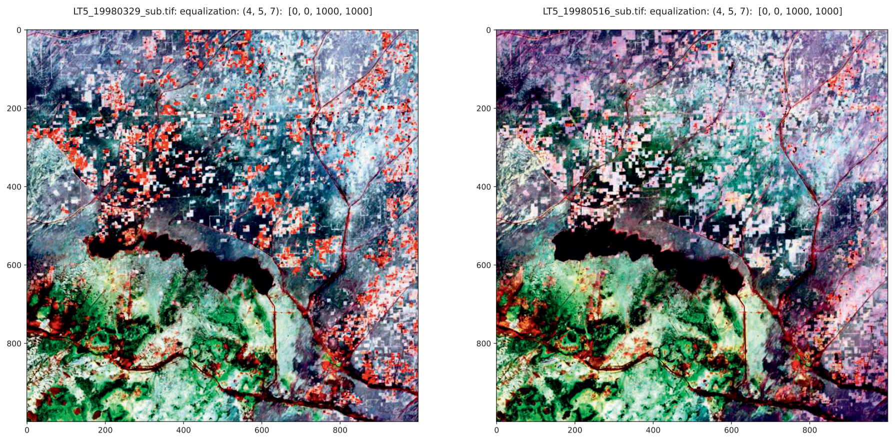

LT5_19980329_sub.tif: equalization: (4, 5, 7): [0, 0, 1000, 1000]
LT5_19980516_sub.tif: equalization: (4, 5, 7): [0, 0, 1000, 1000]
0
200
400
600
800

FIGURE 9.1
LANDSAT 5 TM TOA images (RGB composite of bands 4,5, and 7, histogram equalization) over a water reservoir in Hindustan, India; left image acquired on March 29, 1998, right image on May 16, 1998.

DOI: 10.1201/9781003503286-9

required preprocessing steps were discussed in Chapter 5. Two co-registered satellite images are shown in Figure 9.1.

After having performed the necessary preprocessing, it is common to examine various functions of the spectral bands involved (differences, ratios or linear combinations) which in some way bring the change information contained within them to the fore. The shallow flooding at the western edge of the reservoir in Figure 9.1 is evident at a glance. However, other changes have occurred between the two acquisition times and require more image processing to be clearly distinguished. In the present chapter, we will mention some commonly used techniques for enhancing "change signals" in bi-temporal satellite images. Then we will focus our particular attention on the *multivariate alteration detection* (MAD) algorithm (Nielsen et al., 1998; Nielsen, 2007) for visible/infrared imagery and on a change statistic for polarimetric SAR data based on the complex Wishart distribution (Conradsen et al., 2003, 2016). The chapter concludes with an "inverse" application of change detection, in which *unchanged* pixels are used for automatic relative radiometric normalization of multi-temporal imagery (Canty and Nielsen, 2008). For reviews of change detection in a general context, see Radke et al. (2005), Coppin et al. (2004) and, more recently with focus on deep learning methods, Parelius (2023).

9.1 Naive methods

A simple way to detect changes in two suitably corrected and co-registered multispectral images, represented by N-dimensional random vectors $\boldsymbol{F}$ and $\boldsymbol{G}$, is simply to subtract them from each other component by component and then examine the N difference images

$$D_i = F_i - G_i, \quad i = 1 \dots N. \tag{9.1}$$

Small intensity differences indicate no change, large positive or negative values indicate change, and decision thresholds can be set to define significance. The thresholds are usually expressed in terms of standard deviations from the mean difference value, which is taken to correspond to no change. If the detected signals are uncorrelated, the variances of the difference images are simply

$$\mathrm{var}(D_i) = \mathrm{var}(G_i) + \mathrm{var}(F_i), \quad i = 1 \dots N,$$

or about twice as noisy as the individual image bands. When the significant difference signatures in the spectral channels are then combined so as to try to characterize the kinds of changes that have taken place, one speaks of *spectral change vector analysis* (Jensen, 2005).

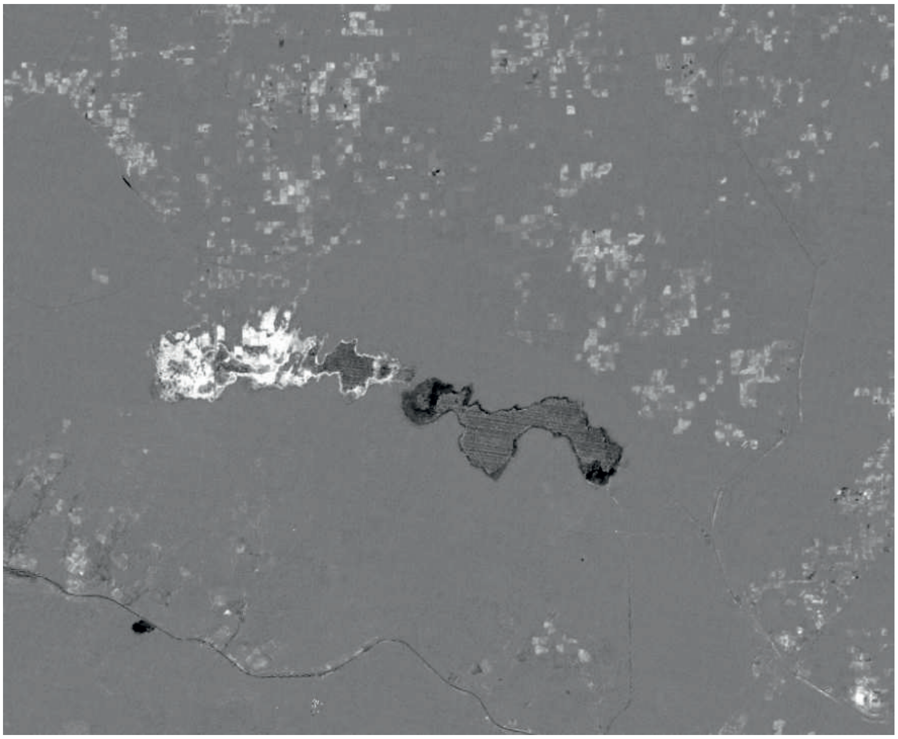


FIGURE 9.2
NDVI difference image for the bi-temporal scene in Figure 9.1.

Alternatively, ratios of intensities

$$\frac{F_k}{G_k}, \quad k = 1 \dots N \tag{9.2}$$

are sometimes formed between successive images. Ratios near unity correspond to no change, while small and large values indicate change. A disadvantage of this method when applied to visible/infrared data is that ratios of random variables are not normally distributed even if the random variables themselves are, so that symmetric threshold values defined in terms of standard deviations are not valid. More formally, for data with additive errors like visible/infrared image pixels, the best statistic for deciding a no-change hypothesis involves a linear combination of observations, not a ratio.

For SAR imagery, the situation is fundamentally different. From Equation (5.26), the variance of the difference of two uncorrelated m-look intensity images is

$$\mathrm{var}(G - F) = \frac{\langle G\rangle^2 + \langle F\rangle^2}{m}.$$

Simple thresholding of the difference image will yield larger errors for a given change in a bright area (large mean intensity) than in a darker area (small mean intensity). Indeed, it turns out that image ratios are a much better choice for detection of changes in multi-look SAR intensity images. This will be illustrated in Section 9.6.1 in the present chapter. Oliver and Quegan (2004) give a thorough discussion in Chapter 12 of their book.

More complicated algebraic combinations, such as differences in vegetation indices (NDVI) or tasseled cap transforms, are also in use. Manipulations of this kind can be performed conveniently on the GEE servers using some of the many mathematical image operations exposed in the API. For example, with the images of Figure 9.1:

```
import ee
import IPython.display as disp

ee.Initialize()

im1 = ee.Image('projects/.../assets/CRC5/LT5_19980329_
sub')
im2 = ee.Image('projects/.../assets/CRC5/LT5_19980516_
sub')
ndvi1 = im1.normalizedDifference(['b4', 'b3'])
ndvi2 = im2.normalizedDifference(['b4', 'b3'])
url = ndvi1.subtract(ndvi2) \
   .getThumbURL({'min':-0.3,'max':0.3})
disp.Image(url=url)
```

The notebook output cell is in Figure 9.2.

If two co-registered satellite images have been classified to yield thematic maps using, for instance, one of the algorithms introduced in Chapter 6, then the class labels can be compared to determine land cover changes. This method is however questionable because, if classification is carried out at the pixel level (as opposed to using segments or objects), then classification errors (typically $> 5\%$) may corrupt or even dominate the true change signal, depending on the strength of the latter.

9.2 Principal components analysis (PCA)

In a scatter plot of pixel intensities in band i of two co-registered multispectral images $\boldsymbol{F}$ and $\boldsymbol{G}$ acquired at different times, each point is a realization of the random vector $(F_i, G_i)^\top$. Since unchanged pixels will be highly correlated over time, they will lie in a narrow, elongated cluster along the principal axis, whereas changed pixels will be scattered some distance away from it; see Figure 9.3. The second principal component, which measures

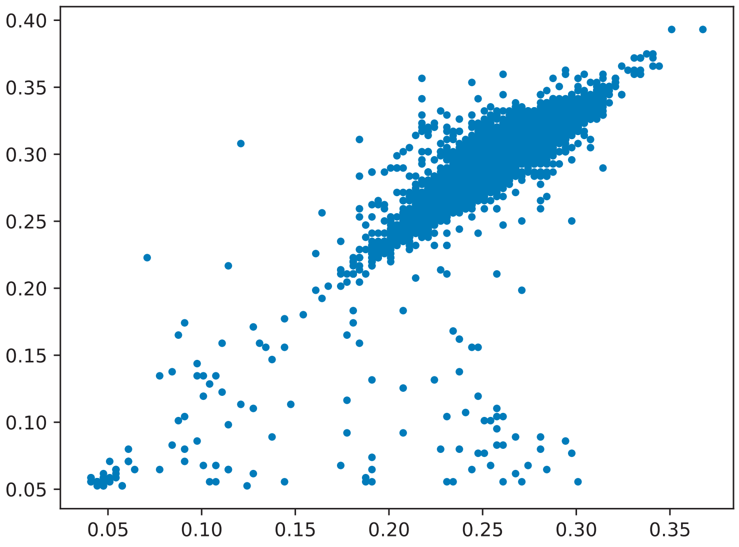


FIGURE 9.3
Scatter plot of spectral bands 4 of the bi-temporal images in Figure 9.1.

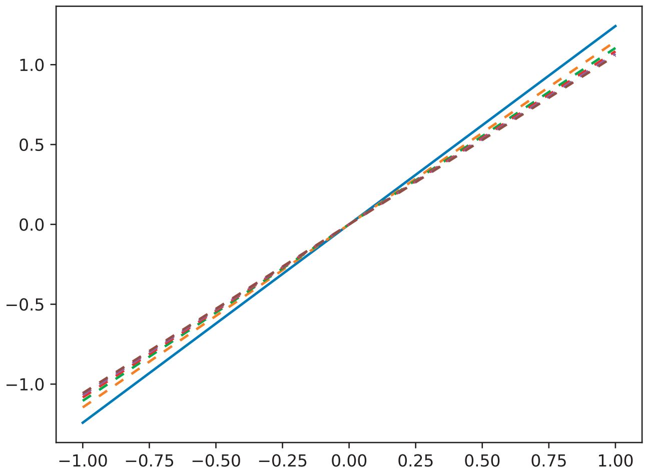


FIGURE 9.4
Iterated PCA. The solid line is the initial principal axis, the dashed lines correspond to five subsequent iterations.

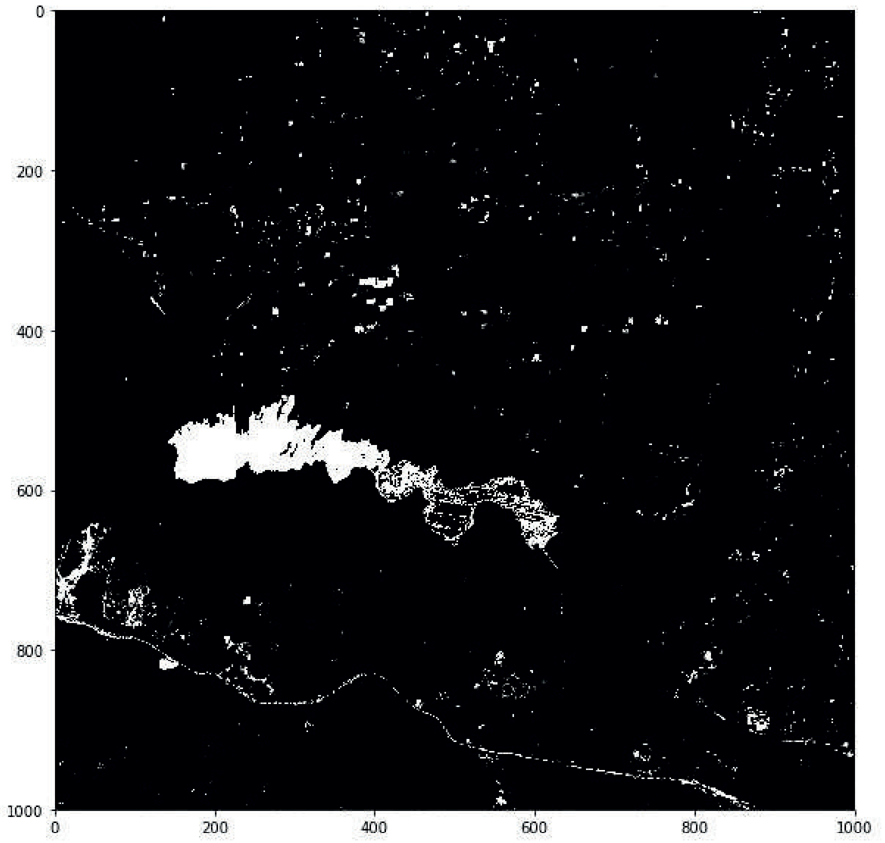


FIGURE 9.5
The change probability for the bi-temporal image in Figure 9.1 after five iterations of principal components analysis.

intensities at right angles to the first principal axis, will therefore quantify the degree of change associated with a given pixel and may even serve as a change image.

9.2.1 Iterated PCA

Since the principal axes are determined by diagonalization of the covariance matrix for all of the pixels, the no-change axis may be poorly defined. To overcome this problem, the principal components can be calculated iteratively using weights for each pixel determined by the degree of change obtained from the preceding iteration.

Listing 9.1 shows part of a Python routine for performing change detection with iterated PCA. After an initial principal components transformation, the EM routine of Section 8.3 is used to cluster the second principal component into two classes. The probability array `U[0,:]` of membership of the pixels to the central cluster is then roughly their probability of no change. The

Listing 9.1: Iterated principal components analysis for change detection (excerpt from the script `ex9_1.py`).

```python
    #  centered data matrix
    G = np.zeros((rows*cols,2))
    G[:,0] = G1-np.mean(G1)
    G[:,1] = G2-np.mean(G2)
    #  initial PCA
    cpm = auxil.Cpm(2)
    cpm.update(G)
    eivs,w = np.linalg.eigh(cpm.covariance())
    eivs = eivs[::-1]
    w = w[:,::-1]
    pcs = G*w
    plt.plot([-1,1],[-np.abs(w[0,1]/w[0,0]),
                     np.abs(w[0,1]/w[0,0])])
    #  iterated PCA
    itr = 0
    while itr<5:
        sigma = np.sqrt(eivs[1])
        U = np.random.rand(2,rows*cols)
    #   cluster the second PC
        unfrozen=np.where(np.abs(pcs[:,1]) >= sigma)[0]
        frozen=np.where( np.abs(pcs[:,1]) < sigma)[0]
        U[0,frozen] = 1.0
        U[1,frozen] = 0.0
        for j in range(2):
            U[j,:]=U[j,:]/np.sum(U,0)
        U=em.em(G,U,0,0,rows,cols,unfrozen=unfrozen)[0]
    #   re-sample the weighted covariance matrix
        cpm.update(G,U[0,:])
        cov = cpm.covariance()
        eivs,w = np.linalg.eigh(cov)
        eivs = eivs[::-1]
        w = w[:,::-1]
    #   weighted PCs
        pcs = G*w
    #   plot the first principal axis
        plt.plot([-1,1],[-np.abs(w[0,1]/w[0,0]),
                np.abs(w[0,1]/w[0,0])],dashes=[4,4])
        itr += 1
```

complement array `U[1,:]` contains the change probabilities. The probabilities of no change for observations having second principal component within one standard deviation of the first principal axis are "frozen" to the value 1 in order to accelerate convergence. Using the no-change probabilities as weights, the covariance matrix is recalculated and the PCA repeated. This procedure is iterated five times in the example script. Results are shown in Figures 9.4 and 9.5. In Figure 9.5, the quantity `U[1,:]` is displayed rather than the second principal component, as it better highlights the changes. The method can be generalized to treat all multispectral bands together (Wiemker, 1997).

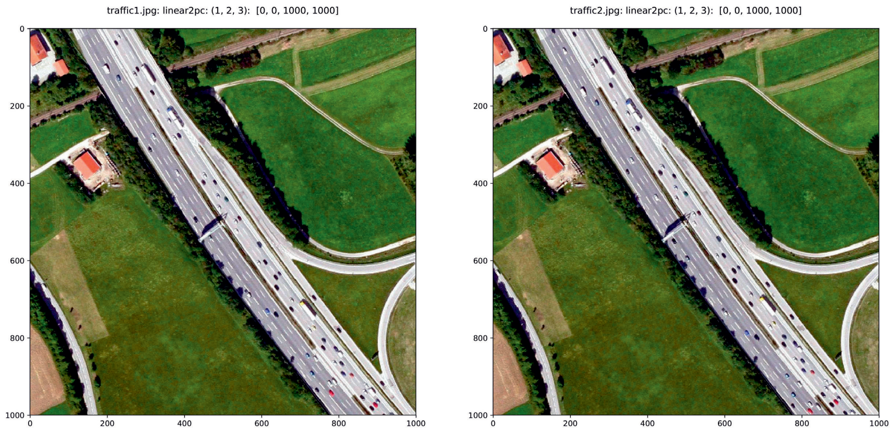

traffic1.jpg: linear2pc: (1, 2, 3): [0, 0, 1000, 1000]
traffic2.jpg: linear2pc: (1, 2, 3): [0, 0, 1000, 1000]
0
200
400
600
800
1000

FIGURE 9.6
Two traffic scenes taken 0.7 seconds apart.

9.2.2 Kernel PCA

For highly nonlinear data, change detection based on linear transformations such as PCA will be expected to give inferior results. As an illustration (Nielsen and Canty, 2008), consider the bi-temporal scene in Figure 9.6. These images were recorded with the airborne DLR 3K-camera system from the German Aerospace Center (DLR) (Kurz et al., 2007), a system consisting of three off-the-shelf cameras arranged on a mount with one camera looking in the nadir direction and two cameras tilted approximately 35^o across track. The 1000×1000 pixel sub-images shown in the figure were acquired 0.7 seconds apart over a busy motorway near Munich, Germany. The images were registered to one another with sub pixel accuracy. The only physical changes on the ground are due to the motion of the vehicles.

Figure 9.7 shows the second principal component for ordinary PCA (left) and kernel PCA (right) for a two-band bi-temporal image consisting of the first band of each of the two traffic scenes. Kernel PCA was discussed in

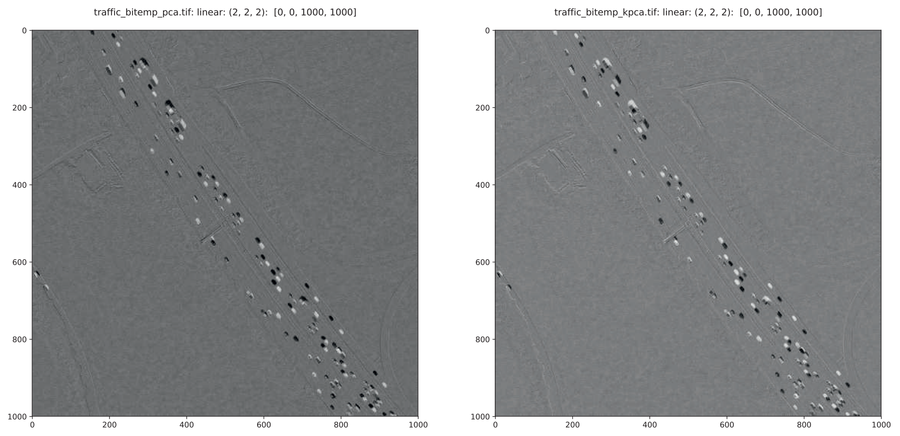

traffic_bitemp_pca.tif: linear: (2, 2, 2): [0, 0, 1000, 1000]
traffic_bitemp_kpca.tif: linear: (2, 2, 2): [0, 0, 1000, 1000]
0
200
400
600
800
1000

FIGURE 9.7
Change detection with principal components analysis, linear data. Left: second principal component for ordinary PCA, right: for kernel PCA.

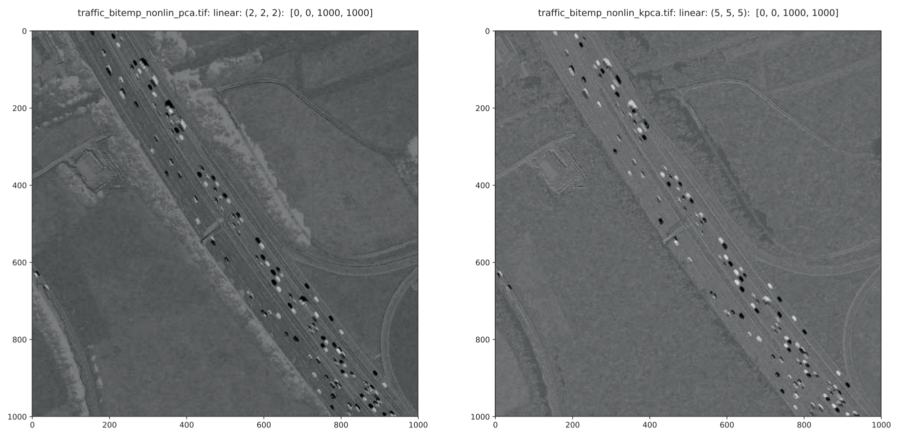

traffic_bitemp_nonlin_pca.tif: linear: (2, 2, 2): [0, 0, 1000, 1000]
traffic_bitemp_nonlin_kpca.tif: linear: (5, 5, 5): [0, 0, 1000, 1000]
0
200
400
600
800
1000

FIGURE 9.8
Change detection with principal components analysis, nonlinear data. Left: second principal component for ordinary PCA, right: fifth principal component for kernel PCA.

Chapter 4. The change sensitivities are seen to be about the same, signaling changes as bright and dark pixels, with no-change indicated as middle gray. In Figure 9.8 the data were artificially "non-linearized" by squaring the intensities in the second band, while leaving those of the first band unchanged. For the ordinary PCA case there are now bright and dark pixels falsely signaling change, whereas with kernel PCA, the fifth kernel principal component indicates the true changes more correctly.

9.3 Multivariate alteration detection

Let us continue to consider two N-band optical/infrared images of the same scene acquired at different times, between which ground reflectance changes have occurred at some locations but not everywhere. We now make a linear combination of the intensities for all N bands in the first image, represented by the random vector $\boldsymbol{G}_1$, thus creating a scalar image characterized by the random variable

$$U = \boldsymbol{a}^\top \boldsymbol{G}_1.$$

The vector of coefficients $\boldsymbol{a}$ is as yet unspecified. We do the same for the second image, represented by $\boldsymbol{G}_2$, forming the linear combination

$$V = \boldsymbol{b}^\top \boldsymbol{G}_2,$$

and then look at the scalar difference image $U - V$. Change information is now contained in a single image. One has, of course, to choose the coefficients $\boldsymbol{a}$ and $\boldsymbol{b}$ in some suitable way. In Nielsen et al. (1998) it is suggested that they be determined by applying standard canonical correlation analysis (CCA), first described by Hotelling (1936), to the two sets of random variables represented by vectors $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$. The resulting linear combinations are then, as we shall see, ordered by similarity (correlation) rather than, as in the original images, by wavelength. This provides a more natural framework in which to look for change. Forming linear combinations has an additional advantage that, due to the Central Limit Theorem (Theorem 2.4), the quantities involved are increasingly well described by the normal distribution.

To anticipate somewhat, in performing CCA on a bi-temporal image, one maximizes the correlation ρ between the random variables U and V given by (see Equation (2.21))

$$\rho = \frac{\text{cov}(U, V)}{\sqrt{\text{var}(U)}\sqrt{\text{var}(V)}}. \tag{9.3}$$

Arbitrary multiples of U and V would clearly have the same correlation, so a constraint must be chosen. A convenient one is

$$\text{var}(U) = \text{var}(V) = 1. \tag{9.4}$$

Note that, under this constraint, the variance of the difference image is

$$\mathrm{var}(U - V) = \mathrm{var}(U) + \mathrm{var}(V) - 2\mathrm{cov}(U, V) = 2(1 - \rho). \tag{9.5}$$

Therefore, the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ which maximize the correlation, Equation (9.3), under the constraints of Equation (9.4) will in fact minimize the variance of the difference image.

The philosophy of making the images as similar as possible before taking their difference is followed in a number of so-called *anomalous change detection* approaches, including the *chronochrome* method of Schaum and Stocker (1997). See Theiler and Matsekh (2009) for a unifying overview.

9.3.1 Canonical correlation analysis (CCA)

Canonical correlation analysis thus entails a linear transformation of each set of image bands $(G_{1_1} \dots G_{1_N})$ and $(G_{2_1} \dots G_{2_N})$ such that, rather than being ordered according to wavelength, the transformed components are ordered according to their mutual correlation; see especially Anderson (2003).

The bi-temporal, multispectral image may be represented by the combined random vector $\begin{pmatrix} \boldsymbol{G}_1 \\ \boldsymbol{G}_2 \end{pmatrix}$. This random vector has a $2N \times 2N$ covariance matrix which can be written in block form:

$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{12}^\top & \boldsymbol{\Sigma}_{22} \end{pmatrix}.$$

Assuming that the means have been subtracted from the image data, $\boldsymbol{\Sigma}_{11} = \langle \boldsymbol{G}_1 \boldsymbol{G}_1^\top \rangle$ is the covariance matrix of the first image, $\boldsymbol{\Sigma}_{22} = \langle \boldsymbol{G}_2 \boldsymbol{G}_2^\top \rangle$ that of the second image, and $\boldsymbol{\Sigma}_{12} = \langle \boldsymbol{G}_1 \boldsymbol{G}_2^\top \rangle$ is the matrix of covariances between the two. We then have, for the transformed variables U and V,

$$\mathrm{var}(U) = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{11} \boldsymbol{a}, \quad \mathrm{var}(V) = \boldsymbol{b}^\top \boldsymbol{\Sigma}_{22} \boldsymbol{b}, \quad \mathrm{cov}(U, V) = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{12} \boldsymbol{b}.$$

CCA now consists of maximizing the covariance $\boldsymbol{a}^\top \boldsymbol{\Sigma}_{12} \boldsymbol{b}$ under constraints $\boldsymbol{a}^\top \boldsymbol{\Sigma}_{11} \boldsymbol{a} = 1$ and $\boldsymbol{b}^\top \boldsymbol{\Sigma}_{22} \boldsymbol{b} = 1$. If, following the usual procedure, we introduce the Lagrange multipliers $\nu/2$ and $\mu/2$ for each of the two constraints, then the problem becomes one of maximizing the unconstrained Lagrange function

$$L = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{12} \boldsymbol{b} - \frac{\nu}{2}(\boldsymbol{a}^\top \boldsymbol{\Sigma}_{11} \boldsymbol{a} - 1) - \frac{\mu}{2}(\boldsymbol{b}^\top \boldsymbol{\Sigma}_{22} \boldsymbol{b} - 1).$$

Setting derivatives with respect to $\boldsymbol{a}$ and $\boldsymbol{b}$ equal to zero gives

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{a}} &= \boldsymbol{\Sigma}_{12} \boldsymbol{b} - \nu \boldsymbol{\Sigma}_{11} \boldsymbol{a} = \boldsymbol{0} \\ \frac{\partial L}{\partial \boldsymbol{b}} &= \boldsymbol{\Sigma}_{12}^\top \boldsymbol{a} - \mu \boldsymbol{\Sigma}_{22} \boldsymbol{b} = \boldsymbol{0}. \end{aligned} \tag{9.6}$$

Multiplying the first of the above equations from the left with the vector $\boldsymbol{a}^\top$, the second with $\boldsymbol{b}^\top$, and using the constraints leads immediately to

$$\nu = \mu = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b} = \rho.$$

Therefore, we can write Equations (9.6) in the form

$$\begin{aligned} \boldsymbol{\Sigma}_{12}\boldsymbol{b} - \rho\boldsymbol{\Sigma}_{11}\boldsymbol{a} &= \boldsymbol{0} \\ \boldsymbol{\Sigma}_{12}^\top\boldsymbol{a} - \rho\boldsymbol{\Sigma}_{22}\boldsymbol{b} &= \boldsymbol{0}. \end{aligned} \tag{9.7}$$

Next, multiply the first of Equations (9.7) by ρ and the second from the left by $\boldsymbol{\Sigma}_{22}^{-1}$. This gives

$$\rho\boldsymbol{\Sigma}_{12}\boldsymbol{b} = \rho^2\boldsymbol{\Sigma}_{11}\boldsymbol{a} \tag{9.8}$$

and

$$\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{12}^\top\boldsymbol{a} = \rho\boldsymbol{b}. \tag{9.9}$$

Finally, combining Equations (9.8) and (9.9), we obtain the following equation for the transformation coefficient $\boldsymbol{a}$,

$$\boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{12}^\top\boldsymbol{a} = \rho^2\boldsymbol{\Sigma}_{11}\boldsymbol{a}. \tag{9.10}$$

A similar argument (Exercise 1) leads to the corresponding equation for $\boldsymbol{b}$, namely

$$\boldsymbol{\Sigma}_{12}^\top\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}\boldsymbol{b} = \rho^2\boldsymbol{\Sigma}_{22}\boldsymbol{b}. \tag{9.11}$$

Equations (9.10) and (9.11) are generalized eigenvalue problems similar to those that we already met for the minimum noise fraction (MNF) and maximum autocorrelation factor (MAF) transformations of Chapter 3; see Equations (3.55) and (3.72). Note, however, that they are coupled via the eigenvalue ρ^2. The desired projections $U = \boldsymbol{a}^\top\boldsymbol{G}_1$ are given by the eigenvectors $\boldsymbol{a}_1 \dots \boldsymbol{a}_N$ of Equation (9.10) corresponding to eigenvalues $\rho_1^2 \ge \rho_2^2 \ge \dots \ge \rho_N^2$. Similarly, the desired projections $V = \boldsymbol{b}^\top\boldsymbol{G}_2$ are given by the eigenvectors $\boldsymbol{b}_1 \dots \boldsymbol{b}_N$ of Equation (9.11) corresponding to the *same* eigenvalues.

Solution of the eigenvalue problems generates new multispectral images $\boldsymbol{U} = (U_1 \dots U_N)^\top$ and $\boldsymbol{V} = (V_1 \dots V_N)^\top$, the components of which are called the *canonical variates* (CVs). The CVs are ordered by similarity (correlation) rather than, as in the original images, by wavelength. The canonical correlations $\rho_i = \mathrm{corr}(U_i, V_i)$, $i = 1 \dots N$, are the square roots of the eigenvalues of the coupled eigenvalue problem. The pair (U_1, V_1) is maximally correlated, the pair (U_2, V_2) is maximally correlated subject to being orthogonal to (uncorrelated with) both U_1 and V_1 (see below), and so on. Taking paired differences then generates a sequence of transformed difference images

$$M_i = U_i - V_i, \quad i = 1 \dots N, \tag{9.12}$$

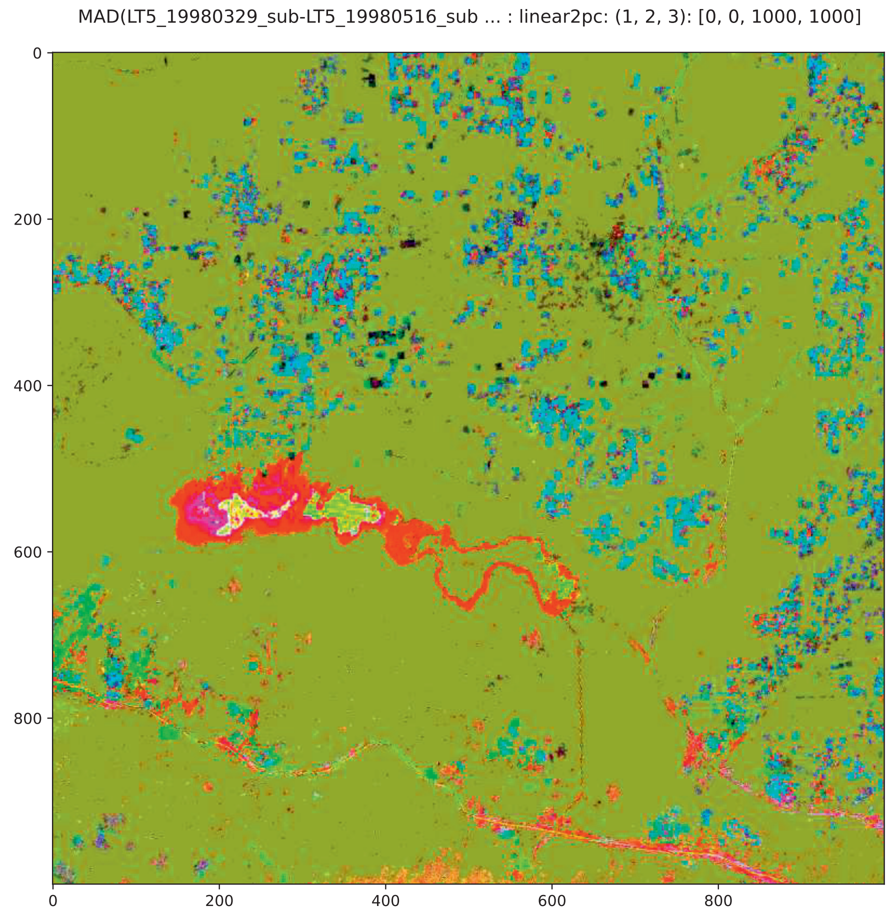

MAD(LT5_19980329_sub-LT5_19980516_sub ... : linear2pc: (1, 2, 3): [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800

FIGURE 9.9
iMAD change map using the six non-thermal bands of the bi-temporal scene of Figure 9.1. The MAD variates are thresholded at significance level 0.0001 (see Section 9.3.3) and MAD variates 1, 2, and 3 are displayed in an RGB 2% saturated linear stretch. Bright and dark colored pixels signify change, green background no change.

referred to as the *multivariate alteration detection* (MAD) *variates* (Nielsen et al., 1998).[*] Since we are dealing with change detection, we want the pairs of canonical variates U_i and V_i to be positively correlated, just like the original image bands. This is easily achieved by appropriate choice of the relative signs of the eigenvector pairs $\boldsymbol{a}_i, \boldsymbol{b}_i$ so as to ensure that $\boldsymbol{a}_i^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b}_i > 0$, $i = 1 \dots N$.

[*]Nielsen et al. (1998) originally numbered the MAD variates in reverse order, least correlated canonical variates first.

A Python script iMad.py for multivariate alteration detection is documented in Appendix C. An example is shown in Figure 9.9, obtained with the commands:

```
# Run the iMAD transformation
%run scripts/iMad -i 50 -n imagery/LT5_19980329_sub.tif \
                           imagery/LT5_19980516_sub.tif
# Set a significance level and calculate change map
%run scripts/iMadmap -m \
imagery/MAD(LT5_19980329_sub-LT5_19980516_sub).tif 0.0001
```

This code actually runs the *iteratively re-weighted MAD* method which will be explained shortly.

9.3.2 Orthogonality properties

Equations (9.10) and (9.11) are of the form

$$\boldsymbol{\Sigma}_1\boldsymbol{a} = \rho^2\boldsymbol{\Sigma}\boldsymbol{a}, \tag{9.13}$$

where both $\boldsymbol{\Sigma}_1$ and $\boldsymbol{\Sigma}$ are symmetric and $\boldsymbol{\Sigma}$ is positive definite. Equation (9.13) can be solved in the same way as for the MNF transformation in Chapter 3. We repeat the procedure here for convenience. First, write Equation (9.13) in the form

$$\boldsymbol{\Sigma}_1\boldsymbol{a} = \rho^2\boldsymbol{L}\boldsymbol{L}^\top\boldsymbol{a},$$

where $\boldsymbol{\Sigma}$ has been replaced by its Cholesky decomposition $\boldsymbol{L}\boldsymbol{L}^\top$. The matrix $\boldsymbol{L}$ is positive definite, lower triangular. Equivalently,

$$\boldsymbol{L}^{-1}\boldsymbol{\Sigma}_1(\boldsymbol{L}^\top)^{-1}\boldsymbol{L}^\top\boldsymbol{a} = \rho^2\boldsymbol{L}^\top\boldsymbol{a}$$

or, with $\boldsymbol{d} = \boldsymbol{L}^\top\boldsymbol{a}$ and the commutativity of inverse and transpose,

$$[\boldsymbol{L}^{-1}\boldsymbol{\Sigma}_1(\boldsymbol{L}^{-1})^\top]\boldsymbol{d} = \rho^2\boldsymbol{d},$$

a standard eigenvalue problem for the symmetric matrix $\boldsymbol{L}^{-1}\boldsymbol{\Sigma}_1(\boldsymbol{L}^{-1})^\top$. Let its eigenvectors be $\boldsymbol{d}_i$. Since they are orthogonal and normalized, we have

$$\delta_{ij} = \boldsymbol{d}_i^\top\boldsymbol{d}_j = \boldsymbol{a}_i^\top\boldsymbol{L}\boldsymbol{L}^\top\boldsymbol{a}_j = \boldsymbol{a}_i^\top\boldsymbol{\Sigma}\boldsymbol{a}_j. \tag{9.14}$$

From Equation (9.14), taking $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}_{11}$ and then $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}_{22}$, it follows that

$$\begin{aligned} \text{cov}(U_i, U_j) &= \boldsymbol{a}_i^\top\boldsymbol{\Sigma}_{11}\boldsymbol{a}_j = \delta_{ij} \\ \text{cov}(V_i, V_j) &= \boldsymbol{b}_i^\top\boldsymbol{\Sigma}_{22}\boldsymbol{b}_j = \delta_{ij}. \end{aligned} \tag{9.15}$$

Furthermore, according to Equation (9.9),

$$\boldsymbol{b}_i = \frac{1}{\rho_i}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}\boldsymbol{a}_i.$$

Therefore, we have, with Equation (9.10),

$$\mathrm{cov}(U_i, V_j) = \boldsymbol{a}_i^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b}_j = \boldsymbol{a}_i^\top \frac{1}{\rho_j}\boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}\boldsymbol{a}_j = \rho_j\ \boldsymbol{a}_i^\top \boldsymbol{\Sigma}_{11}\boldsymbol{a}_j = \rho_j\ \delta_{ij}. \tag{9.16}$$

Thus we see that the canonical variates are *all mutually uncorrelated* except for the pairs (U_i, V_i), and these are ordered by decreasing correlation. The MAD variates themselves are consequently also mutually uncorrelated, their covariances being given by

$$\mathrm{cov}(M_i, M_j) = \mathrm{cov}(U_i - V_i, U_j - V_j) = 0,\ i \neq j = 1 \ldots N, \tag{9.17}$$

and their variances by

$$\sigma^2_{M_i} = \mathrm{var}(U_i - V_i) = 2(1 - \rho_i), \quad i = 1 \ldots N. \tag{9.18}$$

The first MAD variate has minimum variance in its pixel intensities. The second MAD variate has minimum spread subject to the condition that its pixel intensities are statistically uncorrelated with those in the first variate; the third has minimum spread subject to being uncorrelated with the first two, and so on. Depending on the type of change present, any of the components may exhibit significant change information. Interesting small-scale anthropogenic changes, for instance, will generally be unrelated to dominating seasonal vegetation changes or stochastic image noise, so it is quite common that such changes will be concentrated in lower-order MAD variates. In fact, one of the nicest aspects of the method is that it sorts different categories of change into different, uncorrelated image components.

9.3.3 Iteratively re-weighted MAD

Let us now imagine two images of the same scene, acquired at different times under similar conditions, but for which no ground reflectance changes have occurred whatsoever. Then the only differences between them will be due to random effects like instrument noise and atmospheric fluctuation. In such a case we would expect that the histogram of any difference component that we generate will be very nearly Gaussian. In particular, the MAD variates, being uncorrelated, should follow a multivariate, zero mean normal distribution with diagonal covariance matrix. Change observations would deviate more or less strongly from such a distribution. Just as for the iterated principal components method of Section 9.2.1, we might therefore expect an improvement in the sensitivity of the MAD transformation if we can establish an increasingly better background of no change against which to detect change (Nielsen, 2007). This can again be done in an iteration scheme in which, when calculating the means and covariance matrices for the next iteration

```
------------IRMAD ------------
Sun Jun 30 12:26:15 2024
first scene:  imagery/LT5_19980329_sub.tif
second scene: imagery/LT5_19980516_sub.tif
rho: [0.99183095 0.96437037 0.9305281  0.87602746 0.85445809 0.4660925 ]
result written to: imagery/MAD(LT5_19980329_sub-LT5_19980516_sub).tif
elapsed time: 11.810488939285278
```

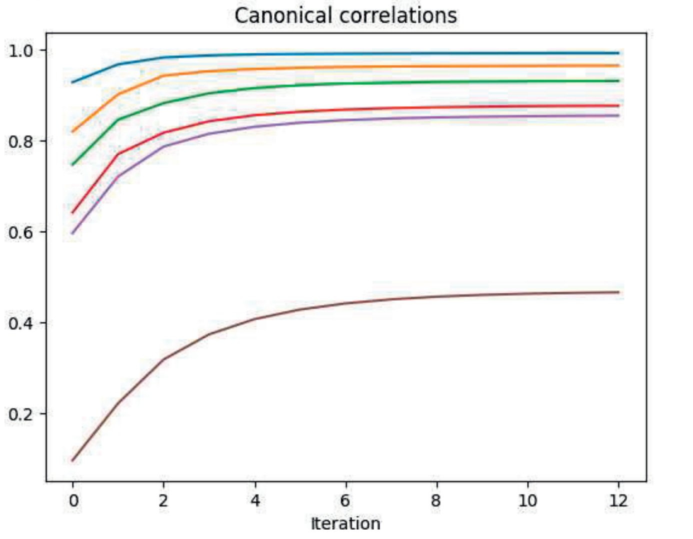


FIGURE 9.10
The canonical correlations ρ_i, $i = 1 \ldots 6$, under 12 iterations of the MAD transformation of the bi-temporal image of Figure 9.1.

of the MAD transformation, observations are weighted in some appropriate fashion.*

One way to determine the weights is to perform an unsupervised classification of the MAD variates using, for example, the Gaussian mixture algorithm of Chapter 8 with K *a priori* clusters, one for no change and $K-1$ clusters for different categories of change. The pixel class membership probabilities for no change could then provide the weights for iteration. This method, being unsupervised, will not distinguish which clusters correspond to change and which to no change, but the no-change cluster can be identified by its compactness, e.g., by its partition density; see Section 8.3.3. This strategy has, however, the disadvantage that a computationally expensive clustering procedure must be repeated for every iteration. Moreover, one has to make a more or less arbitrary decision as to how many no-change clusters are present and tolerate the unpredictability of clustering results generally.

*This was in fact the main motivation for allowing for weights in the provisional means algorithm described in Chapter 2.

An alternative scheme is to continue to examine the MAD variates directly. Let the random variable Z represent the sum of the squares of the standardized MAD variates,

$$Z = \sum_{i=1}^{N} \left(\frac{M_i}{\sigma_{M_i}} \right)^2,$$

where σ_{M_i} is given by Equation (9.18). Then, since the no-change observations are expected to be normally distributed and uncorrelated, the realizations of Z corresponding to no-change observations should be chi-square distributed with N degrees of freedom (Theorem 2.6). In fact, Z can be interpreted as a likelihood ratio test statistic for change. Assuming

$$\boldsymbol{M} = (M_1, \dots M_N)^\top \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

where $\boldsymbol{\Sigma}$ is known and diagonal, consider the hypothesis test

$H_0 : \boldsymbol{\mu} = \boldsymbol{0}$ no change

$H_1 : \boldsymbol{\mu} \neq \boldsymbol{0}$ change.

Since we make for each pixel exactly one observation $\boldsymbol{M}$, the maximum-likelihood estimate of the mean is simply $\hat{\boldsymbol{\mu}} = \boldsymbol{M}$. With Definition 2.7, the likelihood ratio test statistic is therefore

$$Q = \frac{L(\boldsymbol{\mu} = \boldsymbol{0})}{L(\hat{\boldsymbol{\mu}} = \boldsymbol{M})} = \frac{\exp(-\frac{1}{2}\boldsymbol{M}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{M})}{\exp(-\frac{1}{2}(\boldsymbol{M} - \boldsymbol{M})^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{M} - \boldsymbol{M}))}$$
$$= \exp(-\frac{1}{2}\boldsymbol{M}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{M}),$$

from which we derive the chi-square distributed test statistic

$$Z = -2 \ln Q = \boldsymbol{M}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{M} = \sum_{i=1}^{N} \frac{M_i^2}{\sigma_{Mi}^2}. \tag{9.19}$$

The P-value for an observation z of the statistic, i.e., the probability that a sample z drawn from the chi-square distribution could be that large or larger, is therefore

$$\mathcal{P} = 1 - P_{\chi^2;N}(z). \tag{9.20}$$

Small P-values signal change, so after each iteration the P-value itself can be used to weight each pixel before re-sampling to determine the means and covariance matrices for the next iteration, thus gradually reducing the influence of the change observations on the MAD transformation. Iteration continues until some stopping criterion is met, such as lack of significant change in the canonical correlations ρ_i, $i = 1 \dots N$. Nielsen (2007) refers to this procedure as the *iteratively re-weighted* MAD (IR-MAD or iMAD) algorithm.

The Python script `iMad.py` documented in Appendix C iterates the MAD transformation in this way. The iterated canonical correlations for the reservoir scene, Figure 9.1, are shown in Figure 9.10 and a corresponding change map in Figure 9.9.

9.3.4 Scale invariance

An additional advantage of the MAD procedure stems from the fact that the calculations involved are invariant under linear and affine transformations of the original image intensities. This implies insensitivity to linear differences in atmospheric conditions or sensor calibrations at the two acquisition times. Scale invariance also offers the possibility of using the MAD transformation to perform relative radiometric normalization of multitemporal imagery in a fully automatic way, as will be explained in Section 9.7 below.

We can illustrate the invariance as follows. Suppose the second image $\boldsymbol{G}_2$ is transformed according to some linear transformation

$$\boldsymbol{H} = \boldsymbol{T}\boldsymbol{G}_2,$$

where $\boldsymbol{T}$ is a constant nonsingular matrix.* The relevant covariance matrices for the coupled generalized eigenvalue problems, Equations (9.10) and (9.11), are then

$$\begin{aligned} \boldsymbol{\Sigma}_{1h} &= \langle \boldsymbol{G}_1\boldsymbol{H}^\top \rangle = \boldsymbol{\Sigma}_{12}\boldsymbol{T}^\top \\ \boldsymbol{\Sigma}_{11} &\quad \text{unchanged} \\ \boldsymbol{\Sigma}_{hh} &= \langle \boldsymbol{H}\boldsymbol{H}^\top \rangle = \boldsymbol{T}\boldsymbol{\Sigma}_{22}\boldsymbol{T}^\top. \end{aligned}$$

The coupled eigenvalue problems now read

$$\begin{aligned} \boldsymbol{\Sigma}_{12}\boldsymbol{T}^\top(\boldsymbol{T}\boldsymbol{\Sigma}_{22}\boldsymbol{T}^\top)^{-1}\boldsymbol{T}\boldsymbol{\Sigma}_{12}^\top\boldsymbol{a} &= \rho^2\boldsymbol{\Sigma}_{11}\boldsymbol{a} \\ \boldsymbol{T}\boldsymbol{\Sigma}_{12}^\top\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}\boldsymbol{T}^\top\boldsymbol{c} &= \rho^2\boldsymbol{T}\boldsymbol{\Sigma}_{22}\boldsymbol{T}^\top\boldsymbol{c}, \end{aligned}$$

where $\boldsymbol{c}$ is the desired projection for the transformed image $\boldsymbol{H}$. These equations are easily seen to be equivalent to

$$\begin{aligned} \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{12}^\top\boldsymbol{a} &= \rho^2\boldsymbol{\Sigma}_{11}\boldsymbol{a} \\ \boldsymbol{\Sigma}_{12}^\top\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}(\boldsymbol{T}^\top\boldsymbol{c}) &= \rho^2\boldsymbol{\Sigma}_{22}(\boldsymbol{T}^\top\boldsymbol{c}), \end{aligned}$$

which are identical to Equations (9.10) and (9.11) with $\boldsymbol{b}$ replaced by $\boldsymbol{T}^\top\boldsymbol{c}$. Therefore, the MAD components in the transformed situation are, as before,

$$\boldsymbol{a}_i^\top\boldsymbol{G}_1 - \boldsymbol{c}_i^\top\boldsymbol{H} = \boldsymbol{a}_i^\top\boldsymbol{G}_1 - \boldsymbol{c}_i^\top\boldsymbol{T}\boldsymbol{G}_2 = \boldsymbol{a}_i^\top\boldsymbol{G}_1 - (\boldsymbol{T}^\top\boldsymbol{c}_i)^\top\boldsymbol{G}_2 = \boldsymbol{a}_i^\top\boldsymbol{G}_1 - \boldsymbol{b}_i^\top\boldsymbol{G}_2.$$

9.3.5 Regularization

To avoid possible near singularity problems in the solution of Equations (9.10) and (9.11), it is sometimes useful to introduce some form of regularization into the MAD algorithm (Nielsen, 2007). This may in particular be necessary when the number of spectral bands N is large, as is the case for change

*A translation $\boldsymbol{H} = \boldsymbol{G}_2 + \boldsymbol{C}$, where $\boldsymbol{C}$ is a constant vector, will clearly make no difference, since the data matrices are always centered prior to the transformation.

Listing 9.2: Canonical correlation and regularization (excerpt from the Python script `iMad.py`).

```
#       weighted covariance matrices and means
         S = cpm.covariance()
         means = cpm.means()
#       reset prov means object
         cpm.__init__(2*bands)
         s11 = S[0:bands,0:bands]
         s11 = (1-lam)*s11 + lam*np.identity(bands)
         s22 = S[bands:,bands:]
         s22 = (1-lam)*s22 + lam*np.identity(bands)
         s12 = S[0:bands,bands:]
         s21 = S[bands:,0:bands]
         c1 = s12*linalg.inv(s22)*s21
         b1 = s11
         c2 = s21*linalg.inv(s11)*s12
         b2 = s22
#       solution of generalized eigenproblems
         if bands>1:
             mu2a,A = auxil.geneiv(c1,b1)
             mu2b,B = auxil.geneiv(c2,b2)
#           sort a
             idx = np.argsort(mu2a)
             A = (A[:,idx])[:,::-1]
#           sort b
             idx = np.argsort(mu2b)
             B = (B[:,idx])[:,::-1]
             mu2 = (mu2b[idx])[::-1]
         else:
             mu2 = c1/b1
             A = 1/np.sqrt(b1)
             B = 1/np.sqrt(b2)
#       canonical correlations
         mu = np.sqrt(mu2)
         a2 = np.diag(A.T*A)
         b2 = np.diag(B.T*B)
         sigma = np.sqrt( (2-lam*(a2+b2))/(1-lam)-2*mu )
         rho=mu*(1-lam)/np.sqrt( (1-lam*a2)*(1-lam*b2) )
```

detection involving hyper-spectral imagery. The concept of regularization by means of length penalization was met in Section 2.6.4. Here we allow for length penalization of the eigenvectors $\boldsymbol{a}$ and $\boldsymbol{b}$ in CCA by replacing the constraint (9.4) by

$$(1-\lambda)\mathrm{var}(U) + \lambda\|\boldsymbol{a}\|^2 = (1-\lambda)\mathrm{var}(V) + \lambda\|\boldsymbol{b}\|^2 = 1, \tag{9.21}$$

where $\mathrm{var}(U) = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{11}\boldsymbol{a}$, $\mathrm{var}(V) = \boldsymbol{b}^\top \boldsymbol{\Sigma}_{22}\boldsymbol{b}$ and λ is a regularization parameter. To maximize the covariance $\mathrm{cov}(U,V) = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b}$ under this constraint, we now maximize the unconstrained Lagrange function

$$L = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b} - \frac{\nu}{2}((1-\lambda)\boldsymbol{a}^\top \boldsymbol{\Sigma}_{11}\boldsymbol{a} + \lambda\|\boldsymbol{a}\|^2 - 1) - \frac{\mu}{2}((1-\lambda)\boldsymbol{b}^\top \boldsymbol{\Sigma}_{22}\boldsymbol{b} + \lambda\|\boldsymbol{b}\|^2 - 1).$$

Setting the derivatives equal to zero, we obtain

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{a}} &= \boldsymbol{\Sigma}_{12}\boldsymbol{b} - \nu(1-\lambda)\boldsymbol{\Sigma}_{11}\boldsymbol{a} - \nu\lambda\boldsymbol{a} = 0 \\ \frac{\partial L}{\partial \boldsymbol{b}} &= \boldsymbol{\Sigma}_{12}^\top\boldsymbol{a} - \mu(1-\lambda)\boldsymbol{\Sigma}_{22}\boldsymbol{b} - \mu\lambda\boldsymbol{b} = 0. \end{aligned} \tag{9.22}$$

Multiplying the first equation above from the left with $\boldsymbol{a}^\top$ gives

$$\boldsymbol{a}^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b} - \nu((1-\lambda)\boldsymbol{a}^\top \boldsymbol{\Sigma}_{11}\boldsymbol{a} + \lambda\|\boldsymbol{a}\|^2) = 0$$

and, from (9.21), $\nu = \boldsymbol{a}^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b}$. Similarly, multiplying the second equation from the left with $\boldsymbol{b}^\top$, we have $\mu = \boldsymbol{b}^\top \boldsymbol{\Sigma}_{12}^\top \boldsymbol{a} = \nu$. Equation (9.22) can now be written as the single generalized eigenvalue problem ($\boldsymbol{I}$ is the $N \times N$ identity matrix)

$$\begin{pmatrix} \boldsymbol{0} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix} = \mu \begin{pmatrix} (1-\lambda)\boldsymbol{\Sigma}_{11} + \lambda\boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & (1-\lambda)\boldsymbol{\Sigma}_{22} + \lambda\boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix}. \tag{9.23}$$

Note that this equation is equivalent to Equations (9.7) for $\lambda = 0$. Note also that the eigenvalue μ is the covariance of U and V and not the correlation. Thus, with Equation (9.21),

$$\sigma_{MAD}^2 = \mathrm{var}(U) + \mathrm{var}(V) - 2\mathrm{cov}(U,V) = \frac{2 - \lambda(\|\boldsymbol{a}\|^2 + \|\boldsymbol{b}\|^2)}{1-\lambda} - 2\mu \tag{9.24}$$

and the correlation is

$$\rho = \frac{\mu}{\sqrt{\mathrm{var}(U)\mathrm{var}(V)}} = \frac{\mu(1-\lambda)}{\sqrt{(1-\lambda\|\boldsymbol{a}\|^2)(1-\lambda\|\boldsymbol{b}\|^2)}}\ , \tag{9.25}$$

which reduces to $\rho = \mu$ only when $\lambda = 0$. Regularization is included as an option in the Python iMAD script.

Canonical correlation with regularization is illustrated in the Python script excerpt of Listing 9.2. After collecting image statistics with the weighted provisional means method (Section 2.3.2), the covariance matrix and mean vector of the bi-temporal image are extracted from the `cpm` class instance in lines 2 and 3. The (regularized) covariance matrices required for CCA are then constructed (lines 6 to 11) and the generalized eigenvalue problems, Equations (9.10) and (9.11), are solved in lines 18 and 19, or for single band images, in lines 28 to 30. Then, in lines 35 and 36, the MAD standard deviations and canonical correlations are determined as given, respectively, by Equations (9.24) and (9.25).

9.3.6 Postprocessing

The MAD transformation can be augmented by subsequent application of the MAF transformation in order to improve the spatial coherence of the MAD variates (Nielsen et al., 1998). (When image noise is estimated as the difference between intensities of neighboring pixels, the MAF transformation is equivalent to the MNF transformation, as was discussed in Chapter 3.) MAD components post processed in this way are referred to as MAD/MAF.

9.3.7 Unsupervised change classification

Figure 9.11 below shows a bi-temporal LANDSAT scene acquired in early and late summer over the same region as Figure 6.1. Figure 9.12 provides a rather nice visualization of significant changes that have taken place between the two acquisitions. The gray/green (featureless) regions correspond to built-up areas and to forest canopy, neither of which have changed significantly between acquisitions. The significant changes, indicated by the various brightly colored areas, are quite heterogeneous and not easy to interpret, but from their form many are clearly associated with cultivated fields, others with activity in two open face coal mines. We might now go a step further (Canty and Nielsen, 2006) and attempt to cluster the change and no-change pixels in multidimensional MAD or MAD/MAF feature space, using, for instance, the Gaussian mixture algorithm discussed in Chapter 8. Since the no-change cluster will tend to be very dominant, it is to be expected that this will only be sensible for situations in which a good deal of change has taken place and some *a priori*

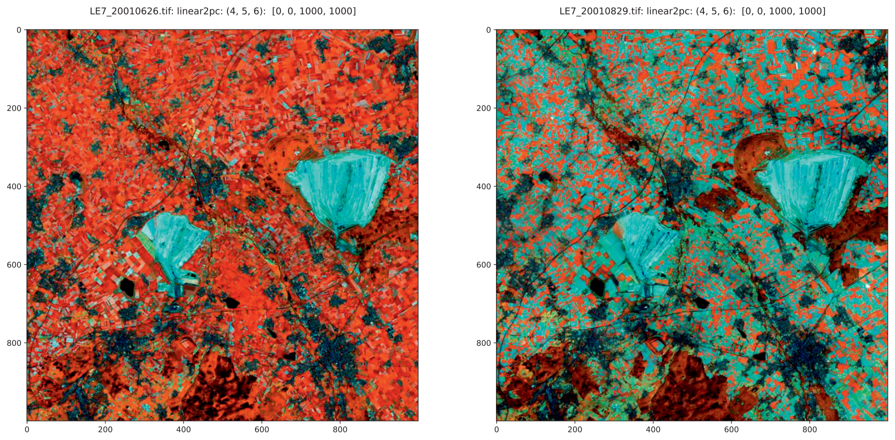

LE7_20010626.tif: linear2pc: (4, 5, 6): [0, 0, 1000, 1000]
LE7_20010829.tif: linear2pc: (4, 5, 6): [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800

FIGURE 9.11
LANDSAT 7 ETM images acquired over Jülich, Germany, on June 26, 2001 and August 29, 2001. RGB color composites of bands 4, 5, and 6 in a linear 2% stretch.

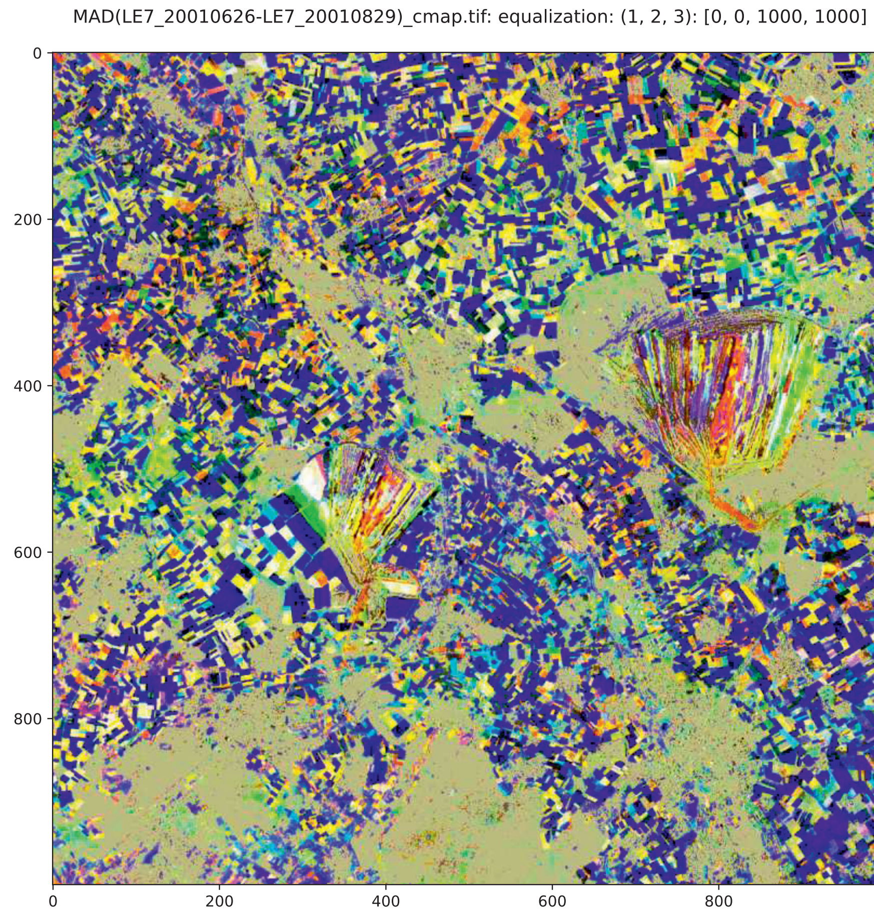

MAD(LE7_20010626-LE7_20010829)_cmap.tif: equalization: (1, 2, 3): [0, 0, 1000, 1000]
0
200
400
600
800
0
200
400
600
800

FIGURE 9.12
RGB composite of iMAD components 1, 2, and 3 for the bi-temporal scene of Figure 9.11. The MAD variates are thresholded at significance level 0.0001.

information about the number of change categories exists. In the present case, the main crops in the scene are sugar beets, corn and cereal grain. The former two are still maturing in August, the time of the second acquisition, whereas the grain fields have been harvested. So we might postulate two main classes of change: maturing crops and harvested crops. Allowing for a no-change class and a "catch-all" class for other kinds of change, we can therefore attempt to classify the iMAD image by assuming four classes in all:

```
run scripts/em -K 4 \
      imagery/MAD(LE7_20010626-LE7_20010829)
```

```
---------------------------
     EM clustering
---------------------------
infile:   imagery/MAD(LE7_20010626-LE7_20010829)
clusters: 4
T0:       0.500000
beta:     0.500000
scale:    2
running EM on 62500 pixel vectors
em iteration 0: dU: 0.836431 loglike: -97996.679848
em iteration 10: dU: 0.959525 loglike: -13378.398171
em iteration 20: dU: 1.000000 loglike: -11130.729298
em iteration 30: dU: 0.999997 loglike: -11338.581835
em iteration 40: dU: 0.411141 loglike: -9477.856172
em iteration 50: dU: 0.021389 loglike: -9393.008548
em iteration 60: dU: 0.003319 loglike: -9395.696867
running EM on 250000 pixel vectors
em iteration 0: dU: 1.000000 loglike: -316164.037620
em iteration 10: dU: 0.000946 loglike: -45944.946901
running EM on 1000000 pixel vectors
em iteration 0: dU: 1.000000 loglike: -935050.848265
em iteration 10: dU: 0.000749 loglike: -191617.238590
...
classified image written to: imagery/MAD(LE7_20010626
                                   -LE7_20010829)_em
elapsed time: 29.9761760235
```

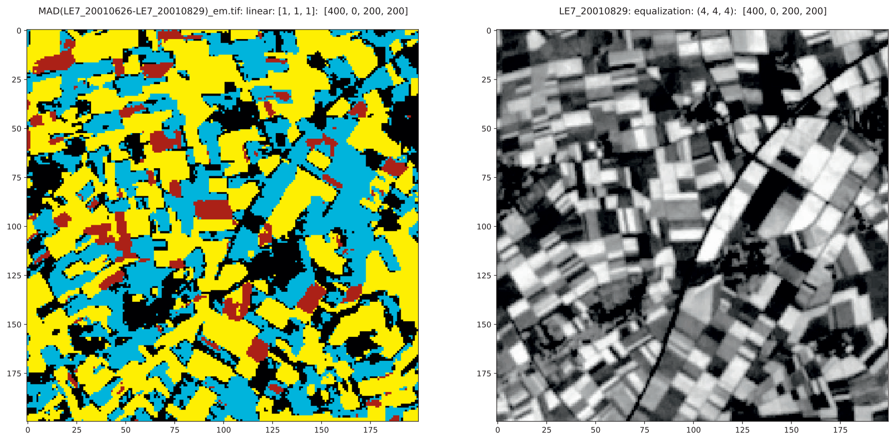

MAD(LE7_20010626-LE7_20010829)_em.tif: linear: [1, 1, 1]: [400, 0, 200, 200]
LE7_20010829: equalization: (4, 4, 4): [400, 0, 200, 200]
0
25
50
75
100
125
150
175

FIGURE 9.13
Left: Gaussian mixture clusters for a spatial subset of the iMAD image of Figure 9.9 assuming 4 clusters in all. Right: the fourth band of the image acquired on August 29, 2001.

A spatial subset of the clustered iMAD image is shown on the left of Figure 9.13 and compared with the band 4 of the second (August) acquisition on the right. The black pixels are no-change observations. The yellow pixels correspond to low reflectance in band 4 (near infrared) and therefore may be fairly confidently associated with harvested grain.

9.3.8 iMAD on the Google Earth Engine

The iteratively re-weighted MAD algorithm is programmed against the GEE Python API in the function

```
imad(current,prev)
```

which can be imported from the Python module `auxil.eeMad.py`:

```
from auxil.eeMad import imad
```

see Appendix C. In order to implement the algorithm, the function `imad()` is iterated over a list of integers of length equal to the maximum number of iMAD iterations:

```
inputlist = ee.List.sequence(1,maxitr)
...
result = ee.Dictionary(inputlist.iterate(imad,first))
```

During the iteration, the elements of `inputlist` constitute the `current` argument in `imad(current, prev)`, and the `prev` argument, initialized in the above statement to `first`, is an `ee.Dictionary` containing the bi-temporal image bands to be processed, the accumulated canonical correlations for each iteration, the current iMAD image, the current chi-square image, as well as a flag to signal when the iteration should stop. For example, here is a wrapper function to access two multispectral images from the GEE data archive and run the iMAD algorithm, returning the final MAD variates:

```
def iMad(cid,poly,sd1,ed1,sd2,ed2,bns,maxitr):
    collection = ee.ImageCollection(cid) \
         .filterBounds(poly) \
         .filterDate(ee.Date(sd1), ee.Date(ed1)) \
         .sort('system:time_start',False)
    image1 = ee.Image(collection.first()).select(bns)
    collection = ee.ImageCollection(cid) \
         .filterBounds(poly) \
         .filterDate(ee.Date(sd2), ee.Date(ed2)) \
         .sort('system:time_start',False)
    image2 = ee.Image(collection.first()).select(bns)
    image2 = image2.register(image1,60)
    inputlist = ee.List.sequence(1,maxitr)
    first = ee.Dictionary({'done':ee.Number(0),
          'image':image1.addBands(image2).clip(poly),
          'allrhos': [ee.List.sequence(1,len(bns))],
```

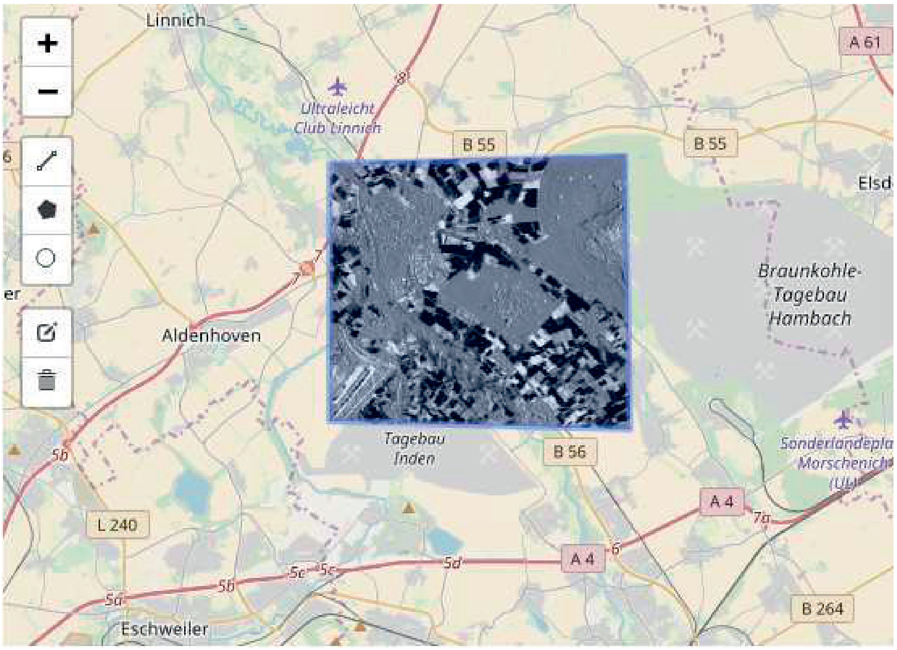


FIGURE 9.14
The first iMAD variate for a subset of the bi-temporal image of Figure 9.11 as calculated using the GEE Python API. Map data (c) OpenStreetMap.

```
                'chi2':ee.Image.constant(0),
                'MAD':ee.Image.constant(0)})
        madnames = ['MAD'+str(i+1) for i in range(len(bns))]
    #  run the algorithm
        result = ee.Dictionary(inputlist.iterate(imad,first))
        MAD = ee.Image(result.get('MAD')).rename(madnames)
        return MAD
```

Here, `poly` is an `ee.Geometry.Polygon` containing the region of interest. Thus, to process the June and August LANDSAT 7 ETM images of Figure 9.11 we execute the notebook cell

```
    collectionid = 'LANDSAT/LE07/C01/T1_RT_TOA'
    bandNames = ['B1','B2','B3','B4','B5','B7']
    startDate1 = '2001-06-25'
    endDate1 = '2001-06-27'
    startDate2 = '2001-08-28'
    endDate2 = '2001-08-30'
    maxitr = 50
    MAD = iMad(collectionid,poly,startDate1,
               endDate1,startDate2,endDate2,
               bandNames,maxitr)
```

In the accompanying notebook, the polygon is selected from an interactive map. The code is not actually executed on the GEE servers until we try to display the result on the map or request export to assets, e.g.,

```
m.add_layer(
    TileLayer(url=GetTileLayerUrl(
        MAD.select('MAD1').visualize(min=-3, max=3) )) )
```

A "slippy map" display of the first iMAD variate in the accompanying Jupyter notebook is shown Figure 9.14. For a detailed, three-part tutorial on using the iMAD transformation on the Google Earth Engine, see

```
https://developers.google.com/earth-engine/tutorials
                                  /community/imad-tutorial-pt1
```

9.4 Change detection with polarimetric SAR imagery

Polarimetric SAR image pixels were represented in Chapter 5 in the form of look-averaged complex covariance matrices (Equation (5.30) or (5.32)). Assuming that the measured scattering amplitudes, which for full quad polarization are given by

$$\boldsymbol{s} = \begin{pmatrix} s_{hh} \\ \sqrt{2}s_{hv} \\ s_{vv} \end{pmatrix}, \tag{9.26}$$

are indeed zero-mean, complex multivariate normally distributed, then we expect that the multilook covariance matrix observations,

$$\bar{\boldsymbol{c}} = \frac{1}{m}\sum_{\nu=1}^{m} \boldsymbol{s}(\nu)\boldsymbol{s}(\nu)^\dagger = \langle \boldsymbol{s}\boldsymbol{s}^\dagger \rangle,$$

being at least approximately complex Wishart distributed, will provide a good characterization of the image statistics. Conradsen et al. (2003) developed a per-pixel likelihood ratio test for changes between two image acquisitions which is based on the complex Wishart distribution and, later, generalized the method to multitemporal image sequences (Conradsen et al., 2016). In the following, their change detection algorithm will be explained and, as usual, programmed in Python.

9.4.1 Scalar imagery: the gamma distribution

We will adopt the approach taken in the appendices of Conradsen et al. (2003, 2016) and begin with the easier case of scalar intensity images or single polarimetry, then proceed in Section 9.4.2 to the multivariate situation which is applicable to dual and quad polarimetric data.

9.4.1.1 Bi-temporal data

Following the notation used to introduce speckle filtering in Section 5.4.3, we represent the pixels in an m look-averaged, single polarimetric SAR intensity image acquired at some initial time by the random variable G_1, with mean $\langle G_1\rangle = x_1$. Here, x_1 is understood to be the look-averaged underlying signal, e.g., $\langle |s_{hh}|^2\rangle$. The density function for G_1, conditional on the value of x_1, is, with Equation (5.34), the gamma density

$$p(g_1 \mid x_1) = \frac{1}{(x_1/m)^m\Gamma(m)} g_1^{m-1}e^{-g_1 m/x_1}. \tag{9.27}$$

For a second image G_2, acquired at a later time, with m looks and $\langle G_2\rangle = x_2$, we have

$$p(g_2 \mid x_2) = \frac{1}{(x_2/m)^m\Gamma(m)} g_2^{m-1}e^{-g_2 m/x_2}. \tag{9.28}$$

We now set up a likelihood ratio test as described in Definition 2.7. Under the null hypothesis H_0 that the pixels are unchanged, $x_1 = x_2 = x$, the likelihood for x is

$$L_0(x) = p(g_1 \mid x)p(g_2 \mid x) = \frac{1}{(x/m)^{2m}\Gamma(m)^2}\, g_1^{m-1}g_2^{m-1}e^{-(g_1+g_2)m/x}, \tag{9.29}$$

and under the alternative hypothesis H_1, $x_1 \neq x_2$, the likelihood for x_1 and x_2 is

$$\begin{aligned} L_1(x_1,x_2) &= p(g_1 \mid x_1)p(g_2 \mid x_2) \\ &= \frac{1}{(x_1/m)^m(x_2/m)^m\Gamma(m)^2}\, g_1^{m-1}g_2^{m-1}e^{-(g_1 m/x_1+g_2 m/x_2)}. \end{aligned} \tag{9.30}$$

By taking derivatives of the log-likelihoods, it is easy to show (Exercise 5(a)) that the likelihood $L_0(x)$ is maximized by

$$\hat{x} = \frac{g_1+g_2}{2}$$

and that $L_1(x_1, x_2)$ is maximized by

$$\hat{x}_1 = g_1, \quad \hat{x}_2 = g_2.$$

Then, according to Equation (2.77) and some simple algebra, the likelihood ratio test has the critical region

$$\frac{L_0(\hat{x})}{L_1(\hat{x}_1,\hat{x}_2)} = \frac{(g_1/m)^m(g_2/m)^n}{\left(\frac{g_1+g_2}{2m}\right)^{2m}} = 2^{2m}\frac{g_1^m g_2^m}{(g_1+g_2)^{2m}} \le t. \tag{9.31}$$

Equivalently,

$$\left(\frac{g_1 g_2}{(g_1+g_2)^2}\right)^m \le \frac{t}{2^{2m}}$$

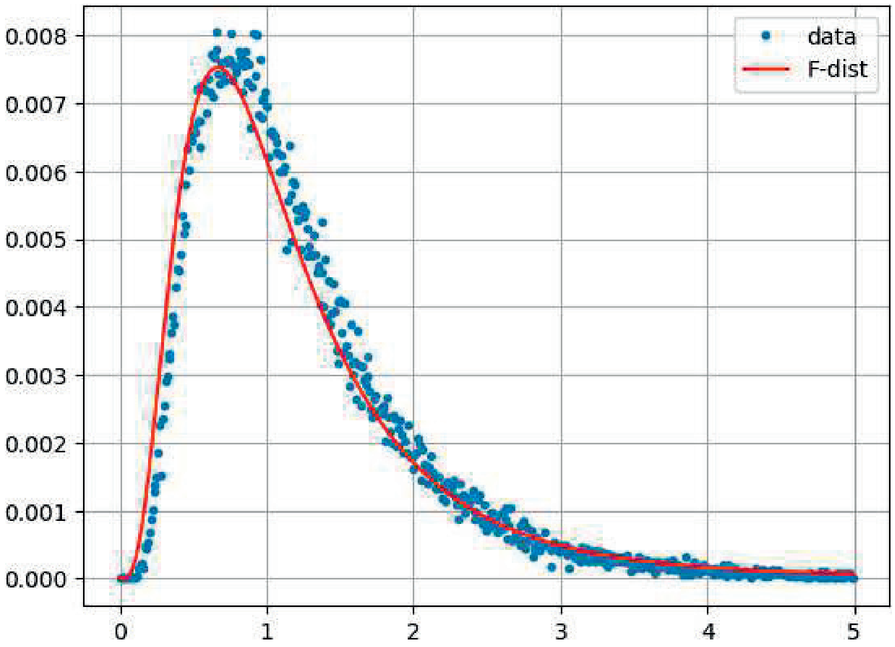


FIGURE 9.15
Ratio of the VV bands of two Sentinel S1 images acquired over a region of (approximate) no-change, compared with a plot of the F-distribution for 2m and 2m degrees of freedom with m=5.

or

$$\frac{g_1 g_2}{(g_1+g_2)^2} \le \tilde{t},$$

where $\tilde{t}$ depends on t. Inverting both sides of the above inequality and simplifying, we obtain

$$\frac{g_1}{g_2} + \frac{g_2}{g_1} \ge \frac{1}{\tilde{t}} - 2.$$

It follows that the critical region has the form

$$\frac{g_1}{g_2} \le c_1 \text{ or } \frac{g_1}{g_2} \ge c_2,$$

where the decision thresholds c_1 and c_2 depend on $\tilde{t}$.

The test statistic is thus the random variable G_1/G_2 corresponding to the ratio of SAR pixel intensities. It tests for a significant increase in intensity between times t_1 and t_2, or for a significant decrease. We still require its distribution. Both G_1 and G_2 have gamma distributions of the form (Equations (9.27) or (9.28))

$$p(g \mid x) = \frac{1}{(x/m)^m \Gamma(m)} g^{m-1} e^{-gm/x}.$$

Making the change of variable $z = 2gm/x$, or, inverting, $g(z) = xz/2m$, we have for the random variable Z, from Theorem 2.1,

$$p(z \mid x) = p(g(z) \mid x) \left|\frac{dg}{dz}\right| = \frac{1}{(x/m)^m\Gamma(m)} \left(\frac{xz}{2m}\right)^{m-1} e^{-z/2} \left|\frac{dg}{dz}\right|.$$

With $|dg/dz| = x/2m$ we obtain

$$p(z \mid x) = \frac{1}{2^m\Gamma(m)} z^{m-1} e^{-z/2}.$$

Comparison with Equation (2.38) shows that Z, and hence, the random variables $2G_1m/x_1$ and $2G_2m/x_2$, are chi-square distributed with $2m$ degrees of freedom. Under the null hypothesis, we have $x_1 = x_2$, so that the test statistic G_1/G_2 is a ratio of two chi-square distributed random variables. Our test statistic is therefore F-distributed with $2m$ and $2m$ degrees of freedom; see Equation (2.83). The percentiles of the F-distribution can therefore be used to set change/no-change decision thresholds to any desired degree of significance. Figure 9.15 shows a specific example for two successive 5-look Sentinel S1 image acquisitions downloaded from the Google Earth Engine.

9.4.1.2 Multi-temporal data

Suppose we have a series of k SAR scalar intensity images and would like to know where and when changes have taken place. A first reaction might be to simply apply the bi-temporal method we developed above to each of the $k-1$ time intervals. The difficulty with this approach is the rate of false positives. If the bi-temporal tests are statistically independent, then the probability of not getting a false positive over a series of length k is the product of not getting one in each of the $k-1$ intervals, i.e., $(1-\alpha)^{k-1}$ and the overall first kind error probability α_T is its complement:

$$\alpha_T = 1 - (1-\alpha)^{k-1}.$$

For example, for $k = 26$, even with a small value of $\alpha = 0.01$ this gives a very large false positive rate of 22.2%. Actually things are a bit worse. The bi-temporal tests are manifestly not independent because consecutive tests have one image in common. The best one can say in this situation is

$$\alpha_T \le (k-1)\alpha,$$

or $\alpha_T \le 25\%$ for $k = 26$ and $\alpha = 0.01$. If we wish to set a false positive rate of at most, say, 1% for the entire series, then each bi-temporal test must have a significance level of $\alpha = 0.0004$ and a correspondingly large false negative rate β. In other words many significant changes may be missed.

To avoid this, we will proceed as follows: Consider the situation in which there are k observations g_i, $i = 1 \dots k$, each parameterized by its mean value

x_i, and we wish to test the null hypothesis that there is no change in the first j observations of the series,

$$H_{0j}:\ x_1 = x_2 = \ldots = x_j = x$$

against the alternative hypothesis

$$H_{1j}:\ x_1 = x_2 = \ldots = x_{j-1} = x,\ x_j \neq x,$$

i.e., that the first change occurred after the $(j-1)$th observation. Then the likelihood function under H_{0j} is

$$L_0 = \frac{1}{(x/m)^{mj}\Gamma(m)^j}\Big(\prod_{i=1}^{j} g_i^{m-1}\Big)\exp\big(-(m/x)\sum_{i=1}^{j} g_i\big), \tag{9.32}$$

with the maximum-likelihood estimate of x given by (see Exercise 5(b))

$$\hat{x} = \frac{1}{j}\sum_{i=1}^{j} g_i.$$

For the alternative H_{1j},

$$L_1 = \frac{1}{\prod_{i=1}^{j-1}(x_i/m)^m (x_j/m)^m \Gamma(m)^j}\Big(\prod_{i=1}^{j} g_i^{m-1}\Big)\exp\big(-m\sum_{i=1}^{j}(g_i/x_i)\big), \tag{9.33}$$

with maximum-likelihood estimators

$$\hat{x}_i = \frac{1}{j-1}\sum_{i=1}^{j-1} g_i,\ i = 1\ldots j-1, \quad \hat{x}_j = g_j.$$

The likelihood ratio test is therefore (the exponential terms cancel exactly; see Exercise 6)

$$\begin{aligned}\frac{\hat{L}_0}{\hat{L}_1} &= \frac{(g_1+\ldots+g_{j-1})^{m(j-1)}(g_j/m)^m(m(j-1))^{-m(j-1)}}{(g_1+\ldots+g_j)(jm)^{-mj}}\\ &= \frac{j^m}{(j-1)^{(j-1)m}}\frac{(g_1+\ldots+g_{j-1})^{m(j-1)}g_j^m}{(g_1+\ldots+g_j)^{mj}}.\end{aligned}$$

Accordingly, we define the following test statistic:

$$R_j = \frac{j^{jm}}{(j-1)^{m(j-1)}}\frac{(G_1+\ldots+G_{j-1})^{m(j-1)}G_j^m}{(G_1+\ldots+G_j)^{mj}}, \quad j = 2\ldots k. \tag{9.34}$$

If we wish to test the null or no-change hypothesis

$$H_0 : x_1 = x_2 = \ldots = x_k$$

against *all* alternatives, that is, that one or any number of changes have occurred, then the likelihood ratio test statistic is

$$Q_k = k^{mk} \frac{\prod_{i=1}^k G_i^m}{(\sum_{j=1}^k G_j)^{km}}, \tag{9.35}$$

which is referred to as an *omnibus test.* The reader is asked to prove this in Exercise 7 and, furthermore, to show that

$$Q_k = \prod_{j=2}^{k} R_j. \tag{9.36}$$

Our sequential change analysis strategy for a time series of k observations will then be first to test Q_k and, if the null hypothesis is not rejected, to stop. Under the alternative, i.e., that at least one change has occurred, we then test R_2, i.e., $x_1 = x_2$ against $x_1 \neq x_2$. If the null hypothesis H_{02} is not rejected, then we proceed to test $x_1 = x_2 = x_3$ against $x_1 = x_2 \neq x_3$ using the statistic R_3, and so on. If and when a null hypothesis is rejected, we note the interval in which the change occurred and re-start the procedure from there. This is referred to this as the *sequential omnibus test procedure.*

Note that the test statistics R_j are independent of one another under the successive null hypotheses. This can be seen by writing Equation (9.34) in the form

$$R_j = \frac{j^m}{(j-1)^{m(j-1)}} U_j^{m(j-1)} (1 - U_j)^m, \quad j = 2 \dots k,$$

where

$$U_j = S_{j-1}/S_j, \text{ with } S_j = G_1 + \dots + G_j.$$

According to Theorem 2.7, the random variables S_j and U_j are independent. Hence, U_j and $S_j + G_{j+1}$ and so U_j and $U_{j+1} = S_j/S_{j+1}$ are also independent. It follows that R_j and R_{j+1} are indeed independent and that Q_k in Equation (9.36) is factored into a product of independent test statistics.

The decision at each time point can be formulated, as in the bi-temporal case in Section 9.4.1.1, as a two-sided test on an F-distributed random variable; see Conradsen et al. (2016). However, we will proceed directly to the more general multivariate case.

9.4.2 Polarimetric imagery: the complex Wishart distribution

In the multivariate situation, where we deal with complex vector observations like Equation (9.26), we represent an observation in an m look-averaged polarimetric image by the random matrix $\boldsymbol{X}$ with a complex Wishart distribution and with realization

$$\boldsymbol{x} = \sum_{\nu=1}^{m} \boldsymbol{s}(\nu)\boldsymbol{s}(\nu)^\dagger = m\bar{\boldsymbol{c}},$$

where $\bar{\boldsymbol{c}}$ is given in the last equality of Equation (5.30). In the polarimetric matrix image, the components of the matrix $\bar{\boldsymbol{c}}$ constitute the image pixel bands.*

9.4.2.1 Bi-temporal data

We need the following result (Goodman, 1963): If

$$\boldsymbol{X} = \sum_{\nu=1}^{m} \boldsymbol{Z}(\nu)\boldsymbol{Z}(\nu)^\dagger, \quad \boldsymbol{Z}(\nu) \sim \mathcal{N}_C(\mathbf{0}, \boldsymbol{\Sigma}),\ \nu = 1\ldots m,$$

is complex Wishart distributed with covariance matrix $\boldsymbol{\Sigma}$ and m degrees of freedom, then the maximum-likelihood estimate for $\boldsymbol{\Sigma}$ is

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{m}\sum_{\nu=1}^{m} \boldsymbol{z}(\nu)\boldsymbol{z}(\nu)^\dagger = \frac{1}{m}\boldsymbol{x}. \tag{9.37}$$

This closely parallels the real random vector case; see, e.g., Equation (2.75). We conclude that $\bar{\boldsymbol{c}} = \hat{\boldsymbol{\Sigma}}$, that is, $\bar{\boldsymbol{c}}$ is the maximum-likelihood estimate of the parameter $\boldsymbol{\Sigma}$.

The density function for the $N \times N$ random matrix $\boldsymbol{X}$ is the complex Wishart distribution given by Theorem 2.11, Equation (2.62). To simplify the notation a little, let us define

$$\Gamma_N(m) = \pi^{N(N-1)/2}\prod_{i=1}^{N}\Gamma(m+1-i).$$

Then, for two m-look quad polarimetric covariance images $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$, the multivariate densities are

$$\begin{aligned} p(\boldsymbol{x}_1 \mid m, \boldsymbol{\Sigma}_1) &= \frac{|\boldsymbol{x}_1|^{m-N}\exp(-\mathrm{tr}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{x}_1))}{|\boldsymbol{\Sigma}_1|^m\Gamma_N(m)} \\ p(\boldsymbol{x}_2 \mid m, \boldsymbol{\Sigma}_2) &= \frac{|\boldsymbol{x}_2|^{m-N}\exp(-\mathrm{tr}(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{x}_2))}{|\boldsymbol{\Sigma}_2|^m\Gamma_N(m)}. \end{aligned}$$

We now define the null (or no-change) simple hypothesis

$$H_0: \quad \boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \boldsymbol{\Sigma},$$

against the alternative composite hypothesis

$$H_1: \quad \boldsymbol{\Sigma}_1 \neq \boldsymbol{\Sigma}_2.$$

*Note that $\boldsymbol{x}$ is now an observation, not a parameter.

Under H_0, the likelihood for $\mathbf{\Sigma}$ is given by

$$\begin{aligned} L_0(\mathbf{\Sigma}) &= p(\boldsymbol{x}_1 \mid m, \mathbf{\Sigma})p(\boldsymbol{x}_2 \mid m, \mathbf{\Sigma}) \\ &= \frac{|\boldsymbol{x}_1|^{m-N}|\boldsymbol{x}_2|^{m-N}\exp(-\mathrm{tr}(\mathbf{\Sigma}^{-1}(\boldsymbol{x}_1+\boldsymbol{x}_2)))}{|\mathbf{\Sigma}|^{2m}\Gamma_N(m)^2}. \end{aligned}$$

According to Theorem 2.12, $\boldsymbol{X}_1 + \boldsymbol{X}_2$ is complex Wishart distributed with $2m$ degrees of freedom and therefore, with Equation (9.37), the maximum-likelihood estimate of $\mathbf{\Sigma}$ is

$$\hat{\mathbf{\Sigma}} = \frac{1}{2m}(\boldsymbol{x}_1 + \boldsymbol{x}_2).$$

Hence, the maximum likelihood under the null hypothesis is

$$L_0(\hat{\mathbf{\Sigma}}) = \frac{|\boldsymbol{x}_1|^{m-N}|\boldsymbol{x}_2|^{m-N}\exp(-2m\cdot\mathrm{tr}(\boldsymbol{I}))}{\left(\frac{1}{2m}\right)^{N\cdot 2m}|\boldsymbol{x}_1+\boldsymbol{x}_2|^{2m}\Gamma_N(m)^2},$$

where $\boldsymbol{I}$ is the $N \times N$ identity matrix and $\mathrm{tr}(\boldsymbol{I}) = N$. (Note that $|a\boldsymbol{x}| = a^N|\boldsymbol{x}|$ for constant a and a $N \times N$ matrix $\boldsymbol{x}$.) Under H_1 we obtain, similarly, the maximum likelihood for $\mathbf{\Sigma}_1$ and $\mathbf{\Sigma}_2$ as

$$L_1(\hat{\mathbf{\Sigma}}_1, \hat{\mathbf{\Sigma}}_2) = \frac{|\boldsymbol{x}_1|^{m-N}|\boldsymbol{x}_2|^{m-N}\exp(-2m\cdot\mathrm{tr}(\boldsymbol{I}))}{\left(\frac{1}{m}\right)^{Nm}\left(\frac{1}{m}\right)^{Nm}|\boldsymbol{x}_1|^{m}|\boldsymbol{x}_2|^{m}\Gamma_N(m)^2}. \tag{9.38}$$

Then, again according to Equation (2.77), the likelihood ratio test has the critical region

$$Q = \frac{L_0(\hat{\mathbf{\Sigma}})}{L_1(\hat{\mathbf{\Sigma}}_1, \hat{\mathbf{\Sigma}}_2)} = 2^{2Nm}\frac{|\boldsymbol{x}_1|^m|\boldsymbol{x}_2|^m}{|\boldsymbol{x}_1+\boldsymbol{x}_2|^{2m}} \le t. \tag{9.39}$$

For a large number of looks m, we can apply Theorem 2.15 to obtain the following asymptotic distribution for the test statistic: As $m \to \infty$, the quantity

$$-2\log Q = -2m(2N\log 2 + \log|\boldsymbol{x}_1| + \log|\boldsymbol{x}_2| - 2\log|\boldsymbol{x}_1+\boldsymbol{x}_2|)$$

is a realization of a chi-square random variable with N^2 degrees of freedom:

$$\Pr(-2\log Q \le z) \simeq P_{\chi^2;N^2}(z). \tag{9.40}$$

This follows from the fact that the number of parameters required to specify a $N \times N$ complex covariance matrix is N^2: N for the real diagonal elements and $N(N-1)$ for the complex elements above the diagonal.* Thus, in the notation of Theorem 2.15, the parameter space ω for $(\mathbf{\Sigma}_1, \mathbf{\Sigma}_2)$ has dimension $q = 2N^2$, and the subspace ω_0 for the simple null hypothesis has dimension $r = N^2$, so $q - r = N^2$.

*The elements below the diagonal are their complex conjugates.

9.4.2.2 Multi-temporal data

The situation for time series of polarimetric SAR images exactly parallels that for scalar observations. Assuming as before that the number of looks m is the same for all k images in the series, the complex Wishart distributions of the $(N \times N)$-dimensional observations $\boldsymbol{x} = m\bar{\boldsymbol{c}}$ are completely determined by the parameters $\boldsymbol{\Sigma}_i$, $i = 1 \dots k$, and the test statistic for

$$H_{0j}: \ \boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \dots = \boldsymbol{\Sigma}_j = \boldsymbol{\Sigma}$$

against

$$H_{1j}: \ \boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \dots = \boldsymbol{\Sigma}_{j-1} \neq \boldsymbol{\Sigma}_j$$

is given by

$$R_j = \frac{j^{jmN}}{(j-1)^{(j-1)mN}} \frac{|\boldsymbol{X}_1 + \dots + \boldsymbol{X}_{j-1}|^{m(j-1)} |\boldsymbol{X}_j|^m}{|\boldsymbol{X}_1 + \dots + \boldsymbol{X}_j|^{jm}}. \tag{9.41}$$

Taking minus twice the logarithm of R_j in Equation (9.41), we have

$$\begin{aligned} Z_j = -2\log R_j = m\Big(&N(j\log j - (j-1)\log(j-1)) \\ &+ (j-1)\log\Big|\sum_{i=1}^{j-1} \boldsymbol{X}_i\Big| + \log|\boldsymbol{X}_j| - j\log\Big|\sum_{i=1}^{j} \boldsymbol{X}_i\Big|\Big). \end{aligned} \tag{9.42}$$

For both quad ($N = 3$) and dual ($N = 2$) polarimetric matrices, the parameter space ω of $(\boldsymbol{\Sigma}, \boldsymbol{\Sigma}_j)$ has dimension $q = 2 \times N^2$, and the subspace ω_0 of $\boldsymbol{\Sigma}$ for the null hypothesis has dimension $r = N^2$, so as for the bi-temporal case, $q - r = N^2$. Thus, from Theorem 2.15, the approximate distribution function of Z_j is chi-square with N^2 degrees of freedom

$$\Pr(-2\log R_j \le z) \simeq P_{\chi^2;N^2}(z) \tag{9.43}$$

and the P-value for an observation z_j of Z_j is approximately

$$1 - P_{\chi^2;N^2}(z_j).$$

In practice, m may be rather small for typical look-averaged polarimetric SAR images. Conradsen et al. (2016) give a better (and more complicated) approximation to the distribution of the test statistic based on large sample distribution theory (Box, 1949). For finite m and an $N \times N$ covariance matrix:

$$\Pr(-2\rho_j \log R_j \le z) \simeq P_{\chi^2;N^2}(z) + \omega_{2j}\left[P_{\chi^2;N^2+4}(z) - P_{\chi^2;N^2}(z)\right], \tag{9.44}$$

where

$$\rho_j = 1 - \frac{2N^2 - 1}{6Nm} \cdot \left(1 + \frac{1}{j(j-1)}\right)$$

$$\omega_{2j} = -\frac{N^2}{4}\cdot\left(1-\frac{1}{\rho_j}\right)^2 + \frac{N^2(N^2-1)}{24m^2\rho_j^2}\cdot\left(1+\frac{2j-1}{j^2(j-1)^2}\right).$$

Correspondingly, the P-values are

$$\mathcal{P}_j = 1 - \Pr(-2\rho_j \log R_j \le z). \tag{9.45}$$

The quantities $\rho_j \to 1$ and $\omega_{2j} \to 0$ as $m \to \infty$, so that Equation (9.44) converges to Equation (9.43) for large m.

As before, the R_j are independent under the null hypotheses and

$$\prod_{j=2}^{k} R_j = Q_k = k^{kmN}\frac{\prod_{i=1}^{k}|\boldsymbol{X}_i|^m}{|\sum_{i=1}^{k}\boldsymbol{X}_i|^{km}}, \tag{9.46}$$

where Q_k is the omnibus test statistic for $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \ldots = \boldsymbol{\Sigma}_k$ against all alternatives. The sequential change analysis strategy for a time series of k observations of dual or quad polarimetric images is then the same as for the scalar situation: First test the statistic Q_k given by Equation (9.46) using the analogous approximate distribution to Equation (9.44), namely

$$\begin{aligned}\Pr(-2\rho\log Q_k \le z) &\simeq P_{\chi^2;(k-1)N^2}(z)\\ &\quad + \omega_2\left[P_{\chi^2;(k-1)N^2+4}(z) - P_{\chi^2;(k-1)N^2}(z)\right]\end{aligned} \tag{9.47}$$

with

$$\rho = 1 - \frac{2N^2-1}{6N(k-1)}\cdot\left(\frac{k}{m}-\frac{1}{mk}\right)$$

$$\omega_2 = -\frac{N^2(k-1)}{4}\cdot\left(1-\frac{1}{\rho}\right)^2 + \frac{N^2(N^2-1)}{24m^2\rho^2}\cdot\left(k-\frac{1}{k^2}\right)$$

and with P-value

$$\mathcal{P}_k = 1 - \Pr(-2\rho\log Q_k \le z). \tag{9.48}$$

If the null hypothesis is not rejected, stop. Under the alternative, test R_2 (Equation (9.41) with $j = 2$), i.e., $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2$ against $\boldsymbol{\Sigma}_1 \ne \boldsymbol{\Sigma}_2$. If the null hypothesis H_{02} is not rejected, then proceed to test $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \boldsymbol{\Sigma}_3$ against $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 \ne \boldsymbol{\Sigma}_3$ using the statistic R_3, and so on. When a null hypothesis is rejected, record the interval in which the change occurred and re-start the procedure from there.

Finally, note that all of the results of Subsection 9.4.1 are a special case of the multivariate situation described here and can be recovered by setting $N \to 1$, $\boldsymbol{\Sigma}_i \to x_i$, $|\boldsymbol{X}_i| \to G_i$, and $\boldsymbol{x}_i \to g_i$.

9.4.2.3 The Loewner order

Now we have a method to determine both where and when significant changes occur in polarimetric radar time series, but what about the direction of change? For example a sudden, significant and spatially coherent decrease in reflectance might signal widespread flooding, or perhaps deforestation. It turns out that, for observations of the polarimetric matrix, the idea of direction is ambiguous. Consider the simplest case in which only the diagonal elements of the m-look dual polarization matrix are measured, for instance

$$\boldsymbol{x} = m\bar{\boldsymbol{c}} = m\begin{pmatrix} \langle|s_{vv}|^2\rangle & 0 \\ 0 & \langle|s_{vh}|^2\rangle \end{pmatrix}.$$

There may be a decrease in one component and an increase in the other. Since any difference of successive observations $\boldsymbol{x}_i - \boldsymbol{x}_{i-1}$ is likewise diagonal, its diagonal elements are also its eigenvalues. If the difference is positive definite (both eigenvalues positive), both components have increased, if negative definite then both have decreased. So we can associate positive or negative, definiteness with change direction, but are forced to leave indefinite difference matrices undefined. This *partial ordering* characterization of change is known as the *Loewner order* and it was suggested by Nielsen et al. (2020) to apply it to dual and full polarization matrix time series in general: for complex, symmetric (Hermitian) observation matrices $\boldsymbol{x}_i$, the eigenvalues of the differences $\boldsymbol{x}_i - \boldsymbol{x}_{i-1}$ partially characterize the direction of changes and so, in the event of significant change at time i, we determine the positive or negative definiteness (or indefiniteness) of the difference between consecutive covariance matrix pixels. Moreover, instead of subtracting the value $\boldsymbol{x}_{i-1}$ for the preceding image, we can subtract the average over all preceding values for which no new change has been signaled. This can conveniently be done recursively with the provisional means algorithm of Chapter 2.

9.4.2.4 An improved temporal speckle filter

The running average image needed for the Loewner order determination described above is in fact an *adaptive temporal speckle filter (ATSF)* on the last image in the sequence (Canty et al., 2020). After all bi-temporal changes have been characterized, pixels which have not exhibited significant change over the full sequence have been maximally averaged, and changed pixels have been averaged over the period following the most recent change. In order to reduce the speckle for regions of very recent change, all averages for which the number of contributing pixels is below some threshold may be replaced by the output of a spatial filter, such as one of those described in Chapter 5, applied to the original image. An example, taken from Canty et al. (2020), is shown in Figure 9.16. Note that the moving excavators/back fillers in the filtered image are at the same positions as in the original unfiltered image.

9.4.3 Python software

The Python script `sar_seqQ.py` documented in Appendix C implements the sequential omnibus change detection procedure outlined above for both scalar intensity and polarimetric matrix imagery. In practice, it is convenient to

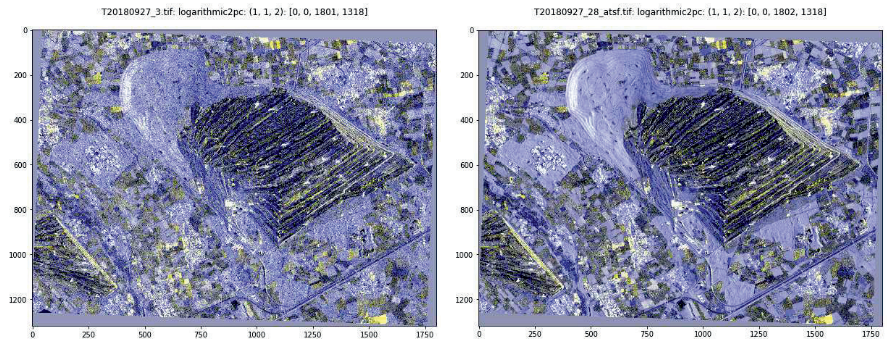

T20180927_3.tif: logarithmic2pc: (1, 1, 2): [0, 0, 1801, 1318]
T20180927_28_atsf.tif: logarithmic2pc: (1, 1, 2): [0, 0, 1802, 1318]

FIGURE 9.16
Left: RGB composite ($R = |S_{vv}|^2, G = |S_{vv}|^2, B = |S_{vvh}|^2$) of a polarimetric Sentinel-1 SAR image acquired 27 Sept 2018 over the Hambach open-cast coal mine near Jülich, Germany. The bright features in the mine pit are very large mobile excavators and back fillers. Right: The adaptive temporal speckle filter (ATSF) determined from a time series of 29 images terminating with the September 27 image.

pre-calculate all of the possible tests in advance, for k observations, namely

$$R^{\ell}_j, \quad \ell = 1 \dots k-1,\ j = \ell+1 \dots k,$$

where ℓ indexes the first image of each subsequence, and then to determine Q_k from their product. The P-values are stored in a (k, k, n) array, where n is the number of pixels. The array is kept in a memory-mapped file in order to save memory. Then, in a second pass over the array, the changes at a given significance level are recorded: Q_k is tested, and if the null hypothesis is rejected, R_2 is tested, i.e., $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2$ against $\boldsymbol{\Sigma}_1 \neq \boldsymbol{\Sigma}_2$. If the null hypothesis H_{02} is not rejected, then the calculation proceeds to test $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \boldsymbol{\Sigma}_3$ against $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 \neq \boldsymbol{\Sigma}_3$ using the statistic R_3, and so on. Whenever a null hypothesis is rejected, the interval in which the change occurred is noted and the procedure restarted from there. In this way, the script keeps track of when and how many changes occur per image pixel. Results are saved in byte format in five image files:

- `cmap.tif`: period of the most recent change (1 band)
- `smap.tif`: period of the first change (1 band)
- `fmap.tif`: total number (or frequency) of changes (1 band)
- `bmap.tif`: Loewner order of the changes in each interval ($k-1$ bands)
- `atsf.tif`: ATSF image (1, 2, 3, 4, or 9 bands)

The first pass, that is the calculation of the P-values, can optionally be run in parallel on two or more IPython engines if multiple CPU cores are available;

see Appendix C. The code for the second pass over the P-values array is shown in Listing 9.3. Note that, in lines 17—18, the Omnibus test on the P-values of Equation (9.47) always determines whether or not a sequential change is indexed.

The Loewner order is determined in a final pass over the P-values: For a given pixel, the average covariance matrix of the no-change observations up to but not including the time of a significant change is accumulated with the method of provisional means (Section 2.3.2). This average is subtracted from the covariance matrix immediately following the signaled change and the positive/negative definiteness (or indefiniteness) of the difference is ascertained. A particularly efficient method involving matrix pivots (Nielsen, 2020) is applied which avoids determining, in the case of full (2×2) and (3×3) covariance matrices, the eigenvalues of the complex matrix differences.

The bash script `run_sar_seq.sh` can be used to create a list of input file names and submit it to `sar_seq.py`. Here, for example, we run the program on a directory containing 8 Sentinel-1 dual polarimetric images acquired over the Camargue in the south of France (Muro et al., 2016); see Figure 9.17. The

Listing 9.3: Determining change maps from a P-values array (excerpt from the Python script `sar_seqQ.py`).

```
 1 def change_maps(pvarray,significance):
 2     import numpy as np
 3     k = pvarray.shape[0]
 4     n = pvarray.shape[2]
 5     #  map of most recent change occurrences
 6     cmap = np.zeros(n,dtype=np.byte)
 7     #  map of first change occurrence
 8     smap = np.zeros(n,dtype=np.byte)
 9     #  change frequency map
10     fmap = np.zeros(n,dtype=np.byte)
11     #  bitemporal change maps
12     bmap = np.zeros((n,k-1),dtype=np.byte)
13     for ell in range(k-1):
14         pvQ = pvarray[ell,k-1,:]
15         for j in range(ell,k-1):
16             pv = pvarray[ell,j,:]
17             idx = np.where((pv<=significance)&
18                    (pvQ<=significance)&(cmap==ell))
19             fmap[idx] += 1
20             cmap[idx] = j+1
21             bmap[idx,j] = 1
22             if ell==0:
23                 smap[idx] = j+1
24     return (cmap,smap,fmap,bmap)
```

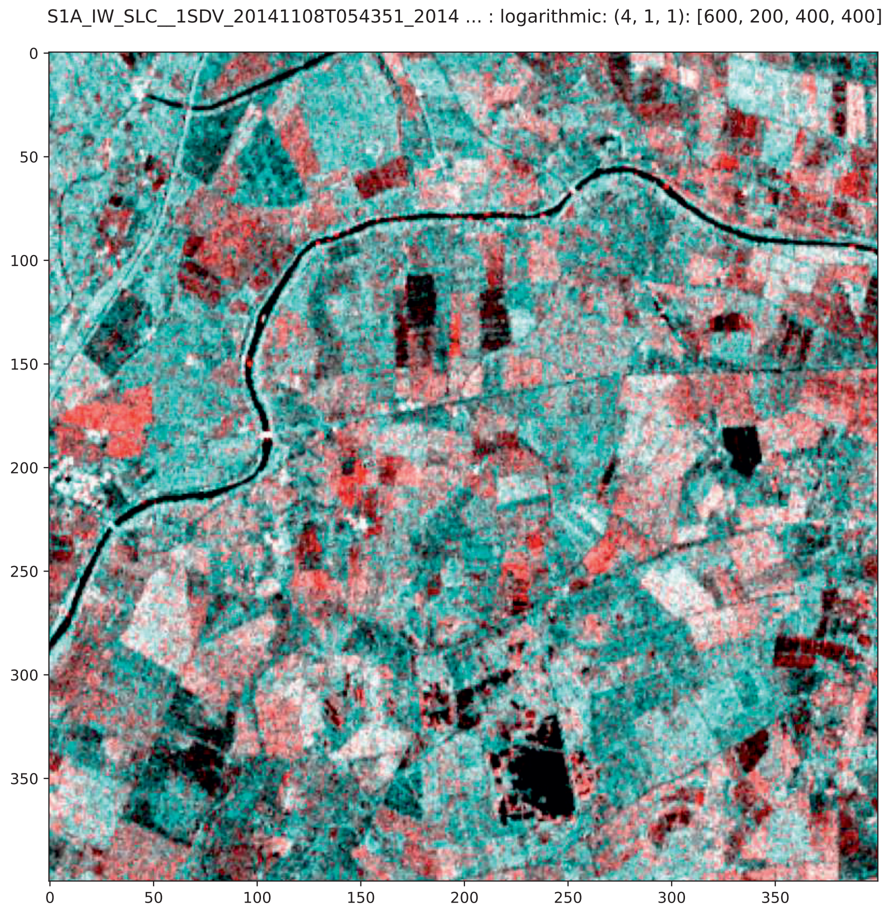

S1A_IW_SLC__1SDV_20141108T054351_2014 ... : logarithmic: (4, 1, 1): [600, 200, 400, 400]
0
50
100
150
200
250
300
350
0
50
100
150
200
250
300
350

FIGURE 9.17
RGB composite of bands ($|s_{vh}|^2$, $|s_{vv}|^2$, $|s_{vv}|^2$) of a Sentinel-1 image acquired over the Camargue on November 8, 2014.

ENL is 12, and the significance level is set to 0.01%:

```
# start parallel engines in background
!ipcluster start -n 4 --daemonize

!scripts/run_sar_seqQ.sh S1A imagery/ 12 0.0001

===========================================
     Multi-temporal SAR Change Detection
===========================================
Sat Sep 14 15:07:12 2024
First (reference) filename:  imagery/S1A ... .tif
```

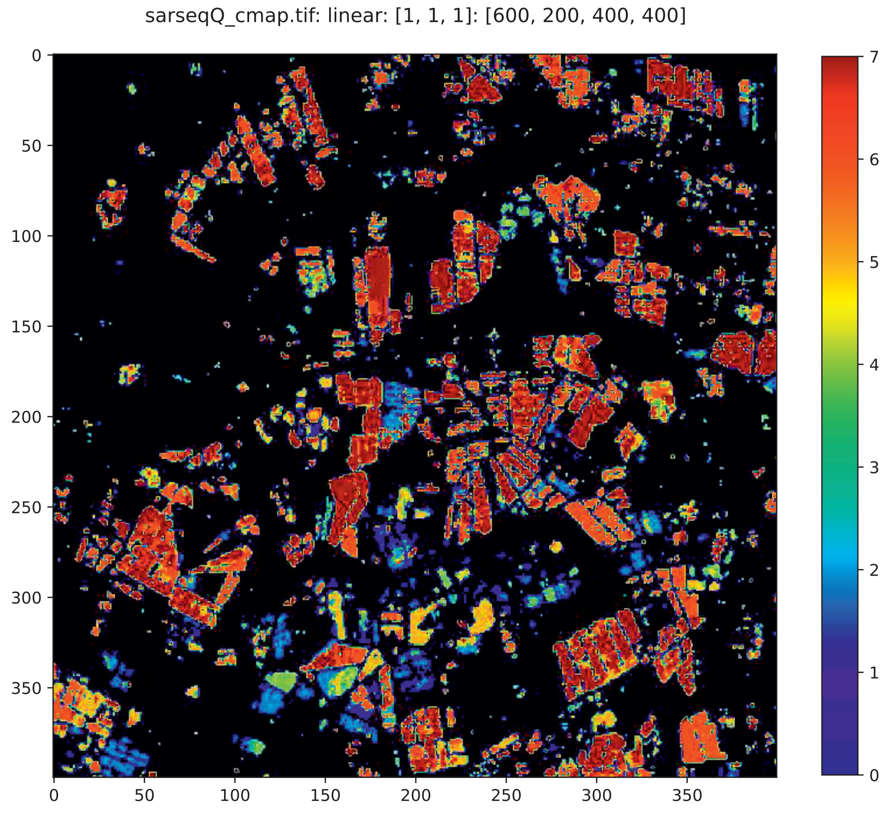


FIGURE 9.18
Change map for the image of Figure 9.17 for a series of 8 Sentinel-1 dual polarimetric images acquired between November, 2014 and October, 2015. The time of the most recent change is color coded, black: no change, brown: last change in interval 7.

```
number of images: 8
equivalent number of looks: 12.000000
significance level: 0.000100
Dual polarizaton
pre-calculating Rj and p-values ...
attempting parallel calculation ...
available engines [0, 1, 2, 3]
ell =  1 2 3 4 5 6 7
elapsed time for p-value calculation: 7.820582389831543
last change map written to: .../imagery/sarseqQ_cmap.tif
frequency map written to: .../imagery/sarseqQ_fmap.tif
bi-temporal map written to: .../imagery/sarseqQ_bmap.tif
```

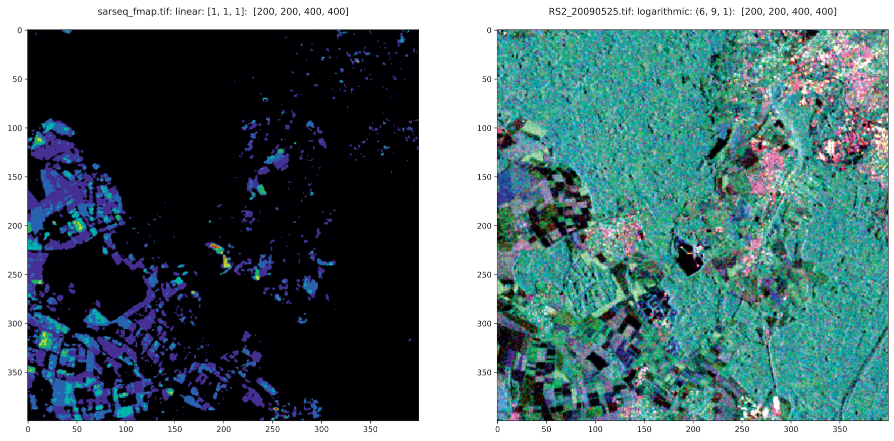

sarseq_fmap.tif: linear: [1, 1, 1]: [200, 200, 400, 400]
RS2_20090525.tif: logarithmic: (6, 9, 1): [200, 200, 400, 400]

FIGURE 9.19
Left: Number (or frequency) of changes in a time series of 12 RADARSAT-2 quad polarimetric images acquired from May, 2009 to October, 2010 over an area southwest of Bonn, Germany. Right: Pauli decomposition display of the first image in the sequence: $R = |s_{hh} - s_{vv}|^2$, $G = |s_{hv}|^2$, $B = |s_{hh} + s_{vv}|^2$. Coniferous forest areas appear green and exhibit no changes. The number of changes is color coded, black: no change through brown: 11 changes.

```
first change map written to: .../imagery/sarseqQ_smap.tif
atsf written to: .../imagery/sarseqQ_atsf.tif
```

A color-coded map of the time of the most recent change is shown in Figure 9.18. Another example is given in Figure 9.19 for a time series of quad polarimetric RADARSAT-2 images; see also the accompanying Jupyter notebook. This time the change map on the left shows the frequency of changes (total number) over the full time period. The “hot spot” near the center (same color coding as in Figure 9.18) corresponds to continual movement of the dredging arms in a flooded sand quarry. Other changes are due to agricultural activity; the forested areas show no significant changes.

9.4.4 SAR change detection on the Google Earth Engine

The GEE platform offers a convenient, free and near-real-time source of sequential SAR data for time series analysis. It archives Sentinel-1a and Sentinel-1b SAR images as soon as they are made available by the European Space Agency.[*] As discussed in Chapter 1, the dual polarimetric Sentinel-1 sensor transmits in only one polarization and receives in two, thus measuring only

[*]Sentinel-1b is no longer functioning but is now being replaced by Sentinel-1c, successfully launched on December 5, 2024.

the bands s_{vv} and s_{vh} or s_{hh} and s_{hv}. In particular, the GEE archived data of which we make use here are acquired in interferometric wide swath (IW) mode and processed to the ground range detected (GRD) product. The spatial resolution is (range by azimuth) 20 m by 22 m and the pixel spacing is 10 m. The IW data are multilooked, the number of looks m is 5 by 1 and the equivalent number of looks is 4.4. Only intensity data are available on the archive, so that the polarimetric covariance matrix representation is diagonal. Moreover, for land surface acquisitions, the vertical emission mode is generally used. Therefore, the look averaged polarimetric matrices are of the form

$$\bar{\boldsymbol{c}} = \begin{pmatrix} \langle |s_{vv}|^2\rangle & 0 \\ 0 & \langle |s_{vh}|^2\rangle \end{pmatrix}.$$

The matrix $m\bar{\boldsymbol{c}}$ is not complex Wishart distributed, but it is can be shown that the test statistics Q_k and R_j discussed above are still approximately valid. For example, in the bi-temporal case, we have observations

$$\boldsymbol{g} = \begin{pmatrix} g_1 & 0 \\ 0 & g_2 \end{pmatrix}, \quad \boldsymbol{h} = \begin{pmatrix} h_1 & 0 \\ 0 & h_2 \end{pmatrix}$$

where g_1, g_2, h_1, h_2 are gamma distributed with equal m and means x_1, x_2, y_1 and y_2. Assuming independence of g_1 and g_2 and of h_1 and h_2,[*] under the null hypothesis $x_1 = y_1 = x$ and $x_2 = y_2 = y$ the likelihood is

$$\begin{aligned} L_0(x,y) &= p(g_1 \mid x)p(g_2 \mid y)p(h_1 \mid x)p(h_2 \mid y) \\ &= \frac{m^{4m}}{(x)^{2m}(y)^{2m}\Gamma(m)^4}\; g_1^{m-1}h_1^{m-1}\; g_2^{m-1}h_2^{m-1}e^{-(g_1+h_1)m/x-(g_2+h_2)m/y}. \end{aligned}$$

The maximum-likelihood estimates are

$$\hat{x} = (g_1+h_1)/2, \quad \hat{y} = (g_2+h_2)/2$$

so that

$$\hat{L}_0(\hat{x},\hat{y}) = \frac{2^{4m}m^{4m}}{(g_1+h_1)^{2m}(g_2+h_2)^{2m}\Gamma(m)^4}(g_1h_1g_2h_2)^{m-1}e^{-4m}.$$

The likelihood for the alternative hypothesis is

$$\begin{aligned} &L_1(x_1,x_2,y_1,y_2) = \\ &\quad \frac{m^{4m}}{(x_1x_2y_1y_2)^m\Gamma(m)^4}\cdot(g_1g_2h_1h_2)^{m-1}e^{-m(g_1/x_1+g_2/x_2+h_1/y_1+h_2/y_2)} \end{aligned}$$

[*]There is some empirical justification that this assumption does not seriously affect the validity of the hypothesis tests with GEE Sentinel-1 data. (Nielsen et al., 2017) investigate histograms of the $-2\log R_j$ values in no-change regions and find good agreement with the expected chi square distributions.

and therefore with $\hat{x}_1 = g_1,\ \hat{x}_2 = g_2,\ \hat{y}_1 = h_1,\ \hat{y}_2 = h_2,$

$$\hat{L}_1(\hat{x}_1, \hat{x}_2, \hat{y}_1, \hat{y}_2) = \frac{m^{4m}}{g_1 g_2 h_1 h_2} e^{-4m}.$$

The likelihood ratio test is then

$$\frac{\hat{L}_0(\hat{x}, \hat{y})}{\hat{L}_1(\hat{x}_1, \hat{x}_2, \hat{y}_1, \hat{y}_2)} = 2^{4m} \frac{(g_1 g_2 h_1 h_2)^m}{(g_1 + h_1)^{2m}(g_2 + h_2)^{2m}} = 2^{4m} \frac{|\boldsymbol{g}|^m |\boldsymbol{h}|^m}{|\boldsymbol{g} + \boldsymbol{h}|^{2m}} < k.$$

This is the same as Equation (9.39) with $N = 2$. Since the (diagonal) covariance matrices $\boldsymbol{h}$ and $\boldsymbol{g}$ have only two degrees of freedom, the statistic $-2\log Q$ is approximately distributed according to

$$\Pr(-2\log Q \le z) \simeq P_{\chi^2;2}(z). \tag{9.49}$$

The same argument applies to $-2\log R_j$ in the multi-temporal case, see Canty et al. (2020) for details.

Again, for a relatively small number of looks, the approximation may not be good. The improved approximations of Equations (9.44) through (9.48) only apply to the full covariance matrices. However, Conradsen et al. (2024) generalize them to include all block diagonal cases, including the simple 2×2 diagonal polarization matrices we are dealing with here. They obtain, for the sequential tests, the parameters

$$\rho_j = 1 - \frac{1}{6m} \cdot \left(1 + \frac{1}{j(j-1)}\right)$$

$$\omega_{2j} = -\frac{1}{2} \cdot \left(1 - \frac{1}{\rho_j}\right)^2$$

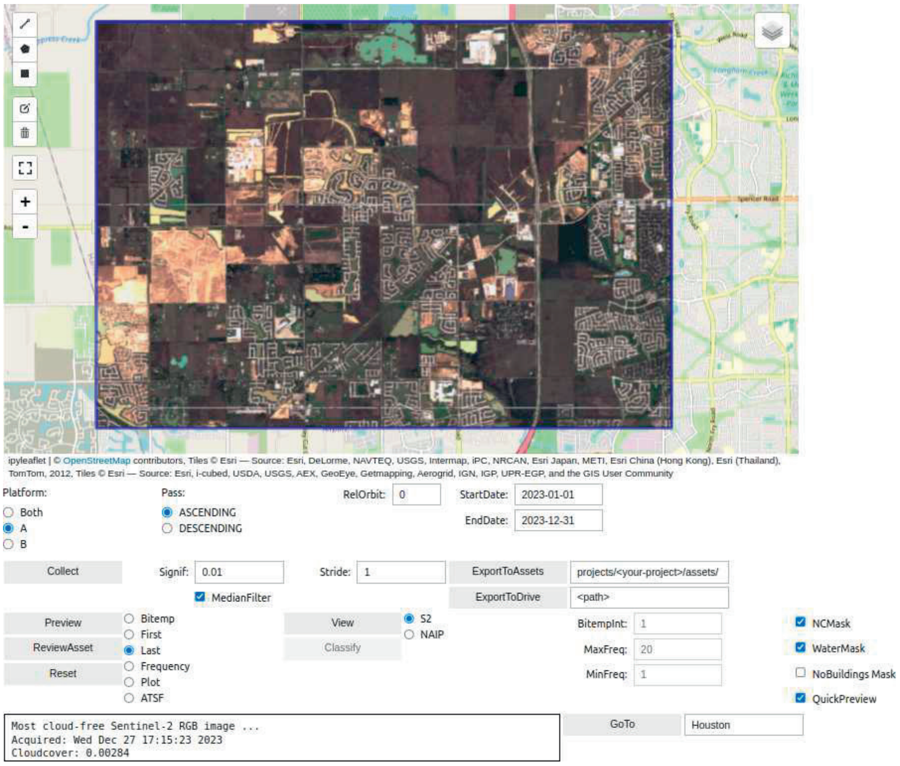
and, for the omnibus test,

$$\rho = 1 - \frac{1}{6(k-1)} \cdot \left(\frac{k}{m} - \frac{1}{mk}\right)$$

$$\omega_2 = -\frac{(k-1)}{2} \cdot \left(1 - \frac{1}{\rho}\right)^2.$$

In the software accompanying this text, the sequential omnibus change detection algorithm is programmed against the GEE Python API in the module `auxil.eesarseq.py`, see Appendix C. Additionally, the module `auxil.application` provides a convenient Jupyter Notebook widget interface which considerably simplifies the process of choosing geographic regions of interest and time intervals:

```
import ee
ee.Initialize()
from auxil.application import run
run()
```



ipyleaflet | © OpenStreetMap contributors, Tiles © Esri — Source: Esri, DeLorme, NAVTEQ, USGS, Intermap, iPC, NRCAN, Esri Japan, METI, Esri China (Hong Kong), Esri (Thailand), TomTom, 2012, Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community

Platform: Both [✓ A] B

Pass: [✓ ASCENDING] DESCENDING

RelOrbit: 0

StartDate: 2023-01-01

EndDate: 2023-12-31

Signif: 0.01

Stride: 1

ExportToAssets: projects/<your-project>/assets/

ExportToDrive: <path>

[✓ MedianFilter]

Bitemp First [✓ Last] Frequency Plot ATSF

[✓ S2] NAIP

BitempInt: 1

MaxFreq: 20

MinFreq: 1

[✓ NCMask] [✓ WaterMask] NoBuildings Mask [✓ QuickPreview]

GoTo: Houston

```
Most cloud-free Sentinel-2 RGB image ...
Acquired: Wed Dec 27 17:15:23 2023
Cloudcover: 0.00284
```

FIGURE 9.20
Jupyter widget interface to the sequential omnibus SAR change detection algorithm for processing Sentinel-1 time series on the GEE.

This generates a widget cell such as that shown in Figure 9.20.

Figure 9.21 gives an example of change detection in a series of Sentinel-1 images calculated from the widget interface of Figure 9.20. The change maps can be exported directly to the user's assets directory and then further processed in the GEE code editor (details in Appendix C).

As a further example, Figure 9.22 shows at the top changes accumulated during the 3-month Russian siege at the Asovstal Works in Ukraine. The red pixels indicate statistically significant increases in both the reflected VV and VH polarization channels (referred to here as positive definite changes), the cyan pixels (negative definite changes) correspond to significant decreases in VV and VH reflectance, and the yellow pixels (indefinite) to an increase in one channel and a decrease in the other. The bottom image plots the fraction of changed pixels at each two-week time interval that occurred for the period January, 2021 through August, 2023. The duration of the siege is well-defined.

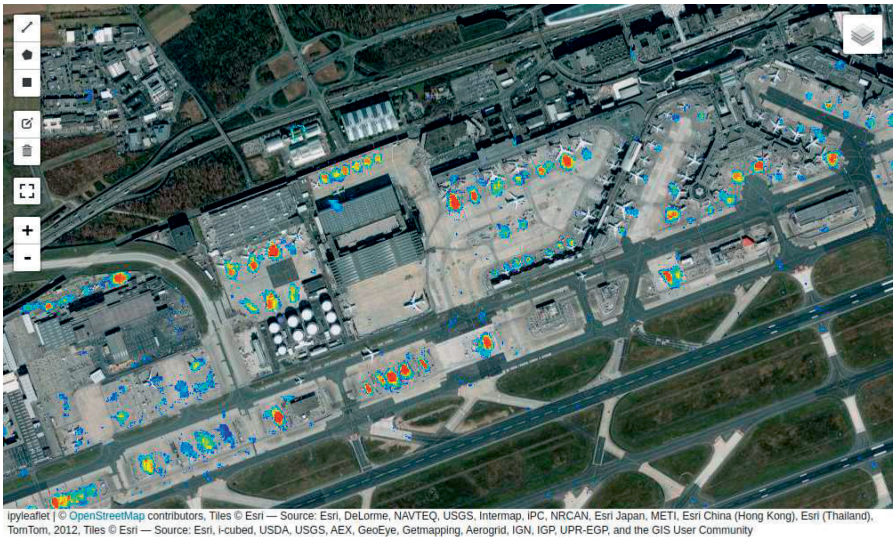
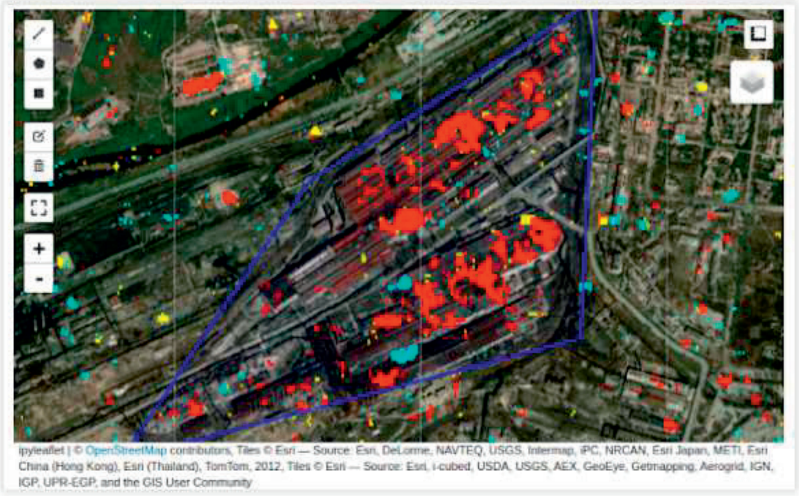

ipyleaflet | © OpenStreetMap contributors, Tiles © Esri — Source: Esri, DeLorme, NAVTEQ, USGS, Intermap, iPC, NRCAN, Esri Japan, METI, Esri China (Hong Kong), Esri (Thailand), TomTom, 2012, Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community

FIGURE 9.21
Change frequency map (blue: few changes, yellow/red: many changes) over the Frankfurt airport for a series of 31 Sentinel-1 images acquired between January and December, 2023. The "hot spots" occur at the aircraft parking positions and boarding gates. Change frequencies less than 5 are masked. Background: Esri World Imagery.

Also due to the Russian invasion of Ukraine, the Zaporizhzhia Nuclear Power Plant on the Dnieper river became the center of an ongoing nuclear safety crisis, described by Ukraine as an act of nuclear terrorism by Russia. The plant, which is the largest of its kind in Europe, has seen destruction of its infrastructure via shelling, damage to its power lines, amounting to what Ukrainian authorities call the largest situation of its kind in history. On 6 June 2023, the Kakhovka Dam was breached causing the depletion of the plant's main water source, the Kakhovka Reservoir, see Figure 9.23, which became too low to use within a few days. The red pixels in the Figure 9.23 signal the characteristic positive definite changes associated with the disappearance of the reservoir water.

See Canty et al. (2019) for additional examples as well as for a discussion of advantages and drawbacks of running the algorithm on the GEE platform.



ipyleaflet | © OpenStreetMap contributors, Tiles © Esri — Source: Esri, DeLorme, NAVTEQ, USGS, Intermap, iPC, NRCAN, Esri Japan, METI, Esri China (Hong Kong), Esri (Thailand), TomTom, 2012, Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community

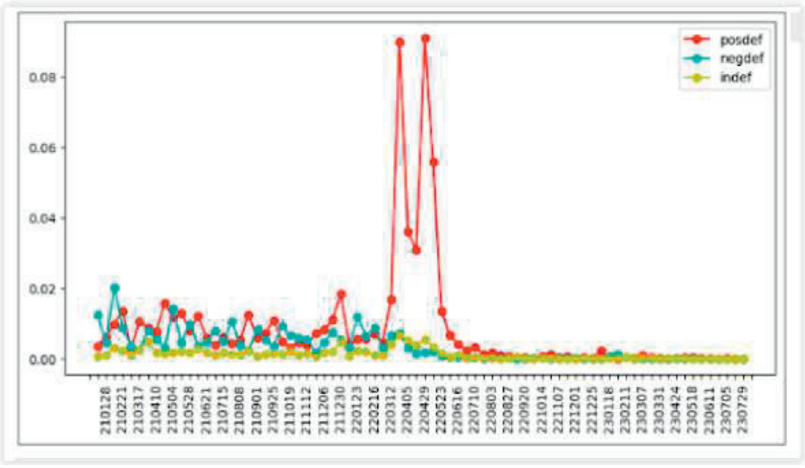


FIGURE 9.22
Top image: changes accumulated during the 3-month Russian siege at the Asovstal Works in Mariupol, Ukraine. Bottom image: the fraction of changed pixels at each (approx. two-week) time interval that occurred for the period January, 2021 through August, 2023.

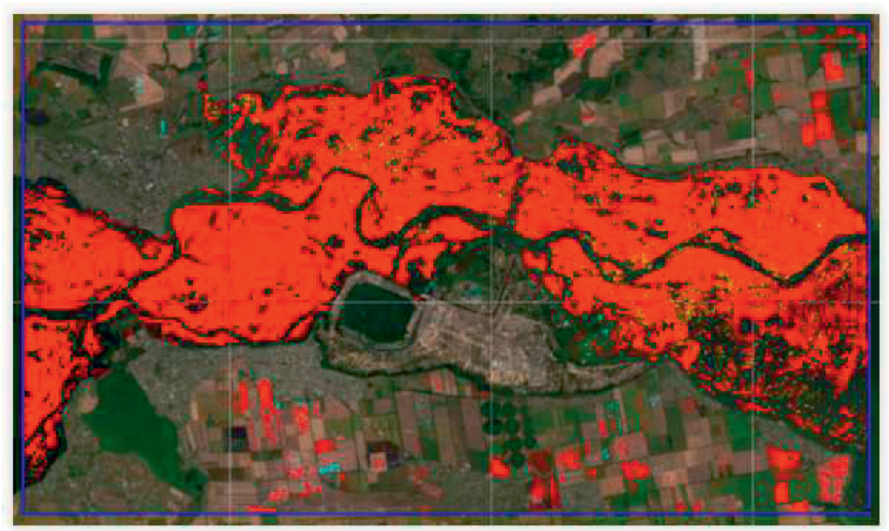


FIGURE 9.23
About 150 km upstream from the Kakhovka dam, the Kakhovka reservoir depletion first shows up clearly as positive definite (red) changes in the period June 8 to June 20, i.e., beginning two days after the dam breach. Note that in this image the Zaporizhzhia reactor cooling pond (near the image center) is unchanged and the Dnieper shows an arm still connecting to it.

9.5 Radiometric normalization of visual/infra-red images

Ground reflectance determination from satellite imagery requires, among other things, an atmospheric correction algorithm and the associated atmospheric properties at the time of image acquisition. For many historical satellite scenes, such data are not available and even for planned acquisitions they may be difficult to obtain. A relative normalization based on the radiometric information intrinsic to the images themselves is an alternative whenever absolute surface reflectances are not required.

In performing relative radiometric normalization, one usually makes the assumption that the relationship between the at-sensor radiances recorded at two different times from regions of constant reflectance can be approximated by linear functions. The critical aspect is the determination of suitable time-invariant features upon which to base the normalization (Schott et al., 1988; Yang and Lo, 2000; Du et al., 2002). We begin by illustrating this with the simple technique of scatter plot matching for red/near-infrared spectral bands. Then we go on to demonstrate how to take advantage of the linear invariance of the MAD transformation to perform fully automatic radiometric normalization across the visual/infrared spectrum.

9.5.1 Scatter plot matching

Maas and Rajan (2010) describe a method for radiometric normalization of the RED and NIR bands of multi-temporal LANDSAT TM and ETM+ images which exploits the characteristic shape of NIR vs. RED scatter plots (bands 4 and 3, respectively). The so-called *bare soil line* (BSL) and *full canopy point* (FCP) derived from the scatter plots of target and reference images are used as invariant features to re-scale the RED and NIR bands of the target to match those of the reference. The procedure works for other platforms too, for instance, with bands 2 and 3 of the ASTER VNIR sensor as illustrated in Figure 9.24. The reference and target scenes in the figure were acquired in different years (July 2001 and September 2005) and with different sensor gains.

With reference to Figure 9.25, one can derive the following linear transformation, which relates a point (x, y) in the target scatter plot to its transform $(\tilde{x}, \tilde{y})$ in the reference scatter plot (Exercise 10):

$$\begin{aligned} \tilde{x} &= X_R + (x - X_T)\frac{L_R}{L_T}\frac{a_T}{a_R} \\ \tilde{y} &= Y_R - (Y_T - y)\frac{L_R}{L_T}. \end{aligned} \tag{9.50}$$

The reference and target features can be extracted from the scatter plots by constructing a histogram of the intensity ratios NIR/RED and using the first percentile for the bare soil line and the 99.9th percentile for the full canopy point; see Canty (2014) for an IDL implementation.

9.5.2 Automatic radiometric normalization

As we have seen in Section 9.3.4, the MAD transformation is invariant under arbitrary linear transformations of the pixel intensities for the images involved. Thus, if one uses MAD for change detection applications, preprocessing by linear radiometric normalization is superfluous. However, radiometric normalization of imagery is important for many other applications, such as mosaicking, tracking vegetation indices over time, comparison of supervised and unsupervised land cover classifications, etc. Furthermore, if some other, noninvariant change detection procedure is preferred, it must generally be preceded by radiometric normalization. Taking advantage of invariance, one can apply the MAD transformation to select the no-change pixels in un-normalized bi-temporal images, and then use them for relative radiometric normalization. The procedure is simple, fast, and completely automatic and compares very favorably with normalization using hand-selected, time-invariant features (Canty et al., 2004; Schroeder et al., 2006; Canty and Nielsen, 2008). See also Philpot and Ansty (2013) for a physical interpretation of the invariant features detected by the MAD algorithm.

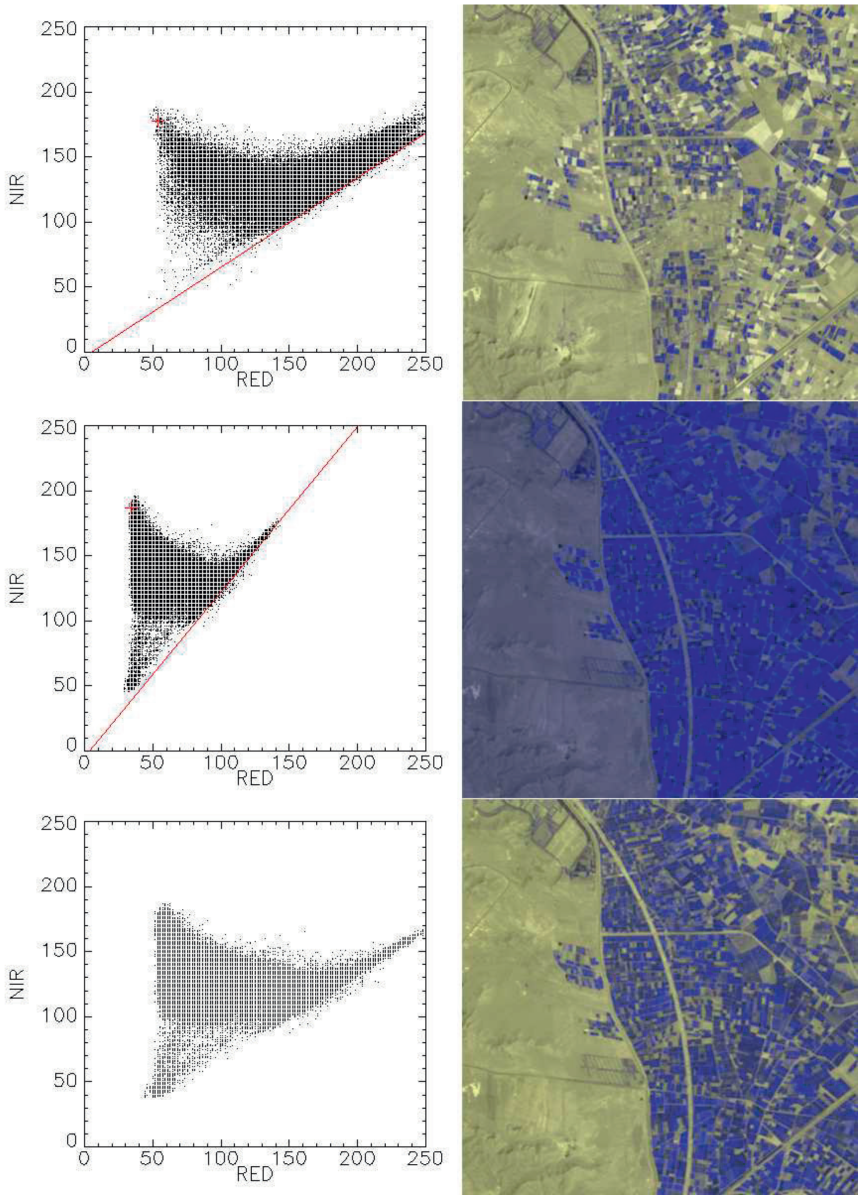


FIGURE 9.24
Scatter plot matching of two ASTER images over a region near Isfahan, Iran. Top row: RGB composite of the July 2001 reference image (bands 2,3,2 in a 0-255 byte linear stretch) along with the NIR vs. RED scatter plot showing the full canopy point (cross) and the bare soil line. Middle row: the September 2005 target image. Bottom row: the normalized target.

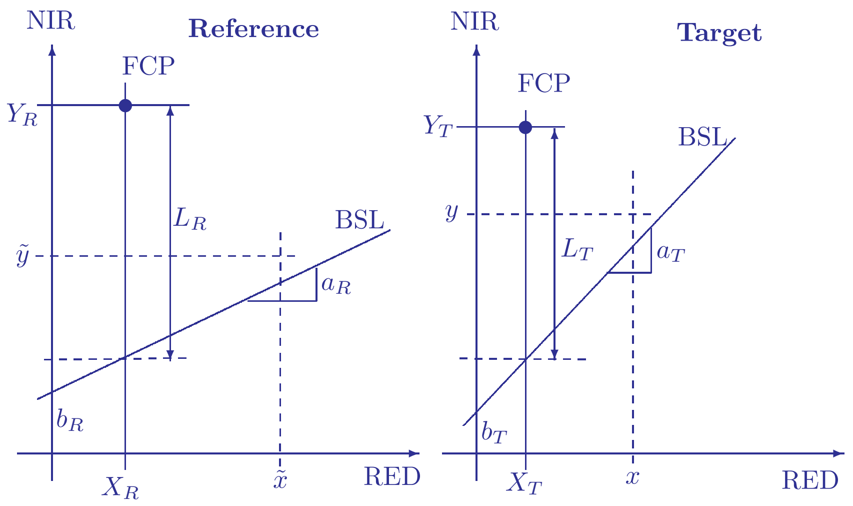


FIGURE 9.25
Principle of scatter plot matching. The coordinates of the full canopy point for reference and target are (X_R, Y_R) and (X_T, Y_T), respectively, and the respective slopes and intercepts of the bare soil line are a_R, b_R and a_T, b_T.

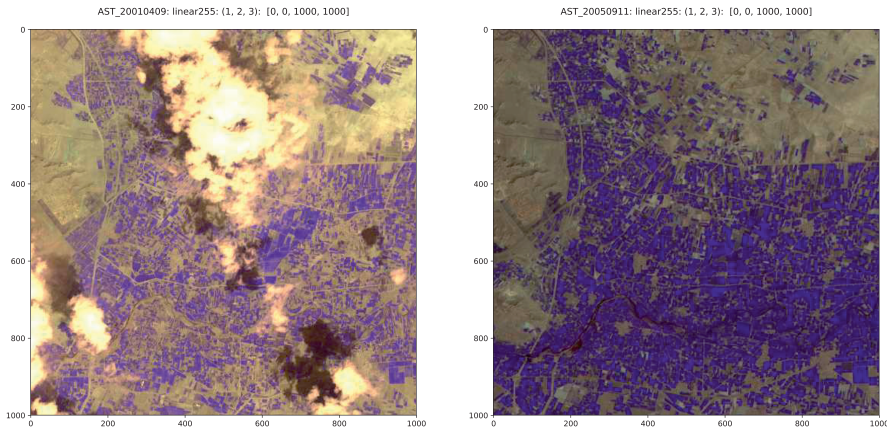

AST_20010409: linear255: (1, 2, 3): [0, 0, 1000, 1000]
AST_20050911: linear255: (1, 2, 3): [0, 0, 1000, 1000]

FIGURE 9.26
RGB composite ASTER images (VNIR bands 1,2 and 3) over an agricultural area near Isfahan in Iran. The image shown on the left was acquired on April 9, 2001, that on the right on September 11, 2005. No image enhancement is applied to the digital numbers.

A Python script `radcal.py` for automatic radiometric normalization is documented in Appendix C. The program reads the output from a previous iMAD transformation which has been performed on overlapping portions of the

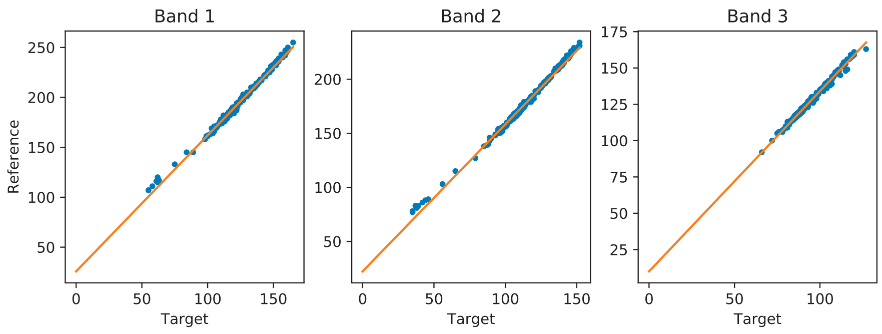


FIGURE 9.27
Jupyter notebook output cell from the script `radcal.py`.

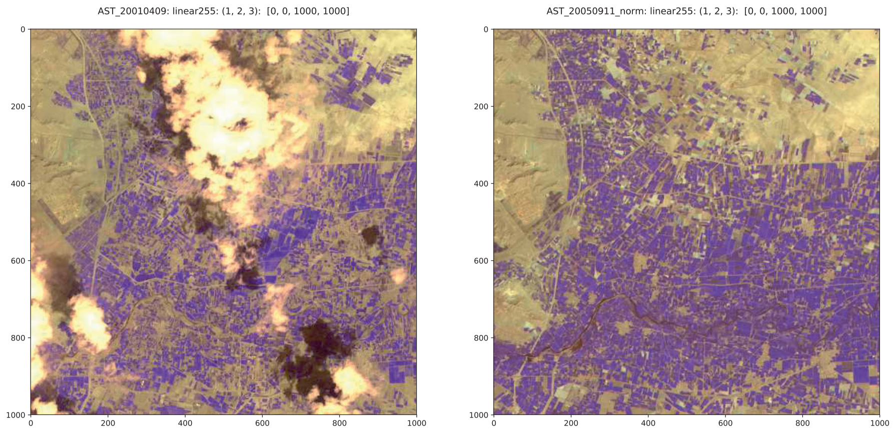

AST_20010409: linear255: (1, 2, 3): [0, 0, 1000, 1000]
AST_20050911_norm: linear255: (1, 2, 3): [0, 0, 1000, 1000]

FIGURE 9.28
As Figure 9.26 after radiometric normalization.

images to be normalized. Then Equations (9.19) and (9.20) are used to select pixels with a high P-value in order to ensure as little contamination with change observations as possible, typically ≥ 0.95. By regressing the reference image onto the target image at the no-change locations, slope and intercept parameters are obtained with which to perform the linear normalization. The preferred regression method is in this case *orthogonal linear regression*; see Appendix A, as both variables involved have similar uncertainties. Figure 9.26 shows ASTER VNIR images taken over the same area of Iran, near Isfahan. The September, 2005 image (target) is to be normalized to the April, 2001 scene (reference). The reference image has substantial cloud cover; however, this will play no role in the determination of the invariant pixels as the iMAD iteration procedure will eliminate cloud pixels completely:

```
run scripts/iMad -p [1,2,3] imagery/AST_20010409 \
                            imagery/AST_20050911

------------IRMAD -------------
Mon Oct 15 14:05:08 2018
first scene:  imagery/AST_20010409
second scene: imagery/AST_20050911
rho: [0.99867231 0.98997623 0.90816593]
result written to: imagery/MAD(AST_20010409-AST_20050911)
elapsed time: 29.2923579216

run scripts/radcal -p [1,2,3] \
                     imagery/MAD(AST_20010409-AST_20050911)
Mon Oct 15 14:10:08 2018
reference: imagery/AST_20010409
target   : imagery/AST_20050911
P-value threshold: 0.95
no-change pixels for training: 301, for testing: 151
band   slope   intercept correlation P(t-test) P(F-test)
 1   1.362209  25.738943  0.996609   0.793053   0.454460
 2   1.361127  22.251960  0.996928   0.845095   0.476366
 3   1.240756  9.984808   0.996258   0.862584   0.443266
result written to: imagery/AST_20050911_norm
elapsed time: 0.485175132751
```

The regression lines in the output cell from the `radcal.py` script are shown in Figure 9.27 and the images after normalization in Figure 9.28. In order to evaluate the normalization procedure the program holds back one third of the no-change pixels for testing purposes, 151 in this case. These are used to calculate means and variances before and after normalization and also to perform statistical hypothesis tests for equal means and variances of the invariant pixels in the reference and normalized target images. These tests were discussed in Section 2.5. As can be seen from the above output, the P-values for the t-test for equal means and for the F-test for equal variances indicate that the hypotheses of equality cannot be rejected for any of the spectral bands.

For more examples and an interactive tutorial on the use of the MAD transformation for radiometric normalization, see

https://developers.google.com/earth-engine/tutorials/community/imad-tutorial-pt3

9.6 Exercises

1. Demonstrate the validity of Equation (9.11) for the MAD transformation vector $\boldsymbol{b}$.

2. The requirement that the correlations of the canonical variates be positive, namely
$$\boldsymbol{a}_i^\top \boldsymbol{\Sigma}_{12}\boldsymbol{b}_i > 0, \quad i = 1\dots N,$$
does not completely remove the ambiguity in the signs of the transformation vectors $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$, since if we invert both their signs simultaneously, the condition is still met. The ambiguity can be resolved by requiring that the sum of the correlations of the first image (represented here by $\boldsymbol{G}$) with each of the canonical variates $U_j = \boldsymbol{a}_i^\top \boldsymbol{G}$, $j = 1\dots N$, be positive:
$$\sum_{\nu=1}^{N} \mathrm{corr}(G_i, U_j) > 0, \quad j = 1\dots N. \tag{9.51}$$
This condition is implemented in the Python script `imad.py`.

(a) Show that the matrix of correlations
$$\boldsymbol{C} = \begin{pmatrix} \mathrm{corr}(G_1,U_1) & \mathrm{corr}(G_1,U_2) & \cdots & \mathrm{corr}(G_1,U_N) \\ \mathrm{corr}(G_2,U_1) & \mathrm{corr}(G_2,U_2) & \cdots & \mathrm{corr}(G_2,U_N) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{corr}(G_N,U_1) & \mathrm{corr}(G_N,U_2) & \cdots & \mathrm{corr}(G_N,U_N) \end{pmatrix}$$
is given by
$$\boldsymbol{C} = \boldsymbol{D}\boldsymbol{\Sigma}_{11}\boldsymbol{A},$$
where $\boldsymbol{A} = (\boldsymbol{a}_1, \boldsymbol{a}_2 \dots \boldsymbol{a}_N)$, $\boldsymbol{\Sigma}_{11}$ is the covariance matrix for $\boldsymbol{G}$ and
$$\boldsymbol{D} = \begin{pmatrix} \frac{1}{\sqrt{\mathrm{var}(G_1)}} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sqrt{\mathrm{var}(G_2)}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sqrt{\mathrm{var}(G_N)}} \end{pmatrix}.$$

(b) Let $s_j = \sum_i C_{ij}$, $j = 1\dots N$, be the column sums of the correlation matrix. Show that Equation (9.51) is fulfilled by replacing $\boldsymbol{A}$ by $\boldsymbol{AS}$, where
$$\boldsymbol{S} = \begin{pmatrix} \frac{s_1}{|s_1|} & 0 & \cdots & 0 \\ 0 & \frac{s_2}{|s_2|} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{s_N}{|s_N|} \end{pmatrix}.$$

3. Consider the following experiment: The Python script `iMad.py` is used to generate MAD variates from two co-registered multispectral images. Then a principal components transformation of one of the images is performed and the MAD transformation is repeated. Will the MAD variates have changed? Why or why not?

4. The following code simulates no-change pixels by copying a spatial subset of one image to another, and adding some Gaussian noise.

```python
import numpy as np
from osgeo import gdal
from osgeo.gdalconst import GA_ReadOnly, GDT_Float32

im1 = 'imagery/AST_20010409'
im2 = 'imagery/AST_20050911'
im2_toy = 'imagery/AST_20050911_toy'
dim = 400
gdal.AllRegister()
inDataset1 = gdal.Open(im1,GA_ReadOnly)
inDataset2 = gdal.Open(im2,GA_ReadOnly)
cols = inDataset1.RasterXSize
rows = inDataset1.RasterYSize
bands = inDataset1.RasterCount
G1 = np.zeros((rows,cols,bands))
G2 = np.zeros((rows,cols,bands))
for k in range(bands):
    band = inDataset1.GetRasterBand(k+1)
    G1[:,:,k] = band.ReadAsArray(0,0,cols,rows)
    band = inDataset2.GetRasterBand(k+1)
    G2[:,:,k] = band.ReadAsArray(0,0,cols,rows)
G2[:dim,:dim,:] = G1[:dim,:dim,:] + \
               0.1*np.random.randn(dim,dim,bands)
driver = inDataset1.GetDriver()
outDataset = driver \
   .Create(im2_toy,cols,rows,bands,GDT_Float32)
for k in range(bands):
            outBand = outDataset.GetRasterBand(k+1)
            outBand.WriteArray(G2[:,:,k],0,0)
            outBand.FlushCache()
```

Ideally, the iteratively re-weighted MAD scheme should identify these pixels unambiguously. Experiment with with different subset sizes to see the extent to which this is the case, using both the `iMad.py` and `radcal.py` scripts.

5. (a) Show that the likelihood functions, Equations (9.29) and (9.30), are maximized by $\hat{x} = (g_1 + g_2)/2$ and by $x_1 = g1,\ x_2 = g_2$, respectively.

   (b) Prove that $\hat{x} = (1/j)\sum_j g_j$ maximizes the likelihood function L_0 given by Equation (9.32). *Hint:* Maximize $\ln(L_0)$ with respect to x.

6. Show that the exponential terms in Equations (9.32) and (9.33) are identical under the maximum-likelihood estimates for x and x_i, $i = 1 \dots j$.

7. (a) Derive the omnibus test statistic, Equation (9.35), for all alternatives to the no-change hypothesis. *Hint:* The likelihood function for k independent gamma distributed observations under the alternative hypothesis is

$$L_1(x_1 \dots x_k) = \prod_{i=1}^{k} p(g_i \mid x_i) = \frac{(\prod g_i)^{m-1} e^{-\sum g_i m/x_i}}{(\prod x_i/m)^m \Gamma(m)^k}.$$

   With maximum likelihood estimates $\hat{x}_i = g_i,\ i = 1 \dots k$, write down the expression for the maximum likelihood $L_1(\hat{x}_1 \dots \hat{x}_k)$. Under H_0 with $x_1 = \dots = x_k = x$ the likelihood is

$$L_0(x) = \frac{(\prod g_i)^{m-1} e^{-(m/x)\sum g_i}}{(x/m)^{km}\Gamma(m)^k}.$$

   Now determine the maximum likelihood under the null hypothesis by setting $\hat{x} = (1/k)\sum g_i$. Finally confirm the likelihood ratio statistic Equation (9.35) from $L_1(\hat{x}_1 \dots \hat{x}_k)/L_0(\hat{x})$.

   (b) Prove that this test statistic can be expressed as a product of the R_j statistics given by Equation (9.35). *Hint:* You can prove this by induction by noting that $R_2 = Q_2$ and, by direct calculation, $R_2R_3 = Q_2R_3 = Q_3$. Then show that $Q_{j-1}R_j = Q_j$.

8. Write down the critical region Equation (9.39) for the dual polarimetric case. What is the asymptotic distribution of the test statistic?

9. The omnibus change detection algorithm can be used as a simple SAR image edge detector simply by comparing an image with a copy of itself shifted by one row and one column and then computing the change map. Modify the Python script `sar_seq.py` to implement edge detection.

10. Derive Equations (9.50) from the geometry of Figure 9.25.

A

Mathematical Tools

A.1 Cholesky decomposition

Cholesky decomposition is used in some of the routines in this book to solve generalized eigenvalue problems associated with the maximum autocorrelation factor (MAF) and maximum noise fraction (MNF) transformations as well as with canonical correlation analysis. We sketch its justification in the following.

THEOREM A.1
If the $p \times p$ matrix $\boldsymbol{A}$ is symmetric positive definite and if the $p \times q$ matrix $\boldsymbol{B}$, where $q \le p$, has rank q, then $\boldsymbol{B}^\top \boldsymbol{A}\boldsymbol{B}$ is positive definite and symmetric.

Proof. Choose any q-dimensional vector $\boldsymbol{y} \ne \boldsymbol{0}$ and let $\boldsymbol{x} = \boldsymbol{B}\boldsymbol{y}$. We can write this as

$$\boldsymbol{x} = y_1\boldsymbol{b}_1 + \ldots + y_q\boldsymbol{b}_q,$$

where $\boldsymbol{b}_i$ is the ith column of $\boldsymbol{B}$. Since $\boldsymbol{B}$ has rank q, we conclude that $\boldsymbol{x} \ne \boldsymbol{0}$ as well, for otherwise the column vectors would be linearly dependent. But

$$\boldsymbol{y}^\top(\boldsymbol{B}^\top\boldsymbol{A}\boldsymbol{B})\boldsymbol{y} = (\boldsymbol{B}\boldsymbol{y})^\top\boldsymbol{A}(\boldsymbol{B}\boldsymbol{y}) = \boldsymbol{x}^\top\boldsymbol{A}\boldsymbol{x} > 0,$$

since $\boldsymbol{A}$ is positive definite. So $\boldsymbol{B}^\top\boldsymbol{A}\boldsymbol{B}$ is positive definite (and clearly symmetric). □

A square matrix $\boldsymbol{A}$ is *diagonal* if $a_{ij} = 0$ for $i \ne j$. It is *lower triangular* if $a_{ij} = 0$ for $i < j$ and *upper triangular* if $a_{ij} = 0$ for $i > j$. The product of two lower(upper) triangular matrices is lower(upper) triangular. The inverse of a lower(upper) triangular matrix is lower(upper) triangular. If $\boldsymbol{A}$ is diagonal and positive definite, then all of its diagonal elements are positive. Otherwise if, say, $a_{ii} \le 0$ then, for $\boldsymbol{x} = (0\ldots,1,\ldots0)^\top$ with the 1 at the ith position,

$$\boldsymbol{x}^\top\boldsymbol{A}\boldsymbol{x} \le 0$$

contradicting the fact that $\boldsymbol{A}$ is positive definite. Now we state without proof Theorem A.2.

DOI: 10.1201/9781003503286-A

THEOREM A.2
If $\boldsymbol{A}$ is nonsingular, then there exists a nonsingular lower triangular matrix $\boldsymbol{F}$ such that $\boldsymbol{FA}$ is non singular upper triangular.

The proof is straightforward, but somewhat lengthy; see e.g., Anderson (2003), Appendix A.

It follows directly that, if $\boldsymbol{A}$ is symmetric and positive definite, there exists a lower triangular matrix $\boldsymbol{F}$ such that $\boldsymbol{FAF}^\top$ is diagonal and positive definite. That is, from Theorem A.2, $\boldsymbol{FA}$ is upper triangular and non singular. But then, since $\boldsymbol{F}^\top$ is upper triangular, $\boldsymbol{FAF}^\top$ is also upper triangular. But it is clearly also symmetric, so it must be diagonal. Finally, since $\boldsymbol{F}$ is non singular it has full rank, so that $\boldsymbol{FAF}^\top$ is also positive definite by Theorem A.1.

One can now go one step further and claim that, if $\boldsymbol{A}$ is positive definite, then there exists a lower triangular matrix $\boldsymbol{G}$ such that $\boldsymbol{GAG}^\top = \boldsymbol{I}$. To show this, choose an $\boldsymbol{F}$ such that $\boldsymbol{FAF}^\top = \boldsymbol{D}$ is diagonal and positive definite. Let $\boldsymbol{D}'$ be the diagonal matrix whose diagonal elements are the positive square roots of the diagonal elements of $\boldsymbol{D}$. Choose $\boldsymbol{G} = \boldsymbol{D}'^{-1}\boldsymbol{F}$. Then $\boldsymbol{GAG}^\top = \boldsymbol{I}$.

THEOREM A.3
(Cholesky Decomposition) *If $\boldsymbol{A}$ is symmetric and positive definite, there exists a lower triangular matrix $\boldsymbol{L}$ such that $\boldsymbol{A} = \boldsymbol{LL}^\top$.*

Proof. Since there exists a lower triangular matrix $\boldsymbol{G}$ such that $\boldsymbol{GAG}^\top = \boldsymbol{I}$, we must have

$$\boldsymbol{A} = \boldsymbol{G}^{-1}(\boldsymbol{G}^\top)^{-1} = \boldsymbol{G}^{-1}(\boldsymbol{G}^{-1})^\top = \boldsymbol{LL}^\top,$$

where $\boldsymbol{L} = \boldsymbol{G}^{-1}$ is lower triangular. □

Cholesky decomposition on a positive definite symmetric matrix is analogous to finding the square root of a positive real number. The Python-Scipy function `scipy.linalg.cholesky(A,lower=True)` performs Cholesky decomposition of a matrix `A` returning `L`. For example:

```
import numpy as np
import scipy.linalg as linalg
A = np.array([[2,1],[1,3]])
L = np.mat(linalg.cholesky(A,lower=True))
print L*L.T
```

```
[[ 2.  1.]
 [ 1.  3.]]
```

A.2 Vector and inner product spaces

The real column vectors of dimension N, which were introduced in Chapter 1 to represent multispectral pixel intensities, provide the standard example of the more general concept of a *vector space*.

DEFINITION A.1 *A set S is a* (real) vector space *if the operations addition and scalar multiplication are defined on S so that, for $\boldsymbol{x}, \boldsymbol{y} \in S$ and $\alpha, \beta \in \mathbb{R}$,*

$$\begin{aligned} \boldsymbol{x}+\boldsymbol{y} &\in S \\ \alpha\boldsymbol{x} &\in S \\ 1\boldsymbol{x} &= \boldsymbol{x} \\ 0\boldsymbol{x} &= \boldsymbol{0} \\ \boldsymbol{x}+\boldsymbol{0} &= \boldsymbol{x} \\ \alpha(\boldsymbol{x}+\boldsymbol{y}) &= \alpha\boldsymbol{x}+\alpha\boldsymbol{y} \\ (\alpha+\beta)\boldsymbol{x} &= \alpha\boldsymbol{x}+\beta\boldsymbol{y}. \end{aligned}$$

The elements of S are called vectors*.*

Another example of a vector space satisfying the above definition is the set of continuous functions $f(x)$ on the real interval $[a, b]$, which is encountered in Chapter 3 in connection with the discrete wavelet transform. Unlike the column vectors of Chapter 1, the elements of this vector space have infinite dimension. Both vector spaces are *inner product spaces* according to Definition A.2.

DEFINITION A.2 *A vector space S is an* inner product space *if there is a function mapping two elements $\boldsymbol{x}, \boldsymbol{y} \in S$ to a real number $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ such that*

$$\begin{aligned} &\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \langle \boldsymbol{y}, \boldsymbol{x} \rangle \\ &\langle \boldsymbol{x}, \boldsymbol{x} \rangle \geq 0 \\ &\langle \boldsymbol{x}, \boldsymbol{x} \rangle = 0 \quad \textit{if and onl if} \quad \boldsymbol{x} = 0. \end{aligned}$$

In the case of the real column vectors, the inner product is defined by Equation (1.8), i.e., $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^\top \boldsymbol{y}$. For the vector space of continuous functions, we define

$$\langle f, g \rangle = \int_a^b f(x)g(x)dx.$$

DEFINITION A.3 *Two elements $\boldsymbol{x}$ and $\boldsymbol{y}$ of an inner product space S are said to be* orthogonal *if $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = 0$. The set of elements $\boldsymbol{x}_i$, $i = 1 \ldots n$ is* orthonormal *if $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle = \delta_{ij}$.*

A finite set S of linearly independent vectors (see Definition 1.2) constitutes a *basis* for the vector space V comprising all vectors that can be expressed as a linear combination of the vectors in S. The number of vectors in the basis is called the *dimension* of V. An orthogonal basis for a finite-dimensional inner product space can always be constructed by the *Gram–Schmidt orthogonalization procedure* (Press et al., 2002; Shawe-Taylor and Cristianini, 2004). If $\boldsymbol{v}_i$, $i = 1 \ldots N$, is an orthogonal basis for V, then for any $\boldsymbol{x} \in V$,

$$\boldsymbol{x} = \sum_{i=1}^{N} \frac{\langle \boldsymbol{x}, \boldsymbol{v}_i \rangle}{\langle \boldsymbol{v}_i, \boldsymbol{v}_i \rangle} \boldsymbol{v}_i.$$

Let W be a subset of vector space V. Then it will have an orthogonal basis $\{\boldsymbol{w}_1, \boldsymbol{w}_2 \ldots \boldsymbol{w}_K\}$. For any $\boldsymbol{x} \in V$, the *projection* $\boldsymbol{y}$ of $\boldsymbol{x}$ onto the subspace W is given by

$$\boldsymbol{y} = \sum_{i=1}^{K} \frac{\langle \boldsymbol{x}, \boldsymbol{w}_i \rangle}{\langle \boldsymbol{w}_i, \boldsymbol{w}_i \rangle} \boldsymbol{w}_i,$$

so that $\boldsymbol{y} \in W$. We define the *orthogonal complement* $W^{\perp}$ of W as the set

$$W^{\perp} = \{\boldsymbol{x} \in V \mid \langle \boldsymbol{x}, \boldsymbol{y} \rangle = 0 \text{ for all } \boldsymbol{y} \in W\}.$$

It is then easy to show that the *residual vector* $\boldsymbol{y}_{\perp} = \boldsymbol{x} - \boldsymbol{y}$ is in $W^{\perp}$, i.e., that $\langle \boldsymbol{y}_{\perp}, \boldsymbol{y} \rangle = 0$ for all $\boldsymbol{y} \in W$. Thus we can always write $\boldsymbol{x}$ as

$$\boldsymbol{x} = \boldsymbol{y} + \boldsymbol{y}_{\perp},$$

where $\boldsymbol{y} \in W$ and $\boldsymbol{y}_{\perp} \in W^{\perp}$.

THEOREM A.4
(Orthogonal Decomposition Theorem) *If W is a finite-dimensional subspace of an inner product space V, then any $\boldsymbol{x} \in V$ can be written uniquely as $\boldsymbol{x} = \boldsymbol{y} + \boldsymbol{y}_{\perp}$, where $\boldsymbol{y} \in W$ and $\boldsymbol{y}_{\perp} \in W^{\perp}$.*

A.3 Complex numbers, vectors, and matrices

A complex number is an expression of the form $z = a + \mathbf{i}b$, where $\mathbf{i}^2 = -1$. The *real part* of z is a and the *imaginary part* is b. The number z can be represented as a point or vector in the *complex plane* with the real part along the x-axis and the imaginary part along the y-axis. Complex number addition then corresponds to addition of two-dimensional vectors:

$$(a_1 + \mathbf{i}b_1) + (a_2 + \mathbf{i}b_2) = (a_1 + a_2) + \mathbf{i}(b_1 + b_2).$$

Multiplication is also straightforward, e.g.,

$$(a_1 + \mathbf{i}b_1)(a_2 + \mathbf{i}b_2) = a_1a_2 - b_1b_2 + \mathbf{i}(a_1b_2 - a_2b_1).$$

The *complex conjugate* of z is $z^* = a - \mathbf{i}b$. Thus

$$z^*z = (a - \mathbf{i}b)(a + \mathbf{i}b) = a^2 + b^2 = |z|^2,$$

where $|z| = \sqrt{a^2 + b^2}$ is the *magnitude* of the complex number z. If θ is the angle that z makes with the real axis, then

$$z = a + \mathbf{i}b = |z|\cos(\theta) + \mathbf{i}|z|\sin(\theta).$$

From the well-known *Euler's Theorem*, we can write this as

$$z = |z|e^{\mathbf{i}\theta}.$$

A *complex vector* is a vector of complex numbers,

$$\boldsymbol{z} = \begin{pmatrix} a_1 + \mathbf{i}b_1 \\ \vdots \\ a_N + \mathbf{i}b_N \end{pmatrix}.$$

Complex matrices similarly are matrices with complex elements. The operations of vector and matrix addition, multiplication, and scalar multiplication carry over straightforwardly from real vector spaces. The operation of transposition is replaced by the *conjugate transpose*

$$\boldsymbol{A}^\dagger = (\boldsymbol{A}^*)^\top.$$

For example, the inner product of two complex vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is

$$\boldsymbol{x}^\dagger\boldsymbol{y} = (x_1^* \dots x_N^*) \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} = x_1^*y_1 + \dots x_N^*y_N$$

and the length or Euclidean norm of $\boldsymbol{x}$ is

$$\|\boldsymbol{x}\| = \sqrt{\boldsymbol{x}^\dagger\boldsymbol{x}} = \sqrt{|x_1|^2 + \dots + |x_N|^2}.$$

The complex analog of a symmetric real matrix is the *Hermitian matrix*, with the property

$$\boldsymbol{A}^\dagger = \boldsymbol{A}.$$

A Hermitian matrix $\boldsymbol{A}$ is positive definite if $\boldsymbol{x}^\dagger\boldsymbol{A}\boldsymbol{x} > 0$ for all nonzero complex vectors $\boldsymbol{x}$. Like positive definite symmetric matrices, positive definite Hermitian matrices have real, positive eigenvalues. The matrix $\boldsymbol{A}$ is said to be *unitary* if $\boldsymbol{A}^\dagger = \boldsymbol{A}^{-1}$.

A.4 Least squares procedures

In this section, ordinary linear regression is extended to a recursive procedure for sequential data. This forms the basis of one of the neural network training algorithms derived in Appendix B. In addition, the orthogonal linear regression procedure used in Chapter 9 for radiometric normalization is explained.

A.4.1 Recursive linear regression

Consider the statistical model given by Equation (2.95), now in a slightly different notation:

$$Y(j) = \sum_{i=0}^{N} w_j x_i(j) + R(j), \quad j = 1 \dots \nu. \tag{A.1}$$

This model relates the independent variables $\boldsymbol{x}(j) = (1, x_1(j) \dots x_N(j))^\top$ to a measured quantity $Y(j)$ via the parameters $\boldsymbol{w} = (w_0, w_1 \dots w_N)^\top$. The index ν is now intended to represent the number of measurements that have been made *so far*. The random variables $R(j)$ represent the measurement uncertainty in the realizations $y(j)$ of $Y(j)$. We assume that they are uncorrelated and normally distributed with zero mean and unit variance ($\sigma^2 = 1$), whereas the values $\boldsymbol{x}(j)$ are exact. We wish to determine the best values for parameters $\boldsymbol{w}$. Equation (A.1) can be written in the terms of a data matrix $\boldsymbol{\mathcal{X}}_\nu$ as

$$\boldsymbol{Y}_\nu = \boldsymbol{\mathcal{X}}_\nu \boldsymbol{w} + \boldsymbol{R}_\nu, \tag{A.2}$$

where

$$\boldsymbol{\mathcal{X}}_\nu = \begin{pmatrix} \boldsymbol{x}(1)^\top \\ \vdots \\ \boldsymbol{x}(\nu)^\top \end{pmatrix},$$

$\boldsymbol{Y}_\nu = (Y(1) \dots Y(\nu))^\top$ and $\boldsymbol{R}_\nu = (R(1) \dots R(\nu))^\top$. As was shown in Chapter 2, the best solution in the least squares sense for the parameter vector $\boldsymbol{w}$ is given by

$$\boldsymbol{w}(\nu) = [(\boldsymbol{\mathcal{X}}_\nu^\top \boldsymbol{\mathcal{X}}_\nu)^{-1} \boldsymbol{\mathcal{X}}_\nu^\top] \boldsymbol{y}_\nu = \boldsymbol{\Sigma}(\nu) \boldsymbol{\mathcal{X}}_\nu^\top \boldsymbol{y}_\nu, \tag{A.3}$$

where the expression in square brackets is the pseudo inverse of $\boldsymbol{\mathcal{X}}_\nu$ and where $\boldsymbol{\Sigma}(\nu)$ is an estimate of the covariance matrix of $\boldsymbol{w}$,

$$\boldsymbol{\Sigma}(\nu) = (\boldsymbol{\mathcal{X}}_\nu^\top \boldsymbol{\mathcal{X}}_\nu)^{-1}. \tag{A.4}$$

Suppose a new observation $(\boldsymbol{x}(\nu+1), y(\nu+1))$ becomes available. Now we must solve the least squares problem

$$\begin{pmatrix} \boldsymbol{Y}_\nu \\ Y(\nu+1) \end{pmatrix} = \begin{pmatrix} \boldsymbol{\mathcal{X}}_\nu \\ \boldsymbol{x}(\nu+1)^\top \end{pmatrix} \boldsymbol{w} + \boldsymbol{R}_{\nu+1}. \tag{A.5}$$

With Equation (A.3), the solution is

$$\boldsymbol{w}(\nu+1) = \boldsymbol{\Sigma}(\nu+1) \begin{pmatrix} \boldsymbol{\mathcal{X}}_\nu \\ \boldsymbol{x}(\nu+1)^\top \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{y}_\nu \\ y(\nu+1) \end{pmatrix}. \tag{A.6}$$

Inverting Equation (A.4) with $\nu \to \nu+1$, we obtain a recursive formula for the new covariance matrix $\boldsymbol{\Sigma}(\nu+1)$:

$$\boldsymbol{\Sigma}(\nu+1)^{-1} = \begin{pmatrix} \boldsymbol{\mathcal{X}}_\nu \\ \boldsymbol{x}(\nu+1)^\top \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{\mathcal{X}}_\nu \\ \boldsymbol{x}(\nu+1)^\top \end{pmatrix} = \boldsymbol{\mathcal{X}}_\nu^\top \boldsymbol{\mathcal{X}}_\nu + \boldsymbol{x}(\nu+1)\boldsymbol{x}(\nu+1)^\top$$

or

$$\boldsymbol{\Sigma}(\nu+1)^{-1} = \boldsymbol{\Sigma}(\nu)^{-1} + \boldsymbol{x}(\nu+1)\boldsymbol{x}(\nu+1)^\top. \tag{A.7}$$

To obtain a similar recursive formula for $\boldsymbol{w}(\nu+1)$ we multiply Equation (A.6) out, giving

$$\boldsymbol{w}(\nu+1) = \boldsymbol{\Sigma}(\nu+1)(\boldsymbol{\mathcal{X}}_\nu^\top \boldsymbol{y}_\nu + \boldsymbol{x}(\nu+1)y(\nu+1)),$$

and replace $\boldsymbol{y}_\nu$ with $\boldsymbol{\mathcal{X}}_\nu \boldsymbol{w}(\nu)$ to obtain

$$\boldsymbol{w}(\nu+1) = \boldsymbol{\Sigma}(\nu+1)\Big(\boldsymbol{\mathcal{X}}_\nu^\top \boldsymbol{\mathcal{X}}_\nu \boldsymbol{w}(\nu) + \boldsymbol{x}(\nu+1)y(\nu+1)\Big).$$

Using Equations (A.4) and (A.7),

$$\begin{aligned} \boldsymbol{w}(\nu+1) &= \boldsymbol{\Sigma}(\nu+1)\Big(\boldsymbol{\Sigma}(\nu)^{-1}\boldsymbol{w}(\nu) + \boldsymbol{x}(\nu+1)y(\nu+1)\Big) \\ &= \boldsymbol{\Sigma}(\nu+1)\Big[\boldsymbol{\Sigma}(\nu+1)^{-1}\boldsymbol{w}(\nu) - \boldsymbol{x}(\nu+1)\boldsymbol{x}(\nu+1)^\top \boldsymbol{w}(\nu) + \boldsymbol{x}(\nu+1)y(\nu+1)\Big]. \end{aligned}$$

This simplifies to

$$\boldsymbol{w}(\nu+1) = \boldsymbol{w}(\nu) + \boldsymbol{K}(\nu+1)\Big[y(\nu+1) - \boldsymbol{x}(\nu+1)^\top \boldsymbol{w}(\nu)\Big], \tag{A.8}$$

where the *Kalman gain* $\boldsymbol{K}(\nu+1)$ is given by

$$\boldsymbol{K}(\nu+1) = \boldsymbol{\Sigma}(\nu+1)\boldsymbol{x}(\nu+1). \tag{A.9}$$

Equations (A.7–A.9) define a so-called *Kalman filter* for the least squares problem of Equation (A.1). For observations

$$\boldsymbol{x}(\nu+1) \quad \text{and} \quad y(\nu+1)$$

the *system response* $\boldsymbol{x}(\nu+1)^\top \boldsymbol{w}(\nu)$ is calculated and compared in Equation (A.8) with the measurement $y(\nu+1)$. Then the *innovation*, that is to say the difference between the measurement and system response, is multiplied by the Kalman gain $\boldsymbol{K}(\nu+1)$ determined by Equations (A.9) and (A.7) and the old estimate $\boldsymbol{w}(\nu)$ for the parameter vector $\boldsymbol{w}$ is corrected to the new value $\boldsymbol{w}(\nu+1)$.

Relation (A.7) is inconvenient as it calculates the inverse of the covariance matrix $\boldsymbol{\Sigma}(\nu+1)$, whereas we require the noninverted form in order to determine the Kalman gain in Equation (A.9). However, equations (A.7) and (A.9) can be reformed as follows:

$$\begin{aligned}\boldsymbol{\Sigma}(\nu+1) &= \Big[\boldsymbol{I} - \boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top\Big]\boldsymbol{\Sigma}(\nu)\\ \boldsymbol{K}(\nu+1) &= \boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1)\Big[\boldsymbol{x}(\nu+1)^\top\boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1)+1\Big]^{-1}.\end{aligned} \tag{A.10}$$

To see this, first of all note that the second equation above is a consequence of the first equation and Equation (A.9). Therefore, it suffices to show that the first equation is indeed the inverse of Equation (A.7):

$$\begin{aligned}&\boldsymbol{\Sigma}(\nu+1)\boldsymbol{\Sigma}(\nu+1)^{-1} = \Big[\boldsymbol{I} - \boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top\Big]\boldsymbol{\Sigma}(\nu)\boldsymbol{\Sigma}(\nu+1)^{-1}\\ &= \boldsymbol{I} - \boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top + \Big[\boldsymbol{I} - \boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top\Big]\boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1)\boldsymbol{x}(\nu+1)^\top\\ &= \boldsymbol{I} - \boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top + \boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1)\boldsymbol{x}(\nu+1)^\top\\ &\quad - \boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top\boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1)\boldsymbol{x}(\nu+1)^\top.\end{aligned}$$

The second equality above follows from Equation (A.7). But from the second of Equations (A.10), we have

$$\boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top\boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1) = \boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1) - \boldsymbol{K}(\nu+1)$$

and therefore

$$\begin{aligned}\boldsymbol{\Sigma}(\nu+1)\boldsymbol{\Sigma}(\nu+1)^{-1} = \boldsymbol{I} &- \boldsymbol{K}(\nu+1)\boldsymbol{x}(\nu+1)^\top + \boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1)\boldsymbol{x}(\nu+1)^\top\\ &- (\boldsymbol{\Sigma}(\nu)\boldsymbol{x}(\nu+1) - \boldsymbol{K}(\nu+1))\boldsymbol{x}(\nu+1)^\top = \boldsymbol{I}\end{aligned}$$

as required.

A.4.2 Orthogonal linear regression

In the model for ordinary linear regression described in Chapter 2, the independent variable x is assumed to be error-free. If we are regressing one spectral band against another, for example, then this is manifestly not the case. If we impose the model

$$Y(\nu) - R(\nu) = a + b(X(\nu) - S(\nu)), \quad i = 1\ldots m, \tag{A.11}$$

with $R(\nu)$ and $S(\nu)$ being uncorrelated, normally distributed random variables with mean zero and equal variances σ^2, then we might consider the analog of Equation (2.45) as a starting point:

$$z(a,b) = \sum_{\nu=1}^{m} \frac{(y(\nu) - a - bx(\nu))^2}{\sigma^2 + b^2\sigma^2}. \tag{A.12}$$

Finding the minimum of Equation (A.12) with respect to a and b is now more difficult because of the nonlinear dependence on b. Let us begin with the derivative with respect to a:

$$\frac{\partial z(a,b)}{\partial a} = 0 = -\frac{2}{\sigma^2(1+b^2)} \sum_{\nu} (y(\nu) - a - bx(\nu))$$

which leads to the estimate

$$\hat{a} = \bar{y} - b\bar{x}. \tag{A.13}$$

Differentiating with respect to b, we obtain

$$0 = \frac{2b}{1+b^2} \sum_{\nu} (y(\nu) - a - bx(\nu))^2 + 2\sum_{\nu} (y(\nu) - a - bx(\nu))x(\nu),$$

which simplifies to

$$\sum_{\nu} (y(\nu) - a - bx(\nu))[b(y(\nu) - a) + x(\nu)] = 0.$$

Now substitute $\hat{a}$ for a using Equation (A.13). This gives

$$\sum_{\nu} [y(\nu) - \bar{y} - b(x(\nu) - \bar{x})][b(y(\nu) - \bar{y} + b\bar{x}) + x(\nu)] = 0.$$

This equation is in fact only quadratic in b, since the cubic term is

$$-b^3\bar{x} \sum_{\nu} (x(\nu) - \bar{x}) = 0.$$

The quadratic term is, with the definition of s_{xy} given in Equation (2.90),

$$b^2 \sum_{\nu} \big((y(\nu) - \bar{y})\bar{x} - (x(\nu) - \bar{x})(y(\nu) - \bar{y}) - (x(\nu) - \bar{x})\bar{x} \big) = -mb^2 s_{xy}.$$

The linear term is, defining

$$s_{yy} = \frac{1}{m} \sum_{\nu=1}^{m} (y(\nu) - \bar{y})^2,$$

given by

$$b \sum_{\nu} (y(\nu) - \bar{y})^2 - (x(\nu) - \bar{x})x(\nu) = mb(s_{yy} - s_{xx}),$$

Listing A.1: Excerpt from the module `auxil.auxil1.py`.

```python
def orthoregress(x,y):
    Xm = np.mean(x)
    Ym = np.mean(y)
    s = np.cov(x,y)
    R = s[0,1]/math.sqrt(s[1,1]*s[0,0])
    lam,vs = np.linalg.eig(s)
    idx = np.argsort(lam)
    vs = vs[:,idx]       # increasing order, so
    b = vs[1,1]/vs[0,1]  # first pc is second column
    return [b,Ym-b*Xm,R]
```

because of the equality $\sum_\nu(x(\nu)-\bar x)x(\nu) = \sum_\nu(x(\nu)-\bar x)(x(\nu)-\bar x)$. Similarly, the constant term is

$$\sum_\nu(y(\nu)-\bar y)x(\nu) = \sum_\nu(y(\nu)-\bar y)(x(\nu)-\bar x) = ms_{xy}.$$

Thus b is a solution of the quadratic equation

$$b^2 s_{xy} + b(s_{xx}-s_{yy}) - s_{xy} = 0.$$

The solution (for positive slope) is

$$\hat b = \frac{(s_{yy}-s_{xx}) + \sqrt{(s_{yy}-s_{xx})^2 + 4s_{xy}^2}}{2s_{xy}}. \tag{A.14}$$

According to Patefield (1977) and Bilbo (1989), the variances in the regression parameters are given by

$$\begin{aligned}\sigma_a^2 &= \frac{\sigma^2\hat b(1+\hat b^2)}{ms_{xy}}\left(\bar x^2(1+\hat\tau) + \frac{s_{xy}}{\hat b}\right)\\ \sigma_b^2 &= \frac{\sigma^2\hat b(1+\hat b^2)}{ms_{xy}}\ (1+\hat\tau)\end{aligned} \tag{A.15}$$

with

$$\hat\tau = \frac{\sigma^2\hat b}{(1+\hat b^2)s_{xy}}. \tag{A.16}$$

If σ^2 is not known *a priori*, it can be estimated by (Kendall and Stuart, 1979)

$$\hat\sigma^2 = \frac{m}{(m-2)(1+\hat b^2)}(s_{yy} - 2\hat b s_{xy} + \hat b^2 s_{xx}). \tag{A.17}$$

It is easy to see that the estimate $\hat{b}$, Equation (A.14), can be calculated as the slope of the first principal component vector $\boldsymbol{u} = (u_1, u_2)^\top$ of the covariance matrix

$$\boldsymbol{s} = \begin{pmatrix} s_{xx} & s_{xy} \\ s_{yx} & s_{yy} \end{pmatrix},$$

that is,

$$\hat{b} = u_2/u_1.$$

Thus orthogonal linear regression on one independent variable is equivalent to principal components analysis. This is the basis for the Python procedure `orthoregress()` in the module `auxil.auxil1.py` shown in Listing A.1, which performs orthogonal regression on the input arrays `x` and `y`. The routine is used in some of the code documented in Appendix C.

A.5 Proof of Theorem 7.1

We need the following inequality:

$$\alpha^r \leq 1 - (1-\alpha)r, \tag{A.18}$$

which holds for any $\alpha \geq 0$ and $r \in [0,1]$. To see this, note that α^r is convex, that is, its second derivative is

$$\frac{d^2}{dr^2}(\alpha^r) = \alpha^r \log(\alpha)^2 \geq 0,$$

so we have the situation shown in Figure A.1.

Proof: (Freund and Shapire, 1997) Applying inequality (A.18) to step (e) in the AdaBoost algorithm (Section 7.3), we obtain

$$\begin{aligned}
\sum_{\nu=1}^m w_{i+1}(\nu) &= \sum_{\nu=1}^m w_i(\nu)\beta_i^{1-[[h_i(\nu)\neq k(\nu)]]} \\
&\leq \sum_{\nu=1}^m w_i(\nu)[1-(1-\beta_i)(1-[[h_i(\nu)\neq k(\nu)]])] \\
&= \sum_{\nu=1}^m w_i(\nu) - (1-\beta_i)\left[\sum_{\nu=1}^m w_i(\nu)(1-[[h_i(\nu)\neq k(\nu)]])\right].
\end{aligned}$$

From steps (a) and (c) in the algorithm, we can write

$$\sum_{\nu=1}^m w_i(\nu)[[h_i(\nu)\neq k(\nu)]] = \sum_{\nu=1}^m \sum_{\nu'=1}^m w_i(\nu')p_i(\nu)[[h_i(\nu)\neq k(\nu)]] = \sum_{\nu'=1}^m w_i(\nu')\epsilon_i.$$

Combining the last two equations then gives

$$\sum_{\nu=1}^{m} w_{i+1}(\nu) \le \sum_{\nu=1}^{m} w_i(\nu)[1-(1-\beta_i)(1-\epsilon_i)]. \tag{A.19}$$

If we apply this inequality successively for $i = 1 \dots N_c$, it follows that

$$\begin{aligned}\sum_{\nu=1}^{m} w_{N_c+1}(\nu) &\le \sum_{\nu=1}^{m} w_1(\nu)\prod_{i=1}^{m}[1-(1-\beta_i)(1-\epsilon_i)] \\ &= \prod_{i=1}^{m}[1-(1-\beta_i)(1-\epsilon_i)],\end{aligned} \tag{A.20}$$

since, in the algorithm, the initial weights sum to unity.

The final voting procedure in step (3) will make an error on training example ν if

$$\sum_{\{i|h_i(\nu)\neq k(\nu)\}} \log(1/\beta_i) \ \ge \sum_{\{i|h_i(\nu)=k(\nu)\}} \log(1/\beta_i).$$

Adding $\sum_{\{i|h_i(\nu)\neq k(\nu)\}} \log(1/\beta_i)$ to both sides,

$$2\left(\sum_{\{i|h_i(\nu)\neq k(\nu)\}} \log(1/\beta_i)\right) \ge \sum_{i=1}^{m} \log(1/\beta_i),$$

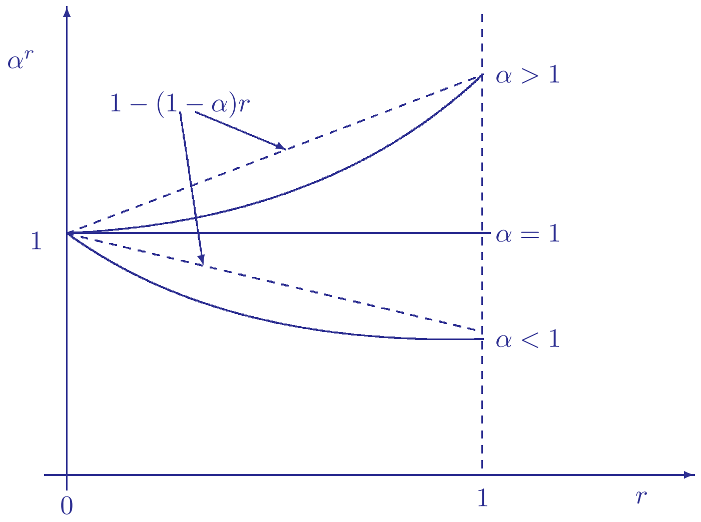


FIGURE A.1
Illustrating inequality (A.18).

or equivalently

$$-2\left(\sum_{\{i|h_i(\nu)\neq k(\nu)\}}\log(\beta_i)\right) \geq -\sum_{i=1}^{m}\log(\beta_i),$$

which we can write in the form

$$-\log\left(\prod_{\{i|h_i(\nu)\neq k(\nu)\}}\beta_i\right) \geq -\frac{1}{2}\log\left(\prod_{i=1}^{N_c}\beta_i\right)$$

or, since the logarithm is a monotonic increasing function of its argument, equivalently as

$$\prod_{i=1}^{N_c}\beta_i^{[[h_i(\nu)\neq k(\nu)]]} \geq \left(\prod_{i=1}^{N_c}\beta_i\right)^{-1/2}. \tag{A.21}$$

From step (e), we have further

$$w_{N_c+1} = w_1(\nu)\prod_{i=1}^{N_c}\beta_i^{1-[[h_i(\nu)\neq k(\nu)]]}. \tag{A.22}$$

Now let U be the set of incorrectly classified training examples for the sequence of N_c classifiers. Then

$$\begin{aligned}\sum_{\nu=1}^{m} w_{N_c+1}(\nu) &\geq \sum_{\nu\in U} w_{N_c+1}(\nu)\\ &= \sum_{\nu\in U} w_1(\nu)\prod_{i=1}^{N_c}\beta_i^{1-[[h_i(\nu)\neq k(\nu)]]} \quad \text{from } (A.22)\\ &= \sum_{\nu\in U} w_1(\nu)\prod_{i=1}^{N_c}\beta_i\prod_{i=1}^{N_c}\beta_i^{-[[h_i(\nu)\neq k(\nu)]]}\\ &\geq \sum_{\nu\in U} w_1(\nu)\left(\prod_{i=1}^{N_c}\beta_i\right)^{1/2} \quad \text{from } (A.21).\end{aligned}$$

But $\sum_{\nu\in U} w_1(\nu) = \epsilon$ and, combining this last inequality with Inequality (A.20), we have

$$\epsilon\left(\prod_{i=1}^{N_c}\beta_i\right)^{1/2} \leq \sum_{\nu=1}^{m} w_{N_c+1}(\nu) \leq \prod_{i=1}^{m}[1-(1-\beta_i)(1-\epsilon_i)]$$

or, solving for ϵ,

$$\epsilon \leq \prod_{i=1}^{N_c}\left(\frac{1-(1-\beta_i)(1-\epsilon_i)}{\sqrt{\beta_i}}\right). \tag{A.23}$$

Since each of the N_c factors is positive, the upper bound will be minimized when

$$\frac{d}{d\beta_i}\left(\frac{1-(1-\beta_i)(1-\epsilon_i)}{\sqrt{\beta_i}}\right)=0, \quad i=1\ldots N_c,$$

with solution

$$\beta_i=\frac{\epsilon_i}{1-\epsilon_i}.$$

Substituting this back into Equation (A.23) gives the upper bound Equation (7.26) and the proof is complete. $\square$

B

Neural Network Training Algorithms

The gradient descent back-propagation algorithm introduced in Chapter 6 is notoriously slow to converge. In this appendix, we will develop two additional training algorithms for the two-layer, feed-forward neural network of Figure 6.11. The first of these, the *scaled conjugate gradient*, makes use of the second derivatives of the cost function with respect to the synaptic weights, i.e., of the Hessian matrix. The second, the *extended Kalman filter* method, takes advantage of the statistical properties of the weight parameters themselves. Both techniques are considerably more efficient than back-propagation.

B.1 The Hessian matrix

We begin with a detailed discussion of the Hessian matrix and how to calculate it efficiently. The Hessian matrix $\boldsymbol{H}$ for a neural network training cost function $E(\boldsymbol{w})$ is given by

$$(\boldsymbol{H})_{ij} = \frac{\partial^2 E(\boldsymbol{w})}{\partial w_i \partial w_j}; \tag{B.1}$$

see Equation (1.56). It is the (symmetric) matrix of second-order partial derivatives of the cost function with respect to the synaptic weights, the latter being thought of as a single column vector

$$\boldsymbol{w} = \begin{pmatrix} \boldsymbol{w}_1^h \\ \vdots \\ \boldsymbol{w}_L^h \\ \boldsymbol{w}_1^o \\ \vdots \\ \boldsymbol{w}_K^o \end{pmatrix}$$

of length $n_w = L(N+1) + K(L+1)$ for the network architecture of Figure 6.11. Since $\boldsymbol{H}$ is symmetric, it is positive definite if and only if its eigenvalues are positive; see Section 1.4. Thus a good way to check if one is at or near a local minimum in the cost function is to examine the eigenvalues of the Hessian matrix.

The scaled conjugate gradient algorithm makes explicit use of the Hessian matrix for more efficient convergence to a minimum in the cost function. The disadvantage of using $\boldsymbol{H}$ is that it is difficult to compute. For example, for a typical classification problem with $(N = 3)$ - dimensional input data, $L = 8$ hidden neurons and $K = 12$ land use categories, there are

$$\big(L(N+1) + K(L+1)\big)\big(L(N+1) + K(L+1) + 1\big)/2 = 9870$$

matrix elements to determine at each iteration (allowing for symmetry). We develop, in the following, an efficient method (Bishop, 1995) not to calculate $\boldsymbol{H}$ directly, but rather the product $\boldsymbol{v}^\top \boldsymbol{H}$ for any vector $\boldsymbol{v}$ having n_w components.

B.1.1 The R-operator

Let us begin by summarizing some results from Chapter 6 for the two-layer, feed-forward network, changing the notation slightly to simplify what follows:

$\boldsymbol{g}' = (g'_1 \dots g'_N)^\top$	input observation vector
$\boldsymbol{g} = \begin{pmatrix} 1 \\ \boldsymbol{g}' \end{pmatrix}$	biased input observation
$\boldsymbol{\ell} = (0 \dots 1 \dots 0)^\top$	class label
$\boldsymbol{I}^h = \boldsymbol{W}^{h\top}\boldsymbol{g}$	activation vector for the hidden layer
$n'_j = f(I^h_j),\ j = 1 \dots L$	output signal vector from the hidden layer
$\boldsymbol{n} = \begin{pmatrix} 1 \\ \boldsymbol{n}' \end{pmatrix}$	biased output signal vector
$\boldsymbol{I}^o = \boldsymbol{W}^{o\top}\boldsymbol{n}$	activation vector for the output layer
$m_k(\boldsymbol{I}^o),\ k = 1 \dots K$	softmax output signal from kth output neuron.

(B.2)

The corresponding activation functions are, for the hidden neurons,

$$f(I^h_j) = \frac{1}{1 + e^{-I^h_j}}, \quad j = 1 \dots L, \tag{B.3}$$

and for the output neurons,

$$m_k(\boldsymbol{I}^o) = \frac{e^{I^o_k}}{\sum_{k'=1}^K e^{I^o_{k'}}}, \quad k = 1 \dots K. \tag{B.4}$$

The first derivatives of the local cross-entropy cost function, Equation (6.34), with respect to the output and hidden weights, Equations (6.36) and (6.40), can be written concisely as

$$\begin{aligned} \frac{\partial E}{\partial \boldsymbol{W}^o} &= -\boldsymbol{n}\boldsymbol{\delta}^{o\top} \\ \frac{\partial E}{\partial \boldsymbol{W}^h} &= -\boldsymbol{g}\boldsymbol{\delta}^{h\top}, \end{aligned} \tag{B.5}$$

where, see Equations (6.38) and (6.42),

$$\boldsymbol{\delta}^o = \boldsymbol{\ell} - \boldsymbol{m} \tag{B.6}$$

and

$$\begin{pmatrix} 0 \\ \boldsymbol{\delta}^h \end{pmatrix} = \boldsymbol{n} \cdot (1 - \boldsymbol{n}) \cdot \boldsymbol{W}^o \boldsymbol{\delta}^o. \tag{B.7}$$

(The dot denotes component-by-component multiplication.) Following Bishop (1995), we introduce the *R-operator* according to the definition

$$R_v\{\boldsymbol{x}\} := \boldsymbol{v}^\top \frac{\partial}{\partial \boldsymbol{w}}\, \boldsymbol{x}, \quad \boldsymbol{v}^\top = (v_1 \dots v_{n_w}).$$

We have

$$R_v\{\boldsymbol{w}\} = \boldsymbol{v}^\top \frac{\partial}{\partial \boldsymbol{w}}\, \boldsymbol{w} = \sum_j v_j \frac{\partial \boldsymbol{w}}{\partial w_j} = \boldsymbol{v}.$$

Note that we are now taking derivatives of vectors. This shouldn't confuse us. For example, the above result in two dimensions is

$$R_v\{\boldsymbol{w}\} = v_1 \frac{\partial}{\partial w_1}(w_1 \boldsymbol{i} + w_2 \boldsymbol{j}) + v_2 \frac{\partial}{\partial w_2}(w_1 \boldsymbol{i} + w_2 \boldsymbol{j}) = v_1 \boldsymbol{i} + v_2 \boldsymbol{j} = \boldsymbol{v}.$$

We adopt the convention that the result of applying the R-operator has the same structure as the argument to which it is applied. Thus, for example,

$$R_v\{\boldsymbol{W}^h\} = \boldsymbol{V}^h,$$

where $\boldsymbol{V}^h$, like $\boldsymbol{W}^h$, is an $(N+1) \times L$ matrix consisting of the first $(N+1)L$ components of the n_w-dimensional vector $\boldsymbol{v}$. Implicitly we set the last $n_w - (N+1)L$ components of $\boldsymbol{v}$ equal to zero.

Next we derive an expression for $\boldsymbol{v}^\top \boldsymbol{H}$ in terms of the R-operator.

$$(\boldsymbol{v}^\top \boldsymbol{H})_j = \sum_{i=1}^{n_w} v_i H_{ij} = \sum_{i=1}^{n_w} v_i \frac{\partial^2 E}{\partial w_i \partial w_j} = \sum_{i=1}^{n_w} v_i \frac{\partial}{\partial w_i} \left(\frac{\partial E}{\partial w_j} \right)$$

or

$$(\boldsymbol{v}^\top \boldsymbol{H})_j = \boldsymbol{v}^\top \frac{\partial}{\partial \boldsymbol{w}} \left(\frac{\partial E}{\partial w_j} \right) = R_v \left\{ \frac{\partial E}{\partial w_j} \right\}, \quad j = 1 \dots n_w.$$

Since $\boldsymbol{v}^\top \boldsymbol{H}$ is a row vector, this can be written

$$\boldsymbol{v}^\top \boldsymbol{H} = R_v \left\{ \frac{\partial E}{\partial \boldsymbol{w}^\top} \right\} \cong \left(R_v \left\{ \frac{\partial E}{\partial \boldsymbol{W}^h} \right\}, R_v \left\{ \frac{\partial E}{\partial \boldsymbol{W}^o} \right\} \right). \tag{B.8}$$

Note the reorganization of the structure in the argument of R_v, namely $\boldsymbol{w}^\top \to (\boldsymbol{W}^h, \boldsymbol{W}^o)$. This is merely for convenience of evaluation. Once the expressions on the right have been evaluated, the result must be "flattened" back to a row vector. Note also that Equation (B.8) is understood to involve the local cost function. In order to complete the calculation, we must sum over all training pairs; see Equation (6.33).

Applying R_v to Equations (B.5),

$$\begin{aligned} R_v\left\{\frac{\partial E}{\partial \boldsymbol{W}^o}\right\} &= -\boldsymbol{n} R_v\{\boldsymbol{\delta}^{o\top}\} - R_v\{\boldsymbol{n}\}\boldsymbol{\delta}^{o\top} \\ R_v\left\{\frac{\partial E}{\partial \boldsymbol{W}^h}\right\} &= -\boldsymbol{g} R_v\{\boldsymbol{\delta}^{h\top}\}, \end{aligned} \tag{B.9}$$

so that, in order to evaluate Equation (B.8), we need expressions for

$$R_v\{\boldsymbol{n}\},\ R_v\{\boldsymbol{\delta}^{o\top}\}\ \text{ and } R_v\{\boldsymbol{\delta}^{h\top}\}.$$

This is somewhat tedious, but well worth the effort.

B.1.1.1 Determination of $R_v\{\boldsymbol{n}\}$

From Equation (B.2), we can write

$$R_v\{\boldsymbol{n}\} = \begin{pmatrix} 0 \\ R_v\{\boldsymbol{n}'\} \end{pmatrix}, \tag{B.10}$$

and from the chain rule,

$$R_v\{\boldsymbol{n}'\} = \boldsymbol{n}' \cdot (1 - \boldsymbol{n}') \cdot R_v\{\boldsymbol{I}^h\}, \tag{B.11}$$

where, by differentiation of $\boldsymbol{I}^h$, we evaluate

$$R_v\{\boldsymbol{I}^h\} = \boldsymbol{V}^{h\top}\boldsymbol{g}. \tag{B.12}$$

Note that, according to our convention, $\boldsymbol{V}^{h\top}$ must be interpreted as an $L \times (N+1)$-dimensional matrix, since the argument $\boldsymbol{I}^h$ is a vector of length L and the result must have the same structure.

B.1.1.2 Determination of $R_v\{\boldsymbol{\delta}^o\}$

With Equations (B.2) and Equation (B.6) we get

$$R_v\{\boldsymbol{\delta}^o\} = -R_v\{\boldsymbol{m}\} = -\boldsymbol{v}^\top \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{w}} = -\boldsymbol{v}^\top \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{I}^o} \cdot \frac{\partial \boldsymbol{I}^o}{\partial \boldsymbol{w}} = -\frac{\partial \boldsymbol{m}}{\partial \boldsymbol{I}^o} \cdot R_v\{\boldsymbol{I}^o\},$$

But from Equation (B.4), it is easy to see that

$$\frac{\partial \boldsymbol{m}}{\partial \boldsymbol{I}^o} = \boldsymbol{m} \cdot (\boldsymbol{1} - \boldsymbol{m})$$

and therefore

$$R_v\{\boldsymbol{\delta}^o\} = -\boldsymbol{m} \cdot (\boldsymbol{1} - \boldsymbol{m}) \cdot R_v\{\boldsymbol{I}^o\}. \tag{B.13}$$

Similarly, with the expression for $\boldsymbol{I}^o$ in Equations (B.2), we get

$$R_v\{\boldsymbol{I}^o\} = \boldsymbol{W}^{o\top} R_v\{\boldsymbol{n}\} + \boldsymbol{V}^{o\top}\boldsymbol{n}, \tag{B.14}$$

where $R_v\{\boldsymbol{n}\}$ is given by Equations (B.10) to (B.12).

B.1.1.3 Determination of $R_v\{\boldsymbol{\delta}^h\}$

We begin by writing Equation (B.7) in the form

$$\begin{pmatrix} 0 \\ \boldsymbol{\delta}^h \end{pmatrix} = \begin{pmatrix} 0 \\ f'(\boldsymbol{I}^h) \end{pmatrix} \cdot \boldsymbol{W}^o \boldsymbol{\delta}^o,$$

where

$$f'(\boldsymbol{I}^h) = (f'(I_1^h) \dots f'(I_L^h))^\top$$

and where the prime on f denotes differentiation with respect to its argument, $f'(x) = df(x)/dx$. Operating with $R_v\{\cdot\}$ and applying the chain rule, we obtain

$$\begin{pmatrix} 0 \\ R_v\{\boldsymbol{\delta}^h\} \end{pmatrix} = \begin{pmatrix} 0 \\ f''(\boldsymbol{I}^h) \end{pmatrix} \cdot \begin{pmatrix} 0 \\ R_v\{\boldsymbol{I}^h\} \end{pmatrix} \cdot \boldsymbol{W}^o \boldsymbol{\delta}^o + \begin{pmatrix} 0 \\ f'(\boldsymbol{I}^h) \end{pmatrix} \cdot \boldsymbol{V}^o \boldsymbol{\delta}^o + \begin{pmatrix} 0 \\ f'(\boldsymbol{I}^h) \end{pmatrix} \cdot \boldsymbol{W}^o R_v\{\boldsymbol{\delta}^o\}. \tag{B.15}$$

Finally, substitute the derivatives of the logistic function

$$f'(\boldsymbol{I}^h) = \boldsymbol{n}'(\boldsymbol{1} - \boldsymbol{n}')$$
$$f''(\boldsymbol{I}^h) = \boldsymbol{n}'(\boldsymbol{1} - \boldsymbol{n}')(\boldsymbol{1} - 2\boldsymbol{n}')$$

into Equation (B.15) to obtain

$$\begin{pmatrix} 0 \\ R_v\{\boldsymbol{\delta}^h\} \end{pmatrix} = \boldsymbol{n} \cdot (\boldsymbol{1} - \boldsymbol{n}) \cdot \left[(\boldsymbol{1} - 2\boldsymbol{n}) \cdot \begin{pmatrix} 0 \\ R_v\{\boldsymbol{I}^h\} \end{pmatrix} \cdot \boldsymbol{W}^o \boldsymbol{\delta}^o + \boldsymbol{V}^o \boldsymbol{\delta}^o + \boldsymbol{W}^o R_v\{\boldsymbol{\delta}^o\} \right], \tag{B.16}$$

in which all of the terms on the right have now been determined. As already mentioned, we have done everything so far in terms of the local cost function. The final step in the calculation involves summing over all of the training examples. This concludes the evaluation of Equation (B.8).

B.1.2 Calculating the Hessian

To calculate the Hessian matrix for the neural network, we evaluate Equation (B.8) successively for the vectors

$$\boldsymbol{v}_1^\top = (1, 0, 0 \dots 0) \quad \dots \quad \boldsymbol{v}_{n_w}^\top = (0, 0, 0 \dots 1)$$

and build up $\boldsymbol{H}$ row for row:

$$\boldsymbol{H} = \begin{pmatrix} \boldsymbol{v}_1^\top \boldsymbol{H} \\ \vdots \\ \boldsymbol{v}_{n_w}^\top \boldsymbol{H} \end{pmatrix}.$$

The excerpt from the Python module `auxil.supervisedclass.py` shown in Listing B.1 implements a vectorized version of the above determination of $\boldsymbol{v}^\top \boldsymbol{H}$ (method `rop()`) and $\boldsymbol{H}$ (method `hessian()`).

Listing B.1: Calculation of the R-operator (excerpt from the module `auxil.-supervisedclass.py`).

```
    def hessian(self):
    #   Hessian of cross entropy wrt synaptic weights
        nw = self._L*(self._N+1)+self._K*(self._L+1)
        v = np.eye(nw,dtype=np.float)
        H = np.zeros((nw,nw))
        for i in range(nw):
            H[i,:] = self.rop(v[i,:])
        return H

    def rop(self,V):
    #   reshape V to dimensions of Wh and Wo, transpose
        VhT = np.reshape(V[:(self._N+1)*self._L],
                         (self._N+1,self._L)).T
        Vo = np.mat(np.reshape(V[self._L*(self._N+1)::],
                         (self._L+1,self._K)))
        VoT = Vo.T
    #   transpose the output weights
        Wo = self._Wo
        WoT = Wo.T
    #   forward pass
        M,n = self.vforwardpass(self._Gs)
    #   evaluation of v^T.H
        Z = np.zeros(self._m)
        D_o = self._ls - M                  #d^o
        RIh = VhT*self._Gs                  #Rv{I^h}
        tmp = np.vstack((Z,RIh))
        RN = n.A*(1-n.A)*tmp.A              #Rv{n}
        RIo = WoT*RN + VoT*n                #Rv{I^o}
        Rd_o = -np.mat(M*(1-M)*RIo.A)       #Rv{d^o}
        Rd_h = n.A*(1-n.A)*( (1-2*n.A)*tmp.A
                *(Wo*D_o).A + (Vo*D_o).A + (Wo*Rd_o).A)
        Rd_h = np.mat(Rd_h[1::,:])          #Rv{d^h}
        REo = -(n*Rd_o.T-RN*D_o.T).ravel() #Rv{dE/dWo}
        REh = -(self._Gs*Rd_h.T).ravel()   #Rv{dE/dWh}
        return np.hstack((REo,REh))         #v^T.H
```

B.2 Scaled conjugate gradient training

The back-propagation algorithm of Chapter 6 attempts to minimize the cost function *locally*, that is, weight updates are made immediately after presentation of a single training pair to the network. We will now consider a *global* approach aimed at minimization of the full cost function, Equation (6.33), which we denote in the following by $E(\boldsymbol{w})$. The symbol $\boldsymbol{w}$ is, as before, the n_w-component vector of synaptic weights.

Let the gradient of the cost function at the point $\boldsymbol{w}$ be $\mathbf{g}(\boldsymbol{w})$ (not to be confused with the observation vector $\boldsymbol{g}$), i.e.,

$$\mathbf{g}(\boldsymbol{w}) = \frac{\partial}{\partial \boldsymbol{w}} E(\boldsymbol{w}).$$

The Hessian matrix

$$(\boldsymbol{H})_{ij} = \frac{\partial^2 E(\boldsymbol{w})}{\partial w_i \partial w_j} \quad i, j = 1 \dots n_w$$

can then be expressed conveniently as the outer product

$$\boldsymbol{H} = \frac{\partial}{\partial \boldsymbol{w}} \mathbf{g}(\boldsymbol{w})^\top. \tag{B.17}$$

B.2.1 Conjugate directions

The search for a minimum in the cost function can be visualized as tracing out a series of points in the space of synaptic weight parameters,

$$\boldsymbol{w}^1, \boldsymbol{w}^2 \dots \boldsymbol{w}^{k-1}, \boldsymbol{w}^k, \boldsymbol{w}^{k+1} \dots \ ,$$

where the point $\boldsymbol{w}^k$ is determined by minimizing $E(\boldsymbol{w})$ along some *search direction* $\boldsymbol{d}^{k-1}$ which originated at the preceding point $\boldsymbol{w}^{k-1}$. This is illustrated in Figure B.1 and corresponds to the vector equation

$$\boldsymbol{w}^k = \boldsymbol{w}^{k-1} + \alpha_{k-1} \boldsymbol{d}^{k-1}. \tag{B.18}$$

Here $\boldsymbol{d}^{k-1}$ is a unit vector along the chosen search direction and the scalar α_{k-1} minimizes the cost function along that direction:

$$\alpha_{k-1} = \arg\min_{\alpha} E\big(\boldsymbol{w}^{k-1} + \alpha \boldsymbol{d}^{k-1}\big).$$

If, starting from $\boldsymbol{w}^k$, we now wish to take the next minimizing step in the weight space, it is not efficient simply to choose, as in back-propagation, the direction of the local gradient $\mathbf{g}(\boldsymbol{w}_k)$ at the new starting point $\boldsymbol{w}^k$. Since the cost function has been minimized along the direction $\boldsymbol{d}^{k-1}$ at the point $\boldsymbol{w}^{k-1} + \alpha_{k-1}\boldsymbol{d}^{k-1}$, its gradient along that direction is zero,

$$\mathbf{g}(\boldsymbol{w}^k)^\top \boldsymbol{d}^{k-1} = 0, \tag{B.19}$$

as indicated in Figure B.1. Since the algorithm has just succeeded in reducing the gradient of the cost function along $\boldsymbol{d}^{k-1}$ to zero, we prefer to choose the new search direction $\boldsymbol{d}^k$ so that the component of the gradient along the old

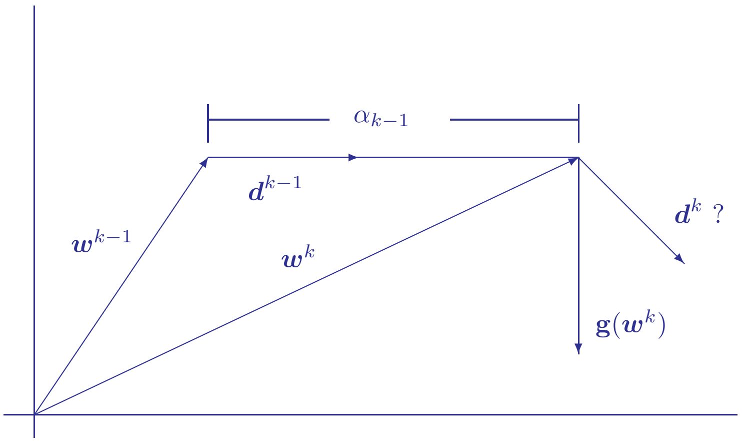


FIGURE B.1
Search directions in weight space.

search direction remains as small as possible. Otherwise, we are undoing what we have just accomplished. Therefore, we choose $\boldsymbol{d}^k$ according to the condition

$$\mathbf{g}\big(\boldsymbol{w}^k + \alpha \boldsymbol{d}^k\big)^\top \boldsymbol{d}^{k-1} = 0. \tag{B.20}$$

But to first order in α we have, with Equation (B.17),

$$\mathbf{g}\big(\boldsymbol{w}^k + \alpha \boldsymbol{d}^k\big)^\top = \mathbf{g}(\boldsymbol{w}^k)^\top + \alpha {\boldsymbol{d}^k}^\top \frac{\partial}{\partial \boldsymbol{w}} \mathbf{g}(\boldsymbol{w}^k)^\top = \mathbf{g}(\boldsymbol{w}^k)^\top + \alpha {\boldsymbol{d}^k}^\top \boldsymbol{H}$$

and Equation (B.20) is, together with Equation (B.19), equivalent to

$${\boldsymbol{d}^k}^\top \boldsymbol{H} \boldsymbol{d}^{k-1} = 0. \tag{B.21}$$

Directions which satisfy Equation (B.21) are referred to as *conjugate directions.*

B.2.2 Minimizing a quadratic function

Although the neural network cost function is not quadratic in the synaptic weights, within a sufficiently small region of weight space it can be approximated as a quadratic function. We describe in the following an efficient procedure to find the global minimum of a quadratic function of $\boldsymbol{w}$ having the general form (see Equation (1.55))

$$E(\boldsymbol{w}) = E_0 + \boldsymbol{b}^\top \boldsymbol{w} + \frac{1}{2} \boldsymbol{w}^\top \boldsymbol{H} \boldsymbol{w}, \tag{B.22}$$

where $\boldsymbol{b}$ and $\boldsymbol{H}$ are constant and the matrix $\boldsymbol{H}$ is positive definite. This will form the basis of the neural network training algorithm presented in the next subsection.

The local gradient of $E(\boldsymbol{w})$ at the point $\boldsymbol{w}$ is given by

$$\mathbf{g}(\boldsymbol{w}) = \frac{\partial}{\partial \boldsymbol{w}} E(\boldsymbol{w}) = \boldsymbol{b} + \boldsymbol{H}\boldsymbol{w},$$

and it vanishes at the global minimum $\boldsymbol{w}^*$,

$$\boldsymbol{b} + \boldsymbol{H}\boldsymbol{w}^* = \mathbf{0}. \tag{B.23}$$

Now let $\{\boldsymbol{d}^k \mid k = 1 \dots n_w\}$ be a set of n_w conjugate directions satisfying Equation (B.21),[*]

$$\boldsymbol{d}^{k\top} \boldsymbol{H} \boldsymbol{d}^{\ell} = 0 \quad \text{for } k \neq \ell, \quad k, \ell = 1 \dots n_w. \tag{B.24}$$

The search directions $\boldsymbol{d}^k$ are in fact linearly independent. In order to demonstrate this, let us assume the contrary, that is, that there exists an index k and constants $\alpha_{k'}$, $k' \neq k$, not all of which are zero, such that

$$\boldsymbol{d}^k = \sum_{\substack{k'=1 \\ k' \neq k}}^{n_w} \alpha_{k'} \boldsymbol{d}^{k'}.$$

Substituting this into Equation (B.24), we have at once

$$\alpha_{k'} \boldsymbol{d}^{k'\top} \boldsymbol{H} \boldsymbol{d}^{k'} = 0 \quad \text{for } k' \neq k$$

and, since $\boldsymbol{H}$ is positive definite,

$$\alpha_{k'} = 0 \quad \text{for } k' \neq k.$$

The assumption leads to a contradiction, hence the $\boldsymbol{d}^k$ are indeed linearly independent. The conjugate directions thus constitute a (non orthogonal) vector basis for the entire weight space.

In the search for the global minimum, suppose we begin at an arbitrary point $\boldsymbol{w}^1$ and express the vector $\boldsymbol{w}^* - \boldsymbol{w}^1$ spanning the distance to the global minimum as a linear combination of the basis vectors $\boldsymbol{d}^k$:

$$\boldsymbol{w}^* - \boldsymbol{w}^1 = \sum_{k=1}^{n_w} \alpha_k \boldsymbol{d}^k. \tag{B.25}$$

Further, define

$$\boldsymbol{w}^k = \boldsymbol{w}^1 + \sum_{\ell=1}^{k-1} \alpha_\ell \boldsymbol{d}^\ell \tag{B.26}$$

[*]It can be shown that such a set always exists; see, e.g., Bishop (1995).

and split Equation (B.25) up into n_w steps

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k + \alpha_k \boldsymbol{d}^k, \quad k = 1 \ldots n_w. \tag{B.27}$$

At the kth step, the search starts at the point $\boldsymbol{w}^k$ and proceeds a distance α_k along the conjugate direction $\boldsymbol{d}^k$. After n_w such steps, the global minimum $\boldsymbol{w}^*$ is reached, since from Equations (B.25) to (B.27) it follows that

$$\boldsymbol{w}^* = \boldsymbol{w}^1 + \sum_{k=1}^{n_w} \alpha_k \boldsymbol{d}^k = \boldsymbol{w}^2 + \sum_{k=2}^{n_w} \alpha_k \boldsymbol{d}^k = \ldots = \boldsymbol{w}^{n_w} + \alpha_{n_w} \boldsymbol{d}^{n_w} = \boldsymbol{w}^{n_w+1}.$$

We get the necessary step sizes α_k from Equation (B.25) by multiplying from the left with ${\boldsymbol{d}^\ell}^\top \boldsymbol{H}$,

$${\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{w}^* - {\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{w}^1 = \sum_{k=1}^{n_w} \alpha_k {\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{d}^k.$$

From Equations (B.23) and (B.24) we can write this as

$$-{\boldsymbol{d}^\ell}^\top (\boldsymbol{b} + \boldsymbol{H}\boldsymbol{w}^1) = \alpha_\ell {\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{d}^\ell,$$

so that an explicit formula for the step sizes is given by

$$\alpha_\ell = -\frac{{\boldsymbol{d}^\ell}^\top (\boldsymbol{b} + \boldsymbol{H}\boldsymbol{w}^1)}{{\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{d}^\ell}, \quad \ell = 1 \ldots n_w.$$

But with Equations (B.24) and (B.26),

$${\boldsymbol{d}^k}^\top \boldsymbol{H}\boldsymbol{w}^k = {\boldsymbol{d}^k}^\top \boldsymbol{H}\boldsymbol{w}^1 + 0,$$

and therefore, replacing index k by ℓ,

$${\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{w}^\ell = {\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{w}^1.$$

The step lengths are thus

$$\alpha_\ell = -\frac{{\boldsymbol{d}^\ell}^\top (\boldsymbol{b} + \boldsymbol{H}\boldsymbol{w}^\ell)}{{\boldsymbol{d}^\ell}^\top \boldsymbol{H}\boldsymbol{d}^\ell}, \quad \ell = 1 \ldots n_w.$$

Finally, using the notation $\mathbf{g}^\ell = \mathbf{g}(\boldsymbol{w}^\ell) = \boldsymbol{b} + \boldsymbol{H}\boldsymbol{w}^\ell$ and substituting $\ell \to k$,

$$\alpha_k = -\frac{{\boldsymbol{d}^k}^\top \mathbf{g}^k}{{\boldsymbol{d}^k}^\top \boldsymbol{H}\boldsymbol{d}^k}, \quad k = 1 \ldots n_w. \tag{B.28}$$

For want of a better alternative, we can choose the first search direction along the negative local gradient

$$\boldsymbol{d}^1 = -\mathbf{g}^1 = -\frac{\partial}{\partial \boldsymbol{w}} E(\boldsymbol{w}^1).$$

(Note that $\boldsymbol{d}^1$ is not a unit vector.) We move, according to Equation (B.28), a distance

$$\alpha_1 = \frac{\boldsymbol{d}^{1\top}\boldsymbol{d}^1}{\boldsymbol{d}^{1\top}\boldsymbol{H}\boldsymbol{d}^1}$$

along this direction to the point $\boldsymbol{w}^2$, at which the local gradient $\mathbf{g}^2$ is orthogonal to $\boldsymbol{d}^1$. We then choose the new conjugate search direction $\boldsymbol{d}^2$ as a linear combination of the two:

$$\boldsymbol{d}^2 = -\mathbf{g}^2 + \beta_1\boldsymbol{d}^1$$

or, at the kth step,

$$\boldsymbol{d}^{k+1} = -\mathbf{g}^{k+1} + \beta_k\boldsymbol{d}^k. \tag{B.29}$$

We get the coefficient β_k from Equations (B.29) and (B.21) by multiplication on the left with $\boldsymbol{d}^{k\top}\boldsymbol{H}$:

$$0 = -\boldsymbol{d}^{k\top}\boldsymbol{H}\mathbf{g}^{k+1} + \beta_k\boldsymbol{d}^{k\top}\boldsymbol{H}\boldsymbol{d}^k,$$

from which follows

$$\beta_k = \frac{\mathbf{g}^{k+1\top}\boldsymbol{H}\boldsymbol{d}^k}{\boldsymbol{d}^{k\top}\boldsymbol{H}\boldsymbol{d}^k}. \tag{B.30}$$

Equations (B.27 to B.30) constitute a recipe with which, starting at an arbitrary point $\boldsymbol{w}^1$ in weight space, the global minimum of the quadratic function, Equation (B.22), is found in precisely n_w steps.

B.2.3 The algorithm

Returning now to the non quadratic neural net cost function $E(\boldsymbol{w})$, we will apply the above method to minimize it. We must take two things into consideration.

First of all, the Hessian matrix $\boldsymbol{H}$ is neither constant nor everywhere positive definite. When $\boldsymbol{H}$ is not positive definite, it can happen that Equation (B.28) leads to a step along the wrong direction—the denominator might turn out to be negative. Therefore, we replace Equation (B.28) with*

$$\alpha_k = -\frac{\boldsymbol{d}^{k\top}\mathbf{g}^k}{\boldsymbol{d}^{k\top}\boldsymbol{H}\boldsymbol{d}^k + \lambda_k\|\boldsymbol{d}^k\|^2}, \quad k = 1\ldots n_w. \tag{B.31}$$

The constant λ_k is supposed to ensure that the denominator in Equation (B.31) is always positive. It is initialized for $k = 1$ with a small numerical value. If, at the kth iteration, it is determined that

$$\delta_k := \boldsymbol{d}^{k\top}\boldsymbol{H}\boldsymbol{d}^k + \lambda_k\|\boldsymbol{d}^k\|^2 < 0,$$

*This corresponds to the substitution $\boldsymbol{H} \rightarrow \boldsymbol{H} + \lambda_k\boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix.

then λ_k is replaced by the larger value $\bar{\lambda}_k$ given by

$$\bar{\lambda}_k = 2\left(\lambda_k - \frac{\delta_k}{\|\boldsymbol{d}^k\|^2}\right). \tag{B.32}$$

This ensures that the denominator in Equation (B.31) becomes positive again. Note that this increase in λ_k has the effect of *decreasing* the step size α_k, as is apparent from Equation (B.31).

Second, we must take into account any deviation of the cost function from its local quadratic approximation. Such deviations are to be expected for large step sizes α_k. As a measure of the *quadricity* of $E(\boldsymbol{w})$ along the chosen step length, we can use the ratio

$$\Delta_k = -\frac{2\big(E(\boldsymbol{w}^k) - E(\boldsymbol{w}^k + \alpha_k \boldsymbol{d}^k)\big)}{\alpha_k {\boldsymbol{d}^k}^\top \mathbf{g}^k}. \tag{B.33}$$

This quantity is precisely 1 for a strictly quadratic function like Equation (B.22). Therefore we can use the following heuristic: For the $k+1$st iteration

$$\begin{array}{rl} \text{if } \Delta_k > 3/4, & \lambda_{k+1} := \lambda_k/2 \\ \text{if } \Delta_k < 1/4, & \lambda_{k+1} := 4\lambda_k \\ \text{else,} & \lambda_{k+1} := \lambda_k. \end{array}$$

In other words, if the local quadratic approximation looks good according to criterion of Equation (B.33), then the step size can be increased (λ_{k+1} is reduced relative to λ_k). If this is not the case, then the step size is decreased (λ_{k+1} is made larger).

All of which leads us, at last, to the following algorithm (Moeller, 1993):

Algorithm (Scaled Conjugate Gradient)

1. Initialize the synaptic weights $\boldsymbol{w}$ with random numbers, set $k = 0$, $\lambda = 0.001$ and $\boldsymbol{d} = -\mathbf{g} = -\partial E(\boldsymbol{w})/\partial \boldsymbol{w}$.

2. Set $\delta = \boldsymbol{d}^\top \boldsymbol{H}\boldsymbol{d} + \lambda\|\boldsymbol{d}\|^2$. If $\delta < 0$, set $\lambda = 2(\lambda - \delta/\|\boldsymbol{d}\|^2)$ and $\delta = -\boldsymbol{d}^\top \boldsymbol{H}\boldsymbol{d}$. Save the current cost function $E1 = E(\boldsymbol{w})$.

3. Determine the step size $\alpha = -\boldsymbol{d}^\top \mathbf{g}/\delta$ and new synaptic weights $\boldsymbol{w} = \boldsymbol{w} + \alpha\boldsymbol{d}$.

4. Calculate the quadricity $\Delta = -(E1 - E(\boldsymbol{w}))/(\alpha\boldsymbol{d}^\top \mathbf{g})$. If $\Delta < 1/4$, restore the old weights: $\boldsymbol{w} = \boldsymbol{w} - \alpha\boldsymbol{d}$, set $\lambda = 4\lambda$, $\boldsymbol{d} = -\mathbf{g}$ and go to 2.

5. Set $k = k + 1$. If $\Delta > 3/4$, set $\lambda = \lambda/2$.

6. Determine the new local gradient $\mathbf{g} = \partial E(\boldsymbol{w})/\partial \boldsymbol{w}$ and the new search direction $\boldsymbol{d} = -\mathbf{g} + \beta\boldsymbol{d}$, whereby, if $k \bmod n_w \neq 0$ then $\beta = \mathbf{g}^\top \boldsymbol{H}\boldsymbol{d}/(\boldsymbol{d}^\top \boldsymbol{H}\boldsymbol{d})$ else $\beta = 0$.

Listing B.2: Scaled conjugate gradient training (excerpt from the module `auxil.supervisedclass.py`).

```
    def train(self):
        try:
            cost = []
            costv = []
            w = np.concatenate((self._Wh.A.ravel(),
                                self._Wo.A.ravel()))
            nw = len(w)
            g = self.gradient()
            d = -g
            k = 0
            lam = 0.001
            while k < self._epochs:
                d2=np.sum(d*d)              # d^2
                dTHd=np.sum(self.rop(d).A*d)# d^T.H.d
                delta = dTHd + lam*d2
                if delta < 0:
                    lam = 2*(lam-delta/d2)
                    delta = -dTHd
                E1 = self.cost()            # E(w)
                dTg = np.sum(d*g)           # d^T.g
                alpha = -dTg/delta
                dw = alpha*d
                w += dw
                self._Wh = np.mat(np.reshape(
                       w[0:self._L*(self._N+1)],
                       (self._N+1,self._L)))
                self._Wo = np.mat(np.reshape(
                       w[self._L*(self._N+1)::],
                       (self._L+1,self._K)))
                E2 = self.cost()           # E(w+dw)
                Ddelta = -2*(E1-E2)/(alpha*dTg)
```

7. If $E(\boldsymbol{w})$ is small enough, stop, else go to 2.

A few remarks on this algorithm:

1. The integer k counts the total number of iterations. Whenever $k \bmod n_w = 0$, then exactly n_w weight updates have been carried out and the minimum of a truly quadratic function would have been reached. This is taken as a good stage at which to restart the search along the negative local gradient $-\boldsymbol{g}$ rather than continuing along the current conjugate direction $\boldsymbol{d}$. One expects that approximation errors will gradually corrupt the determination of the conjugate directions and the "fresh start" is intended to counter this.

Listing B.3: Scaled conjugate gradient training (continued).

```
                if Ddelta < 0.25:
                    w -= dw                # undo
                    self._Wh = np.mat(np.reshape(
                        w[0:self._L*(self._N+1)],
                        (self._N+1,self._L)))
                    self._Wo = np.mat(np.reshape(
                        w[self._L*(self._N+1)::],
                        (self._L+1,self._K)))
                    lam *= 4.0       # decrease step
                    if lam > 1e20:   # step too small
                        k = self._epochs  # give up
                    else:            # else
                        d = -g       # restart
                else:
                    k += 1
                    cost.append(E1)
                    costv.append(self.costv())
                    if Ddelta > 0.75:
                        lam /= 2.0
                    g = self.gradient()
                    if k % nw == 0:
                        beta = 0.0
                    else:
                        beta = np.sum(
                            self.rop(g).A*d)/dTHd
                    d = beta*d - g
            return (cost,costv)
        except Exception as e:
            print ('Error: %s'%e)
            return None
```

2. Whenever the quadricity condition is not filled, i.e., whenever $\Delta < 1/4$, the last weight update is canceled and the search again restarted along $-\mathbf{g}$.

3. Since the Hessian only occurs in the forms $\boldsymbol{d}^\top \boldsymbol{H}$, and $\mathbf{g}^\top \boldsymbol{H}$, these quantities can be determined efficiently with the R-operator method.

Listings B.2 and B.3 show the training method in the Python module `auxil.supervisedclass.py` which implements the scaled conjugate gradient algorithm.

B.3 Extended Kalman filter training

In this section the recursive linear regression method described in Appendix A will be applied to train the feed-forward neural network of Figure 6.10. The appropriate cost function in this case is the quadratic function, Equation (6.30), or more specifically, its local version

$$E(\nu) = \frac{1}{2}\|\boldsymbol{\ell}(\nu) - \boldsymbol{m}(\nu)\|^2; \tag{B.34}$$

however, our algorithm will also minimize the cross-entropy cost function, as will be mentioned later.

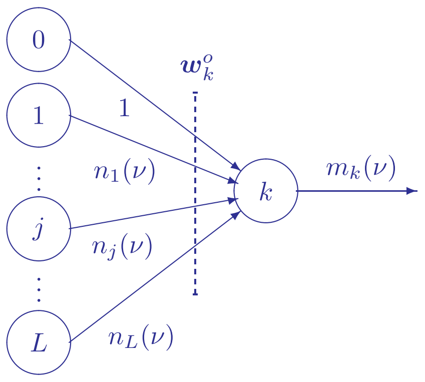


FIGURE B.2
An isolated output neuron.

We begin with consideration of the training process of an isolated neuron. Figure B.2 depicts an output neuron in the network during presentation of the νth training pair $(\boldsymbol{g}(\nu), \boldsymbol{\ell}(\nu))$. The neuron receives its input from the hidden layer (input vector $\boldsymbol{n}(\nu)$) and generates the softmax output signal

$$m_k(\nu) = \frac{e^{\boldsymbol{w}_k^{o\top}(\nu)\boldsymbol{n}(\nu)}}{\sum_{k'=1}^K e^{\boldsymbol{w}_{k'}^{o\top}(\nu)\boldsymbol{n}(\nu)}},$$

for $k = 1 \ldots K$, which is compared to the desired output $\ell_k(\nu)$. It is easy to show that differentiation of m_k with respect to $\boldsymbol{w}_k^o$ yields

$$\frac{\partial}{\partial \boldsymbol{w}_k^o} m_k(\nu) = m_k(\nu)(1 - m_k(\nu))\boldsymbol{n}(\nu) \tag{B.35}$$

and with respect to $\boldsymbol{n}$,

$$\frac{\partial}{\partial \boldsymbol{n}} m_k(\nu) = m_k(\nu)(1 - m_k(\nu))\boldsymbol{w}_k^o(\nu). \tag{B.36}$$

B.3.1 Linearization

We shall drop, for the time being, the indices on $\boldsymbol{w}_k^o$, m_k, and ℓ_k, writing them simply as $\boldsymbol{w}$, m and ℓ. The weight vectors for the other $K-1$ output neurons are considered to be frozen, so that m can be thought of as being a function of $\boldsymbol{w}$ only:

$$m(\nu) = m(\boldsymbol{w}(\nu)^\top \boldsymbol{n}(\nu)).$$

The weight vector $\boldsymbol{w}(\nu)$ is an approximation to the desired synaptic weight vector for our isolated output neuron, one which has been achieved so far in the training process, after presentation of the first ν labeled training observations. A linear approximation to $m(\nu+1)$ can be obtained by expanding in a first-order Taylor series about the point $\boldsymbol{w}(\nu)$,

$$m(\nu+1) \approx m(\boldsymbol{w}(\nu)^\top \boldsymbol{n}(\nu+1)) + \left(\frac{\partial}{\partial \boldsymbol{w}} m(\boldsymbol{w}(\nu)^\top \boldsymbol{n}(\nu+1))\right)^\top (\boldsymbol{w} - \boldsymbol{w}(\nu)).$$

From Equation (B.35), we then have

$$m(\nu+1) \approx \hat{m}(\nu+1) + \hat{m}(\nu+1)(1-\hat{m}(\nu+1))\boldsymbol{n}(\nu+1)^\top(\boldsymbol{w} - \boldsymbol{w}(\nu)), \quad \text{(B.37)}$$

where $\hat{m}(\nu+1)$ is given by

$$\hat{m}(\nu+1) = m(\boldsymbol{w}(\nu)^\top \boldsymbol{n}(\nu+1)).$$

The caret indicates that the signal is calculated from the next (i.e., the $\nu+1$st) training input, but using the current (i.e., the νth) weights. With the definition of the *linearized input*

$$\boldsymbol{a}(\nu) = \hat{m}(\nu)(1-\hat{m}(\nu))\boldsymbol{n}(\nu)^\top, \quad \text{(B.38)}$$

we can write Equation (B.37) in the form

$$m(\nu+1) \approx \boldsymbol{a}(\nu+1)\boldsymbol{w} + [\hat{m}(\nu+1) - \boldsymbol{a}\nu+1)\boldsymbol{w}(\nu)].$$

The term in square brackets is — to first order — the error that arises from the fact that the neuron's output signal is *not* simply linear in $\boldsymbol{w}$. If we neglect it altogether, then we get the *linearized* neuron output signal

$$m(\nu+1) = \boldsymbol{a}(\nu+1)\boldsymbol{w}.$$

Note that $\boldsymbol{a}$ has been defined in Equation (B.38) as a row vector. In order to calculate the synaptic weight vector $\boldsymbol{w}$, we can now apply the theory of recursive linear regression developed in Appendix A. We simply identify the parameter vector $\boldsymbol{w}$ with the synaptic weight vector, y with the desired output ℓ and observation $\boldsymbol{x}(\nu+1)^T$ with $\boldsymbol{a}(\nu+1)$. We then have the least squares problem

$$\begin{pmatrix} \boldsymbol{\ell}_\nu \\ \ell(\nu+1) \end{pmatrix} = \begin{pmatrix} \boldsymbol{\mathcal{A}}_\nu \\ \boldsymbol{a}(\nu+1) \end{pmatrix} \boldsymbol{w} + \boldsymbol{R}_{\nu+1};$$

see Equation (A.5). The Kalman filter equations for the recursive solution of this problem are unchanged:

$$\begin{aligned} \boldsymbol{\Sigma}(\nu+1) &= \big[\boldsymbol{I} - \boldsymbol{K}(\nu+1)\boldsymbol{a}(\nu+1)\big]\boldsymbol{\Sigma}(\nu) \\ \boldsymbol{K}(\nu+1) &= \boldsymbol{\Sigma}(\nu)\boldsymbol{a}(\nu+1)^\top\big[\boldsymbol{a}(\nu+1)\boldsymbol{\Sigma}(\nu)\boldsymbol{a}(\nu+1)^\top + 1\big]^{-1}, \end{aligned} \quad \text{(B.39)}$$

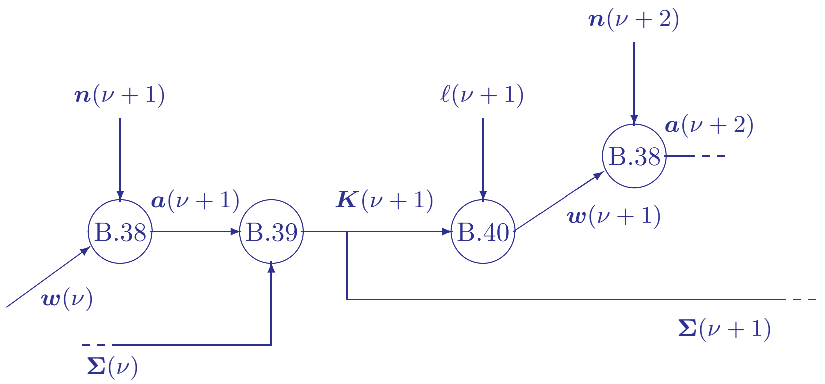


FIGURE B.3
Determination of the synaptic weights for an isolated neuron with the Kalman filter.

while the recursive expression for the parameter vector, Equation (A.8), can be improved somewhat by replacing the linear approximation to the neuron output $\boldsymbol{a}(\nu+1)\boldsymbol{w}(\nu)$ by the actual output for the $\nu+1$st training observation, namely $\hat{m}(\nu+1)$, so we have

$$\boldsymbol{w}(\nu+1) = \boldsymbol{w}(\nu) + \boldsymbol{K}(\nu+1)\big[\ell(\nu+1) - \hat{m}(\nu+1)\big]. \tag{B.40}$$

B.3.2 The algorithm

The recursive calculation of $\boldsymbol{w}$ is depicted in Figure B.3. The input is the current weight vector $\boldsymbol{w}(\nu)$, its covariance matrix $\boldsymbol{\Sigma}_\nu$, and the output vector of the hidden layer $\boldsymbol{n}(\nu+1)$ obtained by propagating the next input observation $\boldsymbol{g}(\nu+1)$ through the network. After determining the linearized input $\boldsymbol{a}(\nu+1)$ from Equation (B.38), the Kalman gain $\boldsymbol{K}_{\nu+1}$ and the new covariance matrix $\boldsymbol{\Sigma}_{\nu+1}$ are calculated with Equation (B.39). Finally, the weights are updated according to Equation (B.40) to give $\boldsymbol{w}(\nu+1)$ and the procedure is repeated.

To make our notation explicit for the output neurons, we substitute

$$\begin{aligned}
\ell(\nu) &\to \ell_k(\nu)\\
\boldsymbol{w}(\nu) &\to \boldsymbol{w}^o_k(\nu)\\
\hat{m}(\nu+1) &\to m_k\left(\boldsymbol{w}^{o\top}_k(\nu)\boldsymbol{n}(\nu+1)\right)\\
\boldsymbol{a}(\nu+1) &\to \boldsymbol{a}^o_k(\nu+1) = \hat{m}_k(\nu+1)(1-\hat{m}_k(\nu+1))\boldsymbol{n}(\nu+1)^\top\\
\boldsymbol{K}(\nu) &\to \boldsymbol{K}^o_k(\nu)\\
\boldsymbol{\Sigma}(\nu) &\to \boldsymbol{\Sigma}^o_k(\nu),
\end{aligned}$$

for $k = 1\dots K$. Then Equation (B.40) becomes

$$\boldsymbol{w}^o_k(\nu+1) = \boldsymbol{w}^o_k(\nu) + \boldsymbol{K}^o_k(\nu+1)\big[\ell_k(\nu+1) - \hat{m}_k(\nu+1)\big], \quad k = 1\dots K. \tag{B.41}$$

Recalling that we wish to minimize the local quadratic cost function, Equation (B.34), note that the expression in square brackets in Equation (B.41) is in fact the negative derivative of $E(\nu)$ with respect to the output signal of the neuron, i.e.,

$$\ell_k(\nu+1) - m_k(\nu+1) = -\frac{\partial E(\nu+1)}{\partial m_k(\nu+1)},$$

so that Equation (B.41) can be expressed in the form

$$\boldsymbol{w}_k^o(\nu+1) = \boldsymbol{w}_k^o(\nu) - \boldsymbol{K}_k^o(\nu+1)\left[\frac{\partial E(\nu+1)}{\partial m_k(\nu+1)}\right]_{\hat{m}_k(\nu+1)}. \tag{B.42}$$

With this observation, we can turn consideration to the hidden neurons, making the substitutions

$$\begin{aligned}
\boldsymbol{w}(\nu) &\rightarrow \boldsymbol{w}_j^h(\nu)\\
\hat{m}(\nu+1) &\rightarrow \hat{n}_j(\nu+1) = f\left(\boldsymbol{w}_j^{h\top}(\nu)\boldsymbol{g}(\nu+1)\right)\\
\boldsymbol{a}(\nu+1) &\rightarrow \boldsymbol{a}_j^h(\nu+1) = \hat{n}_j(\nu+1)(1-\hat{n}_j(\nu+1))\boldsymbol{g}(\nu+1)^\top\\
\boldsymbol{K}(\nu) &\rightarrow \boldsymbol{K}_j^h(\nu)\\
\boldsymbol{\Sigma}(\nu) &\rightarrow \boldsymbol{\Sigma}_j^h(\nu),
\end{aligned}$$

for $j = 1 \ldots L$. Then, analogously to Equation (B.42), the update equation for the weight vector of the jth hidden neuron is

$$\boldsymbol{w}_j^h(\nu+1) = \boldsymbol{w}_j^h(\nu) - \boldsymbol{K}_j^h(\nu+1)\left[\frac{\partial E(\nu+1)}{\partial n_j(\nu+1)}\right]_{\hat{n}_j(\nu+1)}. \tag{B.43}$$

To obtain the partial derivative in Equation (B.43), we differentiate the cost function, Equation (B.34), applying the chain rule:

$$\frac{\partial E(\nu+1)}{\partial n_j(\nu+1)} = -\sum_{k=1}^{K}(\ell_k(\nu+1) - m_k(\nu+1))\frac{\partial m_k(\nu+1)}{\partial n_j(\nu+1)}.$$

From Equation (B.36), noting that $(\boldsymbol{w}_k^o)_j = W_{jk}^o$, we have

$$\frac{\partial m_k(\nu+1)}{\partial n_j(\nu+1)} = m_k(\nu+1)(1-m_k(\nu+1))W_{jk}^o(\nu+1).$$

Combining the last two equations,

$$\frac{\partial E(\nu+1)}{\partial n_j(\nu+1)} = -\sum_{k=1}^{K}(\ell_k(\nu+1)-m_k(\nu+1))m_k(\nu+1)(1-m_k(\nu+1))W_{jk}^o(\nu+1),$$

which we can write more compactly as

$$\frac{\partial E(\nu+1)}{\partial n_j(\nu+1)} = -\boldsymbol{W}_{j\cdot}^o(\nu+1)\boldsymbol{\beta}^o(\nu+1), \tag{B.44}$$

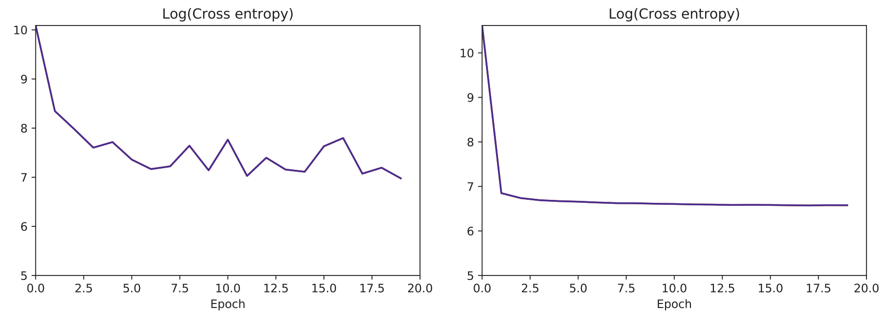


FIGURE B.4
Comparison of cost function minimization. Left: training with back-propagation, right: with the extended Kalman filter.

where $\boldsymbol{W}^o_{j\bullet}$ is the jth *row* (!) of the output-layer weight matrix, and where

$$\boldsymbol{\beta}^o(\nu+1) = (\boldsymbol{\ell}(\nu+1) - \boldsymbol{m}(\nu+1)) \cdot \boldsymbol{m}(\nu+1) \cdot (\boldsymbol{1} - \boldsymbol{m}(\nu+1)).$$

The correct update relation for the weights of the jth hidden neuron is therefore

$$\boldsymbol{w}^h_j(\nu+1) = \boldsymbol{w}^h_j(\nu) + \boldsymbol{K}^h_j(\nu+1)\big[\boldsymbol{W}^o_{j\bullet}(\nu+1)\boldsymbol{\beta}^o(\nu+1)\big]. \tag{B.45}$$

Apart from initialization of the covariance matrices $\boldsymbol{\Sigma}^h_j(0)$, $\boldsymbol{\Sigma}^o_k(0)$, the Kalman training procedure has no adjustable parameters whatsoever. The initial covariance matrices are simply taken to be proportional to the corresponding identity matrices:

$$\boldsymbol{\Sigma}^h_j(0) = Z\boldsymbol{I}^h, \quad \boldsymbol{\Sigma}^o_k(0) = Z\boldsymbol{I}^o, \quad Z \gg 1,\ j = 1\dots L,\ k = 1\dots K,$$

where I^h is the $(N+1)\times(N+1)$ and I^o the $(L+1)\times(L+1)$ identity matrix. We choose $Z = 100$ and obtain the following algorithm:

Algorithm (Kalman Filter Training)

1. Set $\nu = 0$, $\boldsymbol{\Sigma}^h_j(0) = 100\boldsymbol{I}^h, j = 1\dots L$, $\boldsymbol{\Sigma}^o_k(0) = 100\boldsymbol{I}^o,\ k = 1\dots K$ and initialize the synaptic weight matrices $\boldsymbol{W}^h(0)$ and $\boldsymbol{W}^o(0)$ with random numbers.

2. Choose a training pair $(\boldsymbol{g}(\nu+1), \boldsymbol{\ell}(\nu+1))$ and determine the hidden layer output vector

$$\hat{\boldsymbol{n}}(\nu+1) = \begin{pmatrix} 1 \\ f\left(\boldsymbol{W}^h(\nu)^\top \boldsymbol{g}(\nu+1)\right) \end{pmatrix},$$

and with it the quantities

$$\boldsymbol{a}_j^h(\nu+1) = \hat{n}_j(\nu+1)(1-\hat{n}_j(\nu+1))\boldsymbol{g}(\nu+1)^\top, \quad j = 1\dots L,$$

$$\begin{aligned} \hat{m}_k(\nu+1) &= m_k\left(\boldsymbol{w}_k^{o\top}(\nu)\hat{\boldsymbol{n}}(\nu+1)\right) \\ \boldsymbol{a}_k^o(\nu+1) &= \hat{m}_k(\nu+1)(1-\hat{m}_k(\nu+1))\hat{\boldsymbol{n}}(\nu+1)^\top, \quad k=1\dots K \end{aligned}$$

and

$$\boldsymbol{\beta}^o(\nu+1) = (\boldsymbol{\ell}(\nu+1) - \hat{\boldsymbol{m}}(\nu+1))\cdot\hat{\boldsymbol{m}}(\nu+1)\cdot(\mathbf{1}-\hat{\boldsymbol{m}}(\nu+1)).$$

3. Determine the Kalman gains for all of the neurons according to

$$\begin{aligned} \boldsymbol{K}_k^o(\nu+1) &= \boldsymbol{\Sigma}_k^o(\nu)\boldsymbol{a}_k^o(\nu+1)^\top\left[\boldsymbol{a}_k^o(\nu+1)\boldsymbol{\Sigma}_k^o(\nu)\boldsymbol{a}_k^o(\nu+1)^\top+1\right]^{-1}, \\ & k = 1\dots K \\ \boldsymbol{K}_j^h(\nu+1) &= \boldsymbol{\Sigma}_j^h(\nu)\boldsymbol{a}_j^h(\nu+1)^\top\left[\boldsymbol{a}_j^h(\nu+1)\boldsymbol{\Sigma}_j^h(\nu)\boldsymbol{a}_j^h(\nu+1)^\top+1\right]^{-1}, \\ & j = 1\dots L \end{aligned}$$

4. Update the synaptic weight matrices:

$$\begin{aligned} \boldsymbol{w}_k^o(\nu+1) &= \boldsymbol{w}_k^o(\nu) + \boldsymbol{K}_k^o(\nu+1)[\ell_k(\nu+1)-\hat{m}_k(\nu+1)], \quad k=1\dots K \\ \boldsymbol{w}_j^h(\nu+1) &= \boldsymbol{w}_j^h(\nu) + \boldsymbol{K}_j^h(\nu+1)\left[\boldsymbol{W}_{j\boldsymbol{.}}^o(\nu+1)\boldsymbol{\beta}^o(\nu+1)\right], \quad j=1\dots L. \end{aligned}$$

5. Determine the new covariance matrices:

$$\begin{aligned} \boldsymbol{\Sigma}_k^o(\nu+1) &= \left[\boldsymbol{I}^o - \boldsymbol{K}_k^o(\nu+1)\boldsymbol{a}_k^o(\nu+1)\right]\boldsymbol{\Sigma}_k^o(\nu), \quad k=1\dots K \\ \boldsymbol{\Sigma}_j^h(\nu+1) &= \left[\boldsymbol{I}^h - \boldsymbol{K}_j^h(\nu+1)\boldsymbol{a}_j^h(\nu+1)\right]\boldsymbol{\Sigma}_j^h(\nu), \quad j=1\dots L. \end{aligned}$$

6. If the overall cost function, Equation (6.30), is sufficiently small, stop, else set $\nu = \nu+1$ and go to 2.

This method was originally suggested by Shah and Palmieri (1990), who called it the *multiple extended Kalman algorithm* (MEKA),* and explained in detail by Hayken (1994). Its Python implementation is in the class `Ffnekf` which may be imported from the Python module `auxil.supervisedclass.py`; see Appendix C. The cost functions for back-propagation and extended Kalman filter training are compared in Figure B.4. Although both methods use stochastic training, the Kalman filter is considerably more efficient.

* *Multiple*, because the algorithm is applied to multiple neurons, the adjective *extended* characterizes Kalman filter methods that linearize nonlinear models by using a first-order Taylor expansion.

C

Software

This appendix provides installation instructions for the software accompanying the book. It also gives a link to the documentation of all scripts used to illustrate the various algorithms and methods discussed in the text as well as to the solutions to the exercises.

C.1 Installation

By far the easiest way to use the software is to run it in a Docker container on your host operating system. For this, all you need is the Docker runtime which is freely available. It can be easily installed on Linux, Mac, and Windows, see

```
https://docs.docker.com
```

Once you have Docker installed, pull and run the container with

```
docker run -d -p 8888:8888
    -v <path-to-crc5imagery>:/home/imagery/ --name=crc5
        mort/crc5docker
```

This maps the host directory `crc5imagery` to the container directory

```
/home/imagery/
```

and runs the container in detached mode under the name `crc5`. The imagery directory is needed to run the scripts on the image files discussed in the text. Instructions to access the compressed `crc5imagery` directory can be found in the README on the GitHub repository

```
https://github.com/mortcanty/CRC5Docker
```

Point your browser to `http://localhost:8888` to see the JupyterLab home page. Open a notebook to begin work. Stop with

```
docker stop crc5
```

Re-start with

DOI: 10.1201/9781003503286-C

```
docker start crc5
```

For LLM enthusiasts an experimental RAG (retrieval augmented generation) version of the Docker container can be pulled and run with

```
docker run -d -p 8888:8888 -p 7860:7860
    -v <path-to-crc5imagery>:/home/imagery/ --name=crc5_rag
        mort/crc5docker_rag
```

which includes an additional JupyterLab notebook to query the textbook's content informally. Since the LLM is running locally the response time is very slow (minutes).

C.2 Command line utilities

Here we mention some useful command line utilities available in the Docker container. They can be run from the Jupyter notebook interface by opening a local terminal, or in the case of the `gdal` utilities which don't request additional input from `stdin`, directly from an input cell by prepending the command with a "!".

C.2.1 gdal

A set of utilities is automatically installed together with GDAL, the geospatial data abstraction library. See

https://gdal.org/programs/index.html

for a full list and documentation. From an input cell enter:

```
!<utility name> [OPTIONS] <inputs>
```

Example: Read and display image statistics on an LANDSAT 7 ETM+ image. Force computation if no statistics are stored in the image.

```
!gdalinfo -stats imagery/LE7_20010626
```

C.2.2 earthengine

The Google Earth Engine Command Line Interface allows various manipulations of, and provides information about Earth Engine assets and tasks.
Example: Add authentication information to your Docker container. Open a local terminal from the notebook home page and enter:

```
!earthengine authenticate
```

and follow the instructions. More usually you will authenticate and initialize from within the Jupyter notebook Python environment:

```
import ee
ee.Authenticate()
ee.Initialize()
```

C.2.3 ipcluster

Start/stop a parallel processing cluster, see

`https://github.com/ipython/ipyparallel`

Example: Start four IPython engines in a Jupyter notebook cell and detach them from the Jupyter kernel in order to run other cells. Open a notebook cell and enter:

```
!ipcluster start -n 4 --daemonize
```

C.3 Source code and documentation

For those who wish to program the examples given in the exercises, or modify/improve the more extensive scripts accompanying the text, the source code is available at

`https://github.com/mortcanty/CRC5Docker`

and includes the Dockerfile to build a local version of the Docker image. This repo also provides detailed documentation for the individual scripts contained in the `/home/scripts` directory and some of the utilities in the `/home/auxil` directory:

`https://github.com/mortcanty/CRC5Docker/python_scripts.pdf`

The scripts are all command line oriented and are documented in the context of the Jupyter notebook interface.

If you want to avoid using a Docker container altogether, you can clone the entire GitHub repository and create a Python virtual environment, calling it `venvcrc5`. Use the Dockerfile as a guide to installing the necessary utilities.

C.4 Solutions manual

Solutions to selected exercises following each chapter are also available in pdf format on the repo:

https://github.com/mortcanty/CRC5Docker/solutions.pdf

Mathematical Notation

X	random variable
x	realization of X, observation
$\boldsymbol{X}$	random (column) vector
$\boldsymbol{x}$	realization of $\boldsymbol{X}$ (vector observation)
$\boldsymbol{\mathcal{X}}$	data design matrix for $\boldsymbol{X}$
$G, g, \boldsymbol{G}, \boldsymbol{g}, \boldsymbol{\mathcal{G}}$	as above, designating pixel gray-values
κ	valid kernel function
$\boldsymbol{\mathcal{K}}$	kernel matrix with elements $\kappa(\boldsymbol{x}, \boldsymbol{x}')$
$\boldsymbol{x}^\top$	transposed form of the vector $\boldsymbol{x}$ (row vector)
$\lVert\boldsymbol{x}\rVert$	length or (2-)norm of $\boldsymbol{x}$
$\boldsymbol{x}^\top\boldsymbol{y}$	inner product of two vectors (scalar)
$\boldsymbol{x}\cdot\boldsymbol{y}$	Hadamard (component-by-component) product
$\boldsymbol{x}\boldsymbol{y}^\top$	outer product of two vectors (matrix)
$\boldsymbol{C}$	random matrix
$\boldsymbol{c}$	realization of $\boldsymbol{C}$
$\boldsymbol{x}^\top\boldsymbol{C}\boldsymbol{y}$	quadratic form (scalar)
$\lvert\boldsymbol{C}\rvert$	determinant of $\boldsymbol{C}$
$\mathrm{tr}(\boldsymbol{C})$	trace of $\boldsymbol{C}$
$\boldsymbol{I}$	identity matrix
$\mathbf{0}$	column vector of zeroes
$\mathbf{1}$	column vector of ones
$\boldsymbol{\Lambda}$	$\mathrm{Diag}(\lambda_1 \ldots \lambda_N)$ (diagonal matrix of eigenvalues)
$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$	partial derivative of $f(\boldsymbol{x})$ with respect to vector $\boldsymbol{x}$
$f(x)\vert_{x=x^*}$	$f(x)$ evaluated at $x = x^*$
$\mathbf{i}$	$\sqrt{-1}$
$\lvert z\rvert$	absolute value of real or complex number z
z^*	complex conjugate of complex number z
$\boldsymbol{Z}^\dagger$	Hermitian conjugate of complex vector or matrix $\boldsymbol{Z}$
Ω	sample space in probability theory
$\Pr(A \mid B)$	probability of event A conditional on event B
$P(x)$	distribution function for random variable X
$P(\boldsymbol{x})$	joint distribution function for random vector $\boldsymbol{X}$
$p(x)$	probability density function for X
$p(\boldsymbol{x})$	joint probability density function for $\boldsymbol{X}$
$\mathcal{P}$	the P-value
$\langle X\rangle,\ \mu$	mean (or expected) value

$\mathrm{var}(X),\ \sigma^2$	variance
$\langle \boldsymbol{X} \rangle,\ \boldsymbol{\mu}$	mean vector
$\boldsymbol{\Sigma}$	variance–covariance matrix
$\hat{\boldsymbol{\mu}},\ \hat{\boldsymbol{\Sigma}}$	maximum likelihood estimates of $\boldsymbol{\mu},\ \boldsymbol{\Sigma}$
$\Phi(x)$	standard normal probability distribution function
$\phi(x)$	standard normal probability density function
$p_{\chi^2;n}(x)$	chi-square density function with n degrees of freedom
$p_{t;n}(x)$	Student-t density function with n degrees of freedom
$p_{f;m,n}(x)$	F-density function with m and n degrees of freedom
$p_{\mathcal{W}}(\boldsymbol{x}, m)$	Wishart distribution with m degrees of freedom
$p_{\mathcal{W}_C}(\boldsymbol{x}, m)$	complex Wishart distribution with m degrees of freedom
$(\hat{x}(0), \hat{x}(1) \ldots)$	discrete Fourier transform of array $(x(0), x(1) \ldots)$
$\boldsymbol{x} * \boldsymbol{y}$	discrete convolution of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$
$\langle \phi, \psi \rangle$	inner product $\int \phi(x)\psi(x)dx$ of functions ϕ and ψ
$\mathcal{N}_i$	neighborhood of ith pixel
$\{x \mid f(x)\}$	set of elements x that satisfy condition $f(x)$
$\mathcal{K}$	set of class labels $\{1 \ldots K\}$
$\boldsymbol{\ell}$	vector representation of a class label
$u \in U$	u is an element of the set U
$U \otimes V$	Cartesian product set
$V \subset U$	V is a (proper) subset of the set U
$\arg\max_x f(x)$	the set of x which maximizes $f(x)$
$f : A \mapsto B$	the function f which maps the set A to the set B
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of integers
$=:$	equal by definition
$\square$	end of a proof

References

Aanaes, H., Sveinsson, J. R., Nielsen, A. A., Bovith, T., and Benediktsson, A. (2008). Model based satellite image fusion. *IEEE Transactions on Geoscience and Remote Sensing*, 46(5):1336–1346.

Aboufadel, E. and Schlicker, S. (1999). *Discovering Wavelets.* J. Wiley and Sons.

Abrams, M., Hook, S., and Ramachandran, B. (1999). ASTER user handbook. Technical report, Jet Propulsion Laboratory and California Institute of Technology.

Aiazzi, B., Alparone, L., Baronti, S., and Garzelli, A. (2002). Context-driven fusion of high spatial and spectral resolution images based on oversampled multiresolution analysis. *IEEE Transactions on Geoscience and Remote Sensing*, 40(10):2300–2312.

Amro, I., Mateos, J., Vega, M., et al. (2011). A survey of classical methods and new trends in pansharpening of multispectral images. *EURASIP Journal on Advances in Signal Processing*, 79. https://doi.org/10.1186/1687-6180-2011-79.

Anderson, T. W. (2003). *An Introduction to Multivariate Statistical Analysis.* Wiley Series in Probability and Statistics, third edition.

Anfinsen, S., Doulgeris, A., and Eltoft, T. (2009a). Estimation of the equivalent number of looks in polarimetric synthetic aperture radar imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 47(11):3795–3809.

Anfinsen, S., Eltoft, T., and Doulgeris, A. (2009b). A relaxed Wishart model for polarimetric SAR data. In *Proc. PolinSAR, 4th International Workshop on Science and Applications of SAR Polarimetry and Polarimetric Interferometry, Frascati, Italy.*

Beisl, U. (2001). *Correction of Bidirectional Effects in Imaging Spectrometer Data.* Remote Sensing Laboratories, Remote Sensing Series 37, Department of Geography, University of Zurich.

Bellman, R. (1961). *Adaptive Control Procesess, A Guided Tour.* Princeton University Press.

Belousov, A. I., Verzakov, S. A., and von Frese, J. (2002). A flexible classification approach with optimal generalization performance: Support vector machines. *Chemometrics and Intelligent Laboratory Systems*, 64:15–25.

Bilbo, C. M. (1989). Statistisk analyse af relationer mellem alternative antistoftracere. Master's thesis, Informatics and Mathematical Modeling, Technical University of Denmark, Lyngby. In Danish.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition.* Oxford University Press.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning.* Springer.

Bishop, Y. M. M., Feinberg, E. E., and Holland, P. W. (1975). *Discrete Multivariate Analysis, Theory and Practice.* Cambridge Press.

Box, G. E. P. (1949). A general distribution theory for a class of likelihood criteria. *Biometrika*, 36:317–346.

Bradley, A. P. (2003). Shift-invariance in the discrete wavelet transform. In Sun, C., Talbot, H., Ourselin, S., and Adriaansen, T., editors, *Proc. VIIth Digital Image Computing: Techniques and Applications*, pages 29–38.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman, L. (2001). Xception: Deep learning with depthwise separable convolutions. *Computer Vision and Pattern Recognition.* https://doi.org/10.48550/arXiv.1610.02357.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification outputs, with relationships to statistical pattern recognition. In Soulié, F. F. and Hérault, J., editors, *Neurocomputing: Algorithms, Architectures and Applications*, pages 227–236. Springer.

Bruzzone, L. and Prieto, D. F. (2000). Automatic analysis of the difference image for unsupervised change detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(4):1171–1182.

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–699.

Canty, M. J. (2009). Boosting a fast neural network for supervised land cover classification. *Computers and Geosciences*, 35:1280–1295.

Canty, M. J. (2014). CRCENVI: ENVI/IDL scripts for Image Analysis, Classification and Change Detection in Remote Sensing. *https://mortcanty.github.io/CRCENVI/*.

Canty, M. J. and Nielsen, A. A. (2006). Visualization and unsupervised classification of changes in multispectral satellite imagery. *International Journal of Remote Sensing*, 27(18):3961–3975. Internet http://www.imm.dtu.dk/pubdb/p.php?3389.

Canty, M. J. and Nielsen, A. A. (2008). Automatic radiometric normalization of multitemporal satellite imagery with the iteratively re-weighted MAD transformation. *Remote Sensing of Environment*, 112(3):1025–1036. Internet http://www.imm.dtu.dk/pubdb/p.php?5362.

Canty, M. J., Nielsen, A. A., Conradsen, K., and Skriver, H. (2019). Statistical analysis of changes in Sentinel-1 time series on Google Earth Engine. *Remote Sensing*, 12(1):46.

Canty, M. J., Nielsen, A. A., and Schmidt, M. (2004). Automatic radiometric normalization of multitemporal satellite imagery. *Remote Sensing of Environment*, 91(3-4):441–451. Internet http://www.imm.dtu.dk/pubdb/p.php?2815.

Canty, M. J., Nielsen, A. A., Skriver, H., and Conradsen, K. (2020). Wishart-based adaptive temporal filtering of polarimetric sar imagery. *Remote Sensing*, 12(15):2454.

Canty, M. J. and Nieslsen, A. A. (2012). Linear and kernel methods for multivariate change detection. *Computers and Geosciences*, 38:107–114.

Chollet, F. (2017). Random forests. *Machine Learning*, 45:5–321.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46.

Comaniciu, D. and Meer, P. (2002). Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619.

Congalton, R. G. and Green, K. (1999). *Assessing the Accuracy of Remotely Sensed Data: Principles and Practices.* Lewis Publishers.

Conradsen, K., Nielsen, A. A., Schou, J., and Skriver, H. (2003). A test statistic in the complex Wishart distribution and its application to change detection in polarimetric SAR data. *IEEE Transactions on Geoscience and Remote Sensing*, 41(1):3–19. Internet http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=1219.

Conradsen, K., Nielsen, A. A., and Skriver, H. (2016). Determining the points of change in time series of polarimetric SAR data. *IEEE Transactions on Geoscience and Remote Sensing*, 54(5):3007–3024.

Conradsen, K., Nielsen, A. A., and Skriver, H. (2024). Omnibus change detection in block diagonal covariance matrix polsar data illustrated with simulated and sentinel-1 data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 17:16359–16376.

Coppin, P., Jonckheere, I., Nackaerts, K., and Muys, B. (2004). Digital change detection methods in ecosystem monitoring: A review. *International Journal of Remote Sensing*, 25(9):1565–1596.

Cristianini, N. and Shawe-Taylor, J. (2000). *Support Vector Machines and Other Kernel-based Learning Methods.* Cambridge University Press.

Daubechies, I. (1988). Orthonormal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 41:909–996.

Dhillon, I., Guan, Y., and Kulis, B. (2005). A unified view of kernel K-means, spectral clustering and graph partitioning. Technical report UTCS TR-04-25, University of Texas at Austin.

Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923.

Du, Y., Teillet, P. M., and Cihlar, J. (2002). Radiometric normalization of multitemporal high-resolution images with quality control for land cover change detection. *Remote Sensing of Environment*, 82:123–134.

Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis.* J. Wiley and Sons.

Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification.* Wiley Interscience, second edition.

Duda, T. and Canty, M. J. (2002). Unsupervised classification of satellite imagery: Choosing a good algorithm. *International Journal of Remote Sensing*, 23(11):2193–2212.

Dunn, J. C. (1973). A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. *Journal of Cybernetics*, PAM1-1:32–57.

Fahlman, S. E. and LeBiere, C. (1990). The cascade correlation learning architecture. In Touertzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, pages 524–532. Morgan Kaufmann.

Fanning, D. W. (2000). *IDL Programming Techniques.* Fanning Software Consulting.

Fraley, C. (1996). Algorithms for model-based Gaussian hierarchical clustering. Technical report 311, Department of Statistics, University of Washington, Seattle.

Freund, J. E. (1992). *Mathematical Statistics.* Prentice-Hall, fifth edition.

Freund, Y. and Shapire, R. E. (1996). Experiments with a new boosting algorithm. In *Proceedings. Thirteenth International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann.

Freund, Y. and Shapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119–139.

Fukunaga, K. (1990). *Introduction to Statistical Pattern Recognition.* Academic Press, second edition.

Fukunaga, K. and Hostetler, L. D. (1975). The estimation of the gradient of a density function, with applications to pattern recognition. *IEEE Transactions on Information Theory*, IT-21:32–40.

Galloy, M. (2011). *Modern IDL, A Guide to IDL Programming.* M. Galloy. Internet http://michaelgalloy.com/.

Gath, I. and Geva, A. B. (1989). Unsupervised optimal fuzzy clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(3): 773–781.

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow, Second Edition.* O'Reilly.

Géron, A. (2023). *Hands-On Machine Learning with Scikit-Learn and TensorFlow, Third Edition.* O'Reilly.

Gonzalez, R. C. and Woods, R. E. (2017). *Digital Image Processing.* Pearson India.

Goodman, J. W. (1984). Statistical properties of laser speckle patterns. In Dainty, J. C., editor, *Laser Speckle and Related Phenomena*, pages 9–75. Springer.

Goodman, N. R. (1963). Statistical analysis based on a certain multivariate complex Gaussian distribution (An Introduction). *Annals of Mathematical Statistics*, 34:152–177.

Gorelick, N., Hancher, M., Dixon, M., Ilyushchenko, S., Tau, D., and Moore, R. (2017). Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment.* https://doi.org/10.1016/j.rse.2017.06.031.

Green, A. A., Berman, M., Switzer, P., and Craig, M. D. (1988). A transformation for ordering multispectral data in terms of image quality with implications for noise removal. *IEEE Transactions on Geoscience and Remote Sensing*, 26(1):65–74.

Groß, M. H. and Seibert, F. (1993). Visualization of multidimensional image data sets using a neural network. *The Visual Computer*, 10:145–159.

Gumley, L. E. (2002). *Practical IDL Programming.* Morgan Kaufmann.

Haberächer, P. (1995). *Praxis der Digitalen Bildverarbeitungen und Mustererkennung.* Carl Hanser Verlag.

Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *Proceedings of the Fourth Alvey Vision Conference*, pages 147–151.

Harsanyi, J. C. (1993). *Detection and Classification of Subpixel Spectral Signatures in Hyperspectral Image Sequences.* Ph.D. Thesis, University of Maryland, 116 pp.

Harsanyi, J. C. and Chang, C.-I. (1994). Hyperspectral image classification and dimensionality reduction: An orthogonal subspace projection approach. *IEEE Transactions on Geoscience and Remote Sensing*, 32(4):779–785.

Hayken, S. (1994). *Neural Networks, a Comprehensive Foundation.* Macmillan.

Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley.

Hilger, K. B. (2001). *Exploratory Analysis of Multivariate Data.* Ph.D. Thesis, IMM-PHD-2001-89, Technical University of Denmark.

Hilger, K. B. and Nielsen, A. A. (2000). Targeting input data for change detection studies by suppression of undesired spectra. In *Proceedings of a Seminar on Remote Sensing and Image Analysis Techniques for Revision of Topographic Databases, KMS, The National Survey and Cadastre, Copenhagen, Denmark, February 2000.*

Hotelling, H. (1936). Relations between two sets of variates. *Biometrika*, 28:321–377.

Hu, M. K. (1962). Visual pattern recognition by moment invariants. *IEEE Transactions on Information Theory*, IT-8:179–187.

Huang, B., Lu, K., N, A., Khalel, A., Tarabalka, Y., Malof, J., Boulch, A., LeSaux, B., and Collins, L. (2018). Large-scale semantic classification: outcome of the first year of Inria aerial image labeling benchmark. In *IGARSS 2018 - IEEE International Geoscience and Remote Sensing Symposiumm, Jul 2018, Valencia, Spain. pp.1-4.*

Jensen, J. R. (2005). *Introductory Digital Image Analysis: A Remote Sensing Perspective.* Prentice Hall.

Jensen, J. R. (2018). *Introductory Digital Image Analysis: A Remote Sensing Perspective, 4th Edition.* Pearson India.

Kendall, M. and Stuart, A. (1979). *The Advanced Theory of Statistics*, volume 2. Charles Griffen & Company Limited, fourth edition.

Kohonen, T. (1989). *Self-Organization and Associative Memory.* Springer.

Kraskov, A., Stoegbauer, H., and Grassberger, P. (2004). Estimating mutual information. *Physical Review E*, 69(066138):1–16.

Kruse, F. A., Lefkoff, A. B., Boardman, J. B., Heidebrecht, K. B., Shapiro, A. T., Barloon, P. J., and Goetz, A. F. H. (1993). The spectral image processing system (SIPS), interactive visualization and analysis of imaging spectrometer data. *Remote Sensing of Environment*, 44:145–163.

Kurz, F., Charmette, B., Suri, S., Rosenbaum, D., Spangler, M., Leonhardt, A., Bachleitner, M., Stätter, R., and Reinartz, P. (2007). Automatic traffic monitoring with an airborne wide-angle digital camera system for estimation of travel times. In Stilla, U., Mayer, H., Rottensteiner, F., Heipke, C., and Hinz, S., editors, *Photogrammetric Image Analysis*. International Archives of the Photogrammetry, Remote Sensing and Spatial Information Service PIA07, Munich, Germany.

Kwon, H. and Nasrabadi, N. M. (2005). Kernel RX-algorithm: A linear anomaly detector for hyperspectral imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 43(2):388–397.

Lang, H. R. and Welch, R. (1999). Algorithm theoretical basis document for ASTER digital elevation models. Technical report, Jet Propulsion Laboratory and University of Georgia.

Langtangen, H. P. (2009). *Python Scripting for Computational Science*. Springer.

LeCun, Yann and Cortes, Corinna and Burges, C. J. (2010). MNIST handwritten digit database. ATT Labs [Online], 2.

Lee, J.-S., Grunes, M. R., and de Grandi, G. (1999). Polarimetric SAR speckle filtering and its implication for classification. *IEEE Transactions on Geoscience and Remote Sensing*, 37(5):2363–2373.

Lee, J.-S., Grunes, M. R., and Kwok, R. (1994). Classification of multi-look polarimetric SAR imagery based on complex Wishart distribution. *International Journal of Remote Sensing*, 15(11):2299–2311.

Li, H., Manjunath, B. S., and Mitra, S. K. (1995). A contour-based approach to multisensor image registration. *IEEE Transactions on Image Processing*, 4(3):320–334.

Li, S. Z. (2001). *Markov Random Field Modeling in Image Analysis*. Computer Science Workbench. Springer, second edition.

Liao, X. and Pawlak, M. (1996). On image analysis by moments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(3):254–266.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110. https://doi.org/10.1023/B:VISI.0000029664.99615.94.

Maas, S. J. and Rajan, N. (2010). Normalizing and converting image DC data using scatter plot matching. *Remote Sensing*, 2(7):1644–1661.

Mallat, S. G. (1989). A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693.

Mardia, K. V., Kent, J. T., and Bibby, J. M. (1979). *Multivariate Analysis*. Academic Press.

Masters, T. (1995). *Advanced Algorithms for Neural Networks, A C++ Sourcebook*. J. Wiley and Sons.

Mather, P. and Koch, M. (2010). *Computer Processing of Remotely-Sensed Images: An Introduction*. Wiley, fourth edition.

Milman, A. S. (1999). *Mathematical Principles of Remote Sensing*. Sleeping Bear Press.

Moeller, M. F. (1993). A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533.

Müller, K.-R., Mika, S., Rätsch, G., Tsuda, K., and Schölkopf, B. (2001). An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–202.

Muro, J., Canty, M. J., K.Conradsen, Huettich, C., Nielsen, A. A., Skriver, H., Strauch, A., Thonfeld, F., and Menz, G. (2016). Short-term change detection in wetlands using Sentinel-1 time series. *Remote Sensing*, 8(10):795 Open access DOI:10.3390/rs8100795.

Murphey, Y. L., Chen, Z., and Guo, H. (2001). Neural learning using AdaBoost. In *Proceedings. IJCNN apos;01. International Joint Conference on Neural Networks*, volume 2, pages 1037–1042.

Mustard, J. F. and Sunshine, J. M. (1999). Spectral analysis for Earth science: Investigations using remote sensing data. In Rencz, A., editor, *Manual of Remote Sensing*, pages 251–307. J. Wiley and Sons, second edition.

Nielsen, A. A. (2001). Spectral mixture analysis: Linear and semi-parametric full and iterated partial unmixing in multi- and hyperspectral data. *Journal of Mathematical Imaging and Vision*, 15:17–37.

Nielsen, A. A. (2007). The regularized iteratively reweighted MAD method for change detection in multi- and hyperspectral data. *IEEE Transactions on Image Processing*, 16(2):463–478. Internet http://www.imm.dtu.dk/pubdb/p.php?4695.

Nielsen, A. A. (2020). Fast matrix based computation of eigenvalues and the loewner order in polsar data. *IEEE Geoscience and Remote Sensing Letters*, 17(10):1727–1731.

Nielsen, A. A. and Canty, M. J. (2008). Kernel principal component analysis for change detections. In *SPIE Europe Remote Sensing Conference, Cardiff, Great Britain, 15-18 September*, volume 7109.

Nielsen, A. A., Canty, M. J., Skriver, H., and Conradsen, K. (2017). Change detection in multi-temporal dual polarization Sentinel-1 data. In *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium, IGARSS, Fort Worth Texas*, pages 3901–3904.

Nielsen, A. A., Conradsen, K., and Simpson, J. J. (1998). Multivariate alteration detection (MAD) and MAF post-processing in multispectral, bitemporal image data: New approaches to change detection studies. *Remote Sensing of Environment*, 64:1–19. Internet http://www.imm.dtu.dk/pubdb/p.php?1220.

Nielsen, A. A., Skriver, H., and Conradsen, K. (2020). The loewner order and direction of detected change in sentinel-1 and radarsat-2 data. *IEEE Geoscience and Remote Sensing Letters (Early Access June 2019)*, 17(2):242–246. http://doi.org/10.1109/LGRS.2019.2918636.

Núnez, J., Otazu, X., Fors, O., Prades, A., Palà, V., and Arbiol, R. (1999). Multiresolution-based image fusion with additive wavelet decomposition. *IEEE Transactions on Geoscience and Remote Sensing*, 37(3):1204–1211.

Oliver, C. and Quegan, S. (2004). *Understanding Synthetic Aperture Radar Images*. SciTech.

Palubinskas, G. (1998). K-means clustering algorithm using the entropy. In *SPIE European Symposium on Remote Sensing, Conference on Image and Signal Processing for Remote Sensing, September, Barcelona*, volume 3500, pages 63–71.

Parelius, E. J. (2023). A review of deep-learning methods for change detection in multispectral remote sensing images. *Remote Sensing*, 15(8).

Patefield, W. M. (1977). On the information matrix in the linear functional problem. *Journal of the Royal Statistical Society, Series C*, 26:69–70.

Philpot, W. and Ansty, T. (2013). Analytical description of pseudoinvariant features. *IEEE Transactions on Geoscience and Remote Sensing*, 51(4):2016–2021.

Pitz, W. and Miller, D. (2010). The TerraSAR-X satellite. *IEEE Transactions on Geoscience and Remote Sensing*, 48(2):615–622.

Polikar, R. (2006). Ensemble-based systems in decision making. *IEEE Circuits and Systems Magazine*, Third Quarter 2006.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2002). *Numerical Recipes in C++*. Cambridge University Press, second edition.

Price, D., Knerr, S., Personnaz, L., and Dreyfus, G. (1995). Pairwise neural network classifiers with probabilistic outputs. In Fischler, M. A. and Firschein, O., editors, *Neural Information Processing Systems*, pages 1109–1116. MIT Press.

Prokop, R. J. and Reeves, A. P. (1992). A survey of moment-based techniques for unoccluded object representation and recognition. *Graphical Models and Image Processing*, 54(5):438–460.

Quam, L. H. (1987). Hierarchical warp stereo. In Fischler, M. A. and Firschein, O., editors, *Readings in Computer Vision*, pages 80–86. Morgan Kaufmann.

Radke, R. J., Andra, S., Al-Kofahi, O., and Roysam, B. (2005). Image change detection algorithms: A systematic survey. *IEEE Transactions on Image Processing*, 14(4):294–307.

Rall, L. B. (1981). Automatic differentiation: Techniques and applications. *Lecture Notes in Computer Science, Springer*, 120.

Ranchin, T. and Wald, L. (2000). Fusion of high spatial and spectral resolution images: The ARSIS concept and its implementation. *Photogrammetric Engineering and Remote Sensing*, 66(1):49–61.

Reddy, B. S. and Chatterji, B. N. (1996). An FFT-based technique for translation, rotation and scale-invariant image registration. *IEEE Transactions on Image Processing*, 5(8):1266–1271.

Redner, R. A. and Walker, H. F. (1984). Mixture densities, maximum likelihood and the EM algorithm. *SIAM Review*, 26(2):195–239.

Reed, I. S. and Yu, X. (1990). Adaptive multiple band CFAR detection of an optical pattern with unknown spectral distribution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(10):1760–1770.

Riano, D., Chuvieco, E., Salas, J., and Aguado, I. (2003). Assessment of different topographic corrections in LANDSAT-TM data for mapping vegetation types. *IEEE Transactions on Geoscience and Remote Sensing*, 41(5):1056–1061.

Richards, J. A. (2009). *Remote Sensing with Imaging Radar.* Springer.

Richards, J. A. (2012). *Remote Sensing Digital Image Analysis: An Introduction.* Springer.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks.* Cambridge University Press.

Schaum, A. and Stocker, A. (1997). Spectrally selective target detection. In *Proceedings of the International Symposium on Spectral Sensing Research.*

Schölkopf, B., Smola, A., and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(5):1299–1319.

Schott, J. R., Salvaggio, C., and Volchok, W. J. (1988). Radiometric scene normalization using pseudo-invariant features. *Remote Sensing of Environment*, 26:1–16.

Schowengerdt, R. A. (1997). *Remote Sensing, Models and Methods for Image Processing*. Academic Press.

Schowengerdt, R. A. (2006). *Remote Sensing, Models and Methods for Image Processing*. Academic Press, second edition.

Schroeder, T. A., Cohen, W. B., Song, C., Canty, M. J., and Zhiqiang, Y. (2006). Radiometric calibration of Landsat data for characterization of early successional forest patterns in western Oregon. *Remote Sensing of Environment*, 103(1):16–26.

Schwenk, H. and Bengio, Y. (2000). Boosting neural networks. *Neural Computation*, 12(8):1869–1887.

Settle, J. J. (1996). On the relation between spectral unmixing and subspace projection. *IEEE Transactions on Geoscience and Remote Sensing*, 34(4):1045–1046.

Shah, S. and Palmieri, F. (1990). MEKA: A fast, local algorithm for training feed forward neural networks. *Proceedings of the International Joint Conference on Neural Networks, San Diego*, I(3):41–46.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Shekarforoush, H., Berthod, M., and Zerubia, J. (1995). Subpixel image registration by estimating the polyphase decomposition of the cross power spectrum. Technical report 2707, Institut National de Recherche en Informatique et en Automatique (INRIA).

Siegel, S. S. (1965). *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill.

Singh, A. (1989). Digital change detection techniques using remotely-sensed data. *International Journal of Remote Sensing*, 10(6):989–1002.

Sirko, W., Kashubin, S., Ritter, M., Annkah, A., Bouchareb, Y., Dauphin, Y., Keysers, D., Neumann, M., Cisse, M., and Quinn, J. (2021). Continental-scale building detection from high resolution satellite imagery. *Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2107.12283v2.

Smith, S. M. and Brady, J. M. (1997). SUSAN: A new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78.

Solem, J. E. (2012). *Programming Computer Vision with Python.* O'Reilly.

Strang, G. (1989). Wavelets and dilation equations: A brief introduction. *SIAM Review*, 31(4):614–627.

Strang, G. and Nguyen, T. (1997). *Wavelets and Filter Banks.* Wellesley-Cambridge Press, second edition.

Stuckens, J., Coppin, P. R., and Bauer, M. E. (2000). Integrating contextual information with per-pixel classification for improved land cover classification. *Remote Sensing of Environment*, 71(2):82–96.

Sulsoft (2003). AsterDTM 2.0 installation and user's guide. Technical report, SulSoft Ltd, Porto Alegre, Brazil.

Tao, C. V. and Hu, Y. (2001). A comprehensive study of the rational function model for photogrammetric processing. *Photogrammetric Engineering and Remote Sensing*, 67(12):1347–1357.

Teague, M. (1980). Image analysis by the general theory of moments. *Journal of the Optical Society of America*, 70(8):920–930.

Teillet, P. M., Guindon, B., and Goodenough, D. G. (1982). On the slope-aspect correction of multispectral scanner data. *Canadian Journal of Remote Sensing*, 8(2):84–106.

Theiler, U. and Matsekh, A. M. (2009). Total least squares for anomalous change detection. Technical report LA-UR-10-01285, Los Alamos National Laboratory.

Tondewad, P. S. and Dale, M. P. (2020). Remote sensing image registration methodology: Review and discussion. *Procedia Computer Science*, 171:2390–2399. https://doi.org/10.1016/j.procs.2020.04.259.

Tran, T. N., Wehrens, R., and Buydens, L. M. C. (2005). Clustering multispectral images: A tutorial. *Chemometrics and Intelligent Laboratory Systems*, 77:3–17.

Tsai, V. J. D. (1982). Evaluation of multiresolution image fusion algorithms. In *Proceedings of the International Symposium on Remote Sensing of Arid and Semi-Arid Lands, Cairo, Egypt*, pages 599–616.

van Niel, T. G., McVicar, T. R., and Datt, B. (2005). On the relationship between training sample size and data dimensionality: Monte Carlo analysis of broadband multi-temporal classification. *Remote Sensing of Environment*, 98(4):468–480.

von Luxburg, U. (2006). A tutorial on spectral clustering. Technical report TR-149, Max-Planck-Institut für biologische Kybernetik.

Vrabel, J. (1996). Multispectral imagery band sharpening study. *Photogrammetric Engineering and Remote Sensing*, 62(9):1075–1083.

Wang, Z. and Bovik, A. C. (2002). A universal image quality index. *IEEE Signal Processing Letters*, 9(3):81–84.

Weiss, S. M. and Kulikowski, C. A. (1991). *Computer Systems That Learn.* Morgan Kaufmann.

Welch, R. and Ahlers, W. (1987). Merging multiresolution SPOT HRV and LANDSAT TM data. *Photogrammetric Engineering and Remote Sensing*, 53(3):301–303.

Westra, E. (2013). *Python Geospatial Development (2nd edition).* Packt Publishing.

Wiemker, R. (1997). An iterative spectral-spatial Bayesian labeling approach for unsupervised robust change detection on remotely sensed multispectral imagery. In *Proceedings of the 7th International Conference on Computer Analysis of Images and Patterns*, volume LCNS 1296, pages 263–370.

Winkler, G. (1995). *Image Analysis, Random Fields and Dynamic Monte Carlo Methods.* Applications of Mathematics. Springer.

Wu, T.-F., Lin, C.-J., and Weng, R. C. (2004). Probability estimates for multi-class classification by pairwise coupling. *Journal of Machine Learning Research*, 5:975–1005.

Xie, H., Hicks, N., Keller, G. R., Huang, H., and Kreinovich, V. (2003). An ENVI/IDL implementation of the FFT-based algorithm for automatic image registration. *Computers and Geosciences*, 29:1045–1055.

Yang, X. and Lo, C. P. (2000). Relative radiometric normalization performance for change detection from multi-date satellite images. *Photogrammetric Engineering and Remote Sensing*, 66(8):967–980.

Yocky, D. A. (1996). Artifacts in wavelet image merging. *Optical Engineering*, 35(7):2094–2101.

Zhu, X. X., Tuia, D., Mou, L., Xia, G.-S., Zhang, L., Xu, F., and Fraundorfer, F. (2017). Deep learning in remote sensing. *IEEE Geoscience and Remote Sensing Magazine*, December:8–36.

Index