MACHINE LEARNING IMPRECISELY

THE COMPLEMENT OF PRECISION IS CLARITY

DAN WITZNER HANSEN

Lecture notes for Introduction to Machine Learning
2024

Preface

This note serves as auxiliary material for the Introduction to Machine Learning course at ITU. It assumes that readers are already familiar with vectors in 2D and 3D, as well as differentiation of single variables as taught in level A mathematics at Danish high schools. The content of this document is partially contributed by Andrew Duchowski, Clemson, with permission granted to Dan Witzner Hansen for utilization and modification in the Introduction to Machine Learning course in 2023 nd 2024. Any additional use or distribution of this material is strictly prohibited. Please note that this document has utilized chatGPT for language improvement suggestions.

Throughout the semester, this document will undergo continuous updates, so it is essential to download the latest version regularly. Its primary purpose is to provide supplementary material to help students better comprehend the concepts covered in other resources.

1

Overview of Scalars, Vectors, Matrices, and Tensors

Understanding properties and operations on vectors, matrices and tensors is crucial in numerous fields, including machine learning, engineering, game development, and physics. Throughout the course, these representations are utilized in conjunction with selected topics from linear algebra to establish a formal and practical foundation to comprehend key concepts in machine learning. Other books will be used to introduce and formalize linear algebra. Images are used in this course to illustrate aspects of machine learning. Chapter 2 provides a gentle introduction to representation of images and videos using vectors, matrices and tensors however the same principles also applies to other data types. In this course, tensors primarily serve as a higher order data representation and the mathematical operations on tensors is beyond the scope of the course.

This chapter provides a brief introduction to the fundamental representation of scalars, vectors, matrices, and tensors, and their role in representing and manipulating different types of data, such as sound, images, and videos. A visual representation of a scalar, a vector, a matrix and a tensor is illustrated in Figure 1.1.

SCALARS Scalars are single values and can represent quantities such as mass, temperature, or time. Scalars are often denoted by lowercase letters, e.g., a, b, c. Scalar operations typically involve arithmetic operations like addition, subtraction, multiplication, and division, as commonly learned in primary school.

VECTORS Vectors are one dimensional arrays of scalars and may be considered to have magnitude and direction. They can represent physical quantities like displacement, velocity, or force. Vectors are denoted by boldface lowercase letters, e.g., $\mathbf{v}$, $\mathbf{u}$, $\mathbf{w}$. A n-dimensional real valued vector $\mathbf{x} \in \mathbb{R}^n$ can be represented as an ordered collection of scalar components, usually in Cartesian coordinates or other coordinate systems (as we will discuss further in the next lectures). Vectors will later be used to store data and are often called *feature vectors*. Vector operations include addition, subtraction, dot product, cross

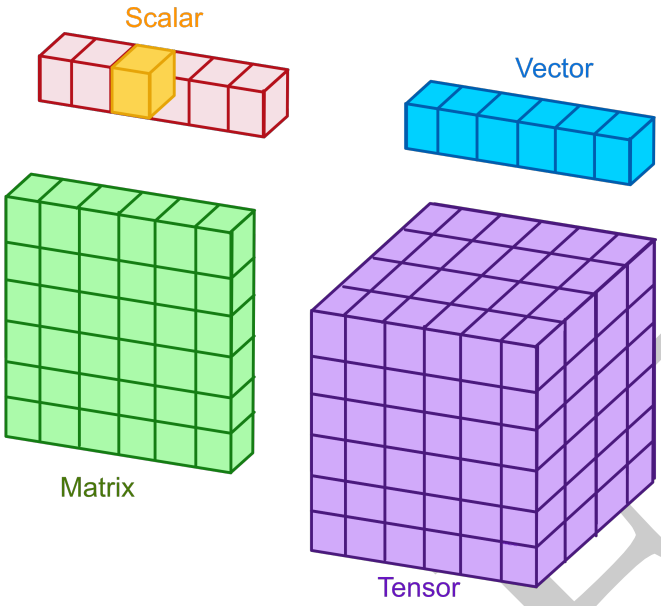

Figure 1.1: Visual representation of mathematical entities: a scalar (a single value), a vector (an array of scalars), a matrix (an array of arrays of scalars), and a tensor (a multi-dimensional array of scalars)

product, and scalar multiplication, as known from high school levels.

MATRICES Matrices are two-dimensional, $M \times N$, arrays of scalars arranged in rows and columns. Matrices are denoted by uppercase letters, e.g. A, B, $C \in \mathbb{R}^{M \times N}$. Each element in a matrix is a scalar. Matrix operations include addition, subtraction, matrix multiplication, and matrix inversion. They are used to represent images, transformations, systems of linear equations, and more. Operations involving scalars, vectors, and matrices will be thoroughly covered during the upcoming weeks when we delve into the topic of linear algebra.

TENSORS Tensors are multidimensional arrays that generalize scalars, vectors, and matrice and are represented by boldface uppercase letters, e.g., $\mathbf{T}$, $\mathbf{S}$, $\mathbf{R}$. For example, a $(M \times N \times k)$ tensor forms a three-dimensional cube of $M \times N \times k$ scalars. Tensors can have any number of dimensions. The dimensions of a tensor are denoted as

$$\overbrace{N_1 \times N_2 \times \cdots \times N_k}^{\text{Tensor dimensions}}.$$

A 4 dimensional tensor $\mathbf{T}$ of real numbers is denoted $\mathbf{T} \in \mathbb{R}^{N_1 \times N_2 \times N_3 \times N_4}$. Tensors are used to represent multi-channel images, video sequences and much more. Each element in a tensor can be a scalar, vector, or another tensor. Tensor operations involve various types of multiplications, contractions, and transformations. While extensive formalization or operations on tensors are beyond the scope of the course, studying tensors independently after-

wards is highly valuable, as they constitute the core representation in modern machine learning frameworks like PyTorch.

Indexing Indexing vectors, matrices, and tensors, is used to get access their elements. Individual elements in vectors can be accessed using a single index value that ranges from 1 to the length of the vector[1]. For example, the element at index 3 of a vector $\mathbf{v}$ can be retrieved by referring to it as $\mathbf{v}_3$. Matrices require two indices (e.g. i and j) to locate an element, A_{ij}. The first index, i corresponds to the row number, while the second index represents the column number. Thus, an element at row 2 and column 4 of a matrix A can be accessed as $A_{2,4}$. Tensors require corresponding indices for each dimension to access specific elements. For instance, in a 3D tensor $\mathbf{T}$, an element at coordinates $(2, 3, 1)$ can be accessed as $\mathbf{T}_{2,3,1}$.

Sorry! To avoid confusion with most material, it's worth noting that historically, indexing conventionally does not use 0-indexing).

Flattening Machine learning methods typically assume data to be in vector form $x \in \mathbb{R}^N$. Flattening involves mapping signals (e.g. matrices, vectors or other structures) to vectors. For example, in the case of images, the pixel intensities or color values are mapped to a one-dimensional vector by concatenating the pixel values row-wise or column-wise as shown in Example 1.1.

Example 1.1: Vectorization

Let $A \in \mathbb{R}^{2\times 3}$ be a 2×3 matrix.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathbb{R}^{2\times 3}$$

The (row)vectorized vector $\mathbf{a} \in \mathbb{R}^6$

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$$

The (column) vectorized vector $\mathbf{b} \in \mathbb{R}^6$

$$\mathbf{b} = \begin{bmatrix} 1 \\ 4 \\ 2 \\ 5 \\ 3 \\ 6 \end{bmatrix}$$

Vectorization provides several benefits. It allows us to leverage techniques developed for vector-based data, enabling efficient computations and analysis. Operations such as matrix multiplication, vector addition, and dot products, which are fundamental operations in linear algebra and signal processing are particularly useful. Additionally, vectorization facilitates the integration of signals (*data fusion*) with other data types by concatenating their respective vectors.

2

Image basics

This chapter provides a brief introduction to images and their representation.

2.1.0 Image representation

In the process of capturing an image, light reflected from the scene in front of the image sensor passes through optics and the aperture, and is measured by a 2D image sensor. There are various methods to measure the incoming light, but ultimately, the image sensor converts the measured light into a $M \times N$ matrix of gray scale or $M \times N$ color pixel values, as illustrated in Figure 2.2. Each pixel value corresponds to a discrete sample of the image's intensity or color.

Image formation Before capturing an image, all sensor cells are reset to ensure that no residual light charges are present. Subsequently, light is allowed to enter the sensor, and the cells start accumulating the incident light. After a specific duration of time, known as the *exposure time*, the incident light is blocked by the *Shutter*. The accumulated light measured by each cell is then converted into the corresponding pixel value. For example, in a color camera, the amount of red, green, and blue light is typically measured.

Resolution The number of pixels used to represent an image is referred to as its *spatial resolution*. Higher spatial resolution corresponds to a larger number of pixels, providing finer details in the image. Conversely, lower spatial resolution implies a relatively smaller number of pixels. The terms "fine" and "coarse" resolution are sometimes used. The resolution would need to be infinite to ensure perfect preservation of the objects in front of the censor. Hence we should rely on some uncertainty of both shapes and color/intensity in the measured images.

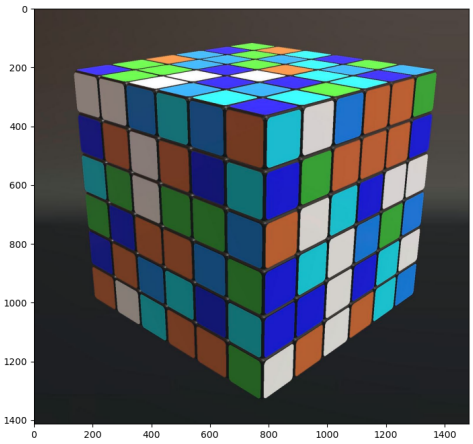

Figure 2.1:

Coordinates As shown in Figure 2.2 different coordinate systems may be used to represent pixel values. In representations where images are treated as matrices, pixel coordinates are specified as (row, column), with the origin $(0, 0)$ placed in the upper left corner. This convention means that the x-coordinate increases towards the right, while the y-coordinate increases downwards. A pixel value can be accessed as $I(r, c)$, where r is the row number and c the column number. On the other hand, the Euclidean coordinate system typically places the origin in the lower-left corner. In this system, the x-coordinate increases when moving to the right, and the y-coordinate increases when moving up. The conversion between coordinate systems where the image origin is in the top-left and Euclidean coordinates in the lower-left can be achieved using the following equations:

$$x_{\text{euclidean}} = x_{\text{image}} \quad \text{(no change)}$$
$$y_{\text{euclidean}} = M - 1 - y_{\text{image}}$$

where $x_{\text{euclidean}}$ and $y_{\text{euclidean}}$ represent the coordinates in the Euclidean system, while x_{image} and y_{image} represent the coordinates in the image system. M is the height of the image in pixels.

Quantization *Quantization* (of colors and intensities) is needed to reduce the continuous range of possible intensity values to a finite set of discrete levels that can be processed, stored, and displayed by computers and imaging devices. Quantization introduces a loss of information and can result in a loss of image quality, particularly in cases where the original continuous intensity values need to be accurately preserved. The choice of quantization levels or bit depth in an

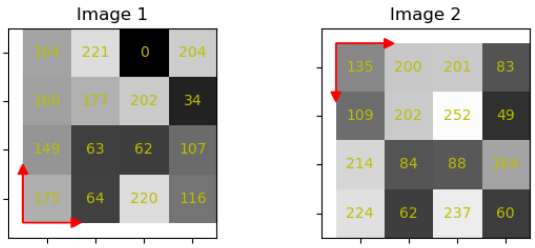

Figure 2.2: (Left) Cartesian and (right) matrix coordinates for gray-scale images. The intensity of each pixel is displayed in each cell.

image affects the trade-off between image quality and storage requirements. Higher bit depths allow for more precise representation of intensity/color values but require more storage space. Lower bit depths reduce storage requirements but result in a coarser representation of intensity levels and potential loss of fine details in the image.

Cells in the imaging sensor might accumulate charges exceeding the maximum measurable charge resulting in an overexposed image. Such cells are typically quantized to the maximum value, 255. In this situation, it becomes impossible to determine the precise amount of incident light in that cell. Care must be taken to avoid this scenario by appropriately setting the shutter and aperture. Saturated cells can affect neighboring pixels by also increasing their charges, resulting in a phenomenon called *blooming*. Additionally, when an object in front of the image sensor is in motion, the light from a particular point on the object may spread out over multiple cells. Hence, a shorter exposure time is generally required to avoid motion blur.

Binary, Grayscale, Color images and Video sequences Gray-level is typically specified in terms of the number of bits. Common gray-level resolutions include 8, 10, and 12 bits, corresponding to 256, 1024, and 4096 gray levels, respectively. 8-bit images are the most prevalent, allowing pixel values to range from black (0) to white (255). Binary images have two possible pixels values usually (0 / 1) or (0 / 255). A pixel, $I(x, y)$ in a color image can be represented as a three-dimensional vector, $[r, g, b]^\top$ containing the amount of red (r), green (g) and blue (b). An 8-bit color image can represent $256^3 = 16,777,216$ different colors.

Alternatively, a color image can be represented by a $M \times N \times 3$ tensor which can be thought of as containing 3 gray-scale images (planes), each representing the measured amount of red, green, or blue light, respectively as shown in Figure 2.3.

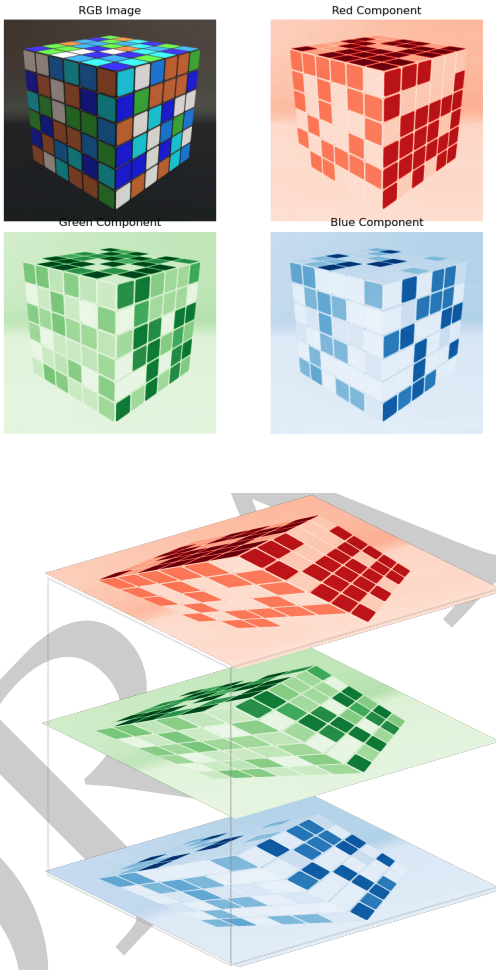

Figure 2.3: (top)Color image and each of the R,G,B color planes. Notice each of these are gray scale images but are here given colors to signify the colorplane. (bottom)Viewing the images as a tensor of RGB image planes.

Image arithmetic

Multi-channel images In certain applications, it becomes beneficial to measure multiple spectral wavelengths individually. These images, known as *multi-spectral* images or *multichannel images*, capture information from various spectral

bands or different measuring sources such as color and depth. Consequently, each pixel contains multiple values $\left[i_1, i_2, \cdots, i_k\right]$. A multi-spectral image can also be considered containing k separate gray scale images (*image planes*) and may be represented by $(M \times N \times k)$ tensor $\mathbf{I}$.

Video Sequence Representation A video sequence is a collection of consecutive frames captured over time. Similar to individual images, each frame in a video sequence can be represented as a matrix or a tensor, depending on whether it is a grayscale or color frame. The representation of a video sequence can be seen as a 3D or 4D tensor (depending on whether it is a color or gray scale image sequence), where the dimensions correspond to the frame index, width, height, and color channels. Each element of the tensor represents the intensity of a specific color channel at a particular pixel in a specific frame. A video sequence o of length T with color images can be represented as a $(M \times N \times 3 \times T)$ tensor. A somewhat correct visualization is shown in Figure 2.4.

Figure 2.4: Color video sequence in "tensor" form. Notice that the illustration is somewhat misleading as each image in the figure actually represent 3 planes.
https://www.mobiles24.co/downloads/s/590592-106-donald_duck_-the_hockey_champ_part_1

2.2.0 Color spaces

Color plays a crucial role in machine learning, image analysis and visual perception of humans. Color spaces provide a systematic way to represent and interpret colors in various coordinate systems. This section describes a few color spaces and delve into the mathematical details of transforming (a.k.a. converting) values between color spaces. The section is related to concepts about vector spaces and transformations that will be covered later in the course.

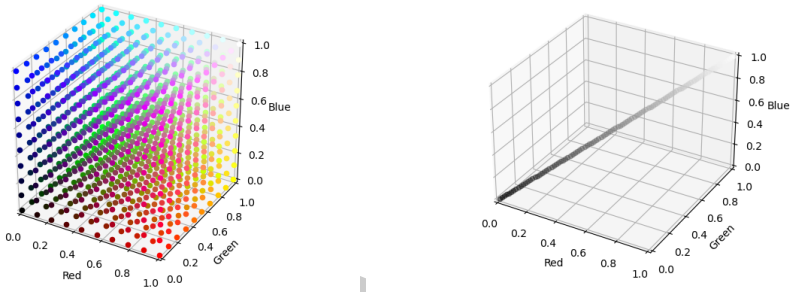

Figure 2.5: (left) The RGB color cube with discrete representation of colors. Each dot corresponds to a particular pixel value. Multiple dots on the same line all have the same color but different levels of illumination (right) gray scale values in the RGB color cube.

The RGB (Red-Green-Blue) color space, shown in Figure 2.5, represents colors by specifying the intensity levels of red, green, and blue primary colors along each direction. For example, RGB colors can be represented as vectors $\left[R, G, B\right]^\top$, where R, G, and B represent the intensity values of the red, green, and blue channels, respectively. Hence a color be represented as a point in a 3D space spanned by the three colors e.g. red along the x-axis, green along the y-axis and blue along the z-axis. As shown in Figure 2.5, the 16 milion colors in an 8 bit image color image can consequently be represented as points in a *color-cube* (left) and where the pure colors are in the corner. The vector on the diagonal from $[0, 0, 0]^\top$ to $[255, 255, 255]^\top$ passes through all gray-scale values and is denoted the *gray-vector*.

2.2.1 Gray scale conversion

There are instances where utilizing only the intensity (gray scale) information of the image can prove advantageous. This choice is often motivated by considerations of time efficiency, storage requirements, and the fact that the intensity image frequently encompasses the most crucial and meaningful information.

It consequently becomes necessary to convert the color image into a grayscale representation.. The process of converting from RGB to grayscale can be simply by using a weighted average (linear combination) of the r,g and b components of the color pixel $\mathbf{c}$ ($\mathbf{c} = [c_r, c_g, c_b]$) :

$$I = w_r \cdot f_r + w_g \cdot c_g + w_b \cdot c_b \tag{2.1}$$

Here, I = represents the intensity of the grayscale image, while w_r, w_g, and w_b are weight factors associated with the red (r), green (g), and blue (b) channels respectively. You may recognize Equation 2.1 as the inner / dot product (later denoted $\mathbf{w}^\top \mathbf{c}$) between the vectors $\mathbf{c} = [c_r, c_g, c_b]$ and $\mathbf{w} = [w_r, w_g, w_b]^\top$.

A common choice is to use equal importance to all three colors, $w_r = w_g = w_b = \frac{1}{3}$ but other weight factors may be needed to enhance specific channels. It is important to note that these weight factors should sum to one $w_r + w_g + w_b = 1$ so the intensity stays within limits (e.g. 8 bits). The significance of colors may vary depending on the specific application. In such cases, it is common to adjust the weight factors accordingly. For example, in image analysis on vegetation data, the green color often contains the most informative content, while metal object images tend to have crucial details primarily in the blue pixels. Similarly, when detecting human skin tones (face and hands), a higher weight may be assigned to the red channel. Determining suitable weights can be aided by analyzing the histograms of each color channel or learned through a machine learning model $f_w(x)$.

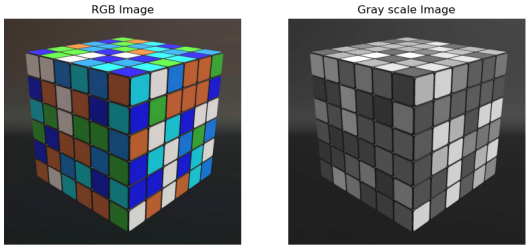

Figure 2.6: (Left) RGB image and (right) the corresponding gray-scale converted image using the weight matrix $\mathbf{w} = [0.299, 0.587, 0.114]$.,

Human visual perception is commonly employed to determine the weights for converting color to grayscale in order to achieve optimal visualization. Major international standardization organizations in the fields of TV, image, and video coding have reached a consensus on a set of weights that strike a satisfactory balance in terms of optimizing visual perception. These weights are typically $\mathbf{w} = [w_r, w_g, w_b] = [0.299, 0.587, 0.114]$. The resulting grayscale value obtained from the linear weighing (dot product) is commonly referred to as "luminance." as shown in Figure 2.6

Transformation is non-invertible It is important to note that he color information is lost during the color-to-grayscale transformation and that once an image has been converted to grayscale, it is generally not possible to revert it back to its original color (e.g. there does not exist an inverse since the transformation is surjective (on-to) and non-injective (one-to-one)).

2.2.2 The Normalized RGB Color Representation

Figure 2.7 shows the RGB color cube and two lines. Observe that the red points $\begin{bmatrix}0,50,0\end{bmatrix}^\top$, $\begin{bmatrix}0,100,0\end{bmatrix}^\top$, and $\begin{bmatrix}0,223,0\end{bmatrix}^\top$ lie on the same line, through the origin, and have the same color (green) but have different levels of illumination. This concept applies to the rest of the color cube as well. For example, the points $\begin{bmatrix}20,240,44\end{bmatrix}^\top$, $\begin{bmatrix}10,120,22\end{bmatrix}^\top$, and $\begin{bmatrix}5,60,11\end{bmatrix}^\top$ also lie on a line and have the same color but with varying levels of illumination.

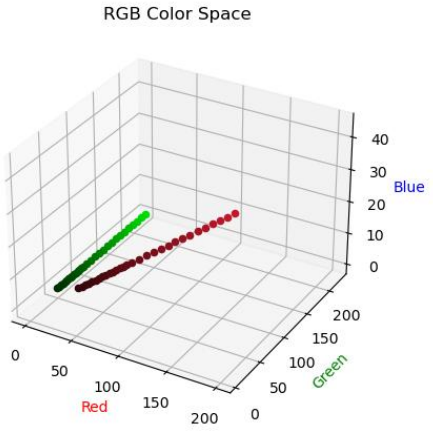

Figure 2.7: RGB color space with sample point drawn from two straight lines.

As shown in Figure 2.8 (left) these are general observations that points on

lines passing through the triangle defined by the points $\left[1,0,0\right]^\top$, $\left[0,1,0\right]^T$, and $\left[0,0,1\right]^\top$ will have the same color but different illumination. The coordinates of the point $[R,G,B]$ where a line intersects the triangle can be found through *normalization*

$$[r,g,b] = \left[\frac{R}{R+G+B}, \frac{G}{R+G+B}, \frac{B}{R+G+B}\right] \tag{2.2}$$

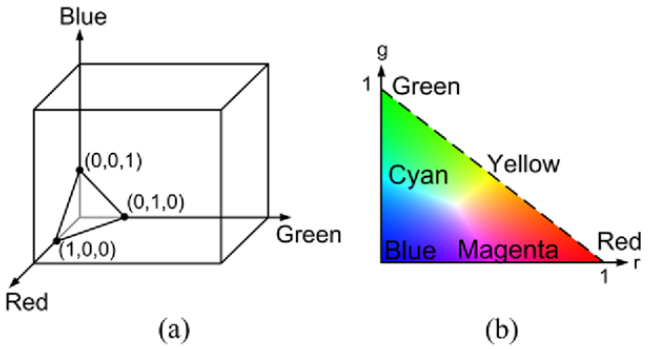

Figure 2.8: Left: The triangle where all color vectors pass through. The value of a point on the triangle is defined using normalized RGB coordinates. Right: The chromaticity plane.

In the normalized RGB color representation, each value falls within the interval $[0,1]$, and the sum of $r+b+c = 1$ (r, g, b). This equation enables the representation of a normalized RGB color using just two values. For instance, by selecting r and g, the triangle in Figure 2.8 (left) can be represented as the chromaticity plane depicted on the right side of Figure 2.8. The edges of this triangle correspond to the pure colors, while colors closer to the center exhibit less purity and eventually become shades of gray. The proximity to the center indicates a higher degree of "pollution" by white light.

Program 2.2.1 shows the conversion from (r,g,b) to (r,g,I) by iterating over all the pixels. Program 2.2.2 performs the inverse transformation from (r,g,I) to (r,g,b).

Program 2.2.1: Convert to normalized values

```
import numpy as np
from PIL import Image

# Load image
image = Image.open('image.jpg')
img = np.array(image)

```

```
# Get image dimensions
M, N, _ = img.shape

# Create output array
out = np.zeros_like(img, dtype=float)

# Perform color channel normalization
for y in range(M):
        for x in range(N):
                total = img[y, x, 0] + img[y, x, 1] + img[y, x, 2]
                out[y, x, 0] = img[y, x, 0] / total
                out[y, x, 1] = img[y, x, 1] / total
                out[y, x, 2] = total / 3

# Convert output array back to image
out_img = Image.fromarray((out * 255).astype(np.uint8))

# Display the images
image.show()  # Show original image
out_img.show()  # Show normalized image
```

Program 2.2.2: From normalized RGB to RGB

```
        # Input image is img, output is out
        for y in range(M):
                for x in range(N):
                        total = 3 * img[y, x, 2]
                        out[y, x, 0] = total * img[y, x, 0]
                        out[y, x, 1] = total * img[y, x, 1]
                        out[y, x, 2] = total * (1 - img[y, x, 0] -
                            img[y, x, 1])
```

2.2.3 Hue, Saturation and Value

Hue and saturation are fundamental concepts in color representation. Instead of representing color in the Cartesian coordinates of RGB color space, colors can be conveniently defined on a circular planar area that is orthogonal to the gray vector, as depicted in Figure 2.10.

The relationship between hue and saturation can be observed by using at the center c of the triangle as reference as shown in Figure 2.10. The third component needed to preserve all RGB colors in the new space is called intensity / value. The HSV (Hue, Saturation, Value) color representation captures the characteristics of colors based on their hue, saturation, and intensity values.

Given a color, p, within the normalized triangle in Figure 2.10

HUE is defined as the angle, θ, between the vectors $\overrightarrow{cr}$ and $\overrightarrow{cp}$, where cr is the direction of the red color. Hue represents the dominant wavelength of light and corresponds to the pure colors located on the edges of the triangle. For instance, a hue of $0°$ corresponds to red, while a hue of $120°$ corresponds to green. Hue is undefined for gray-values e.g. when $r = g = b$. That is

$$\theta = \frac{\overrightarrow{cr} \cdot \overrightarrow{cp}}{||\overrightarrow{cr}|| \, ||\overrightarrow{cr}||}, \tag{2.3}$$

where $a \cdot b$ is the inner product between vectors a and b.

SATURATION represents the purity of a color and is defined as the relative distance

$$\frac{||\vec{cp}||}{||\vec{cp'}||}$$

where $||\cdot||$ denote the length of the vector. Observe that the point c corresponds to $[r, g]^\top = [1/3, 1/3]^\top$ and is "colorless". A point located on the edge of the triangle has maximum saturation, indicating a pure color. Saturation gradually decreases as colors get closer to the center of the triangle. At the center, the saturation is 0, resulting in a color that appears as a shade of gray. Saturation is defined to be zero when the color is black $[r, g, b] = [0, 0, 0]$.

VALUE / INTENSITY is defined as the average of the RGB values e.g. $I = (r + g + b)/3$.

2.2.4 Transformations Between Color Spaces

Colors can be transformed between RGB and HSV colorspaces through a *transformation* $x' = f(x)$. For example a RGB color $x = [R, G, B]^\top$ vector can be (non-linearly) be transformed to a $x' = [H, S, V]$ coordinate in HSV space as shown in Equation 2.6 and with corresponding python example in program 2.2.1.

$$H = \begin{cases} \arccos\left(\frac{1}{2}\frac{(r-g)+(r-b)}{\sqrt{(r-g)^2+(r-b)(g-b)}}\right), & \text{if } g \geq b; \\ 360° - \arccos\left(\frac{1}{2}\frac{(r-g)+(r-b)}{\sqrt{(r-g)^2+(r-b)(g-b)}}\right), & \text{otherwise.} \end{cases} \tag{2.4}$$

$$S = 1 - 3\frac{\min(r, g, b)}{r + g + b} \tag{2.5}$$

$$V = \frac{r + g + b}{3} \tag{2.6}$$

The inverse transformation $f^{-1}(x') = x$ from HSV to RGB is given in equations 2.10

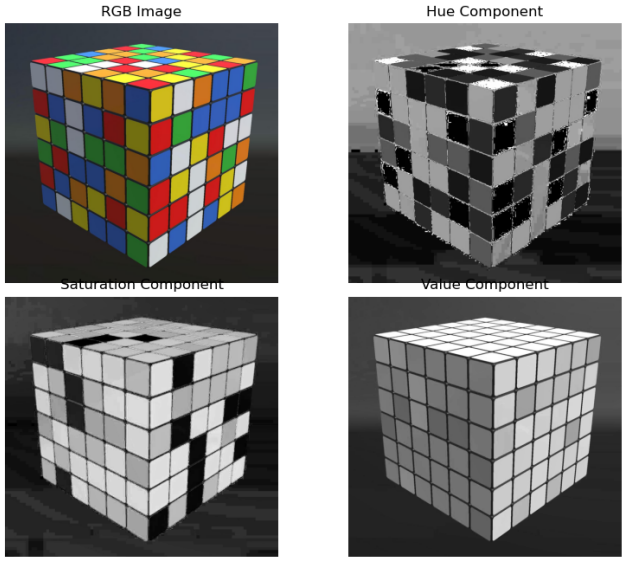

Figure 2.9: (Top left) Color image and its composition into (Top right) Hue, (Bottom left) Saturation and (Bottom right) Value components. Notice how the red colors may be split into bright or dark values depending on which side of the 'red' they lie.

$$H_n = \begin{cases} 0 & \text{if } 0° \leq H \leq 120° \\ H - 120° & \text{if } 120° \leq H \leq 240° \\ H - 240° & \text{if } 120° \leq H \leq 360° \end{cases} \tag{2.7}$$

$$r = \begin{cases} I\left(1 + \frac{S\cos H_n}{\cos(60° - H_n)}\right) & \text{if } 0° \leq H \leq 120° \\ I - IS & \text{if } 120° \leq H \leq 240° \\ 3I - g - b & \text{if } 120° \leq H \leq 360° \end{cases} \tag{2.8}$$

$$g = \begin{cases} 3I - r - b & \text{if } 0° \leq H \leq 120° \\ I\left(1 + \frac{S\cos H_n}{\cos(60° - H_n)}\right) & \text{if } 120° \leq H \leq 240° \\ I - IS & \text{if } 120° \leq H \leq 360° \end{cases} \tag{2.9}$$

$$b = \begin{cases} I - IS & \text{if } 0° \leq H \leq 120° \\ 3I - r - g & \text{if } 120° \leq H \leq 240° \\ I\left(1 + \frac{S\cos H_n}{\cos(60° - H_n)}\right) & \text{if } 120° \leq H \leq 360° \end{cases} \tag{2.10}$$

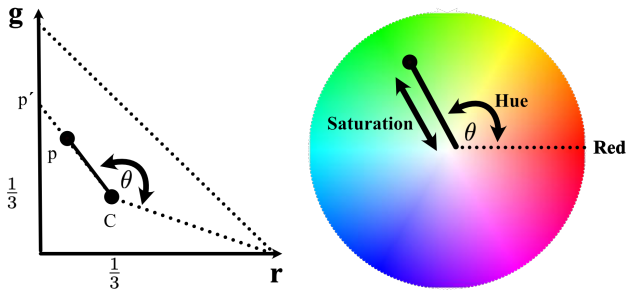

Figure 2.10: HSV color space relation to Normalized RGB colors

Program 2.2.3: Convert RGB to HSV

```
....
# Input image is img,
for y in range(M):
        for x in range(N):
                r = img[y, x, 0]
                g = img[y, x, 1]
                b = img[y, x, 2]

                # Calculate intensity
                intensity = (r + g + b) / 3

                # Calculate hue
                numerator = 0.5 * ((r - g) + (r - b))
                denominator = np.sqrt((r - g) ** 2 + (r - b) *
                    (g - b))
                hue = np.arccos(numerator / denominator)
                if b > g:
                hue = 2 * np.pi - hue

                # Calculate saturation
                min_val = min(r, g, b)
                saturation = 1 - (3 * min_val / (r + g + b))

                # Update HSI image
                HSV[y, x, 0] = hue
                HSV[y, x, 1] = saturation
                HSV[y, x, 2] = intensity
```

2.3.0 Image arithmetic

2.4.0 Image Histograms and Probability

Image histograms provide valuable insights into the underlying image characteristics. This section briefly describe the concept of image histograms and their relationship to probability. The section discusses how image histograms can be used to analyze the distribution of pixel intensities in both color and grayscale images.

An image histogram provides a visual summary of the frequency intensity levels or colors. Mathematically, an image histogram is defined as a function $H(i)$ that represents the number of pixels in the image with intensity i. Hence, H can be thought of as using a vector to store the number of pixels in the image with a particular intensity or color. When provided with an index, H returns the content of the bin with index i which contains the number of pixels with intensity i

For a grayscale image, the intensity i ranges from 0 to 255. In the case of color images, separate histograms are often calculated for each color channel, such as red, green, and blue (as shown in Figure 2.11). However, a more accurate representation is to have bins for each color combination, resulting in a joint histogram that captures the correlation between colors. Figure 2.11 shows the histograms of the red, green, and blue channels, and the grayscale converted image.

While having around 16 million bins for the joint histogram may be impractical, bins are often defined to represent intervals of colors. This allows for a more manageable number of bins while still capturing the overall color distribution.

It is important to note that these individual histograms do not represent the joint histogram of the 16 million possible color combinations, but rather provide insights into the distribution of each channel and the overall grayscale intensity.

Brightness and Contrast The shape and distribution of the histogram can be examined to assess the overall brightness and contrast of an image. A histogram skewed towards lower intensities suggests an overall darker image, while a skew towards higher intensities indicates a brighter image. The spread or width of the histogram indicates the contrast level, with a wider histogram representing higher contrast.

Peaks in the histogram correspond to prominent intensity levels present in the image. These peaks can indicate specific features or objects with distinct intensities. Peak detection algorithms can be employed to identify and extract such features from the image.

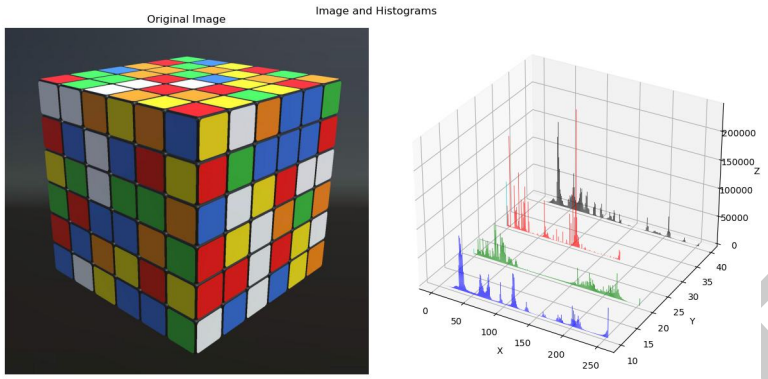

Figure 2.11: Histograms of the individual red, green, and blue channels, and the grayscale converted image.

Probability Density Function (PDF) Histograms and probabilities are closely related. The probability distribution function (PDF) represents the likelihood of a pixel having a specific intensity value. In the context of image histograms, the PDF is calculated by normalizing the histogram, which involves dividing each bin value by the total number of pixels in the image.

$$PDF(i) = \frac{H(i)}{N}$$

where $PDF(i)$ represents the probability of a pixel having intensity value i, $H(i)$ is the number of pixels with intensity value i in the histogram, and N is the total number of pixels in the image. This normalization process ensures that the PDF represents a valid probability distribution, with values ranging from 0 to 1, where the sum of all probabilities is equal to 1.

The PDF provides insights into probability of encountering different intensity values in the image from which it is possible to analyze and understand the distribution of pixel intensities. The cumulative density function (CDF) is derived from the PDF and represents the cumulative probability of pixel intensities up to a given value.

2.5.0 Important Takeways

In combination with chapter 1, this chapter introduces fundamental mathematical concepts and data types that will be revisited in various contexts throughout the course. We begin by introducing the concept of vectors. In the context of data representation, these vectors are referred to as *feature vectors*, used to represent data points, such as color information. As we progress through the course,

feature vectors will be formalized and extended to N-dimensional vectors, enabling us to handle a broader range of data.Vectorization plays a crucial role in transforming various types of data, including matrices and tensors, into a format suitable for machine learning applications. In machine learning formalisms, these transformed data structures are often represented as vectors.

Furthermore, we touch upon the use of functions to transform vectors from one coordinate system (or space) to another. While the transformation demonstrated in this chapter was non-linear, specifically between RGB and HSV color spaces, we will soon delve deeper into both linear and non-linear transformations. Linear functions, in particular, play a crucial role in introductory machine learning, and we will explore their properties more extensively. For now, it is sufficient to envision linear functions as those that preserve the alignment of points along a line, without bending or distorting the data.

The chapter also provides examples of both invertible and non-invertible functions. Specifically, we observe that it is possible to map between RGB and HSV color spaces, allowing us to transform color representations. Additionally, we can map points from the color space to grayscale, but the reverse mapping is not feasible.This example highlights the possibility of functions being *non-invertible*, which means that going back and forth between representations may not be feasible. Intuitively, this implies a loss of information during the transformation process. As we advance in the course, we will explore the concept of invertibility in depth and its importance, particularly in the context of linear functions.

By integrating these foundational concepts and gradually expanding upon them, our goal is to provide you with a comprehensive understanding of images, color spaces, and equip you with a robust mathematical toolkit to effectively address the challenges of machine learning.

3

Transformations

> Functions describe the world.
> YouTube Video: Understanding Linear Transformations.

Whether explicitly specified or implicitly implied, there is a often a need to convert data into different values or formats. *Transformations* are functions that map data from one space to another. [1] These operations encompass a broad spectrum, ranging from simple adjustments of values to the fundamental reshaping of dimensions or the introduction of new components within vectors, matrices, and tensors. Transformations are important for machine learning, where they are used in various tasks, ranging from data preprocessing and camera projections, dimensionality reduction, geometric transformations, and the foundational aspects of neural network architectures. This section introduces several abstract mathematical concepts that may initially appear distant from practical applications. However, they will provide you with valuable tools for your mathematical toolkit. Do not worry if you do not fully comprehend all the details in the first reading. You can always return to this chapter for reference. The subsequent chapters will provide more concrete examples of the theoretical definitions presented in this chapter. Chapter 4 illustrates the concepts of 1D transformations through point processing and chapter 5 illustrates the concepts of 2D and 3D point transformations. It should be easier then to extend these concepts to transformations to N-dimensional inputs.

In these lecture notes, 'transformation' and 'mapping' are used interchangeably.

Recall Matrix Multiplication Before treating the topic of transformations formally lets consider matrix multiplication of vectors as previously described.

Consider a $M \times N$ matrix T and an N-dimensional input vector $\mathbf{x}$. The resulting output vector $\mathbf{y}$ obtained through matrix-vector multiplication is given

by

$$\mathbf{y} = T\mathbf{x}$$

The dimensions of the matrix T directly defines the nature of the transformation in terms of input and output dimensions and the type of transformation done by the matrix. The number of rows (M) in the matrix corresponds to the dimension of the outputs, while the number of columns corresponds to the (required) dimension of the inputs. The dimensions of T define the nature of the transformation matrix multiplication can consequently be understood as a transformation that maps vectors from an N-dimensional space to an M dimensional space. If $M > N$, the transformation is "expanding," potentially stretching or rotating the space. If $M < N$, the transformation is "collapsing," projecting the vector to a lower-dimension.

The interplay between the matrix's dimensions encapsulates how each element in the output vector is formed. By using the *column view* of matrix multiplication observe how the components of $\mathbf{y}$ becomes a linear combination of the columns of T, with coefficients given by the corresponding components of $\mathbf{x}$. To detail this a bit further let's denote the i-th components of $\mathbf{x}$ as x_i and $\mathbf{t}_i$ the i-th column of T such that

$$T = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{t}_1 & \mathbf{t}_2 & \cdots & \mathbf{t}_N \\ | & | & \cdots & | \end{bmatrix}$$

The matrix multiplication $\mathbf{y} = Tx$(column view) becomes

$$\begin{aligned} \mathbf{y} &= x_1\mathbf{t}_1 + x_2\mathbf{t}_2 + \ldots + x_N\mathbf{t}_N \\ &= \sum_{i=1}^{N} x_i\mathbf{t}_i \end{aligned} \tag{3.1}$$

Example 3.1: Transformation as linear combinations

Let

$$T = \begin{bmatrix} 1 & 2 \\ 0 & 2 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

The vector $\mathbf{y} = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 2 \end{bmatrix}$

3.1.0 Linear and non-linear transformations

Linear Transformations *Linear transformations* refers to a set of transformations that fulfill the linear properties described in Definition 3.1 and illustrated in Figure 3.1.

Definition 3.1: Linear Transformation

$T : \mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R}^{\mathbb{M}}$ is a *linear transformation* of $\mathbb{R}^{\mathbb{N}}$ onto $\mathbb{R}^{\mathbb{M}}$ when the following two properties hold true for all u and v in $\mathbb{R}^{\mathbb{N}}$, and for any scalar $c \in \mathbb{R}$.

1. $T(u+v) = T(u) + T(v)$
2. $T(cu) = cT(u)$

Linear transformations have the property that they map vectors from one vector space to another while maintaining the linearity of the original vector space. These are specific transformations that preserve vector addition and scalar multiplication.

$$\mathbf{y} = T(x) = T\mathbf{x}$$

Matrix multiplication as shown in Equation 3.1 encapsulates the essence of a linear transformation and reveals the relationship between input vectors and output vectors. Matrices fulfill the preservation of addition criterion because matrix addition directly corresponds to vector addition. When you add two matrices, the resulting matrix is formed by adding corresponding elements. This mirrors vector addition because each element in the resulting matrix is obtained by adding the corresponding elements in the input matrices. Matrices fulfill the preservation of scalar multiplication criterion because scalar multiplication of a matrix involves multiplying each element of the matrix by the scalar. This operation mirrors scalar multiplication of a vector, which involves multiplying each component of the vector by the scalar. When you apply a matrix transformation to a vector, it involves multiplying the matrix by the vector. This operation distributes the transformation across each component of the vector (as shown in Equation 3.1), which directly corresponds to preserving scalar multiplication. Linear transformations preserve straight lines, planes, hyperplanes, parallel lines and lines passing through the origin.

Note!

Informally, when a transformation is linear, it can be represented by a $M \times N$ matrix, T, such that an input vector $\mathbf{x} \in \mathbb{R}^N$ is mapped to a vector $\mathbf{y} \in \mathbb{R}^M$ through matrix multiplication

[2]

An example of a linear transformation $T(x) = y = 3x$. It can be verified

You can trust me on this point. However, it's important to note that this can also be demonstrated formally if needed.

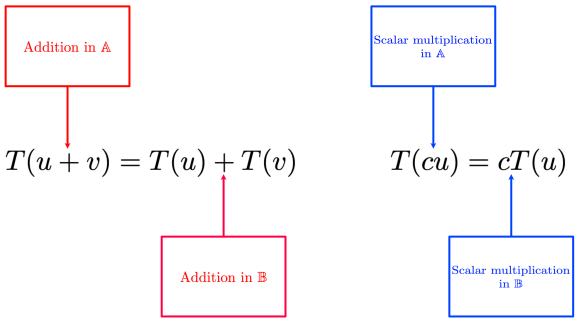

Figure 3.1: Linear transformations

that the two properties hold for this function (try showing it simply by using the definition 3.1). An example of a non-linear function $T(x) = y = \sin(x)$ as $\sin(x_1 + x_2) \neq \sin(x_1) + \sin(x_2)$ for all possible inputs of x_1 and x_2.

Theorem 3.1 shows (without proof) some important properties of linear functions that will become useful later in this chapter. A couple of examples are given in Example 3.5 and Example 3.6. In these examples, we can see that the transformation matrices have dimensions $M \times N$ where M is the dimensionality of the output space and N is the dimensionality of the input space, and they enable us to perform transformations between spaces with different dimensions.

Theorem 3.1: Properties of Linear Transformations

Let T be a linear transformation from $\mathbb{R}^{\mathbb{N}}$ into $\mathbb{R}^{\mathbb{M}}$, where $\mathbf{u}$ and $\mathbf{v}$ are vectors in $\mathbb{R}^N$. Then the properties listed below are true.

1. $T(\mathbf{0}) = \mathbf{0}$
2. $T(-\mathbf{v}) = -T(\mathbf{v})$
3. $T(\mathbf{u} - \mathbf{v}) = T(\mathbf{u}) - T(\mathbf{v})$
4. If $\mathbf{v} = c_1v_1 + c_2v_2 + \cdots + c_nv_n$, is a linear combination then

$$\begin{aligned} T(\mathbf{v}) &= T(c_1v_1) + T(c_2v_2) + \cdots + T(c_nv_n) \\ &= c_1T(v_1) + c_2T(v_2) + \cdots + c_nT(v_n) \end{aligned}$$

The first and last properties of transformations are important to note. The first property means that a transformation preserves the special point 0 (the zero vector) and doesn't change it. The last property indicates that a transformation also preserves points or vectors that lie on a line or a hyperplane. For instance, the transformation $T(x) = ax + b$ is nonlinear because it doesn't preserve 0

(as $T(0) = b$) e.g. it is in conflict with property 1 in theorem 3.1. However, this function is an instance of anther type of transformations that will be treated more formally below, namely *affine transformations*.

3.2.0 Non-linear Transformations

Non-linear transformations are functions that do not preserve vector addition and scalar multiplication (in Definition 3.1). These transformations can generally not be represented by a single matrix. Non-linear transformations often do not preserve straight lines and have more complex behavior, involving curvature, bending, and warping of space. The departure from linearity is often manifested through the violation of properties such as additive relationships, scalar multiplicative properties, or the preservation of the origin and may be achieved simply through vector addition [3]. Examples of non-linear transformations include polynomial functions, trigonometric functions, exponential functions, and other functions that cannot be expressed solely with matrix multiplication. Non-linear functions often involve higher-order terms and cross-terms, leading to more complex relationships between the variables and the function's output. As a result, increasing the inputs, $\mathbf{x}$, may not produce a proportional increase (as stated in theorem 3.1)in the output $\mathbf{y} = T(\mathbf{x})$, making non-linear functions versatile and capable of modeling a wide range of phenomena. However, as you will see later, this added flexibility will also come with a cost when used for machine learning! Understanding whether a function is linear or non-linear (in specific parameters) is of paramount importance as it enables us to apply appropriate machine learning tools, optimize analyses, and gain deeper insights into the underlying dynamics of the systems that needs modelling. In particular this will play an important role in finding the optimal model parameters $\mathbf{w}$. Finding the model parameters, $\mathbf{w}$, is called *learning*. Examples 3.3 and 3.4 show the importance about being clear about which variables a function is linear or non-linear in.

Later in these lecture notes you will see that an affine transformation can in fact be represented by a matrix

Example 3.2: Linear and Affine Functions

This example illustrates the difference between (bivariate) linear and affine transformations.
The function

$$f(x_1, x_2) = w_1 x_1 + w_2 x_2 = \mathbf{w}^\top \mathbf{x}$$

is linear while

$$f(x_1, x_2) = w_1 x_1 + w_2 x_2 + b = \mathbf{w}^\top \mathbf{x} + b$$

is an affine mapping.

Here $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ are variables, and $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ and b are coefficients. The linear function represents a scaling while the affine function represents a translation and scaling of $\mathbf{x}$ in 2D. The coefficients w_1 and w_2 represent the scaling factors along the x_1 and x_2 axes, respectively, and b represents the translation offset.
Concrete examples are:

$$y = 3x_1 - 4x_2 = \begin{bmatrix} 3 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

and

$$y = 3x_1 + -4x_2 - 4 = \mathbf{w}^\top \mathbf{x} + b$$

Notice that the affine mapping is just the equation formula for a straight line.

Example 3.3: Non-linear Functions

For instance, consider the (*bivariate*) function:

$$f(x_1, x_2) = w_1 x_1^2 + w_2 x_2^3$$

In this function, x_1 and x_2 are variables, while w_1 and w_2 are coefficients. This function is an exemplar of non-linearity concerning the variables $\mathbf{x}_i$, as x_1 is squared and x_2 is cubed. However, it's crucial to recognize that f retains linearity in the coefficients w_i, as the terms x_i are perceived as constants in relation to w_i. This nuanced distinction underscores the complexity of non-linear functions and their interactions with various components.

Example 3.4: Linear and non-linear in what?. Polynomials

Consider the polynomial:

$$f(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_n x^n$$

where x is the variable and w_i the coefficients.
Polynomial functions exhibit non-linearity in their input variable x due to the presence of higher-order terms in the relationship between the variables x and the function's output $y = f(x)$. Consequently, as the inputs x are increased, the resulting output y does not follow a strictly proportional increase. However, polynomials are linear in their coefficients. This means that doubling the coefficients w_i results in doubling the corresponding terms in the polynomial.

3.2.1 Affine Mappings

An affine mapping is a specific type of non-linear transformation that combines a linear term (A) with a translation ($\mathbf{b}$). An affine mapping T is given by:

$$T(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$$

where $A \in \mathbb{R}^{M \times N}$ is a M times N matrix, $\mathbf{x} \in \mathbb{R}^N$ a N dimensional vector and $\mathbf{b} \in \mathbb{R}^M$ and $\mathbf{y} \in \mathbb{R}^M$ are M dimensional vectors.

An affine transformation preserves straight lines, parallelism, and ratios of distances along a straight line due to the linear term A [4], but it does not preserve the origin since $T(\mathbf{0}) = A\mathbf{0} + \mathbf{b} = \mathbf{b} \neq 0$. Example 3.2 show examples of 2D linear and affine transformations. Examples 3.5 and 3.6 shows how linear transformations (matrix multiplication) can be used to add and remove dimensions of an input vector.

This will become apparent after reading the next chapters.

Note!

The model parameters w correspond precisely to the coefficients of the matrix A and the vector $\mathbf{b}$ when viewing linear and affine transformations T_w in the context of Equation 3.2.

Example 3.5: 2D-to-3D Transformation

Matrix multiplication can be used to transform a vector to a higher dimensional space. To transform a 2D input vector $\mathbf{x} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ into 3D space, we use the 3×2 transformation matrix $\mathbf{T}$:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Applying the transformation ($\mathbf{x}' = T\mathbf{x}$) yields $x = \begin{bmatrix} 2 \\ 3 \\ 0 \end{bmatrix}$ in 3D space ($\mathbb{R}^3$).

Example 3.6: 3D-to-2D Transformation

Consider a 3D-to-2D transformation (also called T) that transforms a 3D vector $\mathbf{x}$ to 2D space by discarding the 3. coordinate. To achieve this, we can use a 2×3 transformation matrix $\mathbf{T}$:

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

To apply the transformation, we simply multiply the original 3D vector $\mathbf{x}$ by the transformation matrix T.

Let $\mathbf{x} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$ represents a point in $\mathbb{R}^3$.

$$T\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix} = x'$$

Notice how the transformation controls and pics out the first and second element of $\mathbf{x}$ so that after applying the transformation, the original 3D vector $\mathbf{x}$ becomes $x' = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$ in $\mathbb{R}^2$.

3.2.2 General Definition of Transformation

For completeness, the general definition of a transformation (not only matrices) is given in Definition 3.2 and is illustrated in Figure 3.2. The general definition will become particularly useful later in the course when discussing multivariate vector transformations in e.g. neural networks.

Definition 3.2: Transformation

A *Transformation* (T) is a function $T : \mathbb{A} \to \mathbb{B}$ that map elements from a set $\mathbb{A}$ into a elements of another set $\mathbb{B}$. That is applying the function T to $\mathbf{x} \in \mathbb{A}$ yields $\mathbf{y} \in \mathbb{B}$:

$$\mathbf{y} = T(\mathbf{x})$$

where $\mathbf{x} \in \mathbb{A}$ and $\mathbf{y} \in \mathbb{B}$. $\mathbb{A}$ is called the *domain* of T, and $\mathbb{B}$ is the *codomain* of T.
If x is an element of $\mathbb{A}$ and y is an element of $\mathbb{B}$ such that $T(x) = y$, then the set of all images of vectors in $\mathbb{A}$ is the *range* of T, and the set of all x in $\mathbb{A}$ such that $T(x) = y$ is the *preimage* of y .

In this course the domains and codomains are vectors and the transformations may be considered as functions from vectors $\mathbf{x} \in \mathbb{R}^N$ to vectors $\mathbf{y} \in \mathbb{R}^M$

.

Model parameters A transformation T (or *model*) may involve a number (a vector) of *model parameters*, $\mathbf{w}$, that determine how the transformation is applied. The model parameters are sometimes made explicit and denoted with subscript

$$\mathbf{y} = T_w(\mathbf{x}) \tag{3.2}$$

For example the transformation to change the brightness of the TV is governed by a single parameter e.g. to increase or decrease the pixel values in the image. In this case we can consider $\mathbf{x}$ as the input pixel value and $\mathbf{y}$ the output pixel value, and $\mathbf{w}$ as a scalar which determines how much we wish to change the pixel values to improve the brightness of the image. In this case both the domain and codomain are $\mathbb{A} = \mathbb{B} = \mathbb{R}$. The model parameters for a line-model, $T_w(x) = y = ax + b$ are the slope a and intercept b.

Note!

Envision a transformation much like a function or method within programming languages in which a method T accepts inputs of a certain type ($\mathbb{A}$), and produces an output of type $\mathbb{B}$. It's worth noting that the types $\mathbb{A}$ and $\mathbb{B}$ might be the same, such as in the TV example above.

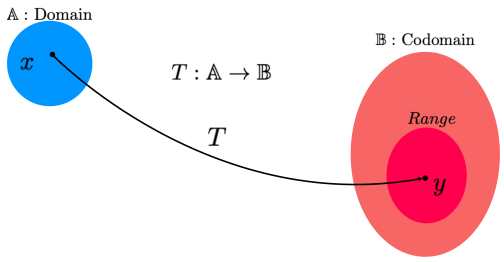

Figure 3.2: Transformations, domain and codomain

4

Intensity Transformations

This chapter illustrates the concepts of linear and non-linear functions in the case where the transformations $T : \mathbb{R} \to \mathbb{R}$ map real numbers to real numbers (univariate).

The chapter uses *Point processing* as a running example to illustrate the concepts. *Point processing* in image analysis are transformations applied to individual pixel values in the image and where the value of the transformed pixel (y) only depends on a single value of the corresponding pixel in the input image as shown in Figure 4.1. When performing point processing on an image I the individual pixel values $I(x, y)$ are manipulated to create a different image I'. For the time being assume the images I and I' are grayscale such that the intensity transformation $T : \mathbb{R} \to \mathbb{R}$ maps intensities to intensities. It is worth pointing out that the transformation T affects the intensity $I(x, y)$ and not the coordinate (x, y). In the TV brightness example above, the input image could be the one stored on your computer (lawfully), and the output image would be what you see on the TV screen. The transformation function would then process these pixel values e.g. to adjust pixel intensities.

Indeed, the observant student may have noticed that while grayscale pixel values are typically in the range of 0 to 255, we have chosen to represent them as real numbers. This intentional choice allows us to transform the discrete pixel values into a continuous scale. By doing so, we can treat the transformation of pixel values as a continuous mathematical function, enabling us to gain a more intuitive understanding of transformations. Moreover, this approach aligns with the idea of interpolation, where we can interpolate pixel values in between the discrete intensity levels to achieve smoother transitions and create various effects on the image.

A convenient way to represent the brightness operations the graphical representation as shown in Figure 4.4. The image in the center is the original image.

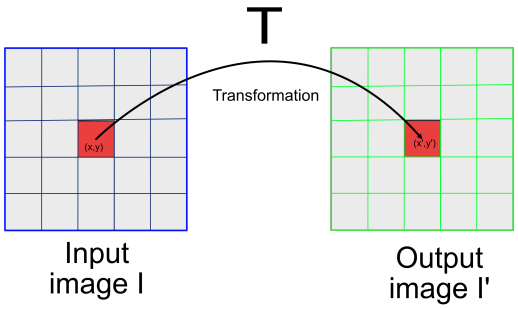

Figure 4.1: The principle of point processing. A pixel in the input image is processed, and the result is stored at the same position in the output image.

Changes vertically represent changes in brightness, and horizontally represent changes in contrast. If b is zero, the resulting image will be equal to the input image (aka the identity function). If b is a negative number, then the resulting image will have decreased brightness, and if b is a positive number, the resulting image will have increased brightness. The graphs illustrates how the pixel value in the input image (horizontal axis) maps to the pixel value in the output image (vertical axis). Pixel values that are mapped to values lager than 255 are set to 255 and pixels values that are mapped to negative values are set to zero.

4.0.1 Linear and Affine Gray-level Transformations

In the movie-watching example from the introduction, each time you press the "+" brightness button on your remote, you effectively increase the brightness, and vice versa when pressing '-'. Mathematically, this can be represented by a transformation parameterized by b:

$$T_b(i) = i + b \tag{4.1}$$

For an intensity value $I(x,y)$ at coordinates (x,y), the transformation is given by

$$I'(x,y) = T_b(I(x,y)) = I(x,y) + b$$

The subscript is used to make it explicit that b is the (only) model parameter

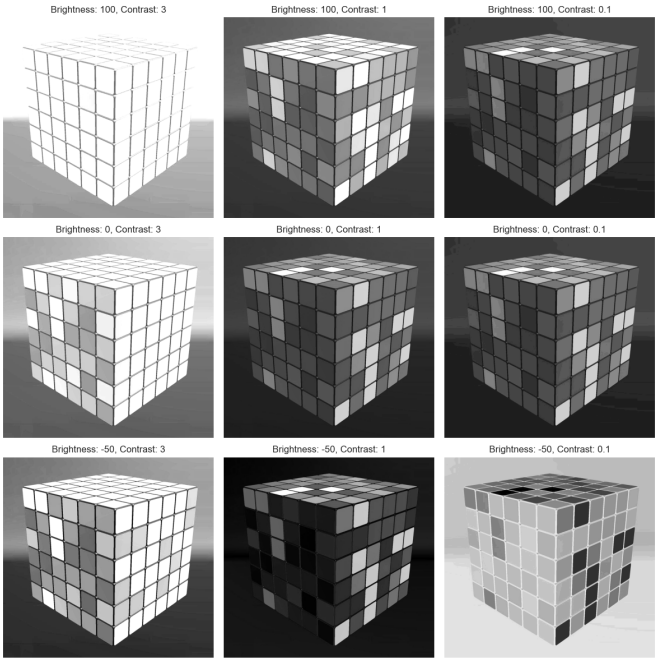

Figure 4.2: Affinely changing brightness and contrast.

of the transformation. Increasing b results in brighter pixel values, while decreasing it darkens the image, as illustrated in the rows of Figure 4.2 and the actual transformation functions are shown in Figure 4.3.

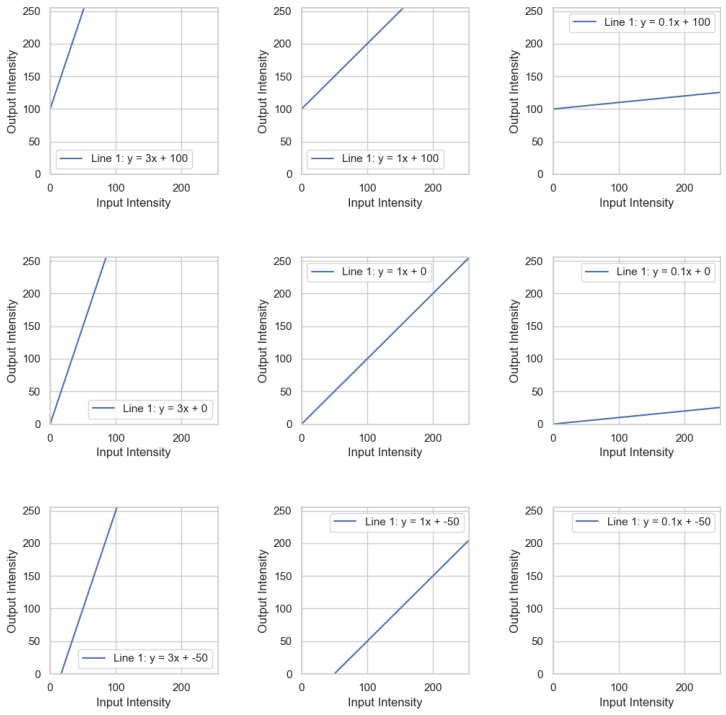

Figure 4.3: The transformation functions use to affinely transform the pixels.

Contrast Adjustment The contrast of an image refers to the difference in intensity between the pixels within the image. It quantifies the degree of variation between the brightest and darkest parts of the image. In simpler terms, an image with *high contrast* will exhibit distinct differences between its lightest and darkest areas, while an image with low contrast will have less pronounced variations in brightness or color intensity. The operation for contrast adjustment of a single pixel value, $I(x, y)$, is represented by a linear transformation:

$$I'(x, y) = T_a(I(x, y)) = a \cdot I(x, y) \tag{4.2}$$

If $a > 1$, the contrast is increased, and if $a < 1$, the contrast is decreased. For instance, when $a = 2$, the pixels 112 and 114 will be mapped to the values 224 and 228, respectively. The difference between the pixel values is multiplied by a factor of 2, leading to an increased contrast. The effect of changing the contrast can be observed in Figure 4.4.

Combining Brightness and Contrast Adjustment into an Affine Mapping

The brightness and contrast equations can be combined to *affinely* transform pixel values such that $T_w(i) = a \cdot i + b$. Applying the transformation to intensity values $I(x, y)$ yields

$$\begin{aligned} I'(x,y) &= T_w(I(x,y)) \\ &= a \cdot I(x,y) + b \end{aligned} \tag{4.3}$$

For clarity "·"represent multiplication by a scalar. This equation represents a straight line in the space of input-output intensities.

Example 4.1: Stretching Intensities

An example of applying the linear intensity transformation in Equation 4.3 is to consider an image I where the contrast is too low. Suppose the minimum (l) , and maximum (h) pixel values are 100 and 150, respectively. We wish to find an intensity transformation that maps intensities from the range $[100, 150]$ to the entire range $[0, 255]$ in the output image. This means setting all pixel values below 100 to zero and pixel values above 150 to 255. In other words, we can subtract 100 and then scale the rest so they span the entire interval. Hence, the values of a and b in Equation 4.3 are set to:

$$a = \frac{255}{h - l} \qquad b = -a \cdot l \tag{4.4}$$

Here, $l = 100$ and $h = 150$.

4.1.0 Non-Linear Gray-Level Mapping

Linear transformations of gray values treat intensity values uniformly, regardless of whether they are dark or bright. However, there are instances when an equal change in intensities is not desired. Non-linear transformations offer a heightened level of control over the image's contrast and overall appearance. This section delves into several widely utilized non-linear mapping techniques employed in image processing. These techniques enable more intricate and customized image transformations, fostering an enhanced degree of sophistication in achieving desired visual effects.

The *gamma transformation* allows for independent contrast adjustments in the dark and light gray levels of an image. The transformation is commonly found in cameras, display devices, and used for image preprocessing in various applications and is defined as

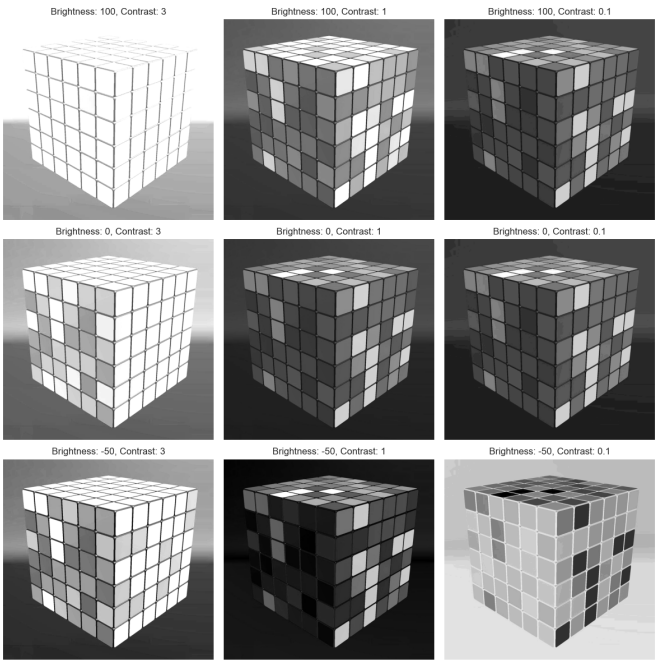

Figure 4.4: Three examples of affine gray-level mappings. The center image is the input. The three other images are the result of applying the three gray-level mappings to the input.

$$I'(x,y) = I(x,y)^{\gamma}$$

where $I'(x,y)$ is the output pixel value, $I(x,y)$ is the input pixel value, and γ is a positive parameter. Figure 4.5 illustrates gamma transformation curves for different values of γ. It is evident that for γ =, the curve corresponds to the *identity mapping*, while for $\gamma < 1$, the curve enhances contrast in the dark regions. Conversely, for $\gamma > 1$, the curve enhances contrast in the bright regions. Applying gamma transformation to images is shown in Figure 4.6.

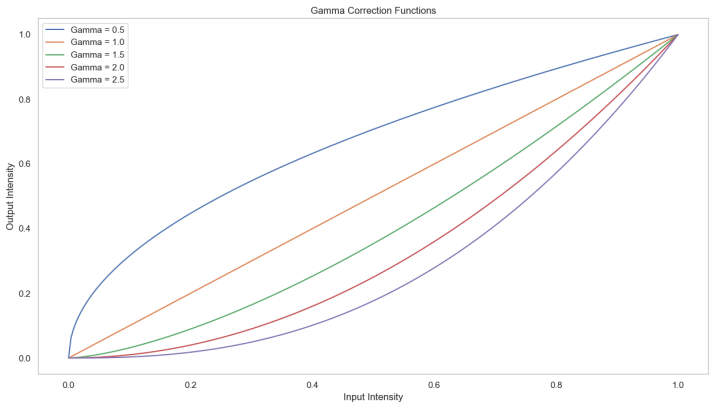

Figure 4.5: Gamma transformations with varying values for the γ parameter.

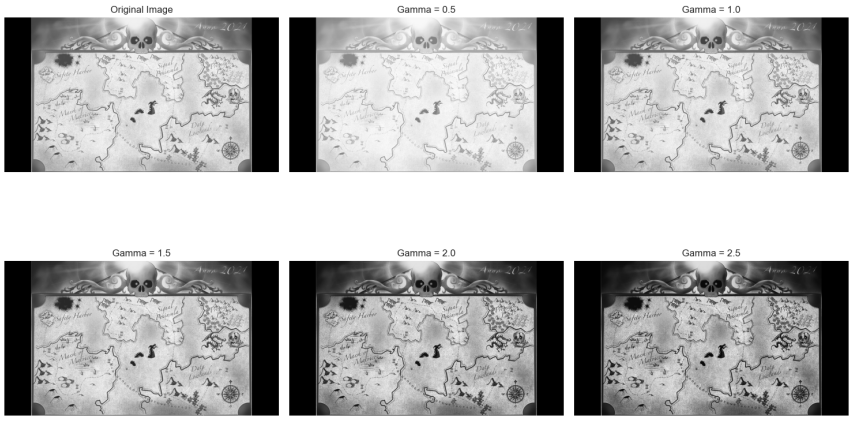

Figure 4.6: Gamma corrected images.

To ensure that the input and output pixel values remain in the range $[0, 1]$, it is necessary to perform normalization before and after the gamma transformation. Before applying gamma mapping, the input pixel values should be scaled to the range $[0, 1]$, and after the transformation, the output values are scaled back to the range $[0, 255]$ to represent an 8-bit image.

4.1.1 Logarithmic and Exponential Transformations

Another approach to non-linear gray-level mapping involves using the logarithm. This method transforms each pixel with the logarithm of its original value. The logarithmic mapping is particularly useful in scenarios where the dynamic range of an image is too great to be displayed effectively or when dealing with images containing a few very bright spots on a darker background. However, it's important to note that the logarithm is not defined for black pixels (having the value of 0). To address this issue, a modified logarithmic mapping is defined as:

$$g(x, y) = c \cdot \log(1 + I(x, y))$$

where c is a scaling constant that ensures the maximum output value is 255 for byte images and where

$$c = \frac{255}{\log(1 + v_{max})}$$

where v_{max} represents the maximum pixel value in the input image.

The behavior of the logarithmic mapping can be further controlled by applying an affine mapping to the input image's pixel values before the logarithmic transformation.

Exponential mapping The exponential transform is defined as:

$$I'(x, y) = (1 + k) \cdot I(x, y)^{-1}$$

where k is a parameter that allows for the adjustment of the transformation curve's shape. The exponential mapping enhances image detail in the light areas while simultaneously reducing detail in the dark areas. This characteristic makes it valuable for specific image processing tasks where selective emphasis on different intensity levels is required.

4.2.0 Thresholding

Thresholding is a fundamental operation often employed for segmenting images into binary regions based on pixel intensity values. Thresholding can be seen as a classification of the pixel values where the color or intensity values are divided into two classes: those below a specified threshold t and those above or equal to it. Thresholding can be defined as:

$$I'(x,y) = \begin{cases} 0, & \text{if } I(x,y) \leq t, \\ 255, & \text{if } I(x,y) > t. \end{cases}$$

where $I(x,y)$ represents the intensity value of the input image at pixel coordinates (x,y), and $I'(x,y)$ is the corresponding intensity value in the output binary image after thresholding and as shown in Figure 4.7. The threshold value t is a crucial parameter as it determines a *hard boundary* between the two classes. Selecting the appropriate threshold value t is critical for successful segmentation. Ideally the image should have a *bimodal histogram* with two distinct peaks corresponding to the background and foreground pixel intensities. In such cases, determining the appropriate value for t is straightforward since the optimal choice for t is between the two peaks. However, real-life images may have more complex histograms with overlapping intensity distributions of foreground and background e.g. due to varying light conditions. By adjusting t, different objects or regions of interest can be extracted from the input image. Setting a fixed threshold may consequently be challenging. Automatic methods that e.g. analyses the color or intensity histogram may be able to minimize the error, but such approaches will not be discussed further here.

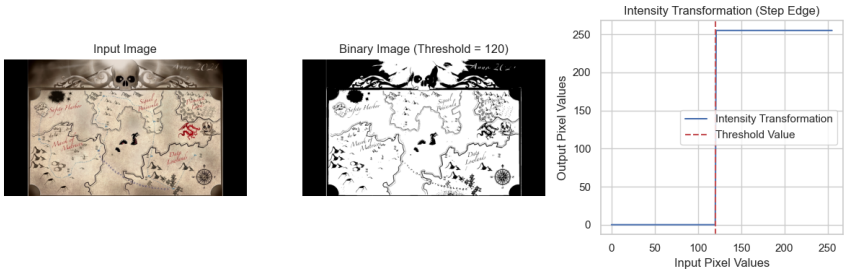

Figure 4.7: Thresholding of an image (left) (middle) thresholded image (right) The intensity transformation function.

Although thresholding can be effective for specific applications, it may result in information loss as finer details in the image are discarded. Careful consideration of the threshold value and image characteristics is necessary to ensure that the segmented regions accurately represent the desired objects or struc-

tures of interest. Thresholding finds applications in a range of image processing tasks, including object recognition, feature extraction, and image enhancement. It is particularly useful in situations where the extraction of specific objects or regions based on their intensity levels is required.

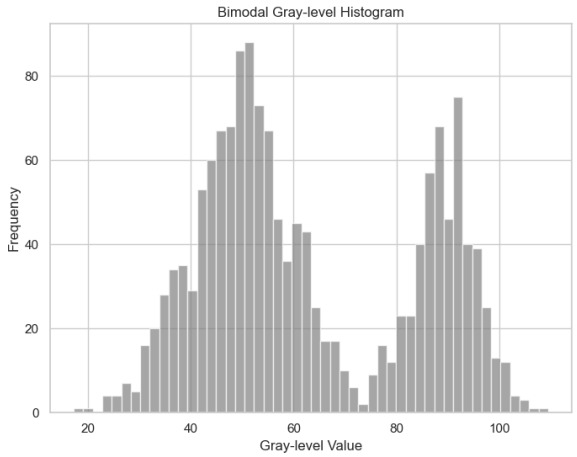

Figure 4.8: Bimodal intensity histogram of an image.

5

2D Geometric Transformations

Most people have tried to perform a geometric transformation of an image when preparing a presentation or manipulating an image. The two most well-known transformations are rotation and scaling, but there are others as well. The term "geometric" transformation refers to a class of image transformations where the image's geometry, represented by the pixel coordinates, undergoes changes while the actual pixel values remain unchanged. That is, the pixel coordinates that are transformed and not the intensity.

In the previous section, we explored transformations $T : \mathbb{R} \rightarrow \mathbb{R}$ through mappings of pixel intensities. In this section, we will delve into 2D transformations $T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$, where (the pixel) coordinates serve as the basis for the transformation. From here it is relatively straight forward to extend transformations to any dimension. A 2D transformation is given by

$$\mathbf{y} = T_{\mathbf{w}}(\mathbf{x}) \tag{5.1}$$

where $\mathbf{x} = \begin{bmatrix} x_1, x_2 \end{bmatrix}^\top$ and $\mathbf{y} = \begin{bmatrix} y_1, y_2 \end{bmatrix}^\top$ and $\mathbf{w}$ is the vector of model parameters.

A geometric transformation determines where the pixel at position (x, y) in I will be located in $I'(x', y')$ Recall from previous chapters that an image is defined as $I(x, y)$, where (x, y) denotes the position of the pixel and $I(x, y)$ its intensity or color value. A geometric transformation determines where the pixel value at position (x, y) in I will be located in I'. After a geometric transformation, only the coordinate is transformed and not the pixel value. So, if $I(2, 3) = 120$, then the pixel value in $I'(2, 3)$ may not have the same value.

5.1.0 Linear and Affine 2D Transformations

This section focuses on linear and affine 2D transformations. Recall from chapter 3 that affine transformations encompass linear transformations with added translations. That is a linear transformation is on the form

$$y = Tx$$

, and affine transformation has the following form

$$y = Tx + b \tag{5.2}$$

where T is a $M \times N$ matrix, $x \in \mathbb{R}^N$ and $y \in \mathbb{R}^M$. Notice, the only difference between linear and affine mappings is the translation. Therefore, $2D$ linear transformations are are detailed in Definition 5.1 and Definition 5.2.

Definition 5.1: Planar 2D Linear Transformation

A *planar linear transformation* is a mapping $T : \mathbb{R}^2 \to \mathbb{R}^2$ that preserves lines and the origin and can be represented by

$$\mathbf{x}' = A\mathbf{x}$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} =$$

That is

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Definition 5.2: Planar 2D Affine Transformation

A *planar affine transformation* ($T : \mathbb{R}^2 \to \mathbb{R}^2$) is linear mapping that also allow points to be translated in space and can be represented by

$$\mathbf{x}' = A\mathbf{x} + \mathbf{b}$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} =$$

That is

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

5.1.1 Specific Linear and affine transformations

The linear transformations encompass scaling, rotation, reflection, and shearing. To elucidate these linear transformations, it is beneficial to illustrate them by multiplying the transformation matrix with a vector. $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ to better comprehend the transformation.

Scaling Scaling is a transformation that stretches or shrinks a vector and can be represented by a diagonal matrix

$$S = \text{diag}(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

where s_x and s_y are scaling factors along the two axes. When the scaling is equal in both directions ($s_x = s_y$), we say the scaling is *isotropic*; otherwise, it is said to be *anisotropic*. Multiplying $S\mathbf{x}$ for a vector $\mathbf{x} \in \mathbb{R}^2$, it becomes evident that the result is a vector

$$\mathbf{y} = \begin{bmatrix} s_x x_1 \\ s_y x_2 \end{bmatrix}$$

Rotation A rotation about the origin is a geometric transformation that pivots a point around the origin by a specific angle θ. A rotation can be represented using a rotation matrix R_θ:

$$R_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{5.3}$$

Reflection Reflection is a transformation that mirrors a point across a line. A reflection can be represented by different matrices depending on the axis of reflection. For example, a reflection across the x-axis is given by matrix

$$A_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

and a reflection about the y-axis,

$$A_y = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

The reflection across the line y = x (45-degree reflection) is given by

$$A_{xy} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Shearing Shearing is a type of linear transformation that distorts the shape of objects by displacing points along a fixed direction. It involves changing the

coordinates of points in a way that maintains the lines parallel to a specified axis while causing other lines to become slanted. Shearing can be described using shear factors that determine how much each point is shifted along the chosen axis.
A *horizontal shear* is given by:

$$S_x = \begin{bmatrix} 1 & x \\ 0 & 1 \end{bmatrix}$$

Similarly a *vertical shear*

$$S_y = \begin{bmatrix} 1 & 0 \\ y & 1 \end{bmatrix}$$

Figure 5.1 shows the effect of applying linear transformations on each pixel coordinate such as scaling, anisotropic scaling (different scaling in x and y direction) rotation, and shear, along with translations. In this case the rotation and scalings are performed relative to the center of the image.

Example 5.1: 2D affine mapping

Consider the affine mapping with

$$A = \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix}$$

and

$$b = \begin{bmatrix} 3 \\ 4 \end{bmatrix}.$$

A square with vertices

$$v_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \quad v_3 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \quad v_4 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

is mapped to a parallelogram with vertices

$$v_1' = \begin{bmatrix} 3 \\ 5 \end{bmatrix}, \quad v_2' = \begin{bmatrix} 5 \\ 5 \end{bmatrix}, \quad v_3' = \begin{bmatrix} 3 \\ 3 \end{bmatrix}, \quad v_4' = \begin{bmatrix} 1 \\ 3 \end{bmatrix}.$$

Translation The only difference between linear and affine mappings is the translations A (pure) translation shifts every point by a constant vector **b**. It is easy to verify that this is equivalent to

$$x' = x + b_x$$
$$y' = y + b_y$$

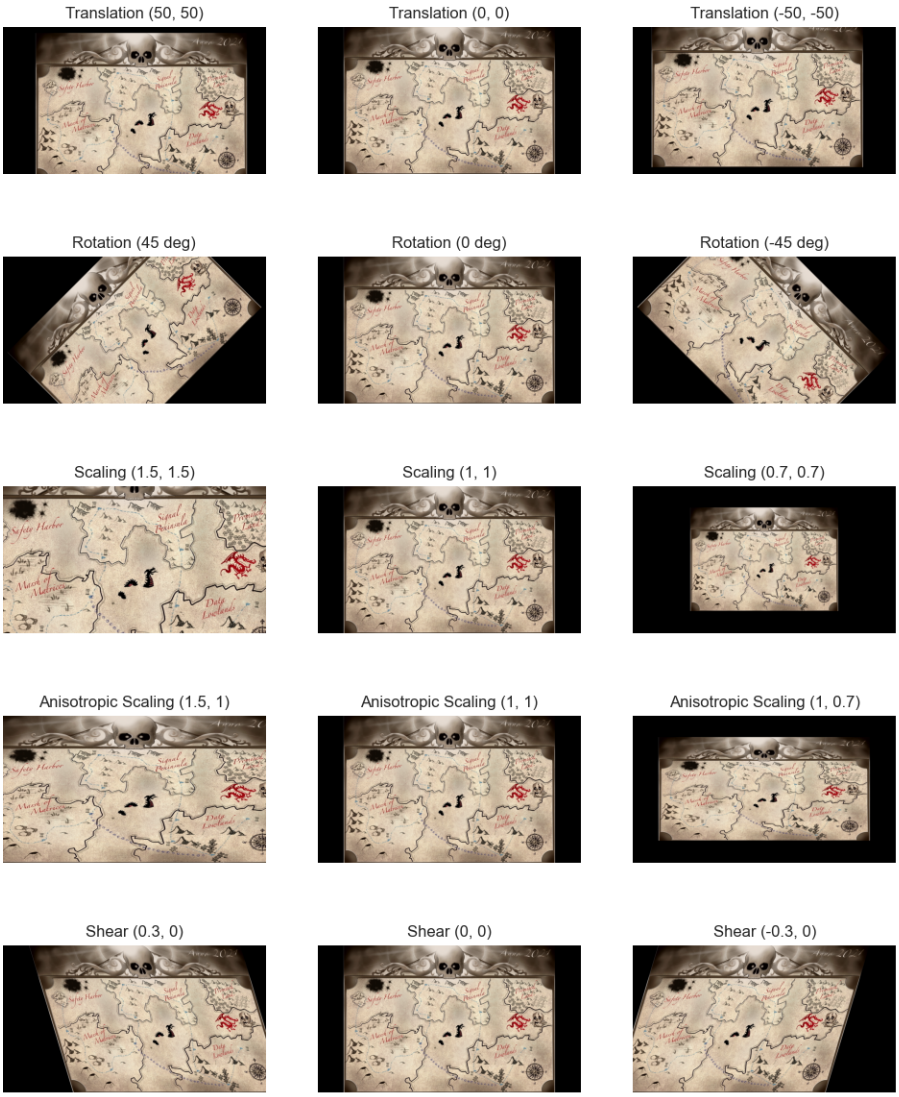

Figure 5.1: Linear and affine mappings of pixel coordinates if the IML Map.

Using the definition 5.2 it is clear that a translation is a specific affine transformation where the matrix $A = I$ is the identity matrix. That is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{5.4}$$

where $b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$ is the translation vector.

When $b_1 = 100$ and $b_2 = 120$, will shift each pixel 100 pixels in the x direction and 120 in the y-direction.

5.2.0 Composition Linear Transformations

Let the matrices A and B represent a linear transformation such a rotations or shearing. *Composition* is when we apply the transformations one after the other. Composing linear transformations is obtained through matrix multiplication giving a new transformation matrix, $C = AB$, that encapsulates the effects of each individual transformation. This is mathematically equivalent to applying transformation B first, followed by transformation A. The order of multiplication reflects the order of application. Suppose we have two 2D linear transformations: a rotation R and a shearing S. That is the composition $S \circ R$

$$S \circ R = SR = C$$

To apply this transformation to a point $p = \begin{bmatrix} x \\ y \end{bmatrix}$, we first rotate it using R and then shear it using S The composition is done from right to left, meaning $Cp = S(Rp) = SRp$

Composing Linear Transformations is NOT commutative I suppose we can agree that the order in which you multiply two real numbers $a * b = b * a$ does not matter. e.g. $6 * 3 = 3 * 6 = 18$. We say that the product of real numbers *commute*. However for matrices the order in which you multiply them matters. Certain combination of matrices may commute $AB = BA$ but generally we can expect that $AB \neq BA$. This *non-commutative* property means that the order in which transformations are applied matters. Non-commutative matrix multiplication extends to any dimension and thus linear transformations in any dimension does not commute e.g.

$$S \circ R = S * R \neq R * S = R \circ S$$

Let's delve into this concept a bit. Consider two distinct 2D linear transformations: a scaling operation S that enlarges an object and a rotation operation R that turns it. If we first apply the shearing operation S and then the rotation operation R to an object, we'll get one result. Reversing the order and first rotate the object by R and then shear it by S, the final outcome will generally be different as demonstrated in Figure 5.2.

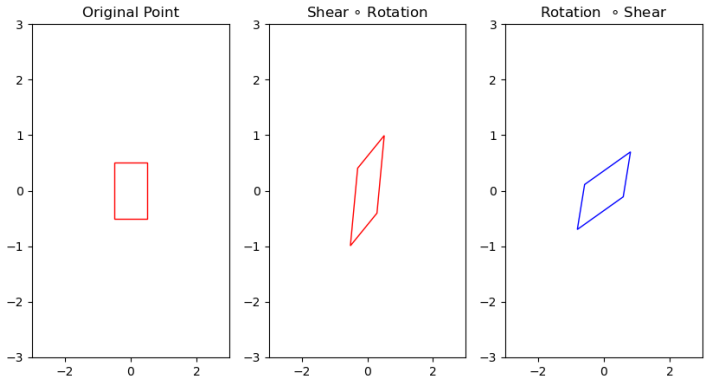

Figure 5.2: Matrix multiplication is non-commutative and therefore the order in which linear transformations are composed matter. (Right) Four points centered around the origin. (middle) The same points after a rotation and shearing. (Left) The points after a shearing and rotation.

5.3.0 2D Transformations around a point

While the previous sections discussed 2D transformations with respect to the origin, it is often necessary to perform transformations around a specific point $\mathbf{p} = \begin{bmatrix} p_x \\ p_y \end{bmatrix}$ for example by perform transformations of rotations, scalings, or translations with $\mathbf{p}$ as the center of these transformations. The general approach for performing a transformation of a point $\mathbf{x}$ around $\mathbf{p}$ is

$$\mathbf{x}' = T(\mathbf{x} - \mathbf{p}) + \mathbf{p}$$

Similarly for affine transformations of $\mathbf{x}$ around $\mathbf{p}$ is

$$\mathbf{x}' = T(\mathbf{x} - \mathbf{p}) + \mathbf{b} + \mathbf{p}$$

The equations can be understood through the following three steps, here using linear transformations as example.

1. TRANSLATION TO ORIGIN: Translate a point $\mathbf{x}$ so that point $\mathbf{p}$ coincides with the origin $\mathbf{0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. This translation ensures that $\mathbf{p}$ becomes the new origin of the coordinate system.

$$\mathbf{x_p} = \mathbf{x} - \mathbf{p}$$

2. TRANSFORMATION: Apply the desired transformation to the points in the translated coordinate system. Since P is now at the origin, the transformation is centered around $\mathbf{p}$

$$\mathbf{x_q} = T\mathbf{x_p}$$

3. TRANSLATION BACK: Finally, translate all points back to their original positions in the original coordinate system. This translation effectively moves them by the negative of the translation applied in step 1, restoring their original positions.

$$\mathbf{x}' = \mathbf{x_q} + \mathbf{p}$$

5.4.0 The Inverse Transformation

Intuitively, the transformation of rotating a point 30 degrees counterclockwise is achieved using matrix multiplication with a matrix R. If we wish to 'undo' this transformation, we can apply another transformation, namely the inverse R^{-1}, which rotates the point 30 degrees clockwise. These transformations are inverses of each other, meaning that $R^{-1}Rx = x$. In essence, combining these transformations results in no net change, effectively applying the identity transformation to the point ($x = I(x)$ where I is the identity function). It's important to note that we can only find the inverse of matrices if the determinant $\det(R) \neq 0$. Intuitively, this means that when an inverse exists, we can transform points back and forth. Otherwise, without a non-zero determinant, there may be no way to recover from the transformation.

5.5.0 Homogeneous coordinates

Composing two affine transformations A_1 and A_2 is less straightforward than composing linear transformations. Finding inverses is also somewhat more involved. The difficulty primarily arises from the non-linearity introduced by the translation component, making direct composition less straightforward. To address this challenge, homogeneous coordinates come to our rescue. By extending the coordinate $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2$ with an additional dimension and setting this to 1, the homogeneous representation $\mathbf{x}'$ of $\mathbf{x}$ becomes:

$$\mathbf{x}' = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \in \mathbb{P}^2$$

Sometimes homogeneous coordinates are also called *projective coordinates* and the set of homogeneous vectors in the plane is denoted $\mathbb{P}^2$.

Transforming Homogeneous Coordinates to Euclidean Coordinates When constructing homogeneous coordinates, we added a dimension to the vector setting it to 1. Going back is almost as simple. The last coordinate is removed provided its value is 1. So what has a value of t?

Homogeneous coordinates assume an equivalence relation between vectors such that vectors:

$$\begin{bmatrix} tx_1 \\ tx_2 \\ t \end{bmatrix}$$

where t is a non zero scalar, are considered equivalent. The vectors

$$\begin{bmatrix} 2 \\ 5 \\ 1 \end{bmatrix}, \begin{bmatrix} 4 \\ 10 \\ 2 \end{bmatrix}, \begin{bmatrix} -6 \\ -15 \\ -3 \end{bmatrix}$$

are all considered equivalent.

A homogeneous vector $\mathbf{v} = \begin{bmatrix} tx \\ ty \\ t \end{bmatrix}$ can be converted to its standardized form $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ simply by multiplying it by $\frac{1}{t}$, such that it becomes $\mathbf{v} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. The transformation to its Euclidean representation is straightforward simply by removing the last coordinate such that $\begin{bmatrix} x \\ y \end{bmatrix}$.

Affine mapping equivalence Consider an affine mapping on the form

$$\mathbf{x}' = A\mathbf{x} = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{5.5}$$

Any such mapping can conveniently be represented by a single matrix product using homogeneous coordinates:

$$Hx = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \tag{5.6}$$

where $H \in \mathbb{R}^{3\times 3}$ and $\mathbf{x}, \mathbf{x}' \in \mathbb{P}^2$.

Using matrix multiplication it is straightforward to verify that the Equation 5.5 and Equation 5.6 are equivalent. Let's illustrate this through a using pure translations as a simplified example.

Observe:

$$\begin{bmatrix} x_1 + b_1 \\ y_1 + b_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

Observe that the two first coordinates are correct but we are left with a 1 in the third coordinate. So lets just remove it and we have estabilised the relation between Equation 5.5 and Equation 5.6.

An isotropic scaling (s), translation ($\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$) and rotation by angle θ can be represented by the matrix

$$\begin{bmatrix} s\cos(\theta) & -\sin(\theta) & b_1 \\ \sin(\theta) & s\cos(\theta) & b_2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{bmatrix}$$

Note!

Notice also that the inverse of affine transformations are easily calculated in one step.

Example 5.2: Transformations using Homogeneous coordinates

This example demonstrates the process of converting a point from Euclidean coordinates to homogeneous coordinates, applying a transformation, and then converting it back to Euclidean coordinates.

Consider a point with Euclidean coordinates $\begin{bmatrix} x \\ y \end{bmatrix}$ and homogeneous representation

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The 3 $\times$ 3 transformation matrix H that performs a translation by $\begin{bmatrix} 4 \\ 5 \end{bmatrix}$ and an anisotropic scaling by factors 2 in the x-direction and 3 in the y-direction is given by

$$H = \begin{bmatrix} 2 & 0 & 4 \\ 0 & 3 & 5 \\ 0 & 0 & 1 \end{bmatrix}$$

Transforming Homogeneous Coordinates The transformed point $\mathbf{p}'$ in homogeneous coordinates can be obtained by multiplying matrix H with the vector $\mathbf{p}$:

$$\mathbf{p}' = H \cdot \mathbf{p}$$

In this representation, $\mathbf{p}'$ will have the coordinates $(2x + 4, 3y + 5, 1)$ in homogeneous coordinates.
This process involves transforming the point using the given matrix H, and the resulting $\mathbf{p}'$ can be considered in either homogeneous or Euclidean coordinates, depending on the application's requirements.
Substituting the values:

$$\mathbf{p}' = \begin{bmatrix} 2 & 0 & 4 \\ 0 & 3 & 5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2x+4 \\ 3y+5 \\ 1 \end{bmatrix}$$

Homogeneous to Euclidean Coordinates Converting this point $\mathbf{p}'$ back to Euclidean coordinates, we divide the first two coordinates by the third:

$$\mathbf{q} = \begin{bmatrix} \frac{2x+4}{1} \\ \frac{3y+5}{1} \end{bmatrix} = \begin{bmatrix} 2x+4 \\ 3y+5 \end{bmatrix}$$

Therefore, the transformed point in Euclidean coordinates is:

$$q' = \begin{bmatrix} 2x+4 \\ 3y+5 \end{bmatrix}$$

Inverse Transformation It turns out that the determinant $\det(H) = 6$, which means that an inverse exists. The inverse

$$\begin{bmatrix} 0.5 & 0 & -2. \\ 0 & \frac{1}{3} & -\frac{3}{3} \\ 0 & 0 & 1 \end{bmatrix}$$

This result is somewhat intuitive since 'undoing' a scaling by factors of 2 and 3 and translating by 4 and 5 should involve scaling by $\frac{1}{2}$ and $\frac{1}{3}$ and applying the appropriate translation in the reverse order, as reflected in the inverse matrix H^{-1}. It is easily verified that $H^{-1}H = 1$.

5.6.0 3D Linear and Affine Transformations

The concepts of linear and affine transformations in two dimensions can be extended to any dimension N. This section will build on these concepts and show their extension to 3 dimensional space. A 3D linear transformation is represented by a 3×3 transformation matrix (or 4×4) if a homogeneous representation is used). Notice that the observations about composing transformations and inverses equally apply to any dimensions due to the generality of linear algebra.

5.6.1 Scaling, Shearing, and Rotation in 3D

Scaling, shearing, and rotation operations in 3D are accomplished through transformation matrices that share similarities with their 2D counterparts. These operations alter the orientation and size of objects in 3D space.

Scaling matrices impact the diagonal elements of the transformation matrix, where each element on the diagonal corresponds to one of the three axes.

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}$$

Shearing matrices introduce off-diagonal terms, affecting the alignment of objects along different axes. These transformations enable complex deformations.

Shearing Matrix:

$$\begin{bmatrix} 1 & Sh_{xy} & Sh_{xz} \\ Sh_{yx} & 1 & Sh_{yz} \\ Sh_{zx} & Sh_{zy} & 1 \end{bmatrix}$$

Rotations around the origin in 3D are characterized by their axes, and rotation matrices are defined for each rotational axis. Observe that the matrices representing these rotations preserve the axis around which the rotation transpires.

ROTATION MATRIX AROUND THE x-AXIS:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

ROTATION MATRIX AROUND THE y-AXIS

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

ROTATION MATRIX AROUND THE z-AXIS

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The translation matrix for 3D translation using homogeneous coordinates is:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here, (t_x, t_y, t_z) represents the translation vector.

5.7.0 Model Parameters in Linear Transformations

In machine learning it is critical to know your model parameters. By learning the appropriate coefficients for a specific transformation, it is possible train the models to fit the data, make predictions, or analyze relationships between variables.

The coefficients of the transformation matrix, T_w, act as the model parameters $\mathbf{w}$ of the transformation thus linearly determining how the transformation

affects the input data. Each coefficient corresponds to a specific element in the matrix, collectively controlling the scaling, rotation, shearing, reflection, or other linear manipulations applied to the input vectors. An affine transformation (Equation 5.2) has the translation vector $\mathbf{b}$ as additional model parameters. Notice that some 2D linear transformations only have a single model parameter even though the matrix has four entries. For example, the 2D rotation matrix has a single model parameter $w = \theta$ that governs the entire transformation as observed in Equation 5.3. Similarly the shear transformation matrices also only have a single model parameter x or y that determines the extent and direction of the shear. The 2D scaling matrix has two parameters s_x and s_y that control scaling along the x and y axes, respectively. In general, higher-dimensional transformations will have a corresponding number of parameters in the transformation matrix, and each parameter influences a specific aspect of the transformation.Whether it is a single parameter governing a rotation or multiple parameters in more complex transformations, the coefficients in the transformation matrix serve as the knobs that allow us to control and interpret the behavior of the linear transformation and the underlying system it represents.

6

Training and Validating Model

It is important to have a good understanding of how to build effective models given the observations. Intuitively, an effective model, $f_{\mathbf{w}}$, is one that is capable of capturing the structure of a given problem while still be able to handle uncertainties in the data during training and deployment. Observations (e.g. data used for training or prediction) are generally assumed to be samples from an observation function $g(x) = h(x) + \epsilon$, where $h(x)$ is the true model and ϵ some noise. *Training* is the process of determining the model parameters,$\mathbf{w}$, of the predictive model $f_{\mathbf{w}}(x)$. During training the model undergoes optimization to learn from the provided data and minimize its predictive errors e.g. the mean squared error (MSE) $Err(x) = \|f_{\mathbf{w}}(x) - h(x)\|^2$, to learn a model $f_{\mathbf{w}}$ that is as close (in a Euclidean sense) to $h(x)$ as possible. The loss, denoted as Err, reaches its minimum value of 0 when the model makes ideal predictions. In all other cases, $Err(x) \geq 0$ is a positive number. Once the model completes its training, it gains the capacity to make predictions. The training dataset becomes less relevant after training because the model parameters have already been fine-tuned to minimize the loss on this data.

Note!

Beware of the temptation to iteratively retrain the model on the training set and evaluate it on the test set in an attempt to optimize "generalizability." This practice can lead to data leakage, as the model gradually becomes tailored to the specific patterns of the test set, resulting in overly optimistic performance estimates.

Overfitting / Underfitting Consider setting the goal of constructing a model $f_{\mathbf{w}}$ that is capable of learning meaningful patterns in a dataset. This is not just any dataset but the best possible one in terms of quantity and quality as to achieve the lowest possible error. You divide the dataset into a training, validation and and a test set. You train a model to the best of its ability and you still have errors that appear to be impossible to get rid of. An immediate solution emerges: Let's create an ultra flexible model with a multitude of adjustable parameters. You train the new model and achieve a remarkable outcome of zero error on the training data. However, upon deployment, customer complaints flood in. The culprit? The model's complexity was increased to achieve a superior training but as a consequence the model needed to interpolate every data point and thus also adapting to the noise. This phenomenon will be referred to as overfitting. We could have used a simpler model but the model may after training lead to systematic and higher training errors (underfitting).

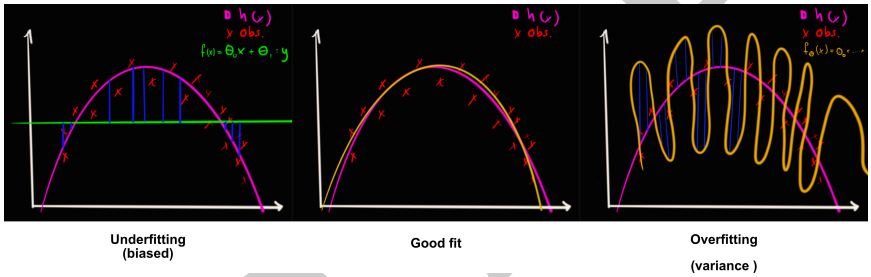

Figure 6.1: Ground truth model $h(x)$ is pink, red crosses are training samples from $g(x)$. Blue lines indicate prediction errors. (left) $f_{\mathbf{w}}(x) = w_0 x + w_1$ is green a (middle) $f_w(x) = w_0 + w_1 x + w_2^2 x$ is a function with an appropriate number of parameters (yellow). (right) $f_w(x)$ is an overly complex model possibly with many parameters (orange). Notice θ in figure is the model parameter until I have time to make new graphics :)

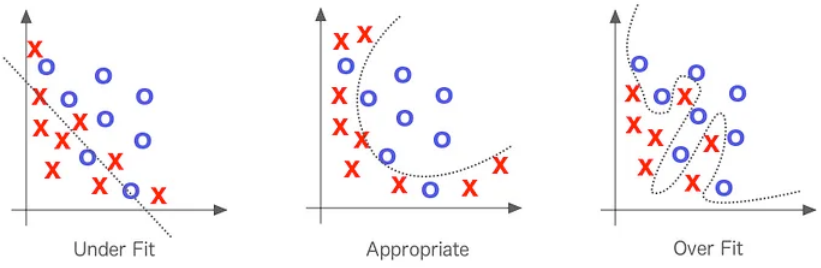

Figure 6.2: Functions to separate red crosses and blue circles.

Figure 6.1 provides some intuition about the importance of bias and variance in terms of regression models. In Figure 6.1 (left) the prediction model f_w is linear. A linear model is clearly too simple to capture the structure of $g(x)$ and will therefore provide biased results (blue). In this case the noise does not significantly influence the results. Figure 6.1 (right) shows the result of a significantly more complex model (yellow) which can capture the variability of the training data (red) very well but since it is too "flexible" it introduces larger errors in the test data (different from the red training data). The preferred model is shown in the middle of Figure 6.1. The model ensures that the there is a nice spread of data around the estimated curve and it seems difficult how the model can be made much better without it following the noise. Observe, this means the model has a low bias and low variance and is quite close the the ground truth model $h(x)$ As shown in Figure 6.2 and Figure 6.1, *Underfitting* arises when a model fails to capture the underlying data trends, often due to excessive simplicity. It may result from inadequate handling of missing data, a lack of outlier treatment, or the removal of relevant features. *Overfitting* occurs when a model learns the training data too closely, capturing noise and random fluctuations rather than the underlying signal. While an overfit model may perform well on the training data, it typically struggles with new, unseen data, rendering it impractical for real-world applications. In such cases, the model demonstrates an outstanding training but exhibits excessive variation when applied for practical purposes (e.g. validation and test sets). Understanding bias and variance is consequently critical in understanding good models. If a model performs well on training data but poorly on validation data, it signals overfitting. So how do we decide on a model with the right complexity - not to simple (underfitting) and not too complex (overfitting) but still performs well on unseen data.

Central to over- and underfitting are the terms bias and variance and the *bias-variance decomposition* of the generalization error. Figure 6.3 shows examples of bias and variance. The intention is that the estimates (purple) are to hit the center of the red target i.e. the bull's eye is the model result we want to achieve that perfectly predicts all the values correctly. As we move away from the bull's eye, our model starts to make more and more wrong predictions

A *bias* (as defined in 6.1) refers to a systematic trend in the estimates' location, often resulting in predictions that consistently deviate from the true center. This bias stems from an oversimplified model that fails to capture the complexity of observational data, leading to substantial errors in both training and test performance.

Conversely, *variance* measures the extent to which the estimates are scattered or spread out. A model with low bias and high variance predicts data points that are widely dispersed around the center, resulting in a diverse but potentially inaccurate set of predictions. In contrast, a model with high bias and low variance may be consistently off-target, but the predictions are closely packed together due to the low variance.

Bias and variance represent opposing characteristics in model performance, with bias indicating systematic errors and variance reflecting the degree of prediction spread as indicated in Figure 6.4.

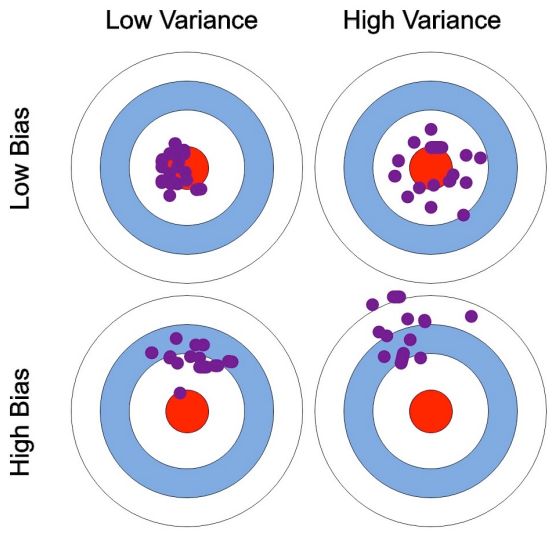

Figure 6.3: Bias and variance influence on estimation. Purple circles are estimates which intend to hit the center of the target.

Underfitting occurs when a model is unable to capture the underlying pattern of the data. This may be caused by having too little data to infer the relevant

parameters yielding high bias. It is typically impossible to optimize bias and variance simultaneously in the presence of noisy observations. The details for this dilemma is captured in the *bias-variance decomposition* where thee main result is that the generalization error (total error), $\text{Err}(x)$, can be decomposed into a systematic error term (bias), a term depending on how much trained models vary (variance) and a term that cannot be done anything about due to noise in the data: [1]

The formal proof requires knowledge of statistics that is a bit outside the scope of this course.

$$\text{Err}(x) = \text{Bias}^2 + \text{Variance} + \text{irreducible error} \tag{6.1}$$

That is the error (in supervised learning) consists of 3 parts, the bias, variance and irreducible error. Irreducible error is a measure of the amount of noise in our data that cannot be removed by creating good models. The other two i.e. Bias and Variance are reducible errors that we can attempt to minimize as much as possible. If we consider that a learned model is "ideal" when it makes no mistakes, *Err* in Equation 6.1 is 0 and the bias and variance terms become 0 and as a consequence there is no noise either. That is "ideal" results can only be achieved when there is no noise. You can have two models, both equally suboptimal yet distinct, with the same Mean Squared Error (MSE): one model may have significantly low bias but high variance ($MSE_1 = \text{Bias}_1^2 + \text{Variance}_1$), while the other could exhibit low variance but high bias ($MSE_2 = \text{Bias}_2^2 + \text{Variance}_2$). Remarkably, both models result in the same MSE ($MSE_1 = MSE_2$), indicating equivalent overall performance scores. However, the preference on whether to prioritize bias reduction or variance reduction oftendepend on the application.

Note!

Occam's Razor is a fundamental principle in the philosophy of science which is relevant when discussing bias, variance, overfitting, and underfitting. Occam's Razor suggests that when faced with multiple competing models (hypotheses), one should prefer the simplest explanation that adequately explains the observed data. For Machine learning this translates to favoring simpler models over complex ones when they demonstrate comparable predictive performance.

The concept of Occam's Razor can be linked to bias and variance in the context of model complexity. A simple model, often associated with high bias, makes strong assumptions about the underlying data distribution, leading to potentially inadequate fits to complex datasets. On the other hand, a complex model, often associated with high variance, possesses greater flexibility to fit intricate patterns in the data but is prone to overfitting by capturing noise and making it less effective at generalizing to unseen data.

Overfitting favors complexity over simplicity and often results in models with unnecessarily intricate structures and hence directly contradicts Occam's Razor. Conversely, underfitting occurs when a model is overly simplistic, aligning more with Occam's Razor but failing to

capture the essential patterns in the data.
In practice, finding the right balance between bias and variance is a key challenge in machine learning. Occam's Razor encourages model selection that leans toward simplicity, emphasizing the importance of choosing models that are appropriately complex for the task at hand. It cautions against the overuse of overly complex models that may lead to overfitting, reinforcing the idea that simpler models often generalize better to new, unseen data—a fundamental principle in achieving effective and robust machine learning models.

The following section is a bit more formal. Let the $y = g(x) = h(x) + \epsilon$ and where ϵ is the noise. Assume that the noise is normally distributed with 0 mean and variance σ. Hence the noise spreads nicely around $h(x)$. The expected loss (e.g. read average) is defined as

$$\text{Err}(x) = \mathbb{E}[(f_w - g(x))^2]$$

Using different data to train the model in the presence of noise may yield different model parameters estimates $\hat{\mathbf{w}}$ depending on which data is used for training. That is, a model $f_{\mathbf{w}}$ trained on a training set Ω_0 will result in an estimate of the model parameters $\hat{\mathbf{w}}_0$. Retraining the model f_w on a different training set Ω_i will result in a different estimate $\hat{\mathbf{w}}_i$. Training a model multiple times will yield a distribution over the model parameters $\hat{\mathbf{w}}_i$. How far the average estimate $\hat{\mathbf{w}}_i$ is from the "true" model parameters w (bias) and how much the estimates vary (variance) tell us something about the stability of the model and whether there are systematic errors as shown in figure Figure 6.3. The model will consider the variance as something to learn from. That is, the model learns too much from the training data, so much so, that when confronted with new (testing) data, it is unable to predict accurately based on it. The average estimate $\mathbb{E}[\hat{\mathbf{w}}]$ may be considered as the solution obtained from least squares and the variance of the model parameters relates to how stable the model is. Hence if the variance is low, the model parameters vary little from the mean hence either overfitted or models the data well.

The intuition about the bias variance decomposition as stated in Equation 6.1 and displayed in Figure 6.4 can more accurately be written as:

$$\text{Err}(x) = \mathbb{E}[f_{\hat{\mathbf{w}}} - h(x)]^2 + \mathbb{E}[(f_{\hat{\mathbf{w}}} - \mathbb{E}[f_{\hat{\mathbf{w}}}])^2] + \sigma_\epsilon^2$$

Figure 6.4 shows the relation between the total error, bias and variance and how to balance between them to get the best possible performance.

Selecting the right model complexity, becomes a matter of balancing bias and variance. As illustrated in Figure 6.4, the ideal model complexity 'k' is where the testing score reaches its peak, indicating the model's capability to perform

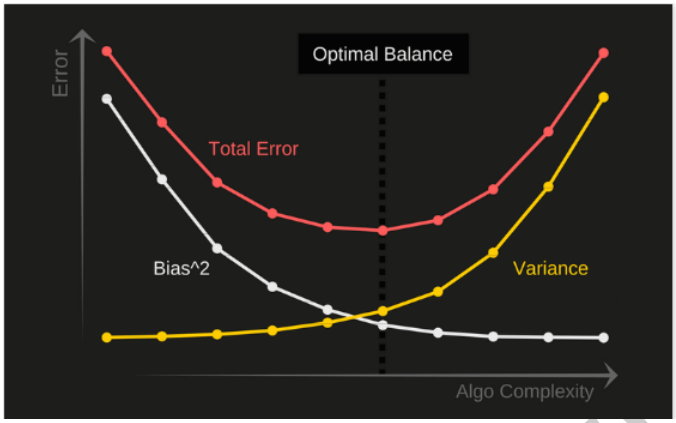

Figure 6.4: Bias and variance represent opposing characteristics in model performance, with bias indicating systematic errors and variance reflecting the degree of prediction spread.

well on unseen data. When the test score aligns closely with the training score, it demonstrates the model's consistency in performance between what it learned during training and its predictions for new, unseen data. Accepting a slightly lower training score is acceptable if it means the model handles new, unseen data well which is crucial because the testing data represents real-world scenarios the model hasn't encountered before. In summary A model with a high bias error underfits data and likely makes very simplistic assumptions on it A model with a high variance error overfits the data and learns too much from it A good model is where both Bias and Variance errors are balanced

Definition 6.1: Bias

Bias is defined as

$$\beta = \mathbb{E}[\hat{X}] - X \tag{6.2}$$

where $\mathbb{E}[X]$ is the expected value (e.g. read average) over a set of estimates X_i and X is the true underlying value.
A prediction function, $f_{\mathbf{w}}$ is called *unbiased* when the bias is 0 and it is either positively or negatively biased depending on whether its is larger or smaller than 0. Hence, the estimate is unbiased when the model on average makes predictions on target. The bias indicates systematic errors.

Definition 6.2: Variance

The **variance** is defined as the difference between the expected value of the squared estimator minus the squared expectation of the estimator:

$$\text{Var}(\hat{X}) = \mathbb{E}[\hat{X}^2] - \mathbb{E}[\hat{X}]^2$$

Note it may be convenient to write the variance in its alternative form:

$$\text{Var}(\hat{X}) = \mathbb{E}[\mathbb{E}[\hat{X}] - \hat{X}]^2.$$

6.1.0 Cross-Validation

When choosing between different machine learning algorithms or model architectures, cross-validation helps compare and select the most suitable one. Error rates and other evaluation metrics may depend on the specific choice of the partition of the test, verification and training sets. In cross-validation the dataset is divided into K subsets, often referred to as *folds*. The idea is to vary which partitions are used for training and validation and asses whether such variations have an effect on the the results as illustrated in Figure 6.5.

In *K-Fold Cross-Validation*, the model undergoes training K times, utilizing $K-1$ folds for training and one fold for validation/testing in each iteration. This process yields K sets of evaluation scores, offering insights into the model's stability across dataset variations. A special case of K-Fold Cross-Validation is *Leave-One-Out Cross-Validation*, where K is set equal to the number of data points (N) in the dataset. In this scenario, the model is trained N times, with each training iteration using $N-1$ data points for training and one data point for testing.To address challenges such as imbalanced datasets and variations in data distribution and location within the full dataset, *Stratified Cross-Validation* comes into play. It ensures that each fold maintains a balanced representation of the distribution found in the entire dataset.

Varying K Performing cross-validation over varying values of K is useful in gauging a model's performance with respect to the size of the dataset but it adds to the computational complexity of the assessment. The following steps can be used as a general guideline, assuming the dataset is thoroughly cleaned, preprocessed, and ready for machine learning. Figure 6.5 shows the steps conducted for a specific choice of K.

1. **Vary K:** Iterate over different values of K

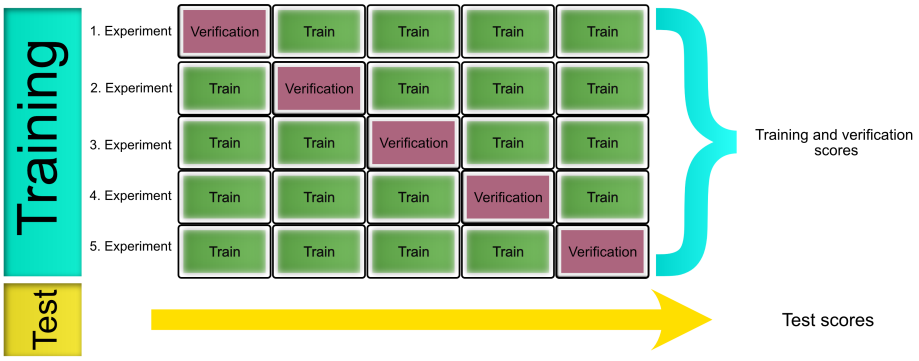

Figure 6.5: K-Fold cross validation (left) and stratified cross validation (right)

a) **Data Splitting:** Divide the dataset into K equal-sized folds, ensuring each fold represents a random and balanced representative subset of the data.
b) **Iteration Through Folds:** For each fold
   i) **Training:** Utilize $K-1$ of the folds as the training set. Train the model on this training data.
   ii) **Validation:** Use the remaining fold as the validation/test set. Evaluate the model's performance using appropriate evaluation metrics (e.g. , accuracy, precision, recall, F1-score, mean squared error) relevant to your problem type (classification or regression).[2]
   iii) **Recording Metrics:** Record the performance metrics for each fold, resulting in K sets of evaluation scores.

More about this later in the course

2. **Analyzing Results:** With the K sets of evaluation scores estimate
   i) **Average / Variance Performance:** Calculate the mean and variance of the K evaluation scores to obtain an overall estimate of your model's performance. This average provides a robust assessment of your model's expected performance on unseen data and the variance how stable it is in doing so. Smaller variation suggests consistent performance, while larger variability may indicate sensitivity to data splitting.
   ii) **Hyperparameter Tuning:** Utilize K-fold cross-validation for hyperparameter tuning. Evaluate performance for different hyperparameter configurations and select the best values based on cross-validation results.
3. **Model Selection:** When comparing multiple models, apply K-fold cross-

validation to each model. Compare their average performance and variability to select the best-performing model with stable performance across folds.

4. **Final Model Training:** After selecting the model and hyperparameters, train the final model using the entire dataset, without splitting it into folds, with the chosen configuration.
5. **Validation on Unseen Data:** Validate the selected model on entirely new, unseen data to assess its generalization capabilities.

Learning Goals

- Training performance and actual performance, which is what truly matters, are not equivalent.
- Training performance assesses the model's performance on the training data, but the real concern is how well it generalizes to entirely new data.
- Exceptional model performance during training can be misleading; it doesn't reveal whether you're overfitting, underfitting, or achieving true optimality.
- Increasing model complexity may result in a noticeable growth in standard deviation (the square root of variance) compared to bias reduction when tested on validation or debugging data. This is a sign of overfitting.
- Conversely, reducing model complexity may lead to bias in the predictions. This is a sign of underfitting.
- The model, representing the optimal balance, is the point where further bias reduction would disproportionately increase standard deviation, and vice versa. At this juncture, you've achieved the best possible equilibrium between bias and variance, maximizing your model's effectiveness. We could call this the "Goldilocks" model.
- The generalization error can be decomposed into a systematic error term (bias), a term depending on how much the trained models vary (variance), and a term that cannot be done anything about due to noise in the data.
-

$$\text{Err}(x) = \text{Bias}^2 + \text{Variance} + \text{irreducible error} \tag{6.3}$$

Note!

The following items indicate methods to identify and remidy overfitting.

TRAIN-VALIDATION-TEST SPLIT: Use a training set, a validation set, and a test set. If the model performs significantly worse on the validation or test set compared to the training set, it indicates overfitting.

LEARNING CURVE: Create learning curves illustrating the model's performance (e.g., accuracy or loss) on both the training and validation sets

as the number of training examples increases. In cases of overfitting, there will usually be gap between the training and validation curves. The training curve often achieves near-perfect performance, while the validation curve either levels off or begins to deteriorate.

CROSS-VALIDATION: Utilize k-fold cross-validation, where the dataset is divided into k subsets (folds), to assess the model performance. If the model performs significantly better on the training folds compared to the validation folds, it may be overfitting. When varying k a smaller the model is trained on relatively larger portions of the dataset , allowing it to potentially overfit to the training data and not generalize well to unseen data. As k becomes larger, each training and validation set becomes smaller. The variance among each fitted model (of fixed complexity) will reveal the stability as the training data becomes smaller.

FEATURE SELECTION: Simplify data by removing less important features or using feature selection techniques. A simpler model is less likely to overfit.

VISUAL INSPECTION: Plot actual vs. predicted values or residual plots (plot of difference) to visually inspect how well the model fits the data. Significant deviations from the diagonal line or patterns in residuals may indicate overfitting.

MODEL COMPLEXITY: Consider the complexity of your model. If you're using a complex model with many parameters (e.g., deep neural networks), it's more prone to overfitting. Simplifying the architecture or using techniques like dropout can help.

INCREASE DATA: Collect more data or use augmentation if possible. A larger dataset can make it harder for the model to memorize the training data.

7

Ridge Regression

In practice, the ground truth model ($h(x)$) is unknown, forcing us to make assumptions about the underlying model. Using the least squares solution may not necessarily lead to optimal model performance because our assumptions about the ground truth model can be inaccurate or even incorrect. The least squares loss function yields different loss values depending on the model parameters $\mathbf{w}$:

$$\overbrace{J(\mathbf{w})}^{\text{Loss function}} = \overbrace{\sum_{n=1}^{N} (y_n - f_w(x_n))^2}^{\text{Mean Squared Error}} \tag{7.1}$$

Learning is consequently about finding the right model parameters, w, such that the model makes as few errors as possible. The optimization algorithm has to deal with this choice and minimize the error from the estimated model to the data. When the model f_w is linear Strang showed that the least squares is solved using the pseudo inverse[1] As shown in chapter 6, models with added complexities may lack relevance when the model is exposed to new, unseen data. Consequently, the model's ability to generalize beyond the boundaries of the training dataset can be compromised, leading to overfitting.

The projection matrix from Strang is also called the *Normal equation*

Note!

Equation 7.1 can conveniently be written in matrix form as

$$J(\mathbf{w}) = (\mathbf{y} - f_w(\mathbf{x}))^\top (\mathbf{y} - f_w(\mathbf{x}))$$

where $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ is the vector of training outputs (labels) and $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ is the vector of inputs.

Penalty Term Regularization is a fundamental set of techniques used to prevent overfitting, improve model generalization, control the complexity of models, and prevent overfitting. This includes using complex models with limited data. When dealing with a small dataset, there's a higher risk of overfitting because the model might capture noise rather than meaningful patterns in the data.

There are many regularization techniques, but for now lets focus on a penalty-based regularization techniques that introduce a *penalty* term into the loss function, ensuring that the function not only includes differences between predicted and actual values but also factors in the significance assigned to the model parameters. The general form of a loss function with a penalty term is:

$$\overbrace{J(\mathbf{w})}^{\text{Loss function}} = \overbrace{E_D(\mathbf{w})}^{\text{Mean Squared Error}} + \overbrace{\lambda E_w(\mathbf{w})}^{\text{Penalty Term}} \tag{7.2}$$

where λ is the regularization coefficient controlling the trade-off between data-dependent error $E_D(w)$ and the regularization term $E_W(w)$. The added term discourages the model from attaining large weights. The regularization term is sometimes called *weight decay*.

The penalty-based regularisation presented in this section is on *Ridge regression* (also called *L2 regularization* since we are using the L2 norm), where the penalty term uses the squared Euclidean distance (L2 norm) of the model parameters. That is Equation 7.2 becomes:

$$\begin{aligned} \overbrace{J(\mathbf{w})}^{\text{Loss function}} &= \overbrace{\sum_{n=1}^{N} (y_n - f_w(x_n))^2}^{\text{Mean Squared Error}} + \lambda \overbrace{(w^\top w)}^{\text{Penalty term } \|w\|^2} \\ &= \|\mathbf{y} - f_w(\mathbf{x})\|^2 + \lambda \|w\|^2 \end{aligned} \tag{7.3}$$

The primary objective is to minimize the loss function, aiming to find model parameters that result in the best possible fit to the training data. For now, let's focus on understanding the purpose of the loss function. Those eager to delve into the solution process can anticipate a straightforward extension of the normal equations, which we'll explore later in this section.

The mean squared error (MSE) quantifies how effectively a model fits the training data by measuring the difference between predicted values and actual targets. In contrast, the regularization term plays a distinct role as a penalty component integrated into the loss function. It penalizes instances where model parameters grow excessively large. The underlying motive of regularization is to discourage the model from adopting undue complexity or acquiring large

parameter values. This introduction of a penalty term encourages the model to favor simpler, more generalizable parameter values. Specifically, L2 regularization (Ridge regularization) encourages model weight values to approach zero unless strongly supported by the data. This characteristic makes Ridge regularization particularly advantageous in scenarios featuring numerous correlated features.

In Figure 7.1, the two components in the minimization of Equation 7.3 are visualized. It exemplifies the delicate balance sought during model training that optimizing the fit to training data (minimizing the loss function) while preventing undue model complexity (minimizing the regularization term).

1. Minimizing the least squares term pushes the model to fit the training data as closely as possible, which helps reduce bias and increases the model's ability to capture complex relationships in the data.
2. Minimizing the regularization term discourages the model from having large parameter values, which helps reduce variance and prevents overfitting.

These two objectives are independent and are fine-tuned by the regularization parameter, which governs their relative importance. The regularization parameter, λ, controls the trade-off between these two objectives. A smaller regularization parameter places more emphasis on minimizing the loss function, potentially resulting in a model that fits the training data very well but is prone to overfitting. A larger regularization parameter places more emphasis on minimizing the regularization term, leading to a simpler model with reduced overfitting but potentially higher bias.

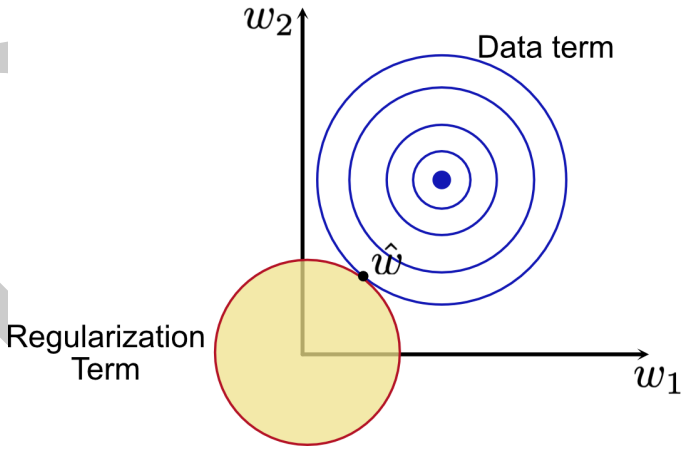

Figure 7.1: Minimizing the regularization loss involves minimization of both terms simultaneously.

Example 7.1: Loss function for Polynomial fit with Ridge regression

In the case of fitting a line polynomial with regularization, the loss function becomes:

$$J(\mathbf{w}) = \sum_{n=1}^{N} \left(y_n - w^\top \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right)^2 + \lambda \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^\top \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

.

Example 7.2: Loss function for Polynomial fit with Ridge regression

In the case of fitting a 3nd-order polynomial with regularization, the loss function becomes:

$$J(\mathbf{w}) = \sum_{n=1}^{N} \left(y_n - w^\top \phi(x_n) \right)^2 + \lambda \mathbf{w}^\top \mathbf{w}$$

where $\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix}$ and $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$

Example 7.3: Regularization for Polynomial regression

For the special case of polynomials, Equation 7.3 becomes:

$$\begin{aligned} J(\mathbf{w}) &= \sum_{n=1}^{N} \left(y_n - \mathbf{w}^\top \phi(x_n) \right)^2 + \lambda (\mathbf{w}^\top \mathbf{w}) \\ &= \|\mathbf{y} - w^\top \phi(\mathbf{x})\|^2 + \lambda \|w\|^2 \end{aligned}$$

where $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}$ and $\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^N \end{bmatrix}$ is the n-th order kernel mapping function.

Figure 7.2 provides an illustrative example of utilizing a limited number of data points, which would typically be insufficient for a Linear Least Squares fit. However, by incorporating regularization techniques, we can effectively constrain the model parameters to smaller values. This regularization-induced constraint is particularly evident in the figure, emphasizing the impact of the

regularization choice on the model parameters.

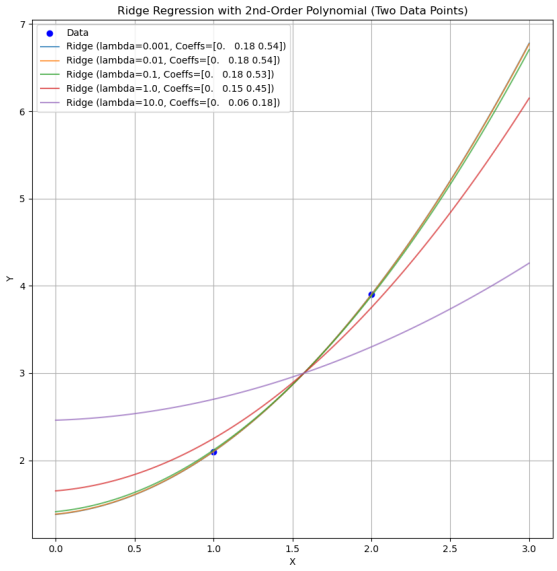

Figure 7.2: Fitting a second-order polynomial to only two data points might seem challenging due to the limited amount of data. However, by carefully choosing suitable values of the regularization parameter, denoted as λ, it is possible to achieve a robust fit for the model, overcoming the data scarcity inherent in such cases.

Figure 7.3 shows an example of fitting polynomials of varying orders (rows) to data with different levels of noise (columns). In the title of each figure are the estimated model parameters. As noise levels increase, the coefficients increase significantly. This happens because the model, in its effort to minimize the training loss, attempts to capture the noise in the data, which in turn forces the coefficients to increase.

Minimization of Loss function The previous paragraphs described the loss function but did not explain how to solve the minimization of the loss function. This section demonstrates, with a provided proof for completeness, that minimizing the least squares of the loss function Equation 7.3 is surprisingly simple and can be solved in closed form using a small extension of the Normal equations used for vanilla linear least squares[2].

Recall the unregularized solution is found using the projection/pseudo inverse / normal equations as $\hat{\mathbf{w}} = (A^\top A)^{-1} A^\top \mathbf{y}$

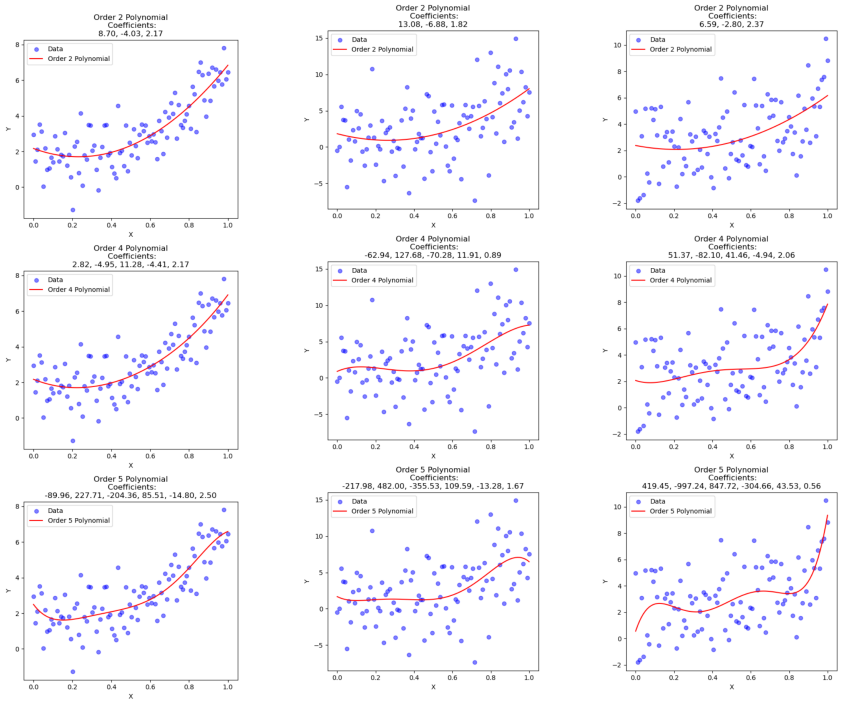

Figure 7.3: Plots of data points with varying levels of noise (left) less and (right) more. Fitted curves (red) of varying order. The ground truth function is a second order one. Notice how the coefficients of the fitted polynomials increase with higher noise levels

Theorem 7.1: Solution to Ridge regression

The solution $\hat{w}$ to a Linear Ridge regression problem can be found in closed form as

$$\hat{\mathbf{w}} = (A^\top A + \lambda I)^{-1} A^\top \mathbf{y}$$

where A is the designmatrix and $\mathbf{y}$ is the vector of labels.

Proof

To minimize the error function and determine the optimal weight vector $\mathbf{w}$, we seek the value of w that minimizes $J(w)$. Since the error function is quadratic in w, its derivatives with respect to w yield linear expressions in the elements of w. This allows us to find the unique solution $\hat{w}$ in closed form.
The proof has been provided due to its clarity but you can skip it at first reading. Rewriting Equation 7.3

$$J(w) = (y - Xw)^\top (y - Xw) + \lambda w^T w$$

Completing the multiplications

$$J(w) = y^T y - 2w^T X^T y + w^T X^T X w + \lambda w^T w$$

To find the optimal w, we take the derivative of $J(w)$ with respect to w and set it equal to zero:

$$\frac{dJ(w)}{dw} = -2X^T y + 2X^T X w + 2\lambda w = 0$$

Now, we can simplify this equation:

$$X^T X w + \lambda w = X^T y$$

Factoring out w, we get:

$$(X^T X + \lambda I) w = X^T y$$

To solve for w, we can take the inverse:

$$w = (X^T X + \lambda I)^{-1} X^T y$$

This is the solution for w in Ridge Regression. ■

8

Multivariate differentiation

This chapter aims to provide a short overview of univariate differentiation and its fundamental principles with the aim of extending these principles to multivariate differentiation.

Univariate differentiation is a foundational concept known from high school (or similar), allowing us to analyze how a single variable impacts a function. It forms the basis for more advanced topics that are important for machine learning and in particular in understanding learning of model parameters in non-linear models.

Let $y = f(x)$ be a (univariate) function. The derivative of y with respect to x is denoted as $f'(x) = \frac{dy}{dx}$. The derivative is based on infintesimal changes h near any point x. That is, the derivative is based on the slope of the tangent line in the infinitesimal small changes of the function around a specific point x

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Linear functions exhibit a constant rate of change throughout their domain, while non-linear functions have varying rates of change depending on the function. Derivatives play a crucial role in understanding these variations.

Example 8.1: Using limits to calculate the derivative of a univaritate function

This example demonstrates the application of derivative principles to calculate the rate of change at a specific point. The derivative of $f(x) = 2x$ at $x = 3$. Following the definition above

$$f'(3) = \lim_{h \to 0} \frac{2(3+h) - 2(3)}{h} = \lim_{h \to 0} \frac{2h}{h} = \lim_{h \to 0} 2 = 2$$

Note!

Univariate differentiation involve a set of differentiation rules.

1. **Sum or Difference Rule:** If $f(x) = u(x) \pm v(x)$, then $f'(x) = u'(x) \pm v'(x)$.
2. **Product Rule:** If $f(x) = u(x) \cdot v(x)$, then $f'(x) = u'(x) \cdot v(x) + u(x) \cdot v'(x)$.
3. **Quotient Rule:** If $f(x) = \frac{u(x)}{v(x)}$, then $f'(x) = \frac{u'(x) \cdot v(x) - u(x) \cdot v'(x)}{(v(x))^2}$.
4. **Chain Rule:** If $y = f(x) = g(u)$ and $u = h(x)$, then $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$.

8.0.1 Partial derivatives

When taking derivatives of multivariate functions (functions in several parameters) $f(\mathbf{x}) = f(x_1, \ldots, x_n)$ we will be talking about the partial derivative $\frac{\partial f}{\partial x_i}$ of f with respect to the parameter x_i. All other parameters than x_i are considered to be constants when differentiating f with respect to x_i ($\frac{\partial f}{\partial x_i}$). Besides from that the rules are quite similar (more about this later in the course). Each partial derivative is a scalar representing rate of change in a particular direction.

Example 8.2: Differentiation of bivariate function

Let

$$f_1(x, y) = 4x^2 + x + \sin(y)$$

then

$$\frac{\partial f_1}{\partial x} = 8x + 1$$
$$\frac{\partial f_2}{\partial y} = cos(y)$$

Example 8.3: Differentiation of bivariate function

Using the notation above and let the function f_2

$$f_2(\mathbf{x}) = 4x_1^2 + x_1 + \sin(x_2) + x_1 x_2$$

then

$$\frac{\partial f_2}{\partial x_1} = 8x_1 + 1 + x_2$$
$$\frac{\partial f_2}{\partial x_2} = \cos(x_2) + x_1$$

8.0.2 Gradient

The gradient (vector) of a multi-variable function (multiple rates of change) has a component for each direction. Thus, a function that takes n variables will have a gradient with n components. The gradient of a multivariate function f is denoted $\nabla f(\mathbf{x})$ and is the vector (function) of all partial derivatives of f w.r.t. the dependent variables x_i. Notice that each component in the gradient is a scalar that determines the slope in a particular direction (e.g. in a particular basis).

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{x}) \end{bmatrix}$$

where $\frac{\partial f}{\partial x_i}$ is the partial derivative of f with respect to the parameter x_i.

The gradient of $f(\mathbf{x}) = 4x_1^2 + x_1 + \sin(x_2) + x_1 x_2$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 8x_1 + 1 + x_2 \\ \cos(x_2) + x_1 \end{bmatrix}$$

Gradient points in the direction of most change An important property of the gradient $\nabla f(\mathbf{x})$ of a multivariate function $f(\mathbf{x}) = f([x_1, \ldots, x_N])$ is its indication of the direction of the greatest increase of f at a given point $\mathbf{a}$:

$$\nabla f(\mathbf{a}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{a}), \frac{\partial f}{\partial x_2}(\mathbf{a}), \ldots, \frac{\partial f}{\partial x_N}(\mathbf{a}) \right]^\top$$

When we take the dot product of the gradient vector with a unit vector in a specific direction, we obtain the *directional derivative* along that direction. Let $\mathbf{u} = [u_1, u_2, \ldots, u_N]^\top$ represent the unit vector denoting the direction in which we want to evaluate the change of f. The directional derivative of f at $\mathbf{a}$ in the direction of $\mathbf{u}$ is given by:

$$D_{\mathbf{u}} f(\mathbf{a}) = \nabla f(\mathbf{a}) \cdot \mathbf{u}$$

The direction in which the function exhibits the steepest ascent corresponds to the direction of the maximum directional derivative. It can be mathematically demonstrated that the dot product of the gradient vector and the unit vector in

this direction is maximized when the unit vector aligns with the gradient vector itself. Therefore, the gradient vector points in the direction of the greatest increase because, in that direction, the rate of change of the function is maximized.

8.1.0 Filters and gradients

This section provides a deeper exploration of the interplay between local processing, convolution, and partial derivatives within the context of discrete functions. While Paulsen and Moslund touch upon these concepts, this discussion aims to offer a more nuanced understanding.

Filtering and gradients have a suprinsing connection in which we can use descrete filters to calculate the gradients in discrete functions. This section devels into these aspects for 1D and 2D (image) signals.

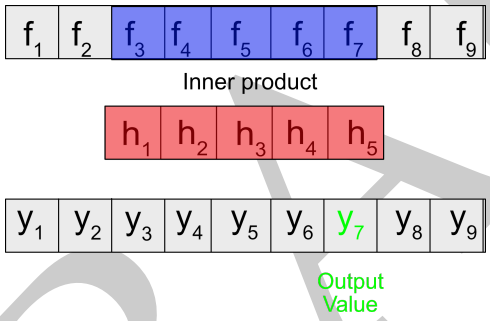

Figure 8.1: Convolution is just inner products calculated with a sliding window (blue) and the filter (red)

1D filtering Assume that we are given a one-dimensional signal $f = [f_1, f_2, \ldots, f_N]$, where N is the length of the signal (e.g. a sound signal). Filtering operates through a filter or kernel $\mathbf{h} = [h_1, h_2, \ldots, h_M]$ of length M. The *convolution* operation is illustrated in Figure 8.4 and is computed as:

$$\mathbf{f} * \mathbf{h}(n) = \mathbf{y}_n = \sum_{m=0}^{M-1} h_m \cdot f_{n-m}$$

Where y_n (green) represents the n-th element of the resulting filtered signal $\mathbf{y}$. The process involves sliding the filter (red) along the input signal (blue), multiplying the filter coefficients with the corresponding signal values, and summing the products to obtain the filtered output at each position. It is important to realize that this process is just calculating the inner product of the filter (red) with the local values (blue) of the signal.

Gradients of discrete functions This paragraph will link the use of filters (convolution) with calculating derivatives of discrete functions (such as image and sound signals).

Consider a discrete univariate function f [1]. The approximation of the partial derivative at a point x is given by:

Represented as values in a vector, matrix, or tensor

$$\frac{dy}{dx} = \frac{f(x+1) - f(x)}{1} \tag{8.1}$$

Visualizing this process in Figure 8.3, the derivative at a specific point is conceptualized as the inner product of the signal f with the filter (vector depicted in yellow). This filter has zeros everywhere except for positions adjacent to the specific point, containing a -1 on the left and a 1 on the right. Effectively, this inner product operation represents the convolution of f with the kernel $\begin{bmatrix}-1 & 1\end{bmatrix}$. The green box of -1 is moved along the yellow vector, computing the gradient at each point in f.

Extending beyond univariate functions, the concept of partial derivatives can be seamlessly applied to images or higher-order discrete functions. For a two-dimensional function $f(x, y)$, the partial derivatives with respect to x and y can be approximated as:

$$\frac{\partial f}{\partial x} = \frac{f(x+1, y) - f(x, y)}{1}$$

$$\frac{\partial f}{\partial y} = \frac{f(x, y+1) - f(x, y)}{1}$$

The convolution operation involves kernels $\begin{bmatrix}-1 & 1\end{bmatrix}$ for $\frac{\partial f}{\partial x}$ and $\begin{bmatrix}-1 \\ 1\end{bmatrix}$ for $\frac{\partial f}{\partial y}$.

For a three-dimensional function $f(x, y, z)$, the partial derivatives with respect to x, y, and z can be approximated in a similar manner:

$$\frac{\partial f}{\partial x} = \frac{f(x+1, y, z) - f(x, y, z)}{1}$$

$$\frac{\partial f}{\partial y} = \frac{f(x, y+1, z) - f(x, y, z)}{1}$$

$$\frac{\partial f}{\partial z} = \frac{f(x, y, z+1) - f(x, y, z)}{1}$$

In practical applications, the use of a simple $\begin{bmatrix} -1 & 1 \end{bmatrix}$ *filter kernel* is sensitive to noise. To overcome this limitation filters such as the Prewitt and Sobel filters in Paulsen and Moslund, are used to computing partial derivatives taking a larger area of the image into account. These filters (or kernels) are applied to the image using convolution to highlight changes in intensity, acting as indicators for edges in gray scale images. Both Prewitt and Sobel filters employ two kernels: one for capturing vertical changes and another for horizontal changes, each comprising weighted values. Hence, applying the derivative filters in the two directions to the image I will yield two output images I_x and I_y containing the derivatives in x and y directions, respectively. The gradient of the the image in a specific pixel is therefore and as shown in Figure 8.2.

$$\nabla I(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x}(x,y) \\ \frac{\partial f}{\partial y}(x,y) \end{bmatrix} = \begin{bmatrix} I_x(x,y) \\ I_y(x,y) \end{bmatrix}$$

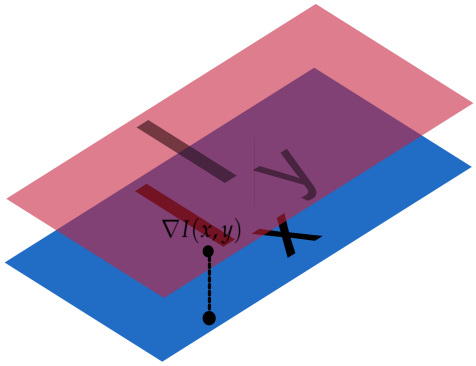

Figure 8.2: The gradient of the image in a specific point (x,y) is calculated through the two images of partial derivatives

8.2.0 Gaussian smoothing and Differentiation

Despite their effectiveness in accentuating edges, Prewit and Sobel filters are prone to noise, resulting in undesirable responses. This help with noise reduction and enhances the accuracy of edge detection, additional preprocessing steps, such as Gaussian smoothing, can be implemented before applying further processing of the signal.

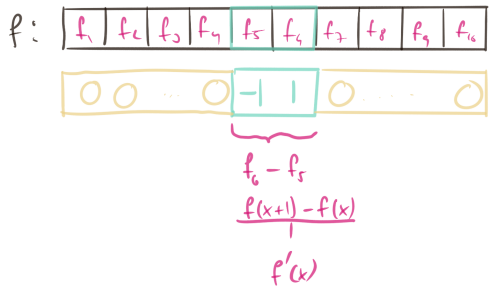

Figure 8.3: Differentiation of a discrete signal f is equivalent to convolution of f with the with a kernel of -1 and 1.

1D filters The univariate Gaussian function is defined as :

$$g_{\sigma,\mu}(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

and shown in Figure 8.4, where σ represents the standard deviation, and μ is the mean. In the context of filtering, the mean is consistently set to $\mu = 0$, and the standard deviation σ becomes a parameter determining the width of the filter. Throughout the discussion that follows, we implicitly assume $\mu = 0$.

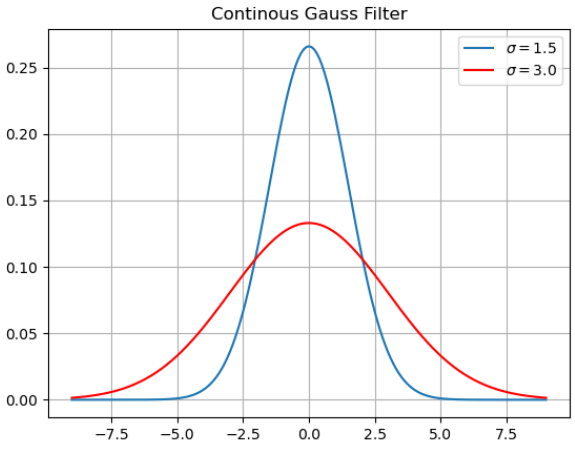

Figure 8.4: Gaussian filters with standard deviation $\sigma = 1.5$ and $\sigma = 3.0$

In cases of discrete signals and filters (e.g. representing the pixels of an image) the discrete Gaussian filter is shown in Figure 8.5 and is expressed as:

$$g_\sigma(u) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{u^2}{2\sigma^2}\right)$$

where u takes integer values within the range $u \in [-3\sigma, \ldots, 3\sigma]$.

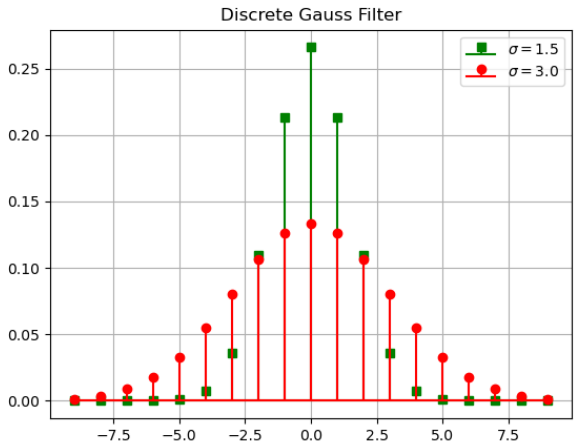

Figure 8.5: Discrete Gaussians with zero mean and standard deviations given in the legend.

The outcome, $h(i)$ of applying discrete Gaussian filtering (inner product / convolution in Equation 8.1) to a 1D discrete signal $f(i)$ is given by:

$$h(i) = \sum_{u=-z}^{z} \overbrace{\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{u^2}{2\sigma^2}\right)}^{\text{Gaussian kernel}} \underbrace{f(i-u)}_{\text{signal function}}$$

The first-order derivative of the (zero-mean) Gaussian function with standard deviation σ, $g_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right)$, can be computed as:

$$\begin{aligned} g'_\sigma(x) &= \frac{\partial g_\sigma(x)}{\partial x} \\ &= -\frac{x}{\sigma^2} \cdot \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right) \\ &= -\frac{x}{\sigma^2} \cdot g_\sigma(x) \end{aligned}$$

Figure 8.6 shows the 1D Gaussian, its first and second order derivatives. Filtering a signal with the derivative of the gaussian with a fixed value of σ es-

sentially performs the same task as first blurring the signal, f, with the Gaussian and then calculates its derivative, that is $\frac{\partial}{\partial x}(f \star g) = \frac{\partial f}{\partial x} \star g = \frac{\partial g}{\partial x} \star f$, where $\star$ is the convolution operator, h

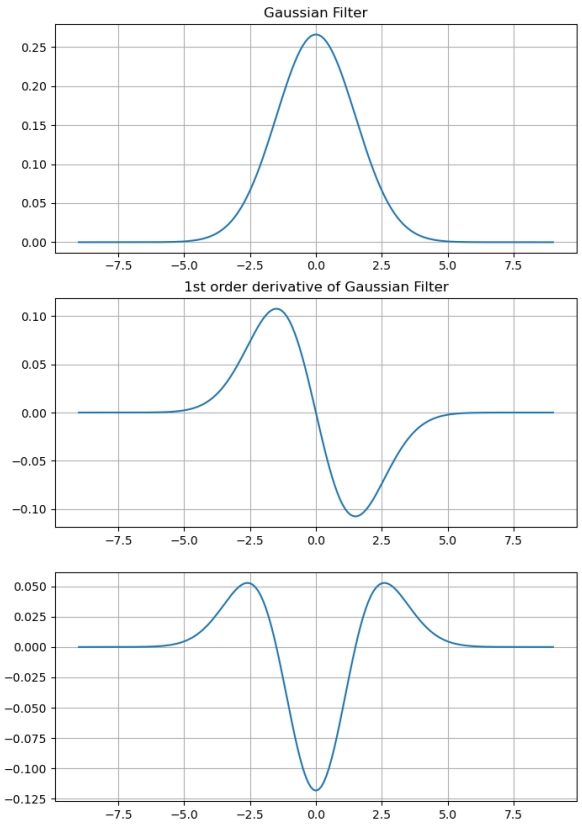

Figure 8.6: 1D Gaussian and its first and second order derivatives.

Convolution of a 2D signal such as an image, with a Gaussian kernel effectively blurs the signal while preserving its overall structure.The 2D Gaussian filter (zero mean and same variance in both direction) is shown in Figure 8.7 and defined by:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

where (x, y) represents the spatial coordinates, and σ is the standard devi-

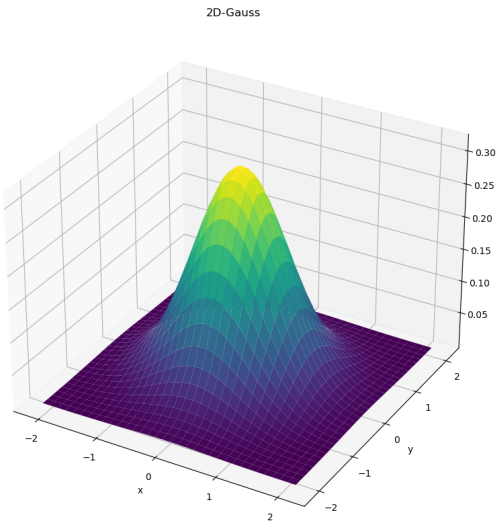

Figure 8.7: 2D Gaussian

ation, determining the spread of the Gaussian. This bell-shaped curve assigns higher weights to central pixels and lower weights to peripheral ones, ensuring that the filter imparts more significance to nearby pixels during the convolution process.

The parial derivativeof the 2D Gaussian respect to x is given by:

$$\frac{\partial G}{\partial x} = -\frac{x}{\pi\sigma^4} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

Similarly, the first derivative with respect to y is:

$$\frac{\partial G}{\partial y} = -\frac{y}{\pi\sigma^4} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

The partial derivatives of the Gaussian is shown in Figure 8.8.

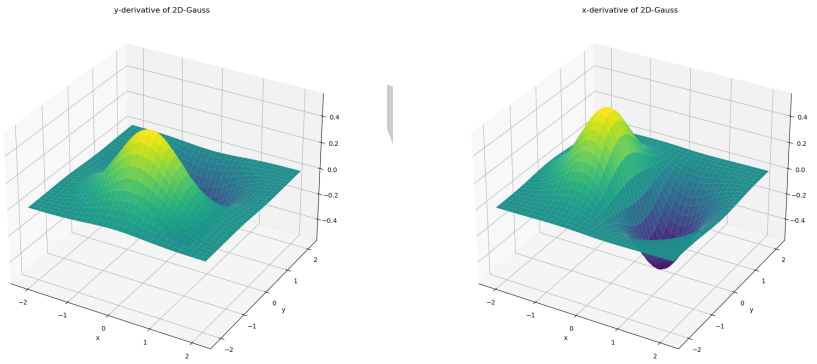

Figure 8.8: Partial derivatives of Gaussian

9

Classification

This chapter aims to fill gaps in existing classification material, connecting content from previous chapters with other reading materials, while acknowledging its intent not to be exhaustive

A **classifier** $f_w(\mathbf{x})$ is a model that categorizes data points into discrete classes, $\mathcal{C}_1, \cdots, \mathcal{C}_k$.

$$f_w : \mathbb{R}^M \to \{\mathcal{C}_1, \cdots, \mathcal{C}_k\}$$

For the special case of *Binary classification* f_w is $f_w : \mathbb{R}^M \to \{\mathcal{C}_1, \mathcal{C}_2\}$ can classify the data into two classes. It is convenient that f_w either returns -1 or 1 (or in some cases 0 and 1). That is $f_w : \mathbb{R}^M \to \{-1, 1\}$.

For example given an image $\mathbf{x}$, $f_w(\mathbf{x})$ may yield 1 if there is a cat in the image and -1 otherwise.

A *linear classifier* categorizes data points by considering a linear combination of features. Its operation is reminiscent of linear regression but with binary outputs. The process of learning the model parameters, $\mathbf{w}$, closely resembles linear least squares regression, but in this case the outputs can only take two possible values.

$$f_w(x) = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases} \tag{9.1}$$

where $\mathbf{w} = \begin{bmatrix} w_n \\ w_{n-1} \\ \vdots \\ w_1 \\ w_0 \end{bmatrix}^\top$ and $\mathbf{x} = \begin{bmatrix} x_n \\ x_{n-1} \\ \vdots \\ x_1 \\ 1 \end{bmatrix}$. Notice that we here use homogeneous

coordinates to represent the hyperplane of the model parameters $\mathbf{w}$ and inputs features $\mathbf{x}$.

Equation 9.1 can be rewritten as

$$f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x}) \tag{9.2}$$

where $\text{sign}(z)$ returns 1 if $z \geq 0$ and -1 otherwise. Notice that the sign function is characterized as an *activation* function - a concept that will be further developed in the following lectures. This classifier assigns a data point to one of two classes based on the sign of the dot product between the feature vector $\mathbf{x}$ and the model parameters $\mathbf{w}$.

Hence given a 2d input $\mathbf{r}$

$$\begin{aligned} f_w(x) &= \text{sign}(\mathbf{w}^\top \mathbf{x}) \\ &= \text{sign}(w_2 x_2 + w_1 x_1 + w_0) \end{aligned}$$

where $\mathbf{w} = \begin{bmatrix} w_2 \\ w_1 \\ w_0 \end{bmatrix}^\top$ and $\mathbf{x} = \begin{bmatrix} r_2 \\ r_1 \\ 1 \end{bmatrix}$.

Notice that the equation $w_2 x_2 + w_1 x_1 + w_0 = 0$ represents a line, serving as the *decision boundary* that separates the two classes as shown in Figure 9.1. Similarly, in the case of an N-dimensional input, Equation 9.2 yields a hyperplane.

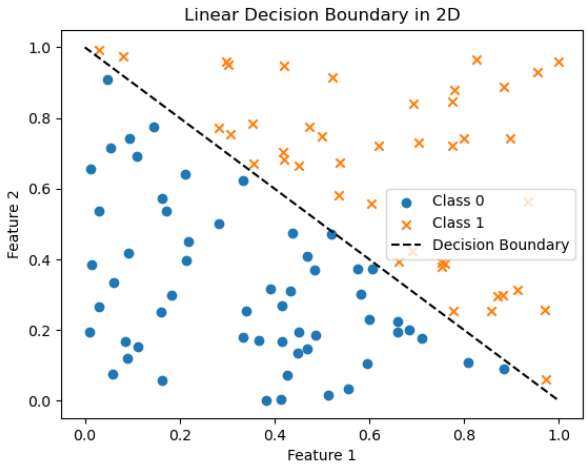

Figure 9.1: Linear decision boundary for classification.

Separability *Separability* refers to the ability to define decision surfaces (boundaries) in a dataset. If the data is *linearly separable*, it means that individual classes

can be cleanly separated by linear decision boundaries (lines, planes and hyperplanes) as shown in Figure 9.1. For data that is not linearly separable, more complex functions are needed to define decision surfaces. Consider each instance in the dataset as a point in a d-dimensional space, where each feature represents a dimension. Linear decision boundaries are linear functions of the input features ($\mathbf{x}$) and are $D-1$ dimensional hyperplanes in a D-dimensional input space. It is rare that the data can be separated cleanly, whether linearly or non-linearly. In a similar way to regression it is possible define a loss for classification to provide a measure of how good the classifier works with specific model parameters $\mathbf{w}$. The goal of classification is to minimize the loss function.

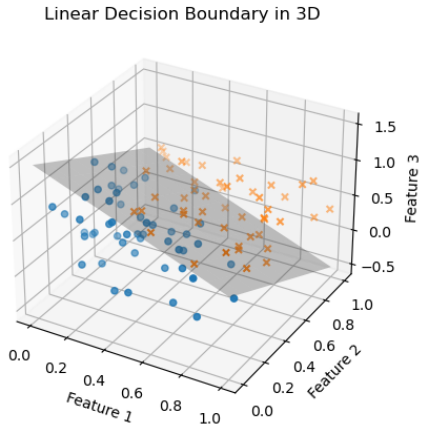

Figure 9.2: 3D decision boundary (plane) separating 3D features.

9.1.0 Polynomial Classification

Similar to regression, polynomials may be used for classification. While linear classifiers are effective for linearly separable data, polynomial classifiers extend the capability to handle more intricate relationships within the data. While linear classifiers excel in scenarios where the data is linearly separable, polynomial classifiers broaden this capability, allowing for the modeling of more complex relationships within the data. Learning the model parameters follows the same procedure as for regression.

Note!

A polynomial (single variate) can be written as

$$P_N(x) = w_0 + w_1 x + w_1 + w_2 x^2 + \cdots + w_N x^N \tag{9.3}$$
$$= \begin{bmatrix} w_0 & w_1 & \cdots w_N \end{bmatrix} \phi(x) \tag{9.4}$$

where $\phi(x)$ is a vector function that given an input value x yields a vector

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \cdots \\ x^N \end{bmatrix}$$

ϕ is called a *feature map* and is essentially mapping an input into a N dimensional space. When dealing with non-linearly separable data, such as the concentric circles, the addition of polynomial features creates a higher-dimensional space where the data might become linearly separable.

Consider a binary classification problem where we have two classes, $\mathcal{C}_1$ and $\mathcal{C}_2$, and a feature vector $\mathbf{x} = [x_1, x_2]$. A linear classifier might use a decision boundary of the form $w_0 + w_1 x_1 + w_2 x_2 = 0$. However, if the data is not linearly separable, we can introduce polynomial terms to the decision function:

$$\begin{aligned} f(\mathbf{x}) &= w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 + w_5 x_1 x_2 + \ldots \\ &= \sum_{i=0}^{k} w_i \phi_i(\mathbf{x}) \\ &= \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) \end{aligned}$$

For a polynomial kernel of degree d, the feature mapping $\boldsymbol{\phi}(\mathbf{x})$ might look like:

$$\boldsymbol{\phi}(\mathbf{x}) = \left[1, x_1, x_2, x_1^2, x_2^2, x_1 x_2, \ldots, x_1^d, x_2^d, (x_1 x_2)^d, \ldots\right]$$

And the weight vector $\mathbf{w}$ would have corresponding values:

$$\mathbf{w} = [w_0, w_1, w_2, w_3, w_4, w_5, \ldots, w_{d-1}, w_d, w_{d+1}, \ldots]$$

Provided that there is enough data to setup the linear equations and the Designmatrix, the values of $\mathbf{w}$ are learned using linear least squares. The kernel

function defines how your input features are transformed into this higher-dimensional space. Different kernels (polynomial, sines etc.) will result in different $\boldsymbol{\phi}$ mappings and, consequently, different types of decision boundaries. Notice that learning the model parameters w is linear. A quadratic classifier, a specific case of polynomial classification, uses a decision boundary represented by a quadratic equation. The decision function takes the form:

$$f(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 = 0$$

As shown in Figure 9.3 the decision boundary is a conic section, often visualized as an ellipse or a hyperbola. By including quadratic terms, the classifier can capture curved decision boundaries, making it suitable for datasets with nonlinear class relationships. Essentially, the polynomial features elevate the data into a higher-dimensional domain where a hyperplane (linear decision boundary) can effectively separate the classes. In the figure, each degree of the polynomial corresponds to a different transformation, projecting the data into progressively higher-dimensional spaces. The trick of using kernls shows that a well-chosen transformation can turn a seemingly complex problem into a more manageable one by revealing hidden patterns in the data (in a higher dimensional space)

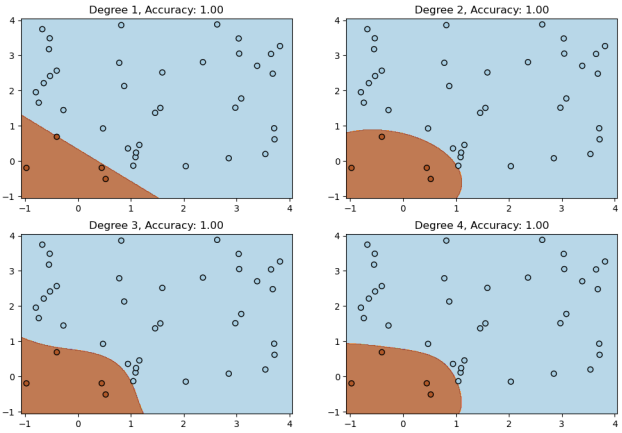

Figure 9.3: Polynomial Classification of different orders

Overfitting Higher order terms will result in more flexible decision boundaries. While polynomial classifiers offer flexibility in capturing complex decision boundaries, there is a risk of overfitting, especially when dealing with high-

degree polynomials. Regularization techniques and careful model evaluation are crucial to mitigate overfitting also in polynomial classifiers.

9.2.0 Unbalanced datasets

Class Imbalance arises when there is an uneven distribution of samples among different classes. Oftentimes achieving perfectly balanced datasets is rare especially when the topic under scrutiny contain inherently 'rare' events precisely because they are challenging to predict. This imbalance can lead to suboptimal model performance, especially for the minority class. Real-world scenarios, such as fraud detection, spam detection, and churn prediction, often involve imbalanced class distributions. For example suppose we are supposed to determine a method that discovers fraudulent credit card transactions. The vast majority of these transactions are legitimate, and only a few are fraudulent. Similarly, if we are testing individuals for cancer, or for the presence of a virus (COVID-19 included), the positive rate will (hopefully) be only a small fraction of those tested.

The example in Figure 9.4 uses a linear classification model, where linear regression is employed to predict class labels. The objective of this model is to minimize the linear least square loss function, denoted as $\mathcal{L}(w)$. This loss function is defined as the sum of squared differences between the actual class labels (y_i) and the predicted values ($f_w(x_i)$) for each data point:

$$\mathcal{L}(w) = \sum_{i=0}^{N} (y_i - f_w(x_i))^2$$

For sake of simplicity let's consider a simplified scenario where the training data is divided into two classes and where the first K data points belong to the larger class, while the remaining data points belong to the smaller class. Since the larger class contributes more data points and, consequently, more losses during the minimization process, the resulting solution tends to move the decision boundary closer to the larger class. This bias arises from the model's effort to reduce the overall loss by prioritizing the larger class, which has a more impact on the total loss. The consequence is a decision boundary that may not adequately represent the underlying data distribution. As a result, these classifiers tend to ignore small classes while concentrating on classifying the large ones accurately.

Figure 9.4: Linear classifier on an unbalanced dataset where the distribution is 90% blue and 10% red.

Under and Oversampling Achieving a balanced distribution is crucial in addressing class imbalance, and two common strategies are employed for this purpose:

UNDER-SAMPLING involves randomly removing samples from the majority class until a balance is achieved with the minority class. While effective in balancing class proportions,information loss from the majority class is a potential drawback.

OVER-SAMPLING Simple approaches entails duplicating random samples from the minority class until balance is reached. However, a notable concern with this technique is the risk of overfitting towards the minority class. To mitigate this risk, recent methods leverage augmentation and generative models to create synthetic data samples for the minority class, such as generating synthetic instances along the line segments connecting existing minority class instances, assuming smoothness in the data.

[1]

More advanced strategies and implementations can be found here `https://imbalanced-learn.org/stable/over_sampling.html`

10

Evaluations and Metrics

This chapter delves into the domain of metrics, which serve as instruments for gauging the degree of similarity or dissimilarity between entities. Whether evaluating the similarity of two vectors of observations or assessing the performance of a supervised model (be it regression or classification) the concept of distance and similarity are central.

10.1.0 Metrics

Suppose we have an observation a, and our objective is to determine whether this observation predominantly aligns with observations of oranges (b) or lemons (c). To achieve this, we require a *distance measure* between two observations a, b and a, c that yields a large value when they are dissimilar and 0 when they are identical. The *dissimilarity* between observations is commonly gauged by the difference between them. A prevalent method for defining distance, or *Norm*, is to consider the magnitude of the difference of observations, $\mathbf{x} = \mathbf{a} - \mathbf{b}$. Norms serve to quantify the size or length of a vector or mathematical object, while similarity measures gauge the resemblance or likeness between two objects. It is important to note that norms and similarities are not necessarily equivalent. The length of a vector can be used as a similarity measure, but they are distinct concepts.

No doubt, the most familiar measure of similarity is the Euclidean distance $\|x\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_M^2}$. As we will shortly see there are other yardsticks to determine what "similar" means. A somewhat more formal approach is to require measure of distance or a *metric* obeys the following rules.

1. Non-negativity: $d(x, y) \geq 0$,
2. Identity of indiscernibles: $d(x, y) = 0$ if and only if $x = y$,

3. Symmetry: $d(x, y) = d(y, x)$,
4. Triangle inequality: $d(x, y) \leq d(x, z) + d(z, y)$.

For instance, we like the distance metric to always be positive or zero. When it is zero the two entities are identical. The triangle inequality is saying the distance from home to the workplace is not greater than the distance from the workplace to the baker and from the baker to home.

The Euclidean distance

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_M^2}.$$

is also called the L2-norm or Euclidean norm. Generally we can define *p-norm*, $\|x\|_p$, where $p \geq 1$:

$$\|x\|_p = (|x_1|^p + |x_2|^p + \ldots + |x_M|^p)^{\frac{1}{p}}$$

Hence by setting $p = 2$ we have the Euclidean norm other distance measures can be obtained by chainging p.

The *L1-norm* ($p = 1$) also known as the *1-norm*, *Manhattan distance* or the *absolute norm* is defined as the sum of the absolute values of its components. For a vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_n \end{bmatrix}^\top$

$$\|\mathbf{x}\|_1 = |x_1| + |x_2| + \ldots + |x_n|$$

where $|\cdot|$ is the absolute value. As shown in Figure 10.1 the L1 norm, also known as the exhibit diamond shaped contours of constant distance. The L1-norm is considered less sensitive to outliers due to absolute values being less influenced by large values. The L1 norm's robustness to outliers, ability to induce sparsity, and computational efficiency make it a valuable when dealing with high-dimensional data or scenarios with noisy or outlier-prone observations.

The *infinity norm* ($p = \infty$) is defined as

$$\|x\|_\infty = \max\{|x_1|, |x_2|, \ldots, |x_M|\}$$

and which measures the largest difference in coordinates of the vector $\mathbf{x}$.

Definition 10.1: p-Distance

The p-distance is defined as

$$d_p(\mathbf{x}, \mathbf{y}) = \begin{cases} \left(\sum_{i=1}^{M} |x_i - y_i|^p\right)^{\frac{1}{p}} & \text{if } 1 \leq p < \infty \\ \max\{|x_1 - y_1|, |x_2 - y_2|, \ldots, |x_M - y_M|\} & \text{if } p = \infty \end{cases}$$

Figure 10.1 shows 1000 observations and those colored in red have a p-distance less than 1 to the point $(0, 0)$, e.g. $d_p(\mathbf{x}, \mathbf{0}) \leq 1$, for six different values of p.

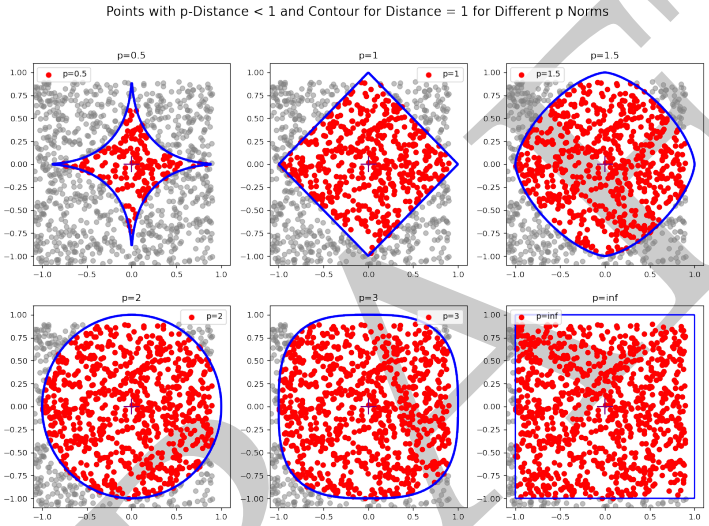

Figure 10.1: 1000 randomly generated points. All the red points have distance 1 or less to the center (purple '+'), according to the given norm.

Note!

Reflecting on the previous weeks, you may now realize that we have exclusively utilized the 2-norm for minimizing loss functions (linear least squares and in extended form for regularization). Now we can change the norm providing different yardsticks to evaluate the models. Such a change is at the moment not so easy. The techniques introduced in Strang's book and previously in these lecture notes are minimizing the 2-norm. Consequently, we find ourselves in need of alternative methods to accommodate different metrics. Do not be discouraged; we will delve into this in a subsequent chapter.

p-norm Similarities Vector similarity measures are fundamental in machine learning to quantify the likeness between vectors or data points. The norms previously presented can be used as measures of similarity using some $p - norm$

$$\text{sim}_p(x, y) = |x - y|_p$$

Manhattan distance (L_1) and Euclidean distance (L_2) are commonly used metrics. Manhattan distance sums absolute differences along each dimension, making it suitable for scenarios with grid-like movement or varying dimension importance. In contrast, Euclidean distance calculates the straight-line distance between points and is ideal when all dimensions contribute equally. In most cases, Manhattan distance yields larger values than Euclidean distance for the same point pair. As data dimensionality increases, Manhattan Distance becomes the preferred choice.

Inner product and Cosine Similarity The inner product is often used similarity metric, giving values in the entire range of $\mathbb{R}$. Larger inner product signify greater similarity, with 0 indicating perpendicular vectors.

The inner product can be derived from the cosine equation by multiplying the cosine of the angle between two vectors by the lengths of both vectors:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= \mathbf{a}^\top \mathbf{b} \\ &= \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta) \end{aligned}$$

The inner product is consequently sensitive to vector length and the angle between the vectors.

The *Cosine similarity* ($\cos(\theta)$) is a metric that ranges from -1 (indicating dissimilarity) to +1 (indicating high similarity) measuring as the angle between two vectors. It is computed by dividing dot of the vectors of their lengths:

$$\text{Cosine Similarity} = \frac{\mathbf{a}^\top \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

The inner product similarity shares similarities with cosine similarity; however, the key distinction lies in how it accounts for vector lengths. While cosine similarity is less sensitive to both vector direction and magnitude, the inner product similarity provides a different perspective by considering the product of corresponding components without normalizing by the vector lengths.

10.2.0 Regression Metrics

Evaluating the performance of a regression model is crucial for assessing how effectively it captures the underlying relationships in the data. The selection of appropriate metrics depends on the specific characteristics of the data and the nature of the regression task. Several metrics are commonly used to assess the accuracy and precision of predictions. A fundamental metric is the *Mean Absolute Error (MAE)*, which measures the average absolute difference between predicted ($\hat{y}_i = f_w(x_i)$) and actual (y_i) values:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

The *Mean Squared Error (MSE)* calculates the average of the squared differences between predicted and actual values:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

MAE and MSE share similarities but differ in their characteristics, often leading to a trade-off between robustness and computational simplicity. MAE assigns equal weight to all errors and maintains the unit of the original data, regardless of their magnitude. Since it considers absolute differences it is less influenced by outliers. On the other hand, MSE squares the errors, giving more weight to larger errors and making it more sensitive to outliers. MSE might be considered simpler in certain contexts due to its differentiability everywhere, facilitating smoother gradients. This property will become important when we cover gradient-based optimization techniques. Additionally, squaring errors aids in achieving numerical stability, particularly when dealing with large or small numbers.

The *Root Mean Squared Error (RMSE)* is then obtained by taking the square root of the MSE:

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

RMSE is expressed in the same units as the target variable, making it easy to interpret. For example, when predicting house prices in kroner, the RMSE will also be in kroner squaring the errors in RMSE gives more weight to large errors. This means that RMSE is sensitive to outliers or large errors, making it a useful metric when such errors are particularly important. In the presense of outlierss alternative metrics like Mean Absolute Error (MAE) or R-squared may be preferred depending on the characteristics of the data and the goals of the analysis.

In scenarios where the prediction errors are expected to exhibit exponential growth or decay, the *Root Mean Squared Logarithmic Error (RMSLE)* can be valuable. This metric calculates the logarithmic differences between predicted and actual values before applying the standard RMSE:

$$\text{RMSLE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(1+y_i) - \log(1+\hat{y}_i))^2}$$

The R^2 metric is a statistical measure ranging from 0 to 1 that assesses the *goodness of fit* of a regression model

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

It indicates the proportion of the variance in target values that is predictable from the input features by the model. y_i is the actual value (labels in the trainingset), $\hat{y}_i = f_w(x)$ is the predicted value, and $\bar{y}$ is the mean of the labels in training data. The numerator calculates the sum of squared residuals (the differences between actual and predicted values), and the denominator calculates the total sum of squared differences between actual values and their mean. $R^2 = 1$ implies a perfect fit, meaning the model perfectly predicts the variation in the outputs. $R^2 = 0$ indicates that the model does not explain any variability in the outputs and is as good as predicting the mean of the labels. $R^2 < 0$ suggests that the model is worse than a simple mean prediction. While a higher R^2 value indicates a better fit of the model to the data, R^2 should be interpreted cautiously. It is recommended to use it in conjunction with other evaluation metrics, especially in cases where model complexity and overfitting might be concerns. One great thing about R^2 is that it has an intuitive scale that does not depend on the units of the target variable. It doesn't matter if you're predicting prices, distance, weight or something else. However it also says nothing about the prediction error of the model, which is quite important in most cases. That's why typically

you may want to compliment R^2 with a metric that helps you understand the error as well.

10.3.0 Binary Classification Metrics

Binary classification involves two classes, here denoted as positive (P) and negative (N). The classifier can either make the correct prediction or not for instances belonging to either the positive or negative class. There are four distinct combinations representing the alignment between the actual class of an observation and the class predicted by the model. The overall performance can be summarized in a 2×2 matrix called the *confusion matrix* distinguishing between true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN)

$$\begin{bmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{bmatrix}$$

A *true positive* are instances that are correctly predicted as positive. The number of true positives is denoted as TP.

A *true negative* Instances that are correctly predicted as negative. The number of true negatives is denoted as TN.

A *false positive* (FP) ("TYPE I ERROR"): Occurs when the classifier predicts the positive class when it should have predicted the negative class. The number of false positives is denoted as FP.

A *false negative* (FN - "TYPE II ERROR"): Occurs when the classifier predicts the negative class when it should have predicted the positive class. The number of false negatives is denoted FN.

The impact of miss-classifications The impact of erroneous classifications varies across applications. The decision to prioritize minimizing False Positives or False Negatives depends on the specific context and the repercussions associated with each type of error. Take, for instance, the development of a spam filter, where the positive class signifies spam. Here, the primary objective is to minimize False Positives. Incorrectly labeling a message as spam (False positive) results in its removal, potentially eliminating pertinent information. Conversely, in medical applications the emphasis leans towards minimizing False Negatives over False Positives. In this context, the positive class denotes the presence of a disease. False Negatives, indicating a failure to detect the disease when it's genuinely present, can be detrimental. A missed diagnosis might lead to delayed medical interventions, deteriorating the patient's health. While a False Positive (incorrectly suggesting illness when it's not present) may lead to unnecessary treatments, patient inconvenience, and increased harm and costs.

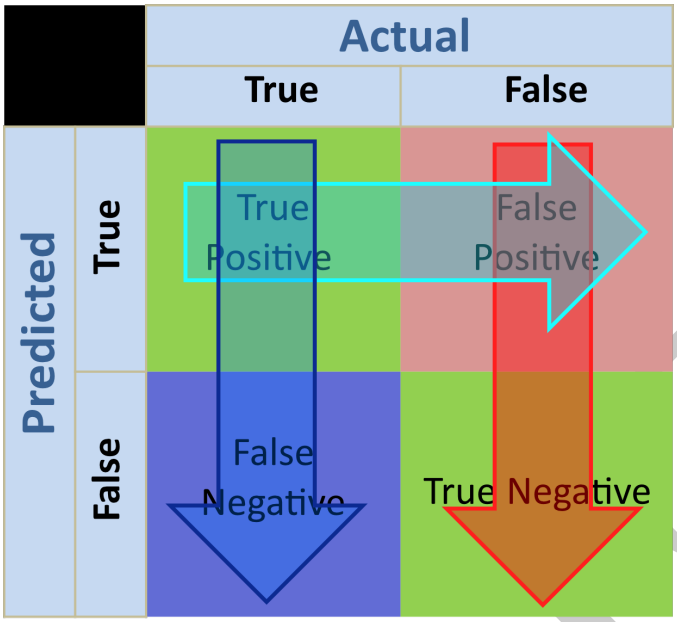

Figure 10.2: Confusion matrix with arrows indicating which measures are included in calculating (yellow) precision, (blue) Recall and (red) specifity and False Positive rate

Fundamental classification metrics Understanding the confusion matrix and associated metrics is crucial for evaluating a classification model. In Figure 10.2, the confusion matrix provides essential information for assessing the performance of a binary classifier. Colored arrows in the figure highlight matrix entries used to calculate specific metrics.

A straightforward method to evaluate a binary classifier is by examining the accuracy, i.e. the percentage of correct predictions, considering both true positives (TP) and true negatives (TN).

$$\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}} = \frac{\text{Number of correct}}{\text{Total \# predictions}}$$

This number indicates how accurately the classifier makes correct predictions. Achieving a high accuracy could certainly be a reason to celebrate! However, the true significance of this accomplishment may be short-lived if we are not careful and only rely on the accuracy. Even a "dummy" classifier that consistently returns the same result, such as always predicting the negative class, can achieve a seemingly high accuracy on an imbalanced dataset. For example, the "dummy" classifier would attain an accuracy of 95% on a dataset with 10,000 samples where only 5% belong to the positive class, and the remaining 95% represent the negative class.

It is evident that accuracy alone doesn't necessarily reflect the model's true

effectiveness. Whether dealing with imbalanced datasets or even in scenarios with balanced data, accuracy might not offer sufficient insights into the model's performance, particularly in identifying specific or systematic errors. A diverse set of metrics based on the confusion matrix can be utilized to comprehensively assess the performance of a classifier. Figure 10.2 displays a confusion matrix with overlaid arrows indicating various metrics. The color of the name below refers to the color of the arrow in the figure used to calculate the terms.

ACCURACY is a measure of the overall accuracy of the model.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{\text{Number of correct}}{\text{Total \# predictions}}$$

PRECISION is defined as the poportion of positive predictions that are correct.

$$\frac{\text{TP}}{\text{TP} + \text{FP}}$$

RECALL (a.k.a. Sensitivity or True Positive Rate) assesses a classifier's ability to correctly identify all positive instances out of the total actual positive instances in a dataset and is calculated (blue arrow) as the proportion of actual positives correctly predicted.

$$\frac{\text{TP}}{\text{TP} + \text{FN}}$$

SPECIFICITY (False Negative Rate) Proportion of actual negatives correctly predicted.

$$\frac{\text{TN}}{\text{TN} + \text{FP}}$$

FALSE POSITIVE RATE (FPR) is the proportion of actual negatives that are incorrectly classified as positive.

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

The True Positive Rate (TPR), or recall, is a key metric in this regard. Conversely, the False Positive Rate (FPR) reveals the model's inclination to misclassify negative instances as positive. Understanding a model's ability to correctly identify positive instances is crucial.

Consider a credit-card fraud detection system, where fraud represents the positive class. A high recall becomes paramount, measuring how effectively the model captures actual cases of fraud. Precision, on the other hand, is pertinent in situations where false positives incur significant costs, such as in medical screenings.

THE F1 SCORE is the harmonic mean of precision and recall and is defined as

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score ranges between 0 (worst) and 1 (best). $F1$ is zero if either the precision or recall is zero and one if both precision and recall are one. The F1

score is a valuable metric for evaluating binary classifiers, especially in situations where false positives and false negatives have different consequences or when dealing with imbalanced datasets. It provides a balance between precision and recall. A high F1 score indicates a good balance between precision and recall. For example, for medical diagnosis, you want to minimize both misclassifying healthy individuals as positive (FP) and missing actual cases (FN). A low F1 score suggests an imbalance between precision and recall. This may happen when the model is biased towards predicting one class over the other. The F1 score is particularly useful when there is an uneven class distribution or when the cost of false positives and false negatives is roughly equal. It can be used to find an optimal threshold for binary classifiers, where a trade-off between precision and recall is necessary. F1 score complements metrics like accuracy, especially when dealing with imbalanced datasets. While accuracy might be high due to the dominance of the majority class, the F1 score considers false positives and false negatives, providing a more nuanced evaluation.

MATTHEWS CORRELATION COEFFICIENT (MCC) The MCC is a correlation coefficient between the observed and predicted binary classifications and is defined as

$$MCC = \frac{\mathrm{TP} \times \mathrm{TN} - \mathrm{FP} \times \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})}}$$

The Matthews Correlation Coefficient is a robust metric for binary classification which considers all four elements of the confusion matrix. MCC ranges between -1 (perfect disagreement) and +1 (perfect agreement). A higher MCC indicates a better overall performance of the classifier. A positive MCC indicates a positive correlation between the predicted and observed classes, while a negative MCC suggests a negative correlation. A value of 0 means no correlation. MCC provides a balance between sensitivity (recall) and specificity. It is particularly useful when classes are imbalanced, and you want to avoid being misled by a high accuracy due to a dominant class. Similar to the F1 score, MCC can be used to find an optimal threshold for binary classifiers. It helps in adjusting the balance between false positives and false negatives.

Example 10.1: Calculating the metrics

Assume that a binary classifier result in a confusion matrix

$$C = \begin{bmatrix} 50 & 10 \\ 5 & 100 \end{bmatrix}$$

Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	$= \frac{50+100}{50+10+5+100} \approx 90.91\%$
False Positive Rate	$\frac{FP}{FP+TN}$	$= \frac{10}{10+100} \approx 9\%$
False Negative Rate	$\frac{FN}{FN+TP}$	$= \frac{5}{5+50} \approx 9\%$
Specificity	$\frac{TN}{TN+FP}$	$= \frac{100}{100+10} \approx 91\%$
Precision	$\frac{TP}{TP+FP}$	$= \frac{50}{50+10} \approx 83\%$
Recall	$\frac{TP}{TP+FN}$	$= \frac{50}{50+5} \approx 91\%$
F1	$\frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$	$= \frac{2\times\frac{5}{6}\times\frac{10}{11}}{\frac{5}{6}+\frac{10}{11}} = \frac{6600}{3795} \approx 1.7$
MCC	$\frac{TP\times TN-FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$	$= \frac{5000-50}{\sqrt{(60)(55)(110)(105)}} \approx 0.343$

Analyzing the Results Assume we are given a classifier (e.g. for detecting cars in images). Depending on the quality of the classifier, we may obtain different precision and recall (and other metrics).

HIGH PRECISION AND HIGH RECALL The classifier correctly identifies almost all cars in the images. Many results are returned, and all of them are labeled correctly.

HIGH RECALL BUT LOW PRECISION The classifier identifies a lot of items as "cars," but many predictions are incorrect. The system thinks various objects, like bikes or pedestrians are also cars. However, it also correctly identifies many actual cars.

HIGH PRECISION BUT LOW RECALL The classifier is selective and only labels a few items as "cars," but those predictions are mostly correct. The system is very picky and might miss many actual cars, resulting in low recall. However, the items it identifies as "cars" are indeed cars, ensuring high precision.

It's essential to note that precision may not necessarily decrease with recall. Lowering the classifier threshold can increase recall by returning more true positive results. However, if the threshold was initially too high, the new results may all be true positives, leading to an increase in precision. Conversely, if the threshold was about right or too low, further lowering it may introduce false positives, decreasing precision.

ROC curve Classifiers commonly employ a decision threshold of 0.5 to assign class labels, and adjusting this threshold becomes crucial in managing the trade-off between false positives and false negatives. The significance of these errors

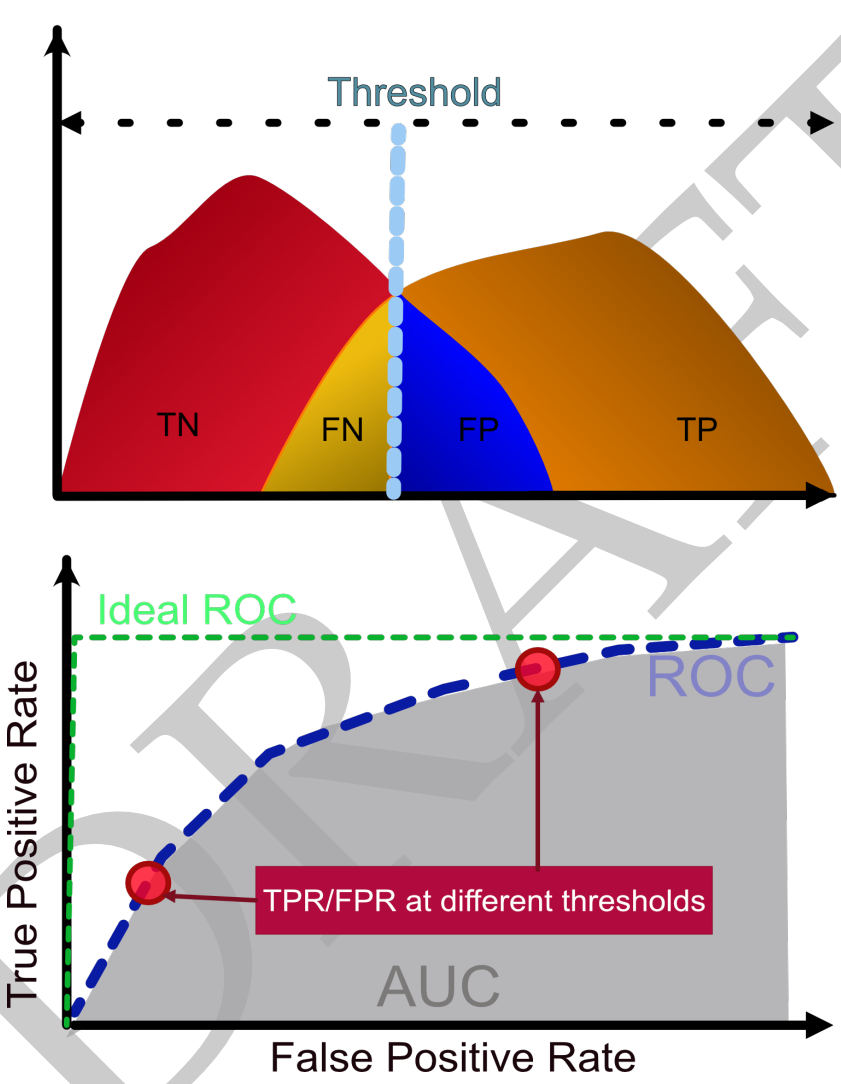

Figure 10.3: When the data lacks clear separability, adjusting the threshold has a direct impact on the trade-off between false negatives and false positives.

varies based on the specific context, and increasing or decreasing the threshold accordingly can impact the classifier's performance. Increasing the threshold reduces false positives but may elevate false negatives, while lowering the threshold has the opposite effect. As illustrated in Figure 10.3, adjusting the threshold to decrease false positives often results in an increase in false negatives, and vice versa. The challenge lies in finding the right threshold that balances true positive rate (TPR) and false positive rate (FPR). In an ideal scenario without overlap between positive and negative classes, setting an appropriate threshold becomes straightforward, leading to a perfect separation of classes with no false positives or false negatives. Unfortunately, achieving this perfection is rare. This trade-off is nicely captured by the Receiver Operating Characteristic (ROC) curve, where the true positive rate is plotted against the false positive rate at different classification thresholds. The closer the curve is to the upper-left corner, the better the model's performance, visually representing the sensitivity-specificity trade-off.

Area Under the Curve The *Area Under the Curve* (AUC) is a useful metric which values range from 0 to 1. An AUC of 1 signifies a perfect score, indicating flawless separation between positive and negative training data. Conversely, an AUC of 0 implies consistently classifying positives as negative, or vice versa. As shown in Figure 10.3, AUC integrates the entire two-dimensional area beneath the ROC curve into an aggregate measure of performance across all thresholds. An AUC of 0.5 indicates a model no better than random, while values above 0.5 signal better-than-random performance.

Precision-Recall curve he Precision/Recall curve, serving as an alternative to the ROC curve, visually illustrates the delicate balance between precision and recall in a classification model. This curve is constructed by adjusting the classification threshold and plotting precision against recall. In the ideal scenario, depicted at the upper-right corner of the curve, the model achieves both high precision and high recall simultaneously. Similar to its counterpart in the ROC curve, the Area Under the Curve (AUC) for the Precision/Recall curve quantifies the model's overall performance across various thresholds. A higher AUC indicates superior performance, highlighting the model's effectiveness in maintaining a balance between precision and recall. This metric is invaluable for holistically evaluating the model's ability to make accurate positive identifications while minimizing false positives. Examples of a ROC and Precision-Recall curve are illustrated in Figure 10.4

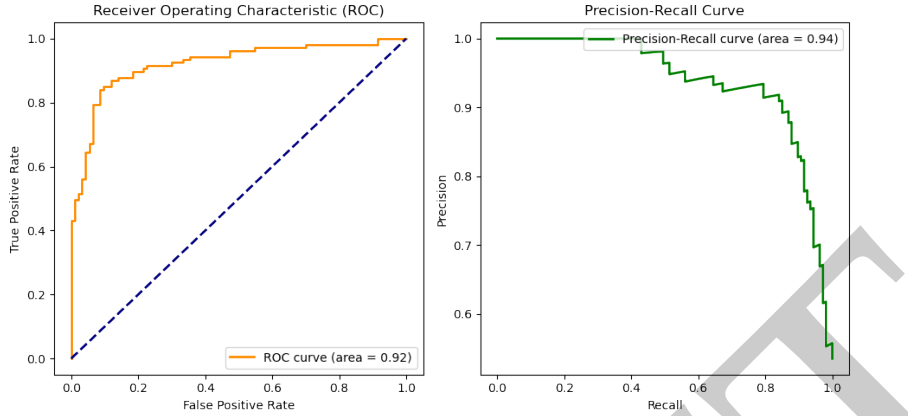

Figure 10.4: ROC(left) and precision-recall (right) curves with their respective AUC given in the captions.

10.4.0 Multiclass Classification Confusion Matrix

In multiclass classification, where there are more than two classes, the confusion matrix is a square matrix with dimensions equal to the number of classes:

$$\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1k} \\ C_{21} & C_{22} & \cdots & C_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ C_{k1} & C_{k2} & \cdots & C_{kk} \end{bmatrix}$$

The metrics for multiclass classification are simple extensions of the binary classification but where the false classifications get divided among more classes.

- **True Positive (TP):** Instances correctly predicted for class i.
- **True Negative (TN):** Instances correctly predicted as not belonging to class i.
- **False Positive (FP):** Instances incorrectly predicted as class i when they belong to another class.
- **False Negative (FN):** Instances belonging to class i but predicted as another class.

Metrics on these observations are similar to binary classification but are focused on a specfic class $\mathcal{C}_i$

- **Accuracy (A):** The overall proportion of correct predictions.

$$A = \frac{\sum_{i=1}^{k} \mathrm{TP}_i}{\sum_{i=1}^{k} (\mathrm{TP}_i + \mathrm{FP}_i)}$$

- **Precision (P):** The proportion of instances predicted as class i that are actually

in class i.

$$P_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}$$

- **Recall (R) or Sensitivity:** The proportion of actual instances of class i that were correctly predicted.

$$R_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

- **F1 Score:** The harmonic mean of precision and recall.

$$F1_i = \frac{2 \times P_i \times R_i}{P_i + R_i}$$

Macro-Averaged F1 Score:
The macro-averaged F1 score is computed as the unweighted average of the F1 scores for each class:

$$\text{Macro-F1} = \frac{1}{k} \sum_{i=1}^{k} F1_i$$

Micro-Averaged F1 Score:
The micro-averaged F1 score is computed by considering all instances and calculating a single F1 score for the entire classification.

$$\text{Micro-F1} = \frac{2 \times \sum_{i=1}^{k} \text{TP}_i}{\sum_{i=1}^{k} (\text{TP}_i + \text{FP}_i) + \sum_{i=1}^{k} (\text{TP}_i + \text{FN}_i)}$$

where TP_i, FP_i, and FN_i are the true positives, false positives, and false negatives for class i, respectively.

11

Kernels

In chapter 9, we observed that polynomial models can be employed for both regression and defining the decision boundary in classification, expressing them in terms of an inner product.

$$f(x) = \mathbf{w}^\top \phi(\mathbf{x}) \tag{11.1}$$

where $\phi(x) = \begin{bmatrix} x^N \\ x^{N-1} \\ \vdots \\ x \\ 1 \end{bmatrix}$ is called the *feature mapping*.

Example 11.1: 2. order polynomial as inner product of model parameters and feature mapping.

A 2. order polynomial

$$f_{\mathbf{w}}(x) = w_2 x^2 + w_1 x + w_0$$

can be rewritten into an inner product

$$f(x) = \mathbf{w}^\top \phi(\mathbf{x})$$

where $\mathbf{w} = \begin{bmatrix} w_2 \\ w_1 \\ w_0 \end{bmatrix}$ and $\phi(x) = \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix}$. Notice that $\phi : \mathbb{R} \to \mathbb{R}^3$ maps the input $x \in \mathbb{R}$

For single variate polynomials the feature mapping takes the input x and map it into a higher dimensional vector $\phi(\mathbf{x})$. As illustrated in Figure 11.1 the

feature mapping is generally a transformation $\phi : \mathbb{R}^M \rightarrow \mathbb{R}^N$ talking M dimensional vectors to some higher dimensional vector N. The output of the feature mapping is a vector called *feature vector*. Equation 11.1 is therefore equivalent to an inner product between the model parameters $\mathbf{w}$ and the feature vector. In the case of a N. order polynomials the feature mapping takes real numbers to vectors of one degree higher than the order of the polynomial $\phi : \mathbb{R} \rightarrow \mathbb{R}^{N+1}$.

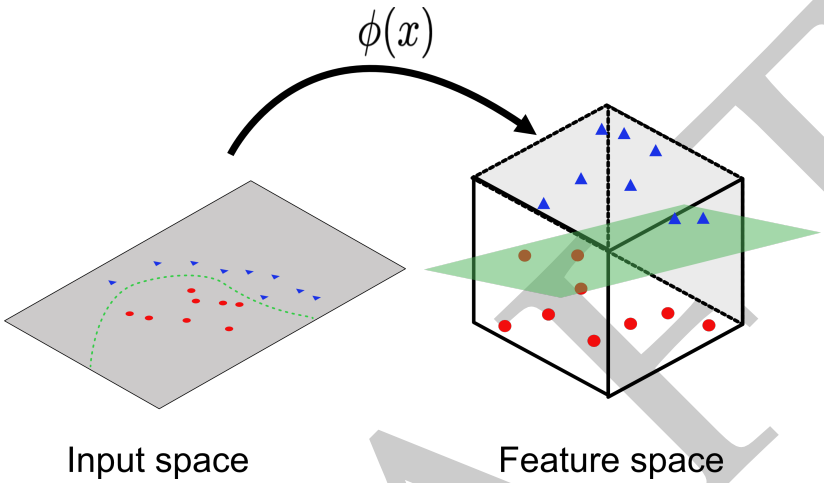

Figure 11.1: Mapping input vectors to the higher dimensional feature space using the feature mapping ϕ

Example 11.2: Learning a non-linear function.

Similarly to the previous example, we can learn non-linear regression functions such as

$$f_w(x) = w_2 \cos(x^2) + w_1 \sin(x) + w_0$$

using linear least squares. In this case, the kernel mapping $\phi(x)$ is given by

$$\phi(x) = \begin{bmatrix} \cos(x^2) \\ \sin(x) \\ 1 \end{bmatrix}$$

This function is non-linear, but learning the model parameters is based on linear least squares. Given a dataset $\{(x_i, y_i)\}^D$, learning the model parameters can be done by solving $A\mathbf{x} = \mathbf{y}$ where the design matrix A is constructed from the kernel vector for each of the D data points such that

$$A = \begin{bmatrix} \cos(x_1^2) & \sin(x_1) & 1 \\ \cos(x_2^2) & \sin(x_2) & 1 \\ \vdots & \vdots & \vdots \\ \cos(x_D^2) & \sin(x_D) & 1 \end{bmatrix}$$

The vector $\mathbf{x}$ contains the unknown model parameters

$$\mathbf{x} = \begin{bmatrix} w_2 \\ w_1 \\ w_0 \end{bmatrix}$$

and the vector $\mathbf{y}$contains the labels

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_D \end{bmatrix}$$

.

Note!

The main observation is that a non-linear function can be learned using linear least squares provided the function is linear in the model parameters.

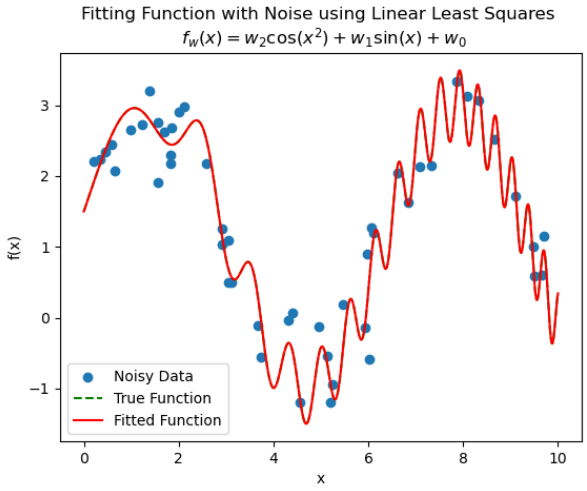

Figure 11.2: Non-linear function whose model parameters are learned with linear least squares.

11.0.1 Kernel Functions

It turns out that the idea of feature mapping is a tremendously useful general concept when working with non-linear data i general.

For simplicity define the vectors $a = \phi(\mathbf{x})$ and $b = \phi(\mathbf{x}')$ whose inner product

is $a^\top b = \phi(\mathbf{x})\phi(\mathbf{x}')$. This leads to the definition 11.1 of the kernel function.

Definition 11.1: Kernel function

The *kernel function* is defined as

$$K(x, x') = \phi(x)^\top \phi(x') = K(x', x) \tag{11.2}$$

where mapping $\phi(x)$ is the feature mapping function.

This definition may initially seem complex. As discussed in paragraph 10.1, the inner product of two vectors is a measure of the similarity between the vectors (cosine of the angle between them multiplied by the length of the vectors). In simpler terms, the kernel function, denoted as K, takes two input vectors $\mathbf{x}$ and $\mathbf{x}'$, maps them to a higher-dimensional feature space using the function ϕ, and then calculates the inner product (cosine distance) between the features in that higher-dimensional space. Essentially, the kernel provides a measure of similarity between the inputs, but now, this similarity is assessed in a higher-dimensional space. It is important to note the symmetry of the kernel, expressed as $K(\mathbf{x}, \mathbf{x}') = K(\mathbf{x}', \mathbf{x})$. This symmetry property shows that the order of input vectors does not affect the resulting similarity measurement.

Kernels define a Metric Kernels define a metric of similarity between input vectors. The simplest example of a kernel function is obtained by letting $\phi(x) = x$ be the identity mapping. In this case, the kernel function $K(x, x') = x^\top x'$ is just the inner product. This kernel is related to the angle between the vectors $\mathbf{x}$ and $\mathbf{x}'$, and we refer to it as the *linear kernel*.

We can define other kernels (metrics) to determine the similarity between vectors in a higher-dimensional space. The choice of the kernel function is crucial and impacts the model's ability to capture underlying patterns in the data. Commonly used kernel functions include:

Linear Kernel

$$K(x, x') = x^\top x'$$

Gaussian Kernel (radial basis function)

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

Polynomial Kernel

$$K(x, x') = (x^\top x' + c)^d$$

Sigmoid Kernel

$$K(x, x') = \tanh(\alpha x^\top x' + c)$$

Note!

The exact form of the feature transformation may not explicitly be known, but the kernel provides a computationally efficient way to work in the transformed space without calculating the feature vectors explicitly. For example it turns out that the Gaussian kernel (a.k.a. Gaussian Radial Basis Function (RBF)) induces an implicit feature transformation, $\phi(\mathbf{x})$, mapping the input space to an infinite-dimensional feature space. Yup, this is hard to imagine!

Feature transformations on a dataset Figure 11.3 shows an example of a simple 1D dataset containing the integer values between -4 and 4. The data points belong to either of the two classes (blue square and green triangle). It is clear that this dataset is not linearly separable since you cannot put a single straight line to separate the classses. While the purpose of kernels is not only intended to ensure class separability, the example will make it easier to comprehend some of the general advantages of kernels.

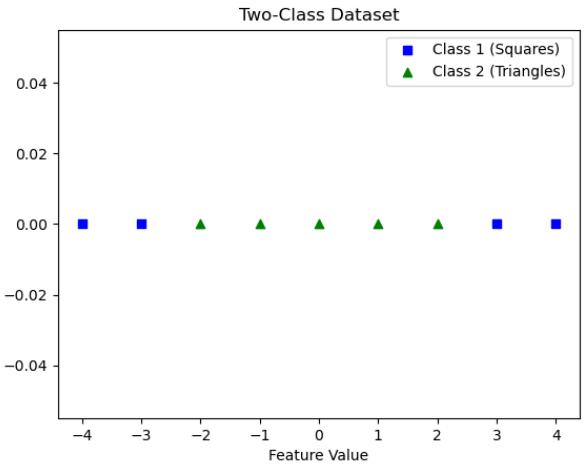

Figure 11.3: Dataset of numbers ($\mathbb{R}$) which is not linearly seperable.

Polynomial feature mapping Define the (polynomial) feature mapping ϕ : $\mathbb{R} \to \mathbb{R}^2$ given by $\phi(x) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$ and observe in Figure 11.4 how the points after being transformed to a higher dimensional feature space $x \to [x, x^2]$ become linearly separable.

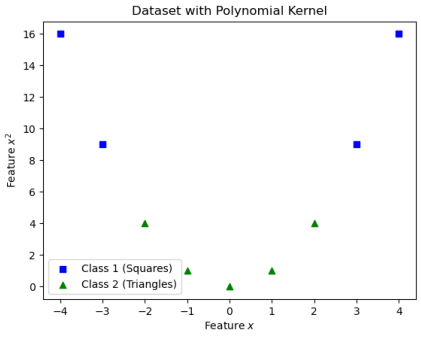

Figure 11.4: 1D features transformed to the 2 dimensionl feature space according to ϕ

Gaussian Radial Basis Feature Mapping Changing the kernel to a Gaussian kernel function (Radial Basis Function or RBF) leads to a different feature space. The Gaussian kernel function is defined as:

$$G_\gamma(x, \ell) = \exp(-\gamma \|x - \ell\|^2) \tag{11.3}$$

where $\gamma = \frac{1}{2\sigma^2}$. The Gaussian is a bell-shaped function varying from 0 (far from the landmark) to 1 at the landmark.

For a specific example with three landmarks ℓ_1, ℓ_2, ℓ_3 and a data point x, the kernel values are:

$$\begin{aligned} K(x, \ell_1) &= \exp\left(-\gamma \|x - \ell_1\|^2\right) \\ K(x, \ell_2) &= \exp\left(-\gamma \|x - \ell_2\|^2\right) \\ K(x, \ell_3) &= \exp\left(-\gamma \|x - \ell_3\|^2\right) \end{aligned}$$

In the concrete example of having two landmark points centered at -2 and 1, and define the feature transformation as $\phi : \mathbb{R} \to \mathbb{R}^2$:

$$\phi_\gamma(x) = \begin{bmatrix} G_\gamma(x, -2), & G_\gamma(x, 1) \end{bmatrix}$$

or equivalently

$$\phi_\gamma(x) = G(\mathbf{x}, \begin{bmatrix} -2, & 1 \end{bmatrix})$$

where $\mathbf{x} = \begin{bmatrix} x, x \end{bmatrix}$ is the vector of repeated values of x.

This feature mapping takes an input x and maps it to a vector, indicating how far x (a value between 1 and 0) is from each of the landmark points -2 and 1. As illustrated in Figure 11.5, it turns out that the data becomes linearly separable in the higher-dimensional feature space. Equivalent counterparts for vector inputs are demonstrated in examples 11.3 and 11.4. The choice of landmarks significantly influences the feature vectors, shaping the separability of data in the higher-dimensional space.

In a dataset with N data points and three landmarks, the *kernel matrix* K is a square matrix where each element K_{ij} represents the Gaussian kernel value between the i-th data point and the j-th landmark:

$$K = \begin{bmatrix} K(x_1, \ell_1) & K(x_1, \ell_2) & K(x_1, \ell_3) \\ K(x_2, \ell_1) & K(x_2, \ell_2) & K(x_2, \ell_3) \\ \vdots & \vdots & \vdots \\ K(x_N, \ell_1) & K(x_N, \ell_2) & K(x_N, \ell_3) \end{bmatrix}$$

where n is the number of data points in your dataset.

In theory, it is possible to define a Radial Basis Function (RBF) for each training data point, thereby expanding the feature vector space to be as large as the number of data points. The feature mapping $\phi(x^*)$ for an unseen data point x^* yields a feature vector with element values representing distances to each of the training data points.

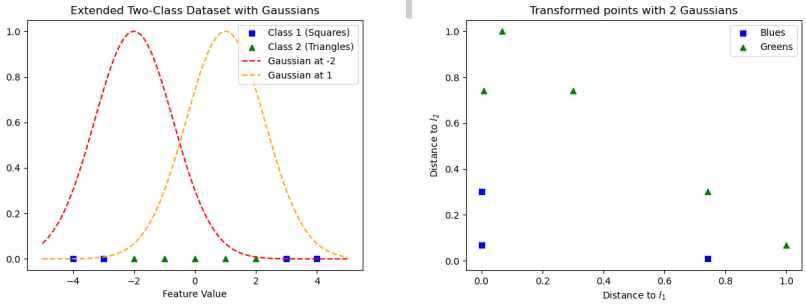

Figure 11.5: (Left) The dataset overlayed the two Gaussians used to create the feature transformation. (Right) the transformed dataset, which becomes linearly separable.

Example 11.3: Kernel mapping and similarity

Define the vectors

$$x_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad x_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \quad x_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Transforming these vectors into a higher-dimensional space through the use of a quadratic polynomial mapping, is expressed as

$$\phi(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}.$$

When mapping the points x_i to the higher dimensional vector space they become:

$$\phi(x_1) = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 6 \\ 9 \end{bmatrix}, \quad \phi(x_2) = \begin{bmatrix} 1 \\ -1 \\ 2 \\ 1 \\ -2 \\ 4 \end{bmatrix}, \quad \phi(x_3) = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The inner product between these mapped vectors is given by

$$K(x_i, x_j) = \phi(x_i)^T \phi(x_j).$$

The *kernel matrix* contains all inner products in the high dimensional spaces

$$\begin{bmatrix} K(x_1, x_1) & K(x_1, x_2) & K(x_1, x_3) \\ K(x_2, x_1) & K(x_2, x_2) & K(x_2, x_3) \\ K(x_3, x_1) & K(x_3, x_2) & K(x_3, x_3) \end{bmatrix} = \begin{bmatrix} 95 & -12 & 15 \\ -12 & 22 & 1 \\ 15 & 1 & 2 \end{bmatrix}$$

Notice how the diagonal (inner product of vectors with them selves) give the highest values.
Figure 11.6 shows the Kernel matrix for the entire Iris dataset for different values of γ using a Gaussian feature mapping. Notice how changing the value γ also changes the overall scale of the similarity values in the matrix. It is also evident that some of the data points (larger values) resemble each other more than others.

Example 11.4: Normalized Kernel mapping

This example shows the influence of normalization. In some cases, normalization of kernelized vectors is done to ensure that the magnitude of

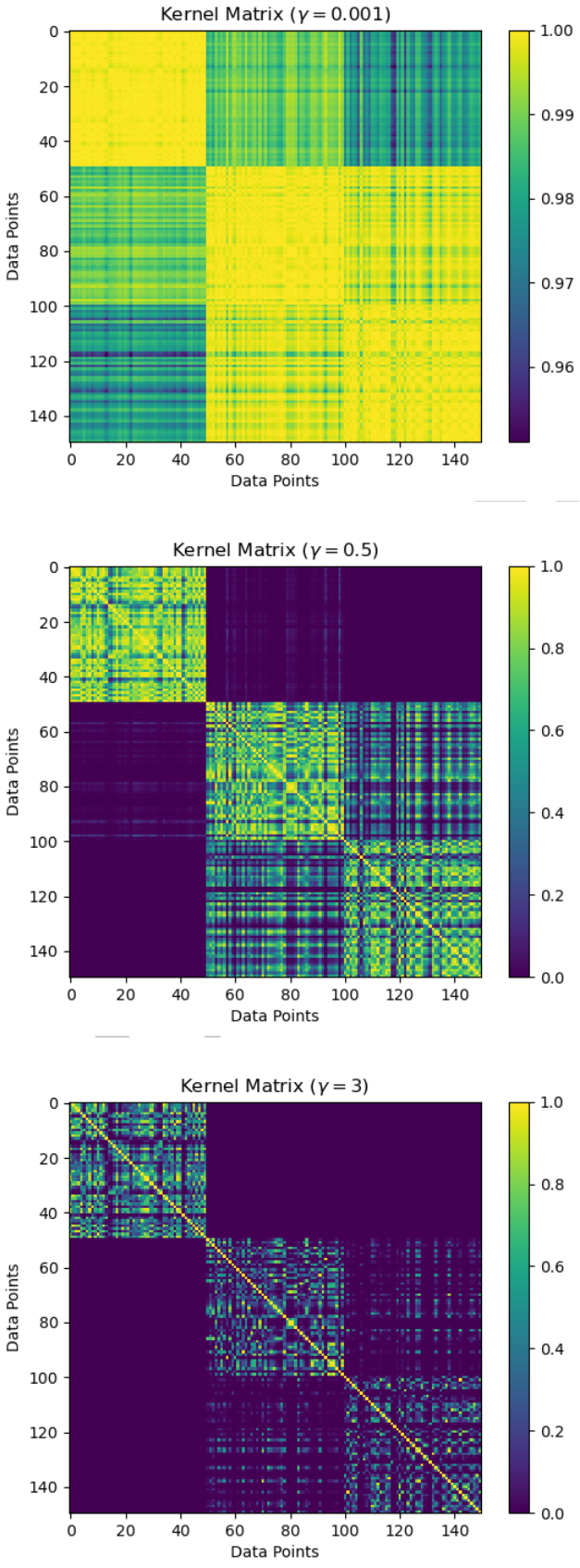

Figure 11.6: Kernel values for the entire iris dataset with variable settings for the scale parameter γ in the Gaussian kernel

the vectors does not affect the similarity measure. The inner product of unit-length vectors is essentially the cosine of the angle between them, providing a measure of similarity that is invariant to the scale of the vectors.
Normalization of each $\phi(x_i)$ to have unit length:

$$\bar{x}_1 = \bar{\phi}(x_1) = \frac{\phi(x_1)}{\|\phi(x_1)\|}, \quad \bar{x}_2 = \bar{\phi}(x_2) = \frac{\phi(x_2)}{\|\phi(x_2)\|}, \quad \bar{x}_3 = \bar{\phi}(x_3) = \frac{\phi(x_3)}{\|\phi(x_3)\|}.$$

The inner product between these normalized mapped vectors is given by

$$K(\bar{x}_i, \bar{x}_j) = \bar{\phi}(x_i)^\top \bar{\phi}(x_j).$$

The *normalized kernel matrix* is:

$$\begin{bmatrix} K(\bar{x}_1, \bar{x}_1) & K(\bar{x}_1, \bar{x}_2) & K(\bar{x}_1, \bar{x}_3) \\ K(\bar{x}_2, \bar{x}_1) & K(\bar{x}_2, \bar{x}_2) & K(\bar{x}_2, \bar{x}_3) \\ K(\bar{x}_3, \bar{x}_1) & K(\bar{x}_3, \bar{x}_2) & K(\bar{x}_3, \bar{x}_3) \end{bmatrix} = \begin{bmatrix} 1 & -\frac{2}{3} & \frac{1}{3} \\ -\frac{2}{3} & 1 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix}$$

Notice how the diagonal (inner product of vectors with themselves) gives the highest values.

11.1.0 Kernel Trick

The concept of a kernel, formulated as an inner product in a (higher dimensional) feature space, allows us to develop interesting non-linear extensions of many machine learning algorithms by leveraging the so-called *kernel trick*. The core insight is that if an algorithm is structured in a way that the input vector x is involved solely through inner products, $x^\top x'$, we can replace that inner product with a kernel $K(x, x')$, thus achieving a non-linear equivalent. For example, the kernel trick can be applied to principal component analysis (will be detailed soon), nearest-neighbor classifiers and regression.

Note!

Kernel functions must adhere to specific mathematical constraints, known as Mercer Kernels, to be considered valid. According to Mercer's theorem, if a function $K(a, b)$ satisfies mathematical conditions known as Mercer's conditions (e.g. K must be continuous and symmetric in its arguments, i.e., $K(a, b) = K(b, a)$), then there exists a function ϕ that maps a and b into another space (potentially of much higher dimensions), such that $K(a, b) = \phi(a)^\top \phi(b)$. This implies that K can be used as a kernel because we know ϕ exists, even if the exact form of ϕ is unknown.

The beauty lies in the fact that you do not need to explicitly perform this mapping! It's worth noting that some commonly used kernels, such as the sigmoid kernel, may not satisfy all of Mercer's conditions, yet they often perform well in practical applications.

11.2.0 Kernel Regression

Kernel regression is a non-parametric method used for estimating the relationship between input variables and an associated output. Unlike linear regression, kernel regression does not assume a specific functional form for the underlying relationship (e.g. there are no model parameters $\mathbf{w}$) but instead the kernel regression model relies on the metric defined by the Kernel and its associated hyperparameters.

The core concept of kernel regression involves assigning weights to data points based on their proximity to a query point $\mathbf{x}$. These weights are determined by the kernel function K, with vectors in close proximity yielding a large kernel function value. Essentially, the kernel function accentuates the impact of nearby points when estimating the output at the query point.

Let (x_i, y_i) be the training data pairs, where x_i is the input and y_i is the corresponding output. Given a query point x, the kernel regression estimate $f(x)$ is computed as a weighted sum of the output values, with weights determined by the kernel function K:

$$f(x) = \sum_{i=1}^{N} K(x, x_i) \cdot y_i \tag{11.4}$$

Here, N is the number of training data points (or an appropriate subset of the nearby points of $\mathbf{x}$), and $K(x, x_i)$ is the kernel function, measuring the similarity between x and x_i. Notice that there are no model parameters $\mathbf{w}$ in this equation. In fact Equation 11.4 is essentially an inner product of the query point $\mathbf{x}$ to all the training points $\mathbf{x}_i$.

$$f(x) = K_x^\top \mathbf{y} \tag{11.5}$$

where $K_x^\top = \begin{bmatrix} K(x, x_1) & K(x, x_2) & \dots & K(x, x_N) \end{bmatrix}^\top$ is the vector of the kernel values of the query vector $\mathbf{x}$ with all the training inputs $\mathbf{x}_i$ and $\mathbf{y} = \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix}^\top$ is the vector of training outputs. Notice that K_x essentially acts as a vector of weights.

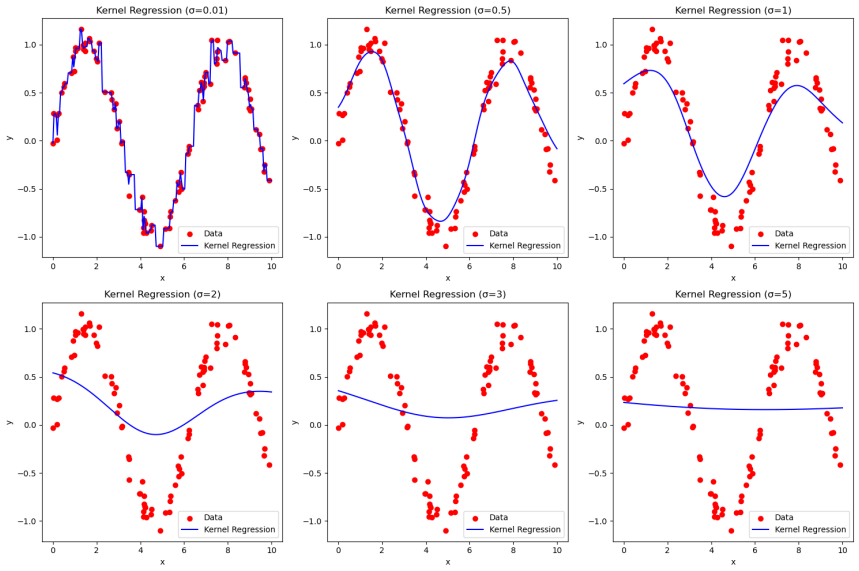

Figure 11.7: Kernel regression with a Gaussian kernel and changing the σ parameter. The σ parameter determines how much the distance between the inputs contributes to the weight.

Figure 11.7 illustrates an example of Kernel regression using a Gaussian kernel. The effectiveness of kernel regression is contingent on the selection of kernel and hyper parameter tuning.In Kernel regression, hyperparameters like the bandwidth (σ) in the Gaussian kernel or parameters (c, d, α) in other kernels play a crucial role. These parameters govern the smoothness and flexibility of the estimated function, and their careful tuning is essential for achieving optimal model performance. Unlike methods assuming a specific functional form, Kernel regression does not impose such constraints on the underlying function, allowing it to capture non-linear relationships. It achieves this by focusing primarily on training inputs that are close to the query point $\mathbf{x}$.

Example 11.5: Kernel Regression of a single point

Consider an example with three data points

$$x_1 = 1, \quad y_1 = 2$$
$$x_2 = 2, \quad y_2 = 1$$
$$x_3 = 3, \quad y_3 = 3$$

The kernel matrix is:

$$\begin{bmatrix} K(x_1,x_1) & K(x_1,x_2) & K(x_1,x_3) \\ K(x_2,x_1) & K(x_2,x_2) & K(x_2,x_3) \\ K(x_3,x_1) & K(x_3,x_2) & K(x_3,x_3) \end{bmatrix}$$

A prediction for an unseen point $x^* = 2.5$ is done as the appropriately weighted sum of the training data inputs and labels

$$\begin{aligned} y^* &= \sum_{i=1}^{3} y_i K(x_i, x^*) \\ &= \sum_{i=1}^{3} y_i K(x_i, 2.5) \\ &= K(1, 2.5) * 2 + 1 * K(2, 2.5) + 3 * K(3, 2.5) \end{aligned}$$

In the case of using a polynomial feature mapping of degree 2 the predicted value becomes $y^* = 1.3875$
Using different input values x_{new} result in a graph similar to Figure 11.8

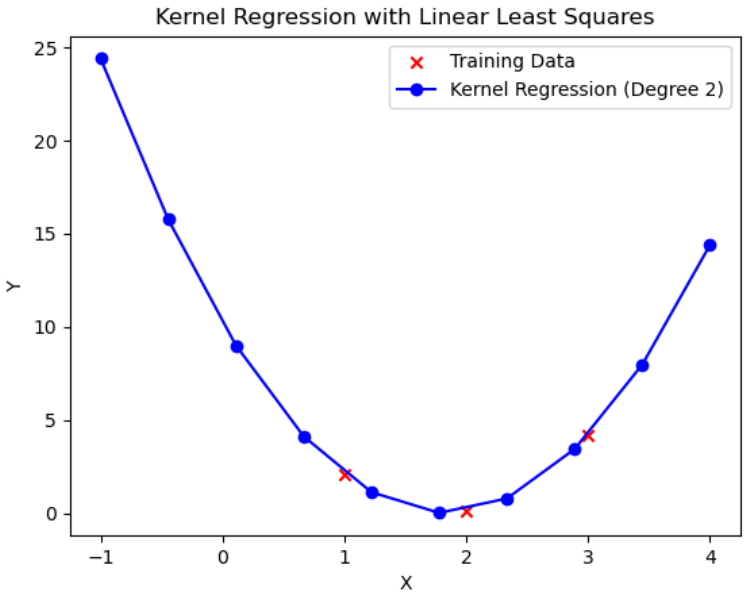

Figure 11.8: Kernel regression in example 11.5 using (red) training data and (blue) unseen points.

Kernels and feature mappings One strategy involves selecting a feature space mapping $\phi(\mathbf{x})$ and subsequently determining the corresponding kernel. In this

case, the kernel function is defined for a one-dimensional input space as

$$K(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{M} \phi_i(\mathbf{x})\phi_i(\mathbf{x}') = \phi(x)^\top \phi(x')$$

where $\phi_i(\mathbf{x})$ the i.th element in high dimensional feature vector $\phi(x)$

Alternatively, another approach is to directly construct kernel functions. In this setting, it is crucial to ensure that the chosen function is a valid kernel, meaning it corresponds to a scalar product in some (possibly infinite-dimensional) feature space. As an illustrative example, consider a kernel function given by the square of the inner product

$$K(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}')^2$$

Consider the specific case of a two-dimensional input space $\mathbf{x} = (x_1, x_2)$ and expand the terms to identify the corresponding nonlinear feature mapping

$$\begin{aligned} K(x, x') &= (\mathbf{x}^\top \mathbf{x}')^2 \\ &= x_1^2 {x_1'}^2 + 2x_1 x_1' x_2 x_2' + x_2 x_2' \\ &= (x_1^2, \sqrt{2}x_1 x_2, x_2)(x_1', \sqrt{2}x_1' x_2', x_2')^\top \\ &= \phi(x)^\top \phi(x'). \end{aligned}$$

Hence the quadratic kernel $K(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}')^2$ corresponds to feature mapping $\phi(\mathbf{x}) = (x_1^2, 2x_1 x_2, x_2)^\top$.

Creating new Kernels from existing kernels (Skip at first reading) A useful approach to creating novel kernels involves composing them from elementary kernels. The kernel $K(x, x')$ must be symmetric and positive semidefinite. Given the validity of kernels $K_1(x, x')$ and $K_2(x, x')$, the following new kernels will also be considered valid, enabling the construction of more complex kernels [1]:

$$\begin{aligned} K(x, x') &= K_1(x, x') + K_2(x, x') \\ K(x, x') &= K_1(x, x') \cdot K_2(x, x') \\ K(x, x') &= c \cdot K_1(x, x') \\ K(x, x') &= f(x) \cdot K_1(x, x') \cdot f(x') \\ K(x, x') &= q(K_1(x, x')) \\ K(x, x') &= \exp(K_1(x, x')) \\ K(x, x') &= x^T A x' \end{aligned}$$

Here, $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with non-negative coefficients, $\phi(x)$ is a function from x to $\mathbb{R}^M$, $K_3(\cdot, \cdot)$ is a valid kernel in

$\mathbb{R}^M$, A is a symmetric positive semidefinite matrix, x_a and x_b are variables (not necessarily disjoint) with $x = (x_a, x_b)$, and K_a and K_b are valid kernel functions over their respective spaces.

12

Support Vector Machines

A Support Vector Machine (SVM) is a robust model suitable for both linear and nonlinear tasks, encompassing classification, regression, and outlier detection. It is particularly efficient for when combined with Kernels. SVMs excel in handling complex datasets, especially those considered small by today's standards. This chapter is dedicated to exploring the application of SVMs specifically for classification. It is intended complement existing reading materials and videos by addressing specific gaps but does not to offer an exhaustive description of support vector machines for regression, optimization methodology or other detailed but useful knowledge.

Consider the dataset $\{(\mathbf{x}_i, y_i)\}$ in Figure 12.1 where the labels $y \in \{\text{red}, \text{blue}\}$ and $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$. The objective is to devise a classifier capable of determining whether the corresponding point should be classified to either class. Previous chapters introduced linear models for regression and classification, where the mapping $y = f_{\mathbf{w}}(\mathbf{x})$ from input $\mathbf{x}$ to output y is determined by the model parameters $\mathbf{w}$. During the training phase, the training data is used to learn the model parameters (by solving the linear set of equations). Once the model is trained, the training data becomes unnecessary. Predictions for new inputs $\mathbf{x}^*$ depend solely on the learned parameter vector $\mathbf{w}$, as in the case of a affine model (using homogeneous coordinates for $\mathbf{x}^*$) where the *decision boundary* is calculated as

$$f_{\mathbf{w}}(\mathbf{x}^*) = \mathbf{w}^\top \mathbf{x}^* \tag{12.1}$$

.

or in the more general case

$$f_{\mathbf{w}}(\mathbf{x}^*) = \mathbf{w}^\top \phi(\mathbf{x}^*).$$

where $\phi(x)$ is a feature mapping.

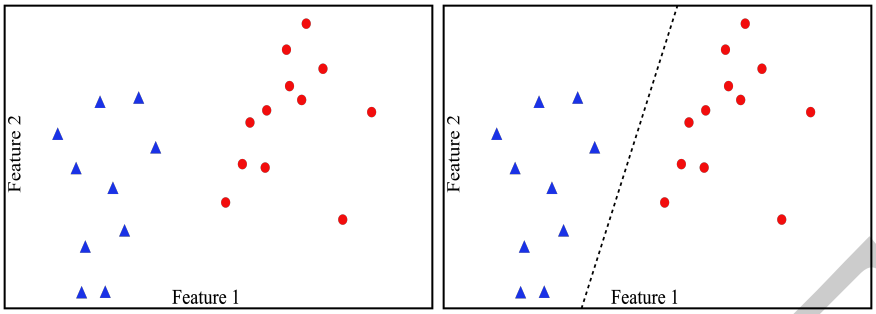

Figure 12.1: (Left) Data set and (right) Linear classifier minimizing the distance to all training data.

When $f_{\mathbf{w}}$ is intended for classification, the *decision function* is given by:

$$\text{sign}(f_{\mathbf{w}}(x)) = \text{sign}(\mathbf{w}^\top \phi(\mathbf{x}))$$

At the other end of the scale are methods that use the training data for making predictions. Among these is the k-nearest neighbor classification, which assigns to each new test vector $\mathbf{x}^*$ the same label as the closest examples from the training set. This is an example of a *memory-based* classification model that stores the entire training set to make predictions for future data points. Memory-based methods typically require a metric to measure the similarity of any two vectors in the input space and are generally fast to 'train.' However, they can be slow at making predictions for test data points because they involve calculations of distances between the test vector and the training data.

Nevertheless, there exists a class of models where a subset of the training data points are reutilized when making predictions. Among these is is the Support Vector Machine. The SVM model presented here is a linear model that employs the kernel trick to obtain a non-linear decision boundary in the input space (and linear in high-dimensional feature space). Understanding the fundamentals of Support Vector Machines (SVM) and their operational principles is facilitated through a straightforward illustration.

Large Margin Classifiers In linear SVMs the decision function is given by

$$f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + \mathbf{b}$$

where $\mathbf{w}$ are model parameters, $\mathbf{x}$ is the input data point, $\mathbf{b}$ is the bias term. A Support Vector Machine learns hyperplane from the training data which serves as the decision boundary: any point situated on one side of it is classified as blue, while any point on the other side is classified as red. So far this seems equivalent to a standard linear classifier. Up to this point, our models have also

assumed the existence of a single decision boundary to minimize the distance to the training data. The core concept of Support Vector Machines (SVMs) is illustrated through Figure 12.2. The model with a yellow decision boundary performs poorly, failing to properly separate the classes while the other models (green) are effective on separating the training set. Even though the green models separate the data they may not be considered equally good. In contrast, the right plot displays the decision boundary of an SVM classifier represented by a solid line.

The margin not only effectively separates the two classes but also maintains maximum distance from the nearest training instances. Conceptually, an SVM classifier aims to fit the widest street (depicted by parallel dashed lines in Figure 12.2) between the classes. This concept is known as *large margin classification*. While the SVM model has as linear model it achieves the optimal model parameters using by maximize the margin [1]. A larger margin implies greater confidence in the classification, and it reduces the risk of overfitting to the training data. This is desirable because it leads to a more robust and generalizable model. Consider training a linear SVM on separable data, denoted as $\{x_i, y_i\}$, where $i = 1, \ldots, l$, $y_i \in \{-1, 1\}$, and $x_i \in \mathbb{R}^d$. The goal is to find a hyperplane that effectively separates positive and negative examples.

By contrast to minimizing sum of squared distances of each training point to the decision boundary

Support Vectors The points that are far away from the decision boundary are pretty safe to classify, however the points that are close to the decision boundary may be more easy be misclassified. These points are called *support vectors* and provide most information (carry most weight) of where the decision boundary should be placed.

In the case where all data is linearly separable all the training data satisfy:

$$\mathbf{w}^\top \mathbf{x}_i + b \geq +1 \text{ for } y_i = +1$$
$$\mathbf{w}^\top \mathbf{x}_i + b \leq -1 \text{ for } y_i = -1$$

These constraints are equivalently expressed as:

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) - 1 \geq 0 \text{ for all } i$$

A data point $\mathbf{x}_i$ is a *support vector* if it satisfies:

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) = 1$$

Note that achieving $y_i(\mathbf{w}^\top \mathbf{x}_i + b) = 1$ for support vectors is always possible by appropriately scaling all model parameters $\mathbf{w}$ and b. The *margin* is defined

as the distance between this hyperplane and the nearest data point from either class. The support vectors are shown in Figure 12.2 (right) as the pink circles around the datapoints.

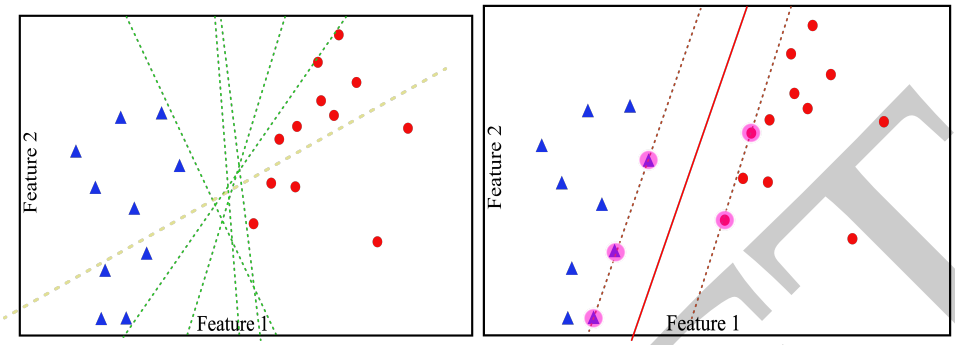

Figure 12.2: (Left) Decision boundaries separating the data. (Right) margin (between dashed lines) and the support vectors (pink).

Note!

Notice that since the objective of SVM aims to maximize the margin between classes then outliers will have less impact on the decision boundary compared to linear least squares models e.g. data points can be really far away from th decision boundary and not have impact. Essentially it is only the support vectors that have influence. For similar reasons SVMs can handle imbalanced datasets well.

Optimizing the model parameters Determining the optimal hyperplane is based on the maximization of margins from the labels. The best hyperplane, represented as a line in the figure is the one that exhibits the greatest distance to the nearest element of each label.

Assuming a separating hyperplane given by $\mathbf{w}^\top x + b = 0$, where $\mathbf{w}$ are the model parameters. $\mathbf{w}$ is normal to the hyperplane and $|b|/\|\mathbf{w}\|$ is the perpendicular distance to the origin where $\|\mathbf{w}\|$ is the Euclidean norm of $\mathbf{w}$. The margin (M) is given by $M = \frac{2}{\|\mathbf{w}\|}$, so minimizing $\|\mathbf{w}\|$ is equivalent to maximizing M. This is because the norm of $\mathbf{w}$ is inversely proportional to the margin. When the norm of $\mathbf{w}$ is minimized subject to the constraint that data points are correctly classified, the SVM is forced to find a decision boundary that maximizes the margin. In addition to maximizing the margin, minimizing the norm of $\mathbf{w}$ also serves as a form of regularization. By minimizing the norm of $\mathbf{w}$, the SVM seeks a simpler decision boundary that generalizes well to unseen data.

The optimization problem for SVM involves minimizing a loss function

$$\mathcal{L}(\mathbf{w}) = \min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b))$$

C is a regularization parameter controlling the trade-off between maximizing the margin and minimizing classification errors. It is beyond the scope of this course and these lecture notes to detail the derivations of the optimization[2]. However, what is important is how predictions are made for a novel input $\mathbf{x}$:

Minimizing $\mathcal{L}$ with respect to $\mathbf{w}$ and b requires the derivatives with respect to α_i to vanish and leads to a convex quadratic programming problem involving Lagrange multipliers.

$$f_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}^\top \mathbf{x}_i + b$$

We can now make use of the kernel trick and replace the inner product (blue) with a kernel K to provide a kernelized decision boundary:

$$f_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i y_i K(\mathbf{x}, \mathbf{x}_i) + b \tag{12.2}$$

The decision function for SVM is given by $\text{sign}(f_{\mathbf{w}}(\mathbf{x}))$ for some input $\mathbf{x}$

Observations Notice how Equation 12.2 for SVM resembles Equation 11.4 for kernel regression. Both SVM and kernel regression involve the use of a kernel function, but their objectives differ slightly. In SVM, the decision function is a sum over all data points N involving the kernel function $K(\mathbf{x}, \mathbf{x}_i)$. This function essentially replaces the inner product $\mathbf{x}^\top \mathbf{x}_i$ with a kernelized measure of similarity. Each term is multiplied by a real number α_i, which is non-zero only for support vectors. The decision function calculates the weighted sum of kernel values between the input $\mathbf{x}$ and all support vectors. SVM selects a subset of the dataset (the support vectors) and uses the kernel to map the input space into a higher-dimensional space to find a hyperplane that captures the data within a margin. On the other hand, kernel regression uses the kernel to determine the weights for neighboring data points to make predictions. Unlike SVM, kernel regression uses all data points, assigning weights based on their proximity to the input $\mathbf{x}$. The objective in kernel regression is to predict the target value based on a weighted sum of values from nearby data points, without the concept of a margin or a hyperplane.

SVMs are a powerful class of machine learning models with notable advantages and considerations.

EFFECTIVE in high-dimensional spaces.
Still effective in cases where the number of dimensions is greater than the number of samples.

USES A SUBSET of training points in the decision function making it memory efficient.

VERSATILE different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

If the number of features is much greater than the number of samples, avoid overfitting by choosing appropriate Kernel functions and regularization terms.

13

Symmetric and Orthogonal Matrices

It is no exaggeration to say that symmetric matrices hold a tremendous position in the theory and application of linear algebra and are furthermore essential in understanding linear dimensionality reduction methods.

Definition 13.1: Definition of a Symmetric and Orthogonal Matrix

A square matrix A is *symmetric* when it is equal to its transpose $A = A^\top$. A square matrix A is *orthogonal* if its transpose $A^\top = A^{-1}$ is equal to its inverse, that is,

$$A^\top A = AA^\top = I$$

Recall that the inner product between two vectors is 0 when they are orthogonal, and it is 1 provided the length of both vectors is also one (unit length). An orthogonal matrix implies that the rows of A are mutually orthogonal unit vectors, and similarly, the columns of A are mutually orthogonal unit vectors. An orthogonal matrix has several important properties. For example, it preserves vector lengths and angles between vectors. An example of an orthogonal matrix is the rotation matrix.

Example 13.1: Orthogonal matrix

The matrix

$$Q = \begin{bmatrix} 0.5 & -0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0.5 & -0.5 \\ 0.5 & 0.5 & 0.5 & 0.5 \end{bmatrix}$$

is orthogonal since

$$Q^\top Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice furthermore that the matrix has determinant 1 $\det(Q) = 1$.
A matrix is *orthonormal* if its columns are orthogonal unit vectors. Let's check the columns of the matrix Q:

$$Q = \begin{bmatrix} 0.5 & -0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0.5 & -0.5 \\ 0.5 & 0.5 & 0.5 & 0.5 \end{bmatrix}$$

The columns are $\mathbf{v}_1 = [0.5, 0.5, -0.5, 0.5]$, $\mathbf{v}_2 = [-0.5, 0.5, 0.5, 0.5]$, $\mathbf{v}_3 = [0.5, -0.5, 0.5, 0.5]$, and $\mathbf{v}_4 = [0.5, -0.5, -0.5, 0.5]$.

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = 0, \quad \mathbf{v}_1 \cdot \mathbf{v}_3 = 0, \quad \mathbf{v}_1 \cdot \mathbf{v}_4 = 0,$$
$$\mathbf{v}_2 \cdot \mathbf{v}_3 = 0, \quad \mathbf{v}_2 \cdot \mathbf{v}_4 = 0, \quad \mathbf{v}_3 \cdot \mathbf{v}_4 = 0$$

$$||\mathbf{v}_1|| = 1, \quad ||\mathbf{v}_2|| = 1, \quad ||\mathbf{v}_3|| = 1, \quad ||\mathbf{v}_4|| = 1$$

This confirms that the columns of the matrix are orthogonal unit vectors, and therefore, the matrix is orthonormal.

Distance matrix A prominent example of a symmetric matrix is the distance matrix. The distance matrix contains the distances between a set of N observations, such as the distance you need to travel between N cities. This matrix is symmetric provided that the distance from city a to city b is the same as the distance from city b to a. Consider N cities labeled as $1, 2, \ldots, N$. The distance matrix D is a symmetric matrix given by:

$$D = \begin{bmatrix} 0 & d_{12} & \cdots & d_{1N} \\ d_{21} & 0 & \cdots & d_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N1} & d_{N2} & \cdots & 0 \end{bmatrix}$$

Here, d_{ij} represents the distance from city i to city j, and $d_{ij} = d_{ji}$ for all i, j since the distance from city i to city j is the same as from city j to city i. Therefore, the distance matrix is symmetric.

Covariance Matrix A *data matrix* $\mathbf{X}$ contains the data where each row corresponds to an observation and each column corresponds to a variable. In this specific example, $\mathbf{X}$ is of shape $N \times D$, where N is the number of observations and D is the number of variables. For instance, if a dataset contains 100 observations with 2 variables, the shape of $\mathbf{X} \in \mathbb{R}^{100\times 2}$. Each row of $\mathbf{X}$ represents a different observation, and each column represents a different variable.

An example of such data matrix $\mathbf{X}$ is

$$\mathbf{X} = \begin{bmatrix} 1.2 & \dots & 0.5 \\ 0.8 & & 0.3 \\ 1.0 & \ddots & 0.7 \\ \vdots & \dots & \vdots \\ & (\text{Row } N) & \end{bmatrix}$$

The covariance matrix relates variables in a dataset by quantifying the degree to which two variables change together (co-vary). Specifically, each entry in the covariance matrix represents the covariance between two corresponding variables, reflecting both the direction and strength of their linear relationship. A positive covariance indicates that the variables tend to increase or decrease together, while a negative covariance suggests an inverted relationship. The covariance matrix summarizes the variability and relationships among the measured variables.

Consider a dataset with n variables, and each observation is a vector $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{in}]$. The covariance matrix, denoted by $\mathbf{C}$, is defined as

$$\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N} \underbrace{(\mathbf{x}_i - \bar{\boldsymbol{x}})(\mathbf{x}_i - \bar{\boldsymbol{x}})^\top}_{\text{outer product}}$$

where $\mathbf{x}\mathbf{x}^\top$ is the *outer product* and $\bar{\boldsymbol{x}}$ is the mean vector obtained by averaging the observations for each variable (column)

$$\bar{\boldsymbol{x}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i$$

The covariance matrix is symmetric, and its diagonal elements represent the variances of individual variables, while off-diagonal elements represent covariances between pairs of variables.

By subtracting $\mathbf{x}'_i = \mathbf{x}_i - \bar{\boldsymbol{x}}$ the mean from each observation $\mathbf{x}_i$, the data becomes *centered* around the mean. The variance of a centered variable $\mathbf{x}$ is

$$\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \bar{\boldsymbol{x}})(\mathbf{x}_i - \bar{\boldsymbol{x}})^\top \tag{13.1}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i\mathbf{x}_i^\top \tag{13.2}$$

$$= \frac{1}{N}X^\top X \tag{13.3}$$

The elements of a covariance matrix, C, quantify the degree to which two variables covary or vary together. Each entry in the covariance matrix represents the covariance between the variables corresponding to its row and column. Covariance is a measure of how changes in one variable relate to changes in another. If a specific entry in the covariance matrix is large and positive, it indicates that the variables covary positively. In practical terms, when one variable increases, the other tends to increase as well. Conversely, if the entry is large and negative, the variables covary negatively, meaning that when one variable increases, the other tends to decrease. On the other hand, if an entry is close to zero, it suggests a weak or no linear relationship between the variables. In such cases, changes in one variable are not strongly associated with changes in the other. When one variable increases or decreases, the other does not show a consistent pattern of change.

13.0.1 Covariance matrix and variance in an arbitrary direction

The covariance matrix of a centered data matrix X is given by $\frac{1}{N}X^\top X$, where X is the centered data matrix. Figure 13.1 illustrates an example dataset. The variance of points in this dataset varies depending on the direction of the vector $\mathbf{v}$.

Theorem 13.1: Covariance measure variance in any direction

The variance in a specific direction $\mathbf{v}$ can be expressed in terms of the covariance matrix C and the vector v as

$$\sigma_v = v^\top C v$$

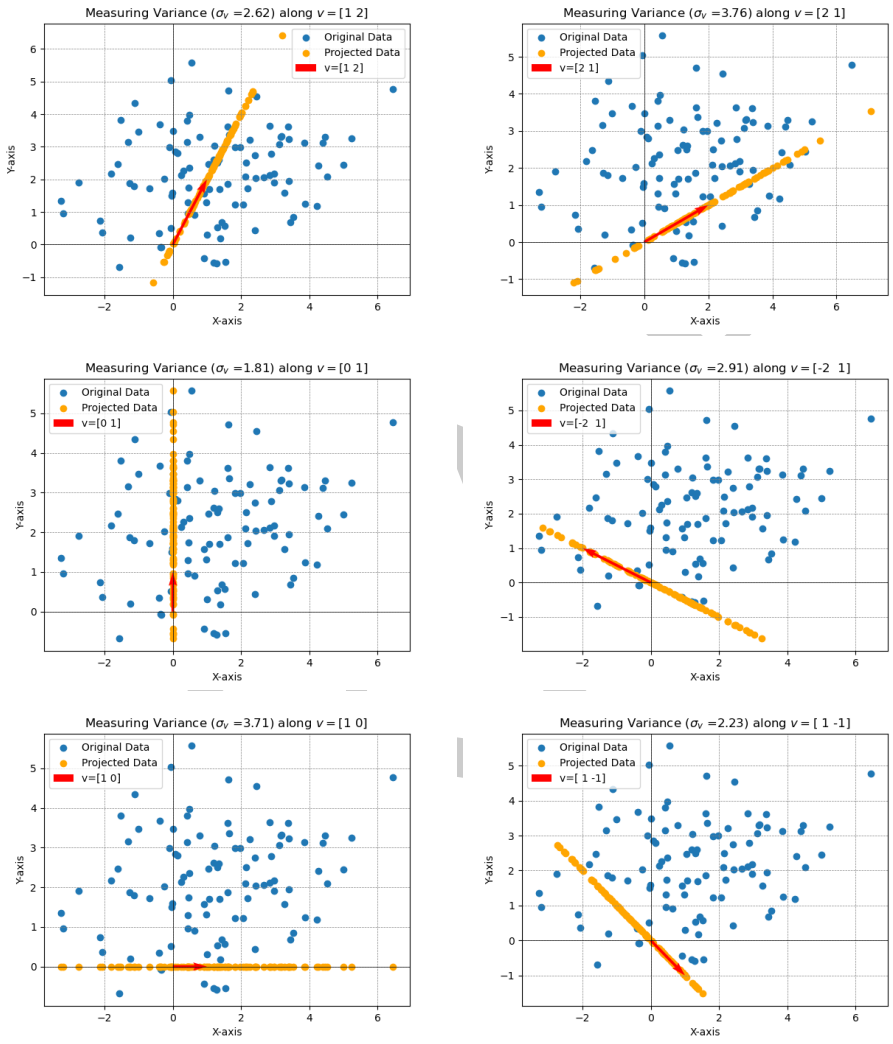

Figure 13.1: Measuring the variance of the data in different directions **v**. The measured variance in each direction is given in the title of each plot.

Proof

Assume the data matrix X has been centered. A line $\mathbf{v}$ spans a one-dimensional subspace of the data and by mapping the data to this line we can analyze the variance in this direction. The data can be mapped to the line using an inner product $v^\top A$. Since the origin is preserved the data remains centralized on the line $\mathbf{v}$ the variance in this direction is given by

$$\sigma_v = v^\top X^\top X v$$

Using the the rules for transpose we have

$$\begin{aligned}\sigma_v &= v^\top X^\top X v \\ &= v^\top X^\top X v \\ &= v^\top C v\end{aligned}$$

where $C = X^\top X$ is the (centralized) covariance matrix.
This shows that the covariance can be used directly to quantifying the variance in specific directions of the dataset.

Example 13.2: Construction of covariance matrix from data

Let's consider a 5D dataset with five data points:

$$\mathbf{x}_1 = \begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix}2\\3\\4\\5\\6\end{bmatrix} \quad \mathbf{x}_3 = \begin{bmatrix}3\\4\\5\\6\\7\end{bmatrix} \mathbf{x}_4 = \begin{bmatrix}4\\5\\6\\7\\8\end{bmatrix} \mathbf{x}_5 = \begin{bmatrix}5\\6\\7\\8\\9\end{bmatrix}$$

The mean vector is therefore

$$\boldsymbol{\mu} = \frac{1}{5}(\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \mathbf{x}_4 + \mathbf{x}_5)$$

$$\boldsymbol{\mu} = \frac{1}{5}\left(\begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix} + \begin{bmatrix}2\\3\\4\\5\\6\end{bmatrix} + \begin{bmatrix}3\\4\\5\\6\\7\end{bmatrix} + \begin{bmatrix}4\\5\\6\\7\\8\end{bmatrix} + \begin{bmatrix}5\\6\\7\\8\\9\end{bmatrix}\right)$$

$$\boldsymbol{\mu} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix}$$

The centralized matrix becomes

$$X = \begin{bmatrix} \mathbf{x}_1 - \boldsymbol{\mu} & \mathbf{x}_2 - \boldsymbol{\mu} & \mathbf{x}_3 - \boldsymbol{\mu} & \mathbf{x}_4 - \boldsymbol{\mu} & \mathbf{x}_5 - \boldsymbol{\mu} \end{bmatrix}$$

$$X = \begin{bmatrix} -2 & -1 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

The construction of the covariance matrix using outer products is given by

$$C = \frac{1}{5} X X^\top$$

$$C = \frac{1}{5} \begin{bmatrix} -2 & -1 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix} \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \\ -1 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$

$$C = \begin{bmatrix} 2.4 & 1.2 & 0 & -1.2 & -2.4 \\ 1.2 & 0.6 & 0 & -0.6 & -1.2 \\ 0 & 0 & 0 & 0 & 0 \\ -1.2 & -0.6 & 0 & 0.6 & 1.2 \\ -2.4 & -1.2 & 0 & 1.2 & 2.4 \end{bmatrix}$$

14

Eigenvalues and Eigenvectors

In chapter 3 we viewed multiplying a vector from the right with a $M \times N$ matrix as a linear transformation from $\mathbb{R}^M$ to $\mathbb{R}^N$ such that

$$Ax = x'$$

where

$$A = \begin{bmatrix} | & | & & | \\ \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \\ | & | & & | \end{bmatrix},$$

An interesting question arises as to whether there are vectors $\mathbf{x}$ for which the vector maintains its direction and possibly only changes its length after the transformation?

This seemingly simple question has found applications in diverse fields such as dimensionality reduction, social networks, communication networks, Google's PageRank algorithm, and spectral graph theory, where it is utilized to study graph characteristics, including connectivity and clustering. Moreover, it plays a crucial role in the modeling and optimization of buildings, particularly in terms of vibrations and seismic analysis.

Eigenvalues and eigenvectors find applications across various domains, yet this chapter specifically emphasizes their role in dimensionality reduction. Rather than delving into the intricacies of computing eigenvectors and eigenvalues, the focus here is on clarifying their fundamental properties in the context of dimensionality reduction and rely on there exist efficient methods to calculate them on a computer.

More formally, for an $N \times N$ square matrix A, are there nonzero vectors $\mathbf{x}$

in $\mathbb{R}^N$ such that $A\mathbf{x} = \lambda\mathbf{x}$, where λ is a scalar multiple of $\mathbf{x}$. The scalar λ is referred to as an eigenvalue of the matrix A, and the nonzero vector $\mathbf{x}$ is called an eigenvector of A corresponding to λ. All vectors are trivially eigenvectors of the identity matrix because, for any vector $\mathbf{x}$, the product $I\mathbf{x} = 1 \cdot \mathbf{x}$. What about other matrices? How many are there for a given matrix? What do they signify?

Definition 14.1: Eigenvalue and Eigenvector

Let A be an $n \times n$ matrix. The scalar λ is an *eigenvalue* of A when there is a nonzero vector $\mathbf{x}$ such that $A\mathbf{x} = \lambda\mathbf{x}$. The vector $\mathbf{x}$ is an *eigenvector* of A corresponding to λ.

Theorem 14.1: Number of Eigenvalues

A square matrix $A \in \mathbb{R}^{N\times N}$ has N eigenvalues and corresponding eigenvectors. The eigenvalues can be real or complex numbers.

Note!

Note that an *eigenvector* cannot be the zero vector. Allowing $\mathbf{x}$ to be zero would make the definition meaningless because $A\mathbf{0} = \lambda\mathbf{0}$ holds for all real values of λ. However, an eigenvalue of $\lambda = 0$ is a valid case.

14.0.1 Eigenvalues and Transformations

To gain insight into eigenvectors and eigenvalues through the lens of transformations, eigenvectors and eigenvalues play a significant role in characterizing a matrix and its corresponding linear mapping. Consider the linear transformation represented by the matrix and illustrated in Figure 14.1

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

The linear transformation represented by A scales the first coordinate while leaving the second coordinate unchanged. It is relatively simple to observe that the standard basis vector along the first coordinate direction

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

is an eigenvector since $A\mathbf{x}_1 = 2\mathbf{x}_1$ keeps its direction when multiplied by A but will be scaled by 2 (its corresponding eigenvalue $\lambda_1 = 2$). Similarly,

$$\mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

is another eigenvector with a corresponding eigenvalue $\lambda_2 = 1$. While it was relatively easy to eye-ball the eigenvalues and eigenvectors in this case, in general, this process is more challenging and often requires elaborate calculations. Typically, these calculations are handled by computers, especially for larger matrices.

Examples 14.2 and 14.3 shows an examples of the eigenvalues and eigenvectors of a reflection matrix R and a rotation matrix and illustrates the geometric relationship between eigenvectors and eigenvalues and the corresponding transformation as illustrated in Figure 14.2.

The eigenvalue λ tells us how the specific vector $\mathbf{x}$ is transformed when multiplied by the matrix A. The eigenvalue indicates whether the vector is stretched (> 1), shrunk (< 1), reversed (negative), or left unchanged by the transformation. It may even not have any real-valued eigenvalues.

Example 14.1: Eigenvalue and eigenvector

We can use definition 14.1 to verify that that for the matrix

$$A = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}$$

that both $\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $\mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ are an eigenvectors of A corresponding to the eigenvalues $\lambda_1 = 2$, and $\lambda_2 = -1$.
Multiplying $\mathbf{x}_1$ on the left by A produces

$$A\mathbf{x}_1 = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} = 2 \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ is therefore an eigenvector of A corresponding to the eigenvalue $\lambda_1 = 2$
Similarly, multiplying $\mathbf{x}_2$ on the left by A produces

$$A\mathbf{x}_2 = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} = -1 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$\mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ is therefore an eigenvector of A corresponding to the eigenvalue $\lambda_2 = -1$

In Example 14.1 the eigenvalues were 2 and -1. Observing what transformation A performs we can see that the first coordinate of $\mathbf{x}$ is exactly scaled by a factor of 2 and the second coordinate with -1. These values exactly correspond to the eigenvalues. Since the basis vectors $i = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $j = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ will maintain their direction but be scaled according to the first and second column of A explains that they are eigenvectors of A.
For the identity matrix I, every vector satisfies $A\mathbf{x} = \mathbf{x}$. In this case, all vectors are eigenvectors of I, and all eigenvalues λ are equal to or greater than 1. This situation is unusual, as most 2×2 matrices have two distinct eigenvector directions and two eigenvalues.

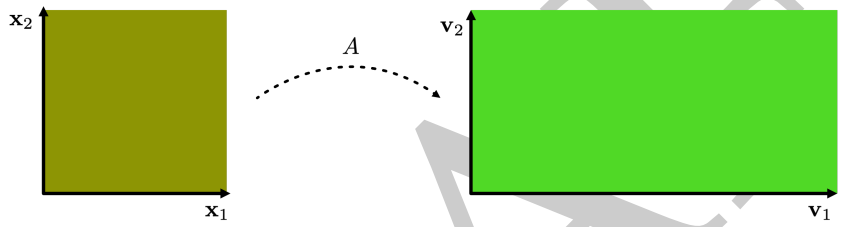

Figure 14.1: Geometric interpretation of eigenvalues. The eigenvectors of A_1 get stretched by the corresponding eigenvalues.

Example 14.2: Eigenvalues of Reflection Matrix

The reflection matrix $R = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ maps $\mathbf{x}' = R\mathbf{x}$. Using the column view of matrix multiplication, we can see that the transformation maps the first coordinate of $\mathbf{x}$ to the second coordinate of $\mathbf{x}'$ and the second coordinate of $\mathbf{x}$ to the first coordinate of $\mathbf{x}'$, e.g.

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + x_2 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}$$

The vector $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is unchanged by R and is consequently an eigenvector of R with eigenvalue 1.
The vector $\mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ is another eigenvector of R but its signs are reversed by R. In summary, the vectors v_1 and v_2 are eigenvectors of R with associated eigenvalues of 1 and -1, respectively. We also observe that a matrix with no negative entries can still have a negative eigenvalue!

Examples 14.1 - 14.3 show that a matrix A has multiple eigenvectors and eigenvalues.

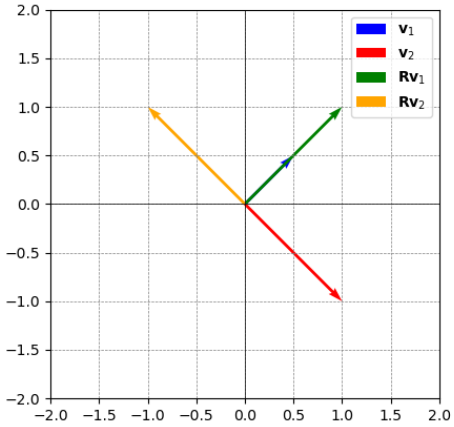

Figure 14.2: The eigenvectors of the reflection matrix

Example 14.3: Eigenvalues of Rotation Matrix

The rotation matrix $Q = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$ has no real-valued eigenvalues. The reason is that a rotation matrix will rotate all points, and thus, there won't be any vector $\mathbf{x}$ such that $Q\mathbf{x} = \lambda\mathbf{x}$ with real λ. However, it does have two complex-valued eigenvectors. While complex-valued eigenvectors may not be directly discussed in these lecture notes, they hold significance in both practical and theoretical considerations.Hence we can also deduce that since a rotation is area preserving then the determinant must be one. Notice that the rotation matrix is not necessarily symmetric.

14.0.2 Eigenvalues and determinants

Linear mappings are illustrative when developing an understanding of determinants, eigenvectors, and eigenvalues. Recall that determinants provide insights into linear transformations and their geometric properties. Specifically, the magnitude of the determinant indicates the factor by which the area is scaled, while the sign of the determinant determines whether the transformation reverses the orientation of the shapes. If the determinant is positive, the linear transformation preserves the orientation, and the area is scaled by the absolute value of the determinant. If the determinant is negative, the transformation reverses the

orientation, resulting in a negative scaling factor. The significance of eigenvalues in describing the scaling behavior of a linear transformation matrix, denoted as A, is shown by Theorem 14.2.

Theorem 14.2: Determinant and Eigenvalues

The determinant of a matrix $A \in \mathbb{R}^{N \times N}$ is the product of its eigenvalues

$$\det(A) = \prod_{i=1}^{n} \lambda_i,$$

where $\lambda_i \in \mathbb{C}$ are (possibly repeated) eigenvalues of A.

This theorem establishes the connection between the product of eigenvalues and the scaling of areas under the influence of the transformation. In essence, Theorem 14.2 provides a unified perspective on the relationship between eigenvalues and the transformation given by a matrix A. Example 14.4 and Figure 14.3 show concrete examples of using Theorem 14.2 to analyze the impact of a transformation.

Example 14.4: Eigenvalues and Determinants

The matrices $A_1 - A_4$ are linear transformations that this example will use to illustrate eigenvalues, eigenvectors, and determinants.

$$A_1 = \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & 1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} \cos\left(\frac{\pi}{6}\right) & -\sin\left(\frac{\pi}{6}\right) \\ \sin\left(\frac{\pi}{6}\right) & \cos\left(\frac{\pi}{6}\right) \end{bmatrix}$$
$$A_3 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad A_4 = \begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{bmatrix}$$

Through automatically computed eigenvectors, the aim is to illustrate how the directions of the eigenvectors, and their associated eigenvalues and the determinant, offer insights into the transformation of space induced by these linear mappings. Figure 14.3 visualized the transformations and their corresponding scaled (by eigenvalue) eigenvectors.

A_1 The matrix A_1 shears the points along the horizontal axis to the right if they are on the positive half of the vertical axis, and to the left vice versa. The eigenvalue $\lambda_1 = 1 = \lambda_2$ is repeated, and the eigenvectors are collinear. The mapping is area-preserving ($\det(A_1) = 1$). This indicates that the mapping acts only along one direction, specifically the horizontal axis.

A_2 The matrix A_2 is equal to $\frac{1}{2}\begin{bmatrix} \sqrt{3} & -1 \\ 1 & \sqrt{3} \end{bmatrix}$ and rotates the points by $\frac{\pi}{6}$ rad (30°) counter-clockwise. It has only complex eigenvalues, as no vectors are preserved, and since a rotation must be volume-

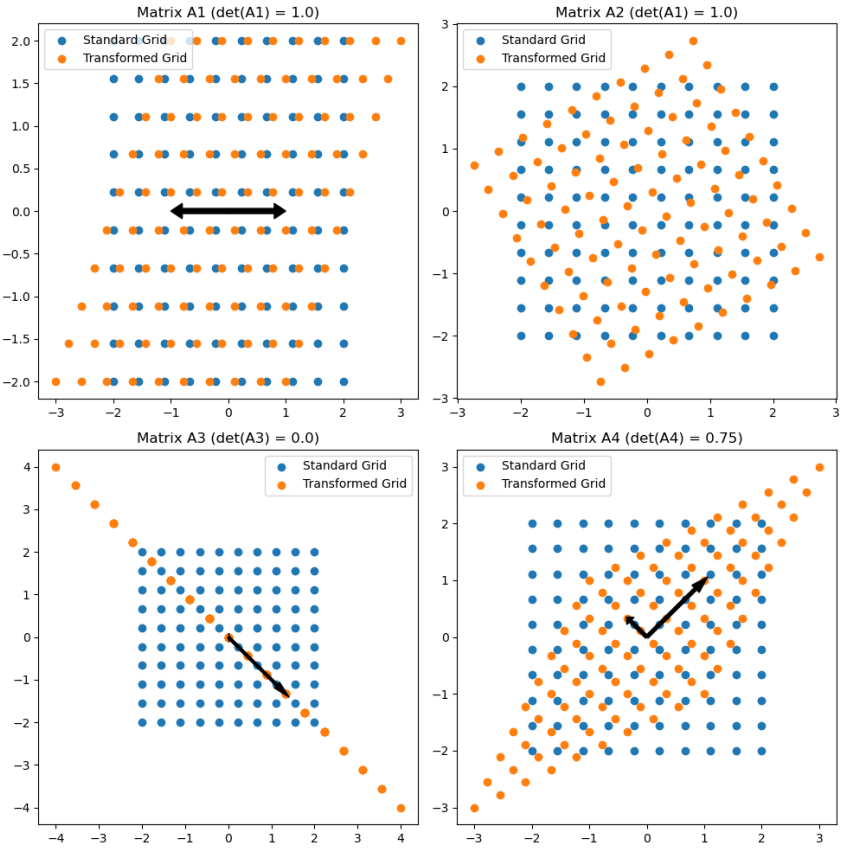

Figure 14.3: Relating eigenvalues, eigenvectors and determinants with transformations as described in Example 14.4

preserving, its determinant is 1.

A_3 The matrix represents a mapping in the standard basis that collapses a two-dimensional domain onto one dimension. The eigenvalues are 2 and 0, and hence the determinant is 0. This shows that the mapping is non-invertible, which makes sense, as mapping the points to a line makes it impossible to transform them again to the standard grid.

A_4 The matrix A_4 is a shear-and-stretch mapping that scales space by 75%, as its determinant is $\frac{3}{4}$. It stretches space along the (red) eigenvector of λ_2 by a factor of 1.5 and compresses it along the orthogonal (blue) eigenvector by a factor of 0.5. So, A_4 has eigenvalues $\frac{3}{2}$ and $\frac{1}{2}$ with corresponding eigenvectors $\begin{bmatrix}1\\1\end{bmatrix}$ and $\begin{bmatrix}1\\-1\end{bmatrix}$.

Example 14.4 also revealed that despite that some eigenvalues are complex-valued then the eigenvectors form a basis of a subspace. This informal observation is captured in theorem 14.3.

Theorem 14.3: Eigenvalues and independence

The eigenvectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of a matrix $A \in \mathbb{R}^{N \times N}$ with N distinct eigenvalues $\lambda_1, \ldots, \lambda_N$ are linearly independent.

Proof

In the context of real eigenvectors, Theorem 14.3 reveals a useful criterion: the presence of non-zero eigenvalues signifies a subspace spanned by a set of eigenvectors. This not only aids in determining whether a combination of eigenvectors forms a subspace but also facilitates the selection of a maximal set of eigenvectors. As a consequence of Theorem 14.2 and Theorem 14.3 it follows that if one of the eigenvalues is 0 then the column vectors of A are linearly dependent and hence A is non-invertible.

Consider a matrix $A \in \mathbb{R}^{n \times n}$ with n distinct eigenvalues $\lambda_1, \ldots, \lambda_n$ and corresponding eigenvectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$. We want to show that these eigenvectors are linearly independent.

Assume that there exist constants $c_1, c_2, \ldots, c_n$, not all zero, such that

$$c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \ldots + c_n\mathbf{x}_n = \mathbf{0}$$

If we pre-multiply both sides of the equation by A, we get

$$A\left(c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \ldots + c_n\mathbf{x}_n\right) = A\mathbf{0}$$

Using the eigenvector-property $A\mathbf{x}_i = \lambda_i\mathbf{x}_i$, this simplifies to

$$c_1\lambda_1\mathbf{x}_1 + c_2\lambda_2\mathbf{x}_2 + \ldots + c_n\lambda_n\mathbf{x}_n = \mathbf{0}$$

Now, since the eigenvalues are distinct, $\lambda_1, \lambda_2, \ldots, \lambda_n$ implies that each term in the sum above is a linear combination of linearly independent vectors. Therefore, the only way for the sum to be zero is if $c_1 = c_2 = \ldots = c_n = 0$. Hence, the eigenvectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are linearly independent.

The eigenvectors associated with non-zero eigenvalues define an important subspace termed the *invariant subspace*. This subspace remains unaffected by the linear transformation, undergoing only a scaling transformation determined by the corresponding eigenvalue. Essentially, vectors within this subspace experience either a stretch or contraction during the transformation. The magnitude of the eigenvalues associated to these eigenvectors indicates the scaling factor: an eigenvalue greater than 1 implies stretching, while a value between 0 and 1 implies contraction.

Each eigenvector in this subset serves as a principal direction, showing the direct effect of the linear transformation—specifically, a simple scaling. These directions are often referred to as the *dominant* or *principal directions* of the transformation.

Furthermore, the linear independence of eigenvectors corresponding to dis-

tinct non-zero eigenvalues is crucial. This independence ensures that the associated principal directions are distinct and do not overlap.

14.1.0 Combining Eigenvectors and subspace transformation

By definition we know that an eigenvector $\mathbf{v}_i$ of the square matrix A and its corresponding eigenvalue λ_i is

$$A\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

where A is a square matrix.

By considering all eigenvectors and eigenvalues, one can aggregate them into a unified matrix product.

$$AV = \Lambda V$$

where Λ is a diagonal matrix containing the eigenvalues,

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix},$$

and V contains all the eigenvectors.

If all eigenvalues are different, the eigenvectors are independent (and hence forms a basis). In this case, we can diagonalize the matrix A using the inverse of the matrix of eigenvectors:

$$V^{-1}AV = \Lambda \tag{14.1}$$

where Λ is a diagonal matrix containing the eigenvalues.

The interpretation of Equation 14.1 is that the matrix A is transformed into a new basis defined by the columns of V. This basis is particularly special because it consists of the eigenvectors of A, forming an orthonormal set. Consequently, the inverse transformation V^{-1} brings the data back to its original coordinate system.

The resulting matrix Λ is diagonal, meaning that the transformation represented by V decouples the original data into independent components along each eigenvector direction. The diagonal elements of Λ are the eigenvalues corresponding to each eigenvector.

15

Dimensionality reduction

The objective of dimensionality reduction is to transform a high-dimensional dataset, originally residing in an N-dimensional space, into a lower-dimensional subspace while retaining as much valuable information as possible. This transformation is achieved through the process of projection onto a k-dimensional subspace, where k is typically much smaller than N. The selection of the appropriate value for k is a critical decision, as it determines the effectiveness of the representation.

This chapter focuses on Principal Component Analysis (PCA), a unsupervised method for dimensionality reduction. PCA operates on the principle of maximizing variance to identify the most informative directions in the data. Figure 15.1 illustrates a 3D dataset in which the data predominantly resides within a 2-dimensional subspace of the overall 3 dimensions. The dashed red lines delineate the subspace, showing that the majority of the variance is concentrated in this 2-dimensional domain. By capturing the directions with the highest variance, PCA aims to preserve the essential structures within the dataset and throws away the remaining components. It turns out that the largest variance is given by the eigenvalues and thus the (ordered) directions of variance are given by the eigenvectors.

It is important to note that various dimensionality reduction methods exist, each employing different criteria for optimization. The choice of the most suitable method depends on the specific characteristics and goals of the dataset at hand. Understanding the nature of the data, whether it's labeled or unlabeled, and the objectives of the analysis will guide the selection of an appropriate dimensionality reduction technique.While PCA emphasizes variance, alternative methods may prioritize the independence of components or other measures of optimality, such as considering class labels and preserving the discriminative information between different classes. In scenarios where the task involves classification or pattern recognition, methods that take class labels may consider

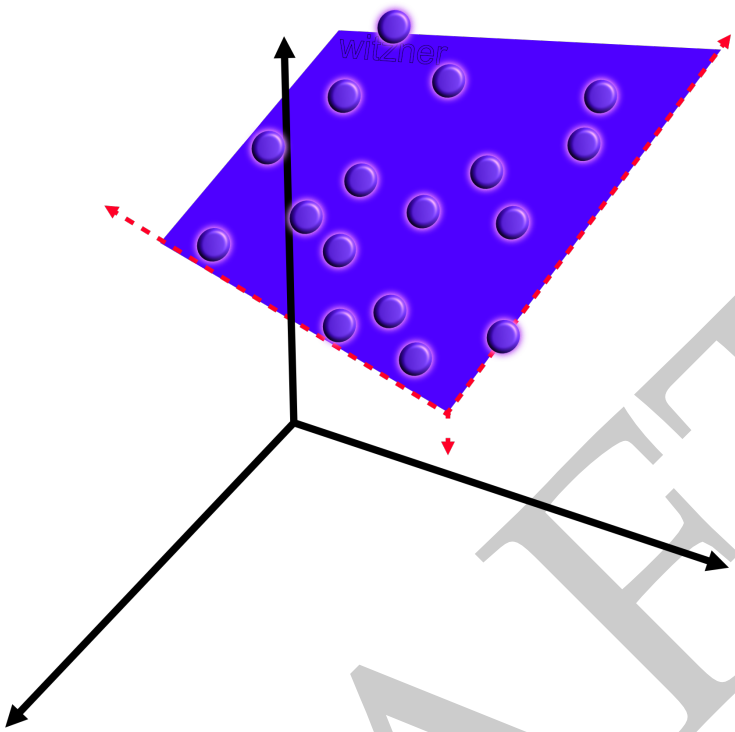

Figure 15.1: 3D dataset but where most of the data lies in a subspace spanned by two of the 3 dimensions. The red vectors is another coordinate system spanning the entire space. The two basis vectors can span most of the data linearly.

to maximize the separation between different classes in addition to capturing variance.

15.1.0 Principal Component Analysis (PCA) Overview

PCA focuses on identifying axes within the entire dataset that maximize variance. The eigenvalues and eigenvectors of the symmetric covariance matrix play an essential role in determining these high-variance directions.

Variance, representing the spread or information content in the data, is illustrated in Figure 15.1. High-variance directions often capture significant features or patterns, while directions with little variance are considered noise or less relevant. Concentrating on directions with the most variance enables us to retain essential information, discarding less informative dimensions. In other words we attempt to find a k dimensional subspace spanned by k basis vectors. As pointed out in Theorem 13.1 the covariance matrix can be used to find variances in any direction. It turns out (see section 15.2) that the eigenvectors represent

directions in the original feature space where the variability of the data is maximized and the magnitude of the corresponding eigenvalue indicates the amount of variance captured by the respective eigenvector. Eigenvalues close to 0 are less informative and may be excluded when constructing the new feature subspace. Larger eigenvalues indicate directions with more variance, while smaller eigenvalues indicate directions with less variance.

The procedure for discovering and reducing the dimensions of a dataset $X \in \mathbb{R}^{N \times D}$, comprising N data points, each with D observed variables, through PCA can be outlined in the following steps and illustrated in Figure 15.3. The steps will be supported with a 3D example dataset shown in Figure 15.2. The arguments and details will be given in section 15.2.

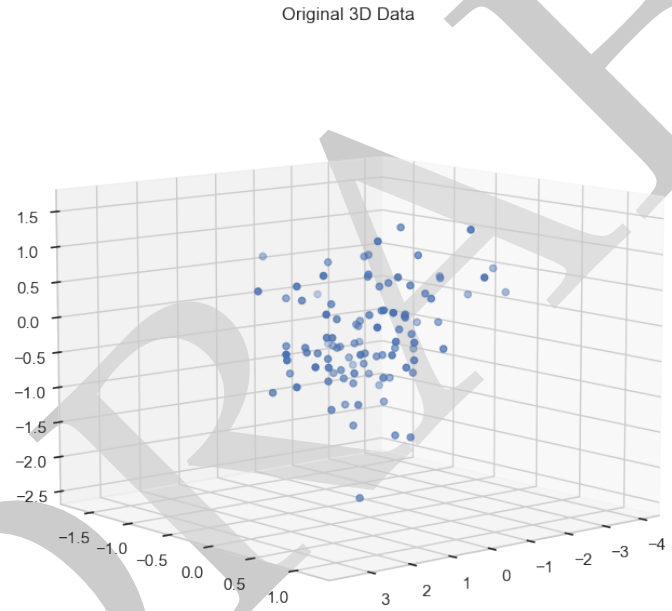

Figure 15.2: 3D dataset

1. CENTER THE DATA Compute the D-dimensional mean vector ($\boldsymbol{\mu}$) by calculating the means for every dimension of the entire dataset and subtracting this from each data point ($\mathbf{x}_i$):

$$\mathbf{x}_i' = \mathbf{x}_i - \boldsymbol{\mu}$$

2. COMPUTE COVARIANCE MATRIX Compute the $D \times D$ covariance matrix (C) of the entire centered dataset using Equation 13.3:

$$C = \frac{1}{N} X^\top X$$

where X is the centralized data matrix. The 3×3 covariance matrix of the dataset is shown in step 2 of Figure 15.3.

3. EIGENVALUES AND EIGENVECTORS Calculate the eigenvectors $\mathbf{v}_i$ and eigenvalues λ_i such that:

$$C = V \Lambda V^\top$$

where:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_D \end{bmatrix}$$

and:

$$V = \begin{bmatrix} | & | & & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_D \\ | & | & & | \end{bmatrix}.$$

Step 3 in Figure 15.3 shows the eigenvalues overlayed the original data. The black and red arrows indicate the eigenvectors. The red arrows have unit size while the black have been scaled with 3 times the eigenvalue $3\lambda_i$. It is clear that some directions possess more variance than others.

4. SORTING VECTORS Sort the eigenvectors by decreasing eigenvalues. The eigenvector associated with the largest eigenvalue corresponds to the direction along which the variance of the data is maximized. Subsequent eigenvectors are associated with decreasing eigenvalues and represent directions of decreasing variance.
Step 4 in Figure 15.3 shows the sorted eigenvalues and the cummulative eigenvalues. The *Scree plot* shows the sorted eigenvalues. The cummulative eigenvalues contains information about how much of the total variance is maintained by keeping the first i eigenvectors.

5. SELECT SUBSPACE Choose k of the D eigenvectors with the largest eigenvalues to form a $D \times K$ dimensional matrix:

$$\Phi = \begin{bmatrix} | & | & & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_k \\ | & | & & | \end{bmatrix}.$$

where each column contains an eigenvector $\mathbf{v}_i$. Since the eigenvectors are orthonormal the matrix Φ consitute an orthonormal basis and a transformation

matrix from the original data space to the subspace spanned by the eigenvectors.

6. TRANSFORM DATA Use the $D \times k$ eigenvector matrix Φ to transform the data samples onto the new subspace spanned by the eigenvectors:

$$\mathbf{y} = \Phi^\top \mathbf{x}$$

where $\mathbf{x}$ is a $D \times 1$-dimensional vector representing one sample, and $\mathbf{y}$ is the transformed $k \times 1$-dimensional sample in the new subspace. Notice that all datapoints in the dataset can be projected to the new subspace by:

$$\mathbf{Y} = \Phi^\top X^\top$$

where $X \in \mathbb{R}^{N \times D}$ contains all the N data points where the mean has been subtracted. The projected 3D data onto the 2D subspace spanned by the eigenvectors is shown in step 6 of Figure 15.3.

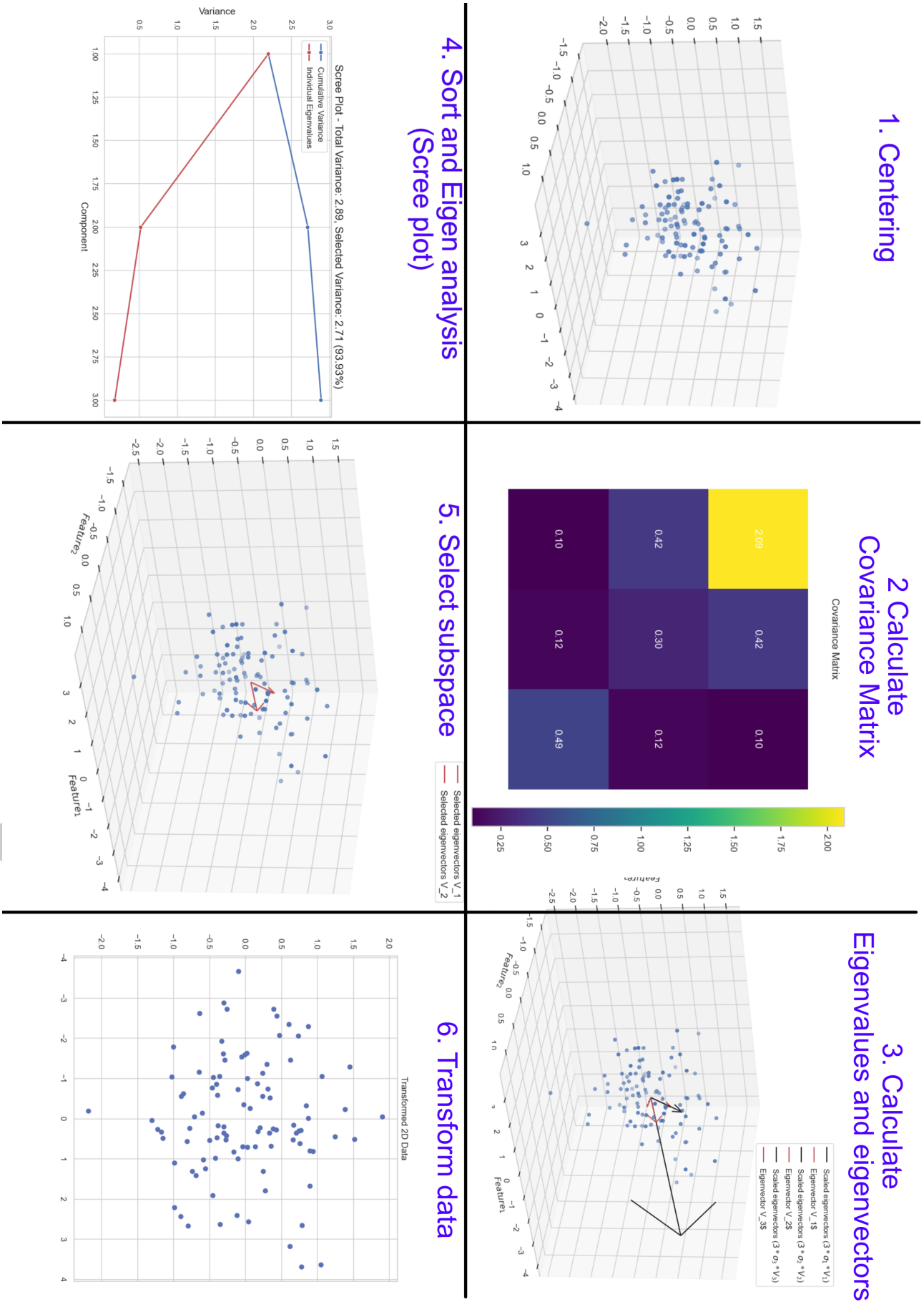

Figure 15.3: The steps of Prinicpal component analysis

15.2.0 Eigenvectors and Covariance Matrix

The previous section presented an overview of PCA. This section will delve deeper into the details of each step and provide a rationale for why the previously introduced steps are correct. The findings of this section can be summarized in the following theorem

Theorem 15.1: Eigenvalues and Eigenvectors of the Covariance Matrix

Let C be the covariance matrix of a dataset $X \in \mathbb{R}^{N \times D}$, where N is the number of data points and D is the number of dimensions. The eigenvectors of C, denoted as V and arranged in columns, along with the corresponding eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_D \geq 0$, satisfy the following properties:

Variance in Direction The eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_D$ are real and non-negative, representing the variance specified by the corresponding eigenvector. The first eigenvector points in the direction of the most variance, the second eigenvector in the direction of the second most variance (orthogonal to the first), and so forth.

Total Variance The *trace* (sum of the diagonal elements) of

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_D \end{bmatrix}$$

measures the total variance of the dataset.

Orthonormal Basis The eigenvectors V form an orthonormal basis, meaning $V^\top V = I$, where I is the identity matrix. This implies that the eigenvectors are mutually orthogonal, and the orthonormality ensures a convenient basis for expressing the dataset.

Eigenvectors of Covariance matrix An eigenvalue λ_i and eigenvector $\mathbf{v_i}$ of the covariance matrix is given by

$$Cv_i = \lambda_i v_i$$

and all the eigenvectors and eigenvalues of C can be combined into

$$CV = \Lambda V$$

where

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_D \end{bmatrix}$$

and:

$$V = \begin{bmatrix} | & | & & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_D \\ | & | & & | \end{bmatrix}.$$

Each eigenvector is associated with an eigenvalue, representing the "length" or "magnitude" of the eigenvector. Denote the eigenvectors of the covariance matrix by $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_D$ and their corresponding eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_D$. For the sake of simplicity (and without loss of generality) assume all eigenvalues ordered such that $\lambda_1 \geq \lambda_2 \geq \ldots \lambda_D \geq 0$ and their corresponding eigenvectors in V are ordered accordingly.

Definition 15.1: Positive Semidefinite

A symmetrix matrix C is called *positive semidefinite* when all its eigenvalues are non-negative. A positive semidefinte matrix C can written in the form $\mathbf{v}^\top C\mathbf{v} \geq 0$ for any $\mathbf{v} \neq \mathbf{0}$.

The covariance is symmetric but it turns out that it is also *positive semidefinite* as argued in the following [1]. The spectral theorem is key in understanding principal component analysis.

recall that proofs are not expected for the exam

Proof

This property arises from the definition of the covariance matrix and its interpretation in terms of variances and covariances.

$$C = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \bar{\boldsymbol{x}})(\mathbf{x}_i - \bar{\boldsymbol{x}})^\top$$

Here, $\mathbf{x}_i$ represents the data points, and $\bar{\boldsymbol{x}}$ is the mean vector. Now, let's express this in terms of matrix operations using the data matrix X, where each row is a data point:

$$C = \frac{1}{N}X^\top X$$

For any vector $\mathbf{v} \neq \mathbf{0}$, the quadratic form $\mathbf{v}^\top C\mathbf{v}$ can be expressed as:

$$\mathbf{v}^\top C\mathbf{v} = \frac{1}{N}(X\mathbf{v})^\top(X\mathbf{v})$$

Since the term inside the parentheses is a sum of squared real numbers, it is always non-negative. Therefore, $\mathbf{v}^\top C\mathbf{v} \geq 0$ for any $\mathbf{v}$, and this implies that the covariance matrix C is positive semidefinite.

Theorem 15.2: Spectral Theorem

Every symmetric matrix C has only real eigenvalues, and the eigenvectors can be chosen to be orthonormal (orthogonal with unit length). The symmetric matrix can be factorized into $C = V\Lambda V^\top$, where Λ contains real eigenvalues, and V consists of orthonormal eigenvectors in its columns. This process is known as *symmetric diagonalization*.

Eigenvectors of the covariance matrix constitute an orthonormal basis As pointed out in Theorem 13.1 the covariance matrix C captures the variance $\sigma_v = \mathbf{v}^\top C\mathbf{v}$ in any direction $\mathbf{v}$. The eigenvectors of the covariance matrix represent directions of variance. According to the spectral Theorem (Theorem 15.2) the eigenvectors in V are orthonormal directions in the original feature space. This implies that $V^\top V = I$, where I is the identity matrix. In simpler terms, any two distinct eigenvectors are orthogonal, and thus, their inner product is 0. This also means that the inverse $V^\top = V^{-1}$. That the eigenvectors in V are orthogonal means that we can think of them as constituting a new coordinate system in the data.

Direction of variance The covariance matrix transformation Cv_i produces a scaled version of the original eigenvector v_i, with the scaling factor being the corresponding eigenvalue λ_i. Therefore, the variance in the direction of

an eigenvector $\mathbf{v}$ is given by $\sigma_v = \mathbf{v}^\top C\mathbf{v}$. This implies that the eigenvalues of the covariance represent the variances along specific directions defined by the eigenvectors.

Considering the expression $Cv_i = V\Lambda V^\top v_i$, where v_i represents the i-th eigenvector of the covariance matrix C, and Λ is a diagonal matrix containing the eigenvalues. As the eigenvectors denoted by V form an orthonormal set, each eigenvector is orthogonal to all others ($v_i^\top v_j = 0$ for $i \neq j$). Thus, when applying the transformation $V\Lambda V^\top$ to an eigenvector v_i, the orthogonality ensures that all other eigenvectors do not contribute to the result. Expanding the expression using the definition of eigenvalues, we get $Cv_i = V\Lambda V^\top v_i$. Since $V^\top v_i$ isolates the i-th component of v_i, the multiplication by Λ scales this component by the corresponding eigenvalue λ_i. Therefore, the resulting vector $V\Lambda V^\top v_i$ captures the variance of the data in the direction of the i-th eigenvector.

Consequently, the first eigenvector points in a direction specified by the data, with the largest eigenvalue. The sum of all eigenvalues contains the total variance of the dataset (trace of Λ).

Eigenvectors mapping to new space The expression $C = V\Lambda V^\top$ arises from the diagonalization of the covariance matrix. In this context, V functions as the transformation matrix, guiding the shift from the original coordinate system to one defined by the eigenvectors (principal components). This transformation is orthogonal, ensuring the preservation of distances and angles. Thus, $C = V\Lambda V^\top$ signifies a change of coordinates to a system where the covariance matrix becomes diagonal. The diagonal elements of Λ represent the variances along the new coordinate axes, and the columns of V indicate the directions of these axes.

By selecting k corresponding to a certain percentage p of the total variance, we can obtain a matrix

$$\Phi = \begin{bmatrix} | & | & & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_k \\ | & | & & | \end{bmatrix}$$

containing the k eigenvectors $\mathbf{v}_i$ spanning a subspace that captures p percent of the total variance of the dataset. A low-dimensional representation of the original data X can consequently be achieved by transforming the data to the subspace using $\Phi^\top X^\top$.

Eigenvectors are mixing components Eigenvectors act as mixing components that define directions in the original feature space, and the eigenvalues determine the importance or weight of each direction in the transformed representation. The eigenvectors provide a new basis for representing the data, and the eigenvalues dictate the importance of each basis vector. By selecting a subset of

these eigenvectors (principal components) based on their corresponding eigenvalues, we can achieve dimensionality reduction while retaining the essential structure of the dataset.

The transformation of a data point $\mathbf{x}$ by the covariance matrix C using the eigenvectors:

$$C\mathbf{x} = V\mathbf{\Lambda}V^T\mathbf{x}$$

can be broken down into two steps:

1. TRANSFORMATION BY V^T The data point $\mathbf{x}$ is projected onto the coordinate system defined by the eigenvectors. Each eigenvector represents a direction in the original space.
2. SCALING BY $\mathbf{\Lambda}$ The resulting vector, obtained after the projection, is scaled by the corresponding eigenvalues. This scaling accounts for the variance in the direction specified by each eigenvector.

The final transformed vector, $C\mathbf{x}$, can be seen as a linear combination of the original features, with the coefficients determined by the components of V^T. Each eigenvector corresponds to a particular mixing component, and the magnitude of the corresponding eigenvalue determines the influence of that component on the overall transformation.

Affine Mapping between Latent Space and Real Space in PCA In Principal Component Analysis (PCA), a new set of orthogonal axes (the *principal components*), is identified through eigendecomposition of the covariance matrix of a centered dataset. A data point $\mathbf{x}$ from the original feature space can be expressed in the context of these principal components as

$$\mathbf{x} = V\mathbf{a} + \boldsymbol{\mu}, \tag{15.1}$$

where V is the matrix containing the eigenvectors in its columns, and $\mathbf{a}$ represents the coordinate vector of $\mathbf{x}$ within the new orthogonal basis spanned by the eigenvectors in V. The term $\boldsymbol{\mu}$ corresponds to the mean vector of the dataset X. $\boldsymbol{\mu}$ is needed because the data matrix was centered by subtracting the mean vector before calculating the covariance matrix as to ensure that the covariance matrix reflects deviations from the dataset's mean rather than from the coordinate origin of the feature space. Given that the columns of V constitute an orthogonal set, it follows that the inverse of V is its transpose, that is, $V^{-1} = V^\top$.

A dataset X with dimensions $D \times N$, where D is the original feature space dimension, and N is the number of data points. The *latent space*, as defined by the principal components, encapsulates a lower-dimensional subspace which

retains a certain amount of the total variance of the original dataset. All eigenvectors are mutually orthogonal. Thus, selecting a subset of these eigenvectors forms an orthogonal subspace of the original N-dimensional data space. The data can be projected onto the latent space, yielding a transformed matrix Φ of dimensions $N \times K$, where K is the desired reduced dimensionality (for example a certain percentage of the total variance of the dataset). The mapping of a point $\mathbf{x}_i$ from the original space to the latent space can be expressed as an affine transformation

$$\mathbf{a}_i = \Phi^\top (\mathbf{x}_i - \boldsymbol{\mu}), \tag{15.2}$$

where Φ is the matrix of k-principal components (eigenvectors) as its column vectors, $\boldsymbol{\mu}$ is the mean vector of the original data and $\mathbf{a}_i \in \mathbb{R}^k$ is the k-dimensional vector representation of $\mathbf{x}_i$ in latent space. The inverse mapping, reconstructing the data in the original space from the latent space, is given by

$$\mathbf{x}_i \approx \Phi \mathbf{a}_i + \boldsymbol{\mu}. \tag{15.3}$$

Notice that this may only be an approximation since the $N - k$ last dimensions were removed when performing the dimensionality reduction. The affine mapping allows for a meaningful interpretation of the latent space. Each column of Φ represents a direction in the original space, and the associated singular value captures the importance of that direction. The latent space coordinates $\mathbf{a}_i$ then represent the data points projected onto these principal directions. The reconstruction error $\mathbf{x_i}$ of a single point $\mathbf{x_i}$ can be measured by first mapping it to latent space using Equation 15.2 and then back to the original space using Equation 15.3 and then observe the difference.

$$\text{Reconstruction Error} = \|X - X_{\text{recon}}\|$$

where X_{recon} is the set of points in X which have been mapped to latent space and back again.

Basis for Generative Models Projecting all data points $\mathbf{x}_i$ to the vectors $\mathbf{a}_i$ in latent space using Equation 15.2 provides a set of points which in some sense are real representations of the data in latent space. However any other points in latent space are governed by the eigenvectors could be valid representations and thus be mapped back to the original space using Equation 15.3. This will in some sense generate new data in the original space which could be valid provided the point in latent space does not deviate too much from what was observed in the original space. As the eigenvalues represent the variance in the directions given by the eigenvectors we can constrain points in latent space to a

given percentage in each direction. That is each element a_i in $\mathbf{a}$ in latent space could be constrained by the variance λ_i in that direction e.g.

The validity of these mappings relies on the assumption that the latent space coordinates do not significantly deviate from the observed variance in the original dataset. Given that the eigenvalues represent the variance along their respective eigenvectors, constraints can be imposed on the points in latent space based on a predefined percentage of the total variance. Specifically, for a vector $\mathbf{a}$ in latent space, each component a_i can be restricted by the associated variance λ_i along the i-th principal component. This establishes a boundary within which the latent coordinates are expected to lie, thereby ensuring that the generated data in the original space remains representative and consistent with what was originally observed e.g.

$$-s\sqrt{\lambda_i} \leq a_i \leq s\sqrt{\lambda_i}, \tag{15.4}$$

where s is a factor determining the level of allowed deviation, typically set based on the desired confidence interval of the normal distribution, thereby reflecting a controlled and meaningful exploration of the data's inherent structure in the latent space.

16

Condition number

coming soon

17

Kernel PCA (Not strictly needed))

18

Mean Shift Clustering

Mean Shift is a non-parametric clustering algorithm that identifies clusters in a dataset by iteratively shifting data points towards the mean of the points in their vicinity. The algorithm is also known as the mode-seeking algorithm because it locates the high-density regions (modes) of the data. One of the primary benefits of Mean Shift is that it does not require prior specification of the number of clusters; it determines the number of clusters based on the data itself, governed by a parameter known as bandwidth.

Motivation Imagine you have a set of points on a plane and you want to find which points belong together in a group. In some places there are lots of data points and in other less so. The areas with highest density are likely to form a cluster. Mean Shift attempts to find clusters of data with high density by following the steps of a hiker trying to find areas where the density of the data is the highest. At every step, the hiker looks around and takes a step in the direction where density is the highest. The hiker can only see a certain distance away (given by a kernel). For a collection of points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, the Mean Shift algorithm starts at each point and changes its location towards where the density is the highest (highest weighted mean). This is done by taking an weighted average of the points around it, giving more importance to points that are closer and less to those that are farther away. The *Mean Shift vector* tells how to move towards the "average" of points nearby

$$\text{Mean Shift at point } \mathbf{x} = \text{Average of all points around } \mathbf{x} - \mathbf{x}. \tag{18.1}$$

This process is repeated several times, and eventually, points that are close to each other move together and form groups. This grouping is useful because it can help understand how the data points are organized without needing to assume any specific structure in advance. It is like letting the points decide for

themselves how they want to be grouped based on where the dense areas are in the data landscape.

The *Mean Shift vector* at a point $\mathbf{x}$ is defined as

$$\mathbf{m}(\mathbf{x}) = \frac{\sum_{i=1}^{n} \mathbf{x}_i K\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)}{\sum_{i=1}^{n} K\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)} - \mathbf{x}. \tag{18.2}$$

and is essentially the vector between the original point and the weighted average of the data points (*centroid*) where K is the kernel function, h is the *bandwidth* parameter and $\mathbf{x}$ is a point. Values that are far away from $\mathbf{x}$ contributes little to the mean shift vector while the oposite is true for the points that are close. The algorithm calculates the Mean Shift vector, which tells each point how to move towards the weighted average of the points nearby. This process is repeated several times, and eventually, points that are close to each other move together and form groups. This grouping is useful because it can help understand how the data points are organized without needing to assume any specific structure in advance. It is like letting the points decide for themselves how they want to be grouped based on where the dense areas are in the data landscape.

The general steps of the Mean Shift algorithm are

1. Initialize a candidate mode (centroid) $\mathbf{x}_0$ for each data point.
2. Compute the Mean Shift vector $\mathbf{m}(\mathbf{x}_0)$ at the location of the candidate mode.
3. Update the candidate mode: $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{m}(\mathbf{x}_t)$.
4. Repeat steps 2 and 3 until the candidate mode converges, that is, when $\mathbf{m}(\mathbf{x}_t)$ is below a small threshold.

19

Gradient Descent

Earlier in the course, we engaged with the optimization problem of minimizing linear functions, specifically those represented in matrix-vector form. These functions can be articulated as follows:

$$\sum_{i=1}^{N}(Ax_i - y_i)^2 = \|Ax - y\|^2, \tag{19.1}$$

where A is a matrix that transforms the vector x_i, and y_i are the observed data points. This formulation simplifies to the squared norm of the difference between the transformed vectors and the data, thereby defining our cost or loss function in terms of the least squares. This is effectively analogous to minimizing the loss function $\mathcal{L}$, which is expressed as:

$$\mathcal{L}(x) = \|Ax - b\|^2.$$

This equivalence arises from the fact that the square root function is monotonically increasing, thus ensuring that minimizing the square of the norm will also minimize the norm itself. We have previously established that linear equations of this sort can be solved by direct methods, yielding a closed-form solution, which is obtained through the application of linear least squares and the pseudoinverse.

19.1.0 Non-linear optimization and Gradient descent

When we encounter more intricate objective functions that exhibit nonlinearity in the parameters (e.g. model parameters $\mathbf{w}$), such as minimizing the least squares error when $F_{\mathbf{w}}(x)$ is non-linear

$$\mathcal{L} = \sum_{i=1}^{N} (F_{\mathbf{w}}(x_i) - y_i)^2, \tag{19.2}$$

we recognize that the linear tools are no longer applicable. Here, $F_{\mathbf{w}}$ symbolizes a nonlinear transformation applied to the input data x_i, and the traditional analytical methods fall short. In such scenarios, we must resort to iterative, nonlinear optimization methods. *Gradient descent* emerges as a prominent example of such methods, allowing us to iteratively approach the values of x that minimize the loss function $\mathcal{L}$:

$$\arg\min_{w} \mathcal{L}(\mathbf{w}) = \|F_{\mathbf{w}}(X) - Y\|^2,$$

where X incorporates all the input data points, and Y contains their corresponding target values.

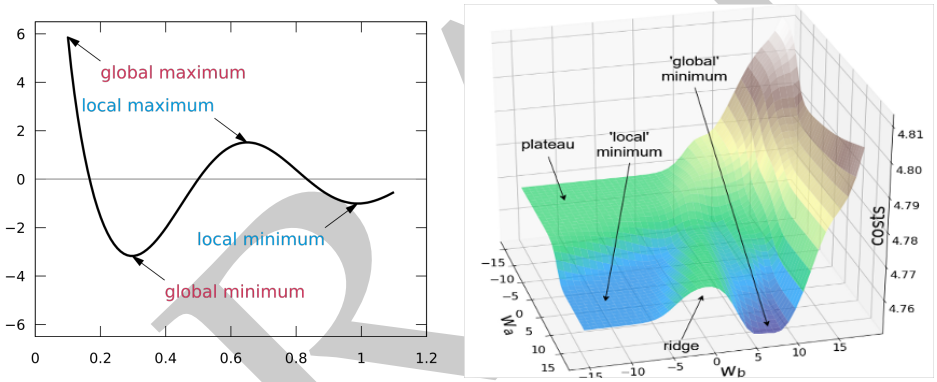

Figure 19.1: Visual representations of local and global minima and maxima in scalar fields.

Figure 19.1 shows the concept that a function may possess multiple extrema, both *local* and *global*. Moreover, in multivariate functions, we may encounter the phenomenon known as a *ridge*, where the gradient in one direction is significantly more pronounced than in another.

Gradient descent is an iterative optimization algorithm to find (local) minima/maxima of a (non-linear) function f.

The intuition The fundamental idea of the gradient descent can be illustrated by a person who stands on a mountain and has to find the way down to the bottom. The person will do this one step at the time but always taking the steepest step downwards. The person will (believe) to be at the bottom when there is no

further steepest step down to be taken. *Gradient ascent* attempts to achieve the opposite where the climber has to reach the highest mountain.

The idea of gradient descent for a *univariate* function in x is illustrated in **??**. Let the "mountain" from the example above be described by the function $f(x)$. The first derivative, $f'(x) = \frac{\partial f}{\partial x}(x)$ gives the rate of change of f in x. That is, $\frac{\partial f}{\partial x}$ tells us how much the function f changes when we make a small change to x. A positive value will tell that we will go uphill with a small positive step in the x-direction and go downhill with a negative derivative. Hence we can reach a minimum by taking small steps proportional to the negative derivative until we reach a point where he derivative is close to 0. Taking steps in the direction of the positive gradient will take us towards a local maximum of the function. In this case the procedure is called *gradient ascent*. In terms of this course the function that we wish to minimize is the loss function $\mathcal{L}(\mathbf{w})$ given the model parameters $\mathbf{w}$.

Gradient points in the direction of most change Intuitively, it seems reasonable that if the function f is sufficiently smooth (differentiable) then if each of the directions in the gradient is pointing in the direction of most change then the combination of all of them will also be pointing in the direction of most change. In other words, the gradient corresponds to the rate of steepest ascent/descent. An instructional and highly recommended proof can be found by clicking on this link.**The really important message about the gradients is that they point in the direction of the largest change.**

19.1.1 The steps of gradient descent

This section describes one type of method, gradient descent, used for iterative minimization of non-linear functions such as error functions.

When performing gradient descent on multivariate functions $f(\mathbf{x}) = f([x_1, \ldots, x_N])$ the direction of greatest increase is no longer just a "forward" or "backward" step along the x axis (as in the univariate case) as shown in **??** . Extending it to multivariate functions is just a small step (pun intended).

Given an initial guess x_0, the gradient descent method iteratively estimates a new estimate x_{i+1} by using the current estimate x_i and update it with a small step in the negative gradient direction e.g.

$$x_{i+1} = x_i - \nu * \nabla f(x_i) \tag{19.3}$$

where ν is a scaling parameter which, in machine learning, is often called the *learning rate*. Gradient descent typically stops when the gradient is close to zero

or when the algorithm has made a predefined number of iterations.

The learning rate ν is important as it controls the step size induced by the gradient. As shown in Figure 19.2, the learning rate may be set too large or too small. A small learning rate may imply more iterations are needed before gradient decent reaches the local maximum (if at all). A learning rate may also be set too large which may result in gradient decent never converges at the local minimum.

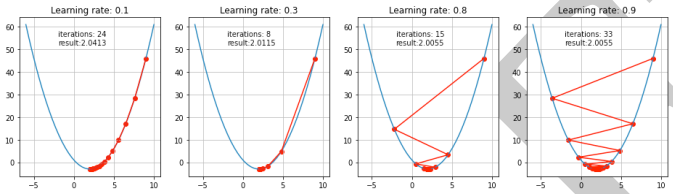

Figure 19.2: Influence of the learning rate on convergence

In summary, the steps of (plain vanilla) gradient descent are:

1. Choose a starting point x and a learning rate ν
2. Update the estimate according to Equation 19.3 until gradient is small enough or for a maximum number of iteration.

For the function

$$f(X) = f(x_1, x_2) = 0.5x_1^2 + \frac{5}{2}x_2^2 - x_1 * x_2 - 2(x_1 + x_2)$$

The gradient is

$$\nabla f(X) = \nabla f(x_1, x_2) = \begin{bmatrix} -2 + x_1 - x_2 \\ -2 - x_1 + 5 * x_2 \end{bmatrix}$$

and the gradient decent steps with the initial value for $x_0 = [2, 1]$ are shown in the figure below

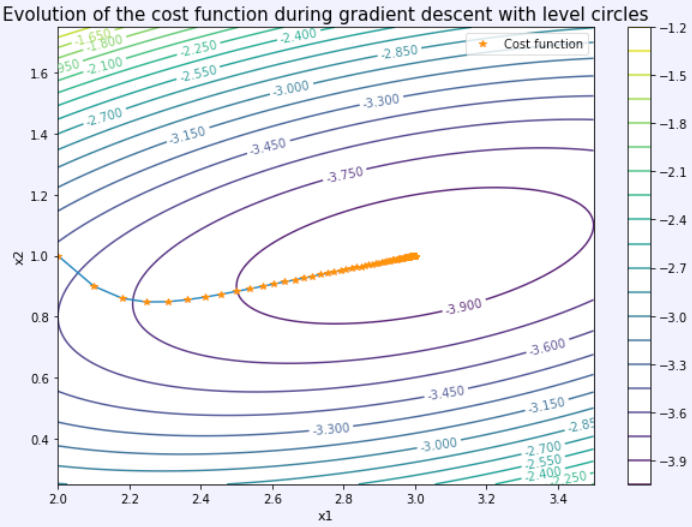

Figure 19.3: Gradient decent on $f(x_1, x_2) = 0.5x_1^2 + \frac{5}{2}x_2^2 - x_1 * x_2 - 2(x_1 + x_2)$

Figure 19.3 shows the isocontours (curves with the same function value) of the objective function f overlayed with each step of the optimization. It is clear that the optimization progresses nicely towards the minimum. The main reasons for this is that gradient descent requires that the objective function ideally should be

- Differentiable
- Convex

Convexity Figure 19.4 shows examples of differentiable univariate functions that are either convex and non-convex. A convex set is a set of points such that, given any two points a, b in that set, the line joining a and b lies entirely within that set. Intuitively, this means that the set is connected (so that you can pass between any two points without leaving the set) and has no dents in its perimeter. In terms of optimization Figure 19.4 (right) shows that if the set is non-convex then gradient descent may end up in local maxima (e.g. start at $x_0 = 15$). A mathematical way to check if a univariate function is convex is to calculate the second derivative and check if its value is always larger than 0.

$$\frac{\partial^2 f(x))}{\partial x^2} > 0$$

It is relatively easy to see that the function $f(x) = 0.5x^2 + y^2$ is convex.

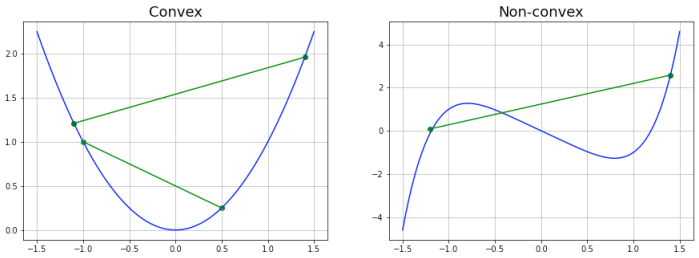

Figure 19.4: The function on the left is convex while the one on the right in not.

The gradient, $\nabla f(x,y) = \begin{bmatrix} x \\ 2y \end{bmatrix}$ shows that the function is changing twice as much in the y direction as in the x direction. The second order partial derivatives with respect to x ($\frac{\partial^2 f}{\partial^2 x}$), y ($\frac{\partial^2 f}{\partial^2 y}$) and x and y, ($\frac{\partial^2 f}{\partial x \partial y}$) are all constants and large than 0 and hence the function is convex in every point.

Placing the second order derivatives in a matrix is called the Hessian. The Hessian of a function in two variables is a 2×2 matrix:

$$H_f(X) \begin{bmatrix} \frac{\partial^2 f(x,y)}{\partial^2 x} & \frac{\partial^2 f(x,y)}{\partial x \partial y} \\ \frac{\partial^2 f(x,y)}{\partial y \partial x} & \frac{\partial^2 f(x,y)}{\partial y \partial y} \end{bmatrix}$$

The Hessian of a function in N variables is given by

$$H_f(X) \begin{bmatrix} \frac{\partial^2 f(X)}{\partial^2 x_1} & \frac{\partial^2 f(X)}{\partial x \partial x_2} & \cdots & \frac{\partial^2 f(X)}{\partial x_1 \partial x_N} \\ \frac{\partial^2 f(X)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(X)}{\partial^2 x_2} & \cdots & \frac{\partial^2 f(X)}{\partial x_1 \partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(X)}{\partial x_N \partial x_1} & \frac{\partial^2 f(X)}{\partial x_N \partial x_2} & \cdots & \frac{\partial^2 f(X)}{\partial^2 x_N} \end{bmatrix}$$

It turns out that if you differentiate a function f with respect to x and the differentiate it with respect to y is the same as first differentiating f with respect to y and then differentiating it with respect to x e.g. $\frac{\partial f}{\partial x_i \partial x_j} = \frac{\partial f}{\partial x_j \partial x_i}$. It therefore becomes evident that the hessian matrix is symmetric.

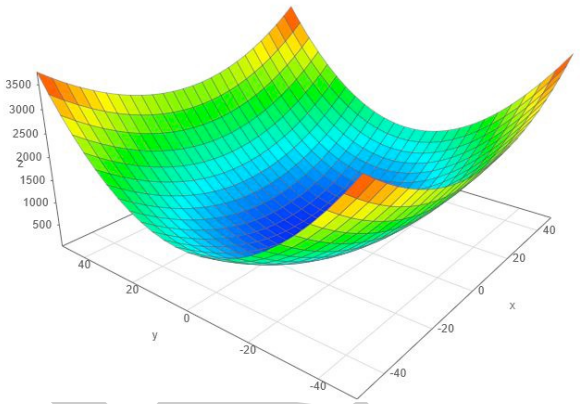

Figure 19.5:

20

Computation Graph

Modern machine learning, particularly deep neural networks, heavily relies on gradient-based optimization techniques to enhance their performance. While calculating gradients analytically for simple models like linear regression is relatively straightforward, complex models demand more sophisticated methods.

20.1.0 Computation Graph Representation

A computational graph is defined as a directed graph where the nodes correspond to mathematical operations. Computational graphs are a way of expressing and evaluating a mathematical expression. In a computation graph, edges represent data or parameters involved in the function, while nodes represent functions applied to incoming edges. In a computational graph nodes are either input values or functions for combining values. Edges receive their weights as the data flows through the graph. Outbound edges from an input node are weighted with that input value; outbound nodes from a function node are weighted by combining the weights of the inbound edges using the specified function. For example, the function $p = x + y$ can be represented as a computational graph as shown in Figure 20.1. The graph has an addition node (indicated by the "+" sign) with two input variables, x and y, and one output, p.

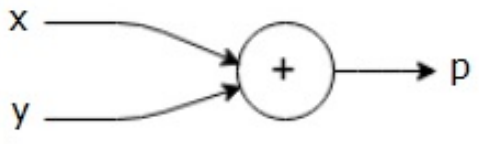

Figure 20.1:

A somewhat more complicated multivariate function

$$f(x,y,z) = (x+y)*z$$

is a composition of two functions g and h as illustrated in Figure 20.3

$$f(x,y,z) = h(g(x,y),z)$$
$$g(i,j) = i+j$$
$$h(p,q) = p*q$$

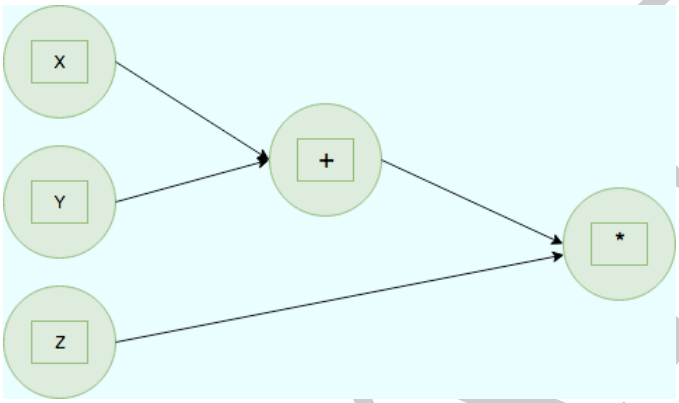

Figure 20.2:

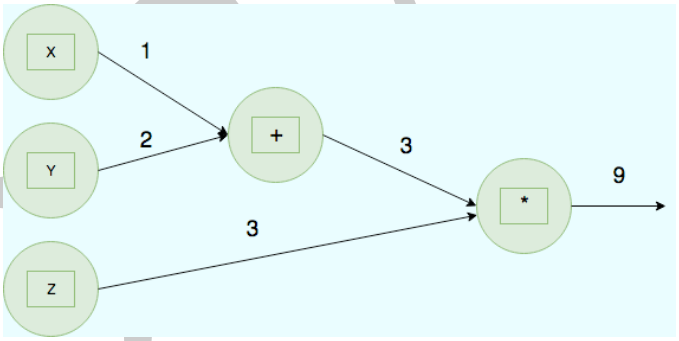

Figure 20.3:

20.1.1 Forward Pass

The process of evaluating the value of the expression represented by the computational graph is called *the forward pass*. It involves passing values from input variables on the left to the output on the right.

By performing a forward pass, we can evaluate the function in the given values. In both notations we can compute the answers to the function, provided

we do so in the correct order. Before knowing the answer to $f(x,y,z)$ first we need the answer to $g(x,y)x + y$= and then $h(g(x,y),z)$. In the notation of the equations these dependencies are given by order of precedence. Similarly in computational graphs we start at the variable nodes and pass the values forward in the graph. We have to wait until all the edges pointing into a node have been assigned a value before computing the output value for that node. Let's look at the example for computing $f(1,2,3)$.

$$
\begin{aligned}
f(1,2,3) &= h(g(1,2),3) \\
g(1,2) &= 1+2=3 \\
f(1,2,3) &= h(3,3) \\
h(3,3) &= 3*3=9 \\
f(1,2,3) &= 9
\end{aligned}
$$

Sigmoid Functions The sigmoid function, $f(z) = \frac{1}{1+e^{-z}}$, where $z = \mathbf{x}^\top \mathbf{w}$ as illustrated in Figure 20.4

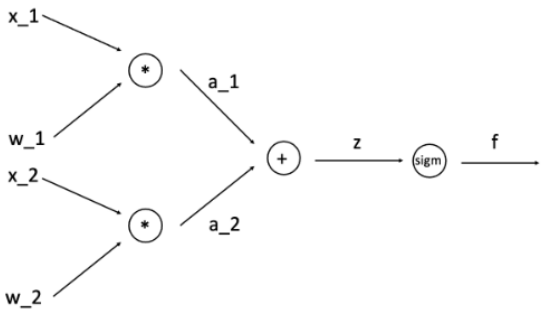

Figure 20.4:

MAKE A COMPLEX FUNCTION AND REPRESENT AS A GRAPH

20.2.0 Derivatives in Computation Graph - backwards pass

The graphical representation of functions provides a convenient representation when computing (partial) derivatives of functions with respect to its inputs.

Recap - Chain Rule The chain rule is essential when computing derivatives and gradients of composed functions. Given single-variable functions, $f : \mathbb{R} \rightarrow$

$\mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ such that $f(g(x))$. The chain rule states that their derivative with respect to x can be expressed as:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}.$$

WRONG!

Illustrate it as a graph

$$\frac{\partial g}{\partial g} = 1$$

This derivative is represented by the identity operation in the computational graph.

Next, we proceed with the backward pass through the multiplication operation. We need to calculate the gradients at nodes corresponding to p and z. Since $g = p \cdot z$, we can determine the following derivatives:

$$\frac{\partial g}{\partial z} = p \quad \text{and} \quad \frac{\partial g}{\partial p} = z$$

For multivariate functions, where $g : \mathbb{R}^n \to \mathbb{R}^m$ and $f : \mathbb{R}^m \to \mathbb{R}$, the chain rule is given by

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial g_j} \cdot \frac{\partial g_j}{\partial x_i} \quad \text{for } i = 1, 2, \ldots, n.$$

Let $q = x + y$ and $f = qz$. Then $\frac{\partial f}{\partial z} = q$. Notice that the derivative of f with respect to z depends only on x and y through the intermediate value q.

Consider a function that performs a weighted sum followed by an activation function, such as the sigmoid function. The computation graph for this layer would involve nodes representing these operations and edges carrying the data (input, weights, and output). By computing gradients at each node, we can efficiently propagate gradients backward through the network.

In the backward pass, our goal is to compute the gradients for each input variable with respect to the final output. These gradients are essential for training neural networks using gradient descent.

For example, we aim to calculate the following gradients: $\frac{\partial x}{\partial g}$, $\frac{\partial y}{\partial g}$, and $\frac{\partial z}{\partial g}$.

The backward pass starts by finding the derivative of the final output with respect to itself, which is always equal to one:

Given the values obtained from the forward pass, i.e., $p = 4$ and $z = -3$, we can compute these derivatives as follows:

$$\frac{\partial g}{\partial z} = 4 \quad \text{and} \quad \frac{\partial g}{\partial p} = -3$$

Now, we want to calculate the gradients at nodes corresponding to x and y, denoted as $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial y}$. To do this efficiently, we can use the chain rule of differentiation. Applying the chain rule, we get:

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial p} \cdot \frac{\partial p}{\partial x}$$

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} \cdot \frac{\partial p}{\partial y}$$

Since $p = x + y$, we can easily compute $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$ as:

$$\frac{\partial p}{\partial x} = 1 \quad \text{and} \quad \frac{\partial p}{\partial y} = 1$$

Therefore, we have:

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial p} \cdot 1 = (-3) \cdot 1 = -3$$

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} \cdot 1 = (-3) \cdot 1 = -3$$

The backward pass efficiently calculates the gradients at x and y using local information, even when these nodes are not directly connected to g. This process is fundamental for training neural networks.

Figure 20.5: Backward Pass (Backpropagation)

The ability is crucial for computing derivatives of earlier inputs using the chain rule.This becomes more powerful when dealing with complex expressions.