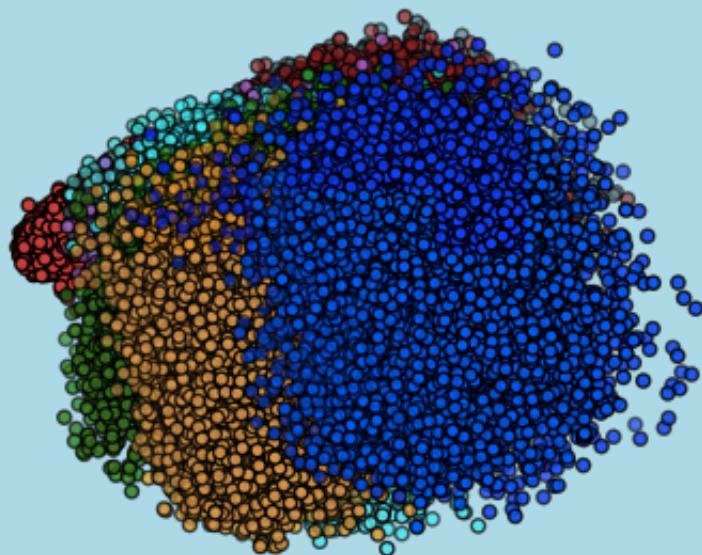


# Math for Data Science

Omar Hijab<sup>\*</sup>



Copyright ©2022 — 2024 Omar Hijab. All Rights Reserved.

# Preface

This text is under construction and is continuously updated. Upon completion of the text, the preface will be enlarged and exercises will be added.

This text is a presentation of the mathematics underlying Data Science. At first, the text assumes minimal math background, and basic math is reviewed. After this, deeper results are presented.

Important principles or results are displayed in these boxes.

The culmination of the text is Chapter 8, where neural networks and machine learning are introduced. Much of the mathematics developed in prior chapters is used here.

The ideas presented are made concrete by interpreting them in **Python** code. The standard Python data science libraries are used, and a Python index lists the Python functions used in the text.

Python code is displayed in these boxes.

Detailed proofs and detailed code snippets are included throughout the text for the same reason: There is value in understanding how things work, and real understanding can only be achieved by going all the way.

Because SQL is usually part of a data scientist's toolkit, an introduction to using SQL from within Python, is included in an appendix.

Throughout, we use *iff* to mean *if and only if*. To help navigate the text, in each section, we use the ship's wheel  to indicate a break, a new idea, or a change in direction.

Sections and figures are numbered sequentially within each chapter, and equations are numbered sequentially within each section, so §3.3 is the third

section in the third chapter, Figure 7.11 is the eleventh figure in the seventh chapter, and (3.2.1) is the first equation in the second section of the third chapter.

If a section contains the alert

★ under construction ★,

then it is incomplete. If a section does not contain this alert, then it is complete except for minor edits. Out of 55 sections, fewer than 5 are incomplete.

# Contents

<b>Preface</b>	<b>iii</b>
<b>List of Figures</b>	<b>xiii</b>
<b>1 Datasets</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 The MNIST Dataset . . . . .	4
1.3 Averages and Vector Spaces . . . . .	9
1.4 Two Dimensions . . . . .	17
1.5 Complex Numbers . . . . .	34
1.6 Mean and Covariance . . . . .	43
1.7 High Dimensions . . . . .	59
<b>2 Linear Geometry</b>	<b>65</b>
2.1 Vectors and Matrices . . . . .	65
2.2 Products . . . . .	73
2.3 Matrix Inverse . . . . .	83
2.4 Span and Linear Independence . . . . .	90
2.5 Zero Variance Directions . . . . .	104
2.6 Pseudo-Inverse . . . . .	109
2.7 Projections . . . . .	118
2.8 Basis . . . . .	125
2.9 Rank . . . . .	133
<b>3 Principal Components</b>	<b>139</b>
3.1 Geometry of Matrices . . . . .	139
3.2 Eigenvalue Decomposition . . . . .	142
3.3 Singular Value Decomposition . . . . .	169

3.4 Principal Component Analysis . . . . .	178
3.5 Cluster Analysis . . . . .	188
<b>4 Counting</b>	<b>193</b>
4.1 Permutations and Combinations . . . . .	193
4.2 Graphs . . . . .	198
4.3 Binomial Theorem . . . . .	212
4.4 Exponential Function . . . . .	219
<b>5 Probability</b>	<b>229</b>
5.1 Binomial Probability . . . . .	229
5.2 Probability . . . . .	240
5.3 Random Variables . . . . .	246
5.4 Normal Distribution . . . . .	263
5.5 Chi-squared Distribution . . . . .	273
<b>6 Statistics</b>	<b>285</b>
6.1 Estimation . . . . .	285
6.2 $Z$ -test . . . . .	290
6.3 $T$ -test . . . . .	303
6.4 Two Means . . . . .	310
6.5 Variances . . . . .	314
6.6 Maximum Likelihood Estimates . . . . .	318
6.7 Chi-Squared Tests . . . . .	319
<b>7 Calculus</b>	<b>325</b>
7.1 Calculus . . . . .	325
7.2 Entropy and Information . . . . .	343
7.3 Multi-variable Calculus . . . . .	351
7.4 Back Propagation . . . . .	357
7.5 Convex Functions . . . . .	369
7.6 Multinomial Probability . . . . .	387
<b>8 Machine Learning</b>	<b>399</b>
8.1 Overview . . . . .	399
8.2 Neural Networks . . . . .	401
8.3 Gradient Descent . . . . .	416
8.4 Network Training . . . . .	425

8.5 Shallow Learning . . . . .	428
8.6 Regression Examples . . . . .	440
8.7 Strict Convexity . . . . .	452
8.8 Accelerated Gradient Descent . . . . .	456
8.9 Stochastic Gradient Descent . . . . .	463
<b>A Appendices</b>	<b>465</b>
A.1 SQL . . . . .	465
A.2 Minimizing Sequences . . . . .	479
A.3 Keras Training . . . . .	486
<b>References</b>	<b>487</b>
<b>Python</b>	<b>489</b>
<b>Index</b>	<b>493</b>



# List of Figures

1.1 Iris dataset [23]. . . . .	2
1.2 Images in the MNIST dataset. . . . .	3
1.3 A portion of the MNIST dataset. . . . .	5
1.4 Original and projections: $n = 784, 600, 350, 150, 50, 10, 1$ . . . . .	6
1.5 The MNIST dataset (3d projection). . . . .	6
1.6 A crude copy of the image. . . . .	8
1.7 HTML colors. . . . .	10
1.8 The vector $v$ joining the points $m$ and $x$ . . . . .	12
1.9 Datasets of points versus datasets of vectors. . . . .	13
1.10 A statistic $f$ valued in a vector space $V$ . . . . .	14
1.11 A dataset with its mean. . . . .	16
1.12 A vector $v$ . . . . .	17
1.13 Vectors $v_1$ and $v_2$ and their shadows in the plane. . . . .	17
1.14 Adding $v_1$ and $v_2$ . . . . .	18
1.15 Scaling with $t = 2$ and $t = -2/3$ . . . . .	20
1.16 The polar representation of $v = (x, y)$ . . . . .	21
1.17 $v$ and its antipode $-v$ . . . . .	22
1.18 Two vectors $v_1$ and $v_2$ . . . . .	23
1.19 Pythagoras for general triangles. . . . .	25
1.20 Proof of Pythagoras for general triangles. . . . .	25
1.21 $v$ and $v^\perp$ . . . . .	26
1.22 Multiplying and dividing points on the unit circle. . . . .	35
1.23 Complex numbers . . . . .	38
1.24 The second, third, and fourth roots of unity . . . . .	41
1.25 The fifth, sixth, and fifteenth roots of unity . . . . .	42
1.26 MSD for the mean (green) versus MSD for a random point (red). . . . .	44
1.27 Projecting a vector $v$ onto the line through $u$ . . . . .	49
1.28 Covariance ellipses and inverse covariance ellipses. . . . .	52

1.29 Covariance ellipse and incovariance ellipse.	55
1.30 A positively correlated dataset $\rho > 0$ .	56
1.31 A negatively correlated dataset $\rho < 0$ .	56
1.32 Level contours of $v \cdot Q^{-1}v$ .	57
1.33 Ellipsoid and axes in 3d.	58
1.34 Disks inside the square	60
1.35 Balls inside the cube	61
2.1 The points 0, $x$ , $Ax$ , and $b$ .	109
2.2 The points $x$ , $Ax$ , the points $x^*$ , $Ax^*$ , and the point $x^+$ .	110
2.3 Projecting onto a line.	118
2.4 Projecting onto a plane, $Pb = ru + sv$ .	119
2.5 Dataset, reduced dataset, and projected dataset, $n < d$ .	123
2.6 Relations between vector classes.	126
2.7 First defect for MNIST.	129
2.8 The dimension staircase with defects.	129
2.9 The dimension staircase for the MNIST dataset.	130
2.10 A $5 \times 3$ matrix $A$ is a linear transformation from $\mathbf{R}^3$ to $\mathbf{R}^5$ .	133
3.1 Image of unit circle with $\sigma_1 = 1.5$ and $\sigma_2 = .75$ .	140
3.2 SVD decomposition $A = USV$ .	142
3.3 Relations between matrix classes.	143
3.4 Inverse covariance ellipse and centered dataset.	153
3.5 $S = \text{span}(v_1)$ and $T = S^\perp$ .	157
3.6 Three springs at rest and perturbed.	161
3.7 Six springs at rest and perturbed.	162
3.8 Two springs along a circle leading to $Q(2)$ .	163
3.9 Five springs along a circle leading to $Q(5)$ .	164
3.10 Plot of eigenvalues of $Q(50)$ .	168
3.11 Density of eigenvalues of $Q(d)$ for $d$ large.	168
3.12 MNIST eigenvalues as a percentage of the total variance.	180
3.13 MNIST eigenvalue percentage plot.	181
3.14 Original and projections: $n = 784, 600, 350, 150, 50, 10, 1$ .	185
3.15 The full MNIST dataset (2d projection).	186
3.16 The Iris dataset (2d projection).	187
4.1 $6 = 3!$ permutations of 3 balls.	193
4.2 Directed and undirected graphs.	198

4.3	A weighed directed graph.	198
4.4	A double edge and a loop.	199
4.5	The complete graph $K_6$ and the cycle graph $C_6$ .	200
4.6	The triangle $K_3 = C_3$ .	201
4.7	Non-isomorphic graphs with degree sequence $(3, 2, 2, 1, 1, 1)$ .	209
4.8	Complete bipartite graph $K_{5,3}$ .	210
4.9	Pascal's triangle.	215
4.10	The exponential function $\exp x$ .	225
4.11	Convexity of the exponential function.	228
5.1	The distribution of $p$ given 7 heads in 10 tosses.	237
5.2	The logistic function.	238
5.3	The logistic function takes real numbers to probabilities.	239
5.4	100,000 sessions, with 5, 15, 50, and 500 tosses per session.	243
5.5	When we sample $X$ , we get $x$ .	246
5.6	$N = 150$ petal lengths and their mean.	247
5.7	Histogram of $N = 150$ petal lengths.	249
5.8	Means of 100,000 batches, of size $n = 1, 5, 15, 50$ .	250
5.9	Distribution of a bernoulli random variable.	251
5.10	Confidence that $X$ lies in interval $[a, b]$ .	252
5.11	Cumulative distribution functions.	252
5.12	Cdf of a bernoulli distribution.	253
5.13	Binary variance.	255
5.14	When we sample $X_1, X_2, \dots, X_n$ , we get $x_1, x_2, \dots, x_n$ .	260
5.15	The standard normal distribution.	264
5.16	$z = \text{norm.ppf}(p)$ and $p = \text{norm.cdf}(z)$ .	266
5.17	Confidence (green) or significance (red) (lower-tail, two-tail, upper-tail).	266
5.18	68%, 95%, 99% confidence cutoffs for standard normal.	268
5.19	Cutoffs, confidence levels, $p$ -values.	268
5.21	68%, 95%, 99% cutoffs for non-standard normal.	269
5.20	$p$ -values at 5% and at 1%.	269
5.22	$(X, Y)$ in the square and in the circle.	274
5.23	Chi-squared distribution with different degrees.	275
5.24	With degree $d \geq 2$ , the chi-squared distribution peaks at $d - 2$ .	277
6.1	Statistics flowchart: $p$ -value $p$ and significance $\alpha$ .	286

6.2	Histogram of sampling $n = 25$ students, repeated $N = 1000$ times.	291
6.3	The error matrix.	300
6.4	$t$ -distribution, against normal (dashed).	304
6.5	Fisher $F$ -distribution.	317
6.6	Contingency table [25].	322
7.1	$f'(a)$ is the slope of the tangent line at $a$ .	325
7.2	Composition of two functions.	327
7.3	Angle $\theta$ in the plane, $P = (x, y)$ .	330
7.4	Increasing or decreasing?	332
7.5	Increasing or decreasing?	333
7.6	Tangent parabolas $p_m(x)$ (green), $p_L(x)$ (red), $L > m > 0$ .	336
7.7	The logarithm function $\log x$ .	338
7.8	The absolute entropy function $H(p)$ .	345
7.9	Asymptotics of binomial coefficients.	347
7.10	The relative information $I(p, q)$ with $q = .7$ .	349
7.11	Composition of multiple functions.	354
7.12	Composition of three functions in a chain.	358
7.13	A network composition [27].	362
7.14	The function $g = \max(y, z)$ .	363
7.15	Forward and backward propagation [27].	364
7.16	Level sets and sublevel sets in two dimensions.	370
7.17	Contour lines in two dimensions.	370
7.18	Line segment $[x_0, x_1]$ .	371
7.19	Convex: The line segment lies above the graph.	372
7.20	Convex hull of $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ .	374
7.21	A convex hull with one facet highlighted.	375
7.22	Convex set in three dimensions with supporting hyperplane.	376
7.23	Hyperplanes in two and three dimensions.	377
7.24	Separating hyperplane theorem.	377
7.25	The third row is the sum of the first and second rows, and the $H$ column is the negative of the $I$ column.	397
8.1	A perceptron with activation function $f$ .	403
8.2	Perceptrons in parallel [18].	404
8.3	Network of neurons.	406
8.4	Incoming and Outgoing signals.	407

8.5	Forward and back propagation between two neurons.	407
8.6	Downstream, local, and upstream derivatives at node $i$ .	410
8.7	A shallow dense layer.	415
8.8	Layered neural network [9].	415
8.9	Double well newton descent.	419
8.10	Double well cost function and sublevel sets at $w_0$ and at $w_1$ .	421
8.11	Double well gradient descent.	424
8.12	Cost trajectory and number of iterations as learning rate varies.	428
8.13	Linear regression neural network.	431
8.14	Logistic regression neural network.	433
8.15	Population versus employed: Linear Regression.	440
8.16	Longley Economic Data [15].	441
8.17	Polynomial regression: Degrees 2, 4, 6, 8, 10, 12.	444
8.18	Hours studied and outcomes.	446
8.19	Exam dataset: $x$ .	446
8.20	Exam dataset: $(x, p)$ [29].	446
8.21	Exam dataset: $(x, x_0)$ .	447
8.22	Hours studied and one-hot encoded outcomes.	447
8.23	Neural network for student exam outcomes.	448
8.24	Equivalent neural network for student exam outcomes.	448
8.25	Exam dataset: $(x, x_0, p)$ .	449
8.26	Convex hulls of Iris classes in $\mathbf{R}^2$ .	450
8.27	Convex hulls of MNIST classes in $\mathbf{R}^2$ .	451
A.1	Dataframe from list-of-dicts.	469
A.2	Menu dataframe and SQL table.	469
A.3	Rawa restaurant.	472
A.4	OrdersIn dataframe and SQL table.	473
A.5	OrdersOut dataframe and SQL table.	475



# Chapter 1

## Datasets

In this chapter we explore examples of datasets and some simple Python code. We also review the geometry of vectors in the plane and properties of  $2 \times 2$  matrices, introduce the mean and covariance of a dataset, then present a first taste of what higher dimensions might look like.

### 1.1 Introduction

Geometrically, a *dataset* is a sample of  $N$  points  $x_1, x_2, \dots, x_N$  in  $d$ -dimensional space  $\mathbf{R}^d$ . Algebraically, a dataset is an  $N \times d$  matrix.

Practically speaking, as we shall see, the following are all representations of datasets

$$\text{matrix} = \text{CSV file} = \text{spreadsheet} = \text{SQL table} = \text{array} = \text{dataframe}$$

Each point  $x = (t_1, t_2, \dots, t_d)$  in the dataset is a *sample* or an *example*, and the components  $t_1, t_2, \dots, t_d$  of a sample point  $x$  are its *features* or *attributes*. As such,  $d$ -dimensional space  $\mathbf{R}^d$  is *feature space*.

Sometimes one of the features is separated out as the label. In this case, the dataset is a *labelled* dataset.



The *Iris dataset* contains 150 examples of four features of Iris flowers, and there are three classes of Irises, *Setosa*, *Versicolor*, and *Virginica*, with 50 samples from each class.

The four features are sepal length and width, and petal length and width (Figure 1.1). For each example, the class is the label corresponding to that example, so the Iris dataset is labelled.

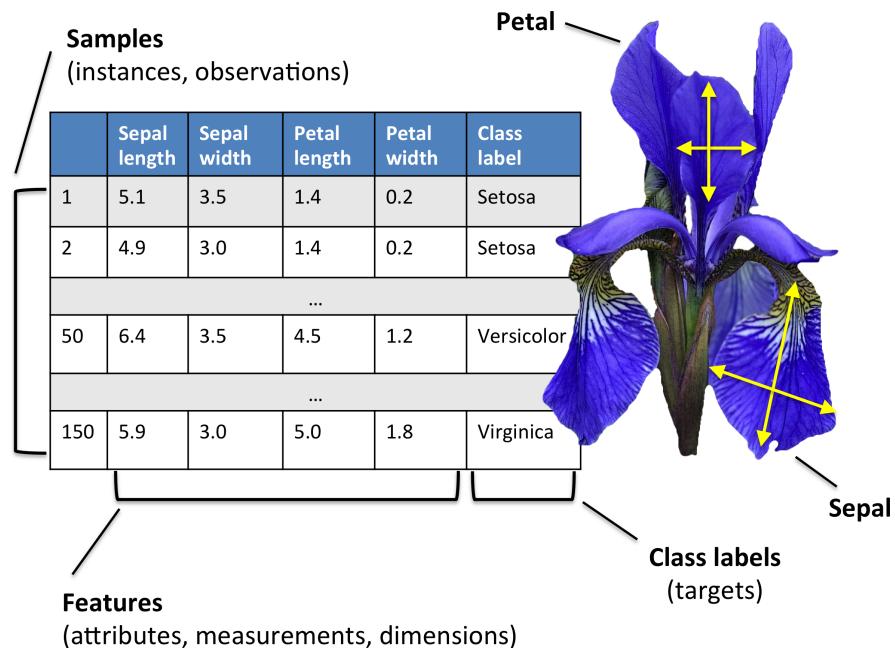


Figure 1.1: Iris dataset [23].

The Iris dataset is downloaded using the code

```
from sklearn import datasets

iris = datasets.load_iris()
dataset = iris["data"]
labels = iris["target"]

dataset, labels
```

If `sklearn` is not installed, you'll need to first run

```
pip install sklearn
```

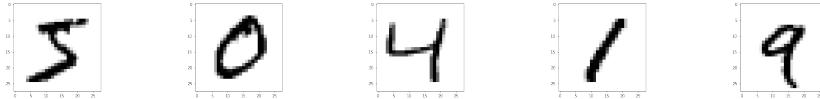


Figure 1.2: Images in the MNIST dataset.



The *MNIST dataset* consists of images of hand-written digits (Figure 1.2). There are 10 classes of images, corresponding to each digit  $0, 1, \dots, 9$ . We seek to compress the images while preserving as much as possible of the images' characteristics.

Each image is a grayscale  $28 \times 28$  pixel image. Since  $28^2 = 784$ , each image is a point in  $d = 784$  dimensions. Here there are  $N = 60000$  samples and  $d = 784$  features.



This subsection is included just to give a flavor. *All unfamiliar words are explained in detail in Chapter 2.* If preferred, just skip to the next subsection.

Suppose we have a dataset of  $N$  points

$$x_1, x_2, \dots, x_N$$

in  $d$ -dimensional feature space. We seek to find a lower-dimensional feature space  $U \subset \mathbf{R}^d$  so that the projections of these points onto  $U$  retain as much information as possible about the data.

In other words, we are looking for an  $n$ -dimensional subspace  $U$  for some  $n < d$ . Among all  $n$ -dimensional subspaces, which one should we pick? The answer is to select  $U$  among all  $n$ -dimensional subspaces *to maximize variability in the data*.

Another issue is the choice of  $n$ , which is an integer satisfying  $0 \leq n \leq d$ . On the one hand, we want  $n$  to be as small as possible, to maximize data compression. On the other hand, we want  $n$  to be big enough to capture most of the features of the data. At one extreme, if we pick  $n = d$ , then we have no compression and complete information. At the other extreme, if we pick  $n = 0$ , then we have full compression and no information.

Projecting the data from  $\mathbf{R}^d$  to a lower-dimensional space  $U$  is *dimensional reduction*. The best alignment, the best-fit, or the best choice of  $U$  is *principal component analysis*. These issues will be taken up in §3.4.



If this is your first exposure to data science, there will be a learning curve, because here there are three kinds of thinking: Data science (Datasets, PCA, descent, networks), math (linear algebra, probability, statistics, calculus), and Python (`numpy`, `pandas`, `scipy`, `sympy`, `matplotlib`). It may help to read the [code examples](#), and the [important math principles](#) first, then dive into details as needed.

To illustrate and make concrete concepts as they are introduced, we use *Python* code throughout. We run Python code in a Jupyter notebook.

*Jupyter* is an IDE, an integrated development environment. Jupyter supports many frameworks, including Python, Sage, Julia, and R. A useful Jupyter feature is the ability to measure the amount of execution time of a code cell by including at the start of the cell

```
%%time
```

The installation procedure is to first [install Python](#), then [install Jupyter](#) using [Python pip](#). Additional frameworks (R, ...) are then installed separately. After this, to make each additional installed framework available from [within Jupyter](#), run `jupyter kernelspec`. Detailed steps depend on the reader's laptop setup.

## 1.2 The MNIST Dataset

The MNIST<sup>1</sup> dataset consists of 70,000 images, split into 60,000 training images and 10,000 testing images. The following code

```
from keras.datasets import mnist
```

---

<sup>1</sup>The National Institute of Standards and Technology (NIST) is a physical sciences laboratory and non-regulatory agency of the United States Department of Commerce.

```
train, test = mnist.load_data()  
  
dataset, labels = train  
  
dataset.shape, labels.shape
```

returns

```
((60000, 28, 28), (60000,))
```

(This code requires `keras`, `tensorflow` and related modules if not already installed.)



Figure 1.3: A portion of the MNIST dataset.

Since this dataset is for demonstration purposes, these images are coarse. Since each image consists of 784 pixels, and each pixel shading is a number, each image is a point  $x$  in  $\mathbf{R}^d = \mathbf{R}^{784}$ .

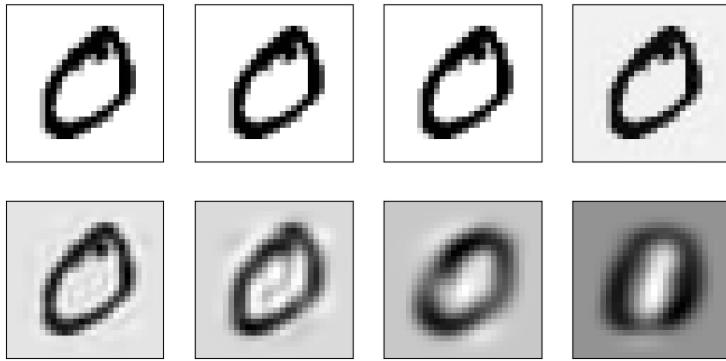


Figure 1.4: Original and projections:  $n = 784, 600, 350, 150, 50, 10, 1$ .

To *compress* the image means to reduce the number of dimensions in the point  $x$  while keeping maximum information. We can think of a single image as a dataset itself, and compress the image, or we can design a compression algorithm based on a collection of images. It is then reasonable to expect that the procedure applies well to any image that is similar to the images in the collection.

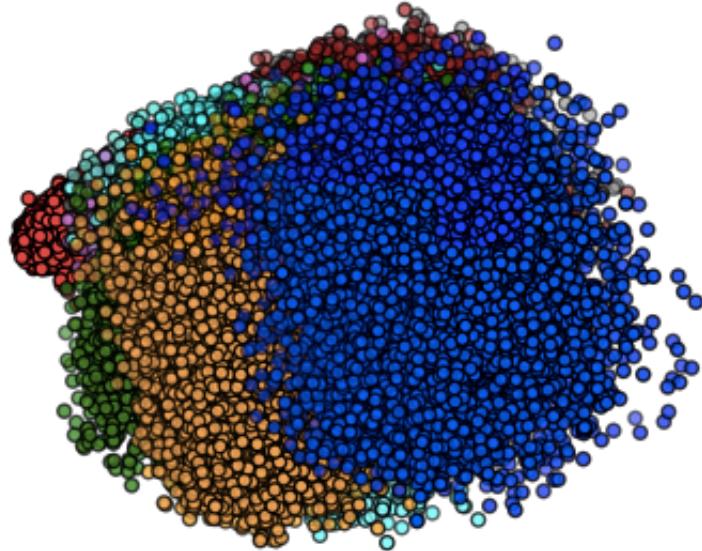


Figure 1.5: The MNIST dataset (3d projection).

For the second image in Figure 1.2, reducing dimension from  $d = 784$  to

$n$  equal 600, 350, 150, 50, 10, and 1, we have the images in Figure 1.4.

Compressing each image to a point in  $n = 3$  dimensions and plotting all  $N = 60000$  points yields Figure 1.5. All this is discussed in §3.4.



The top left image in Figure 1.4 is given by a 784-dimensional point which is imported as an array `pixels` of shape  $(28, 28)$ .

```
pixels = dataset[1]
```

### Live exercise in class

1. Take out your laptops and open Jupyter.
2. In Jupyter, return a two-dimensional plot of the point  $(2, 3)$  using the code

```
from matplotlib.pyplot import *
grid()
scatter(2,3)
show()
```

3. Do `for` loops over  $i$  and  $j$  in `range(28)` and use `scatter` to plot points at location  $(i, j)$  with size given by `pixels[i, j]`, then `show`.

Here is one possible code, returning Figure 1.6.

```
from matplotlib.pyplot import *
from numpy import *
pixels = dataset[1]
grid()
for i in range(28):
    for j in range(28): scatter(i,j, s = pixels[i,j])
```

```
show()
```

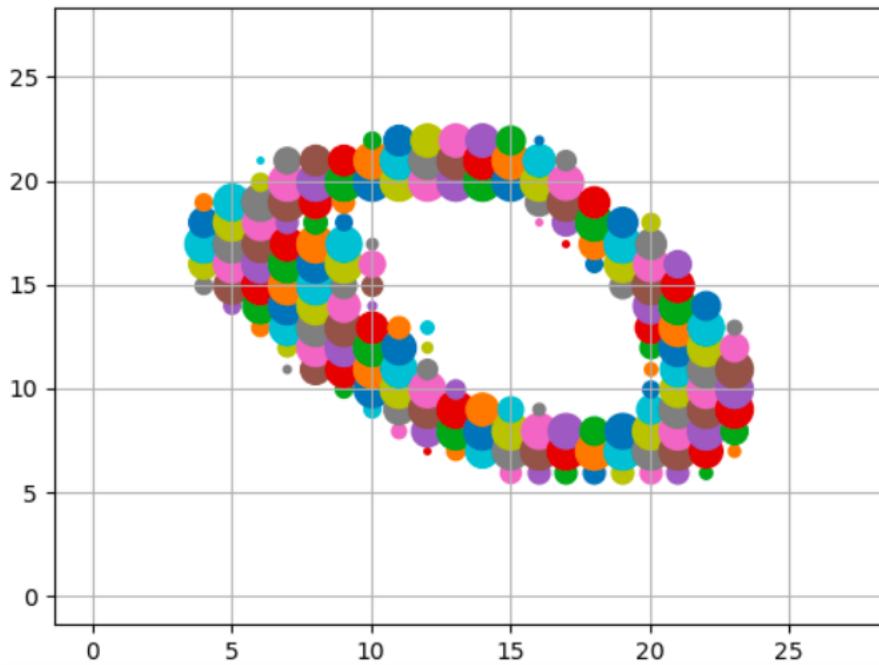


Figure 1.6: A crude copy of the image.

The top left image in Figure 1.4 is returned by the code

```
from matplotlib.pyplot import *
imshow(pixels, cmap="gray_r")
```



We end the section by discussing the Python `import` command. The last code snippet can be rewritten

```
import matplotlib.pyplot as plt  
  
plt.imshow(pixels, cmap="gray_r")
```

or as

```
from matplotlib.pyplot import imshow  
  
imshow(pixels, cmap="gray_r")
```

So we have three versions of this code snippet.

In the second version, it is explicit that `imshow` is imported from the submodule `pyplot` of the module `matplotlib`. Moreoever, the submodule `matplotlib.pyplot` is referenced by a short nickname `plt`.

In the first version `import from *`, many commands, maybe not all, are imported from the submodule `matplotlib.pyplot`.

In the third version, only the command `imshow` is imported. Which import style is used depends on the situation.

In this text, we usually use the first style, as it is visually lightest. To help with online searches, in the Python index, Python commands are listed under their full module path.

## 1.3 Averages and Vector Spaces

Suppose we have a population of things (people, tables, numbers, vectors, images, etc.) and we have a sample of size  $N$  from this population:

```
L = [x_1,x_2,...,x_N].
```

The total population is the *population* or the *sample space*. For example, the sample space consists of all real numbers and we take  $N = 5$  samples from this population

```
L_1 = [3.95, 3.20, 3.10, 5.55, 6.93].
```

Or, the sample space consists of all integers and we take  $N = 5$  samples from this population

```
L_2 = [35, -32, -8, 45, -8].
```

Or, the sample space consists of all rational numbers and we take  $N = 5$  samples from this population

```
L_3 = [13/31, 8/9, 7/8, 41/22, 32/27].
```

Or, the sample space consists of all Python strings and we take  $N = 5$  samples from this population

```
L_4 = ['a2e?', '#%T', '7y5', 'kkk>><</', '[]*+']
```

Or, the sample space consists of all HTML colors and we take  $N = 5$  samples from this population



Figure 1.7: HTML colors.

Here's the code generating the colors

```
# HTML color codes are #rrggbb (6 hexes)
from matplotlib.pyplot import *
from random import choice

def hexcolor():
    return "#" + ''.join([choice('0123456789abcdef') for _ in
                           range(6)])

for i in range(5): scatter(i, 0, c=hexcolor())
show()
```



Let  $L$  be a list as above. The goal is to compute the sample *average* or *mean* of the list, which is

$$\text{mean} = \text{average} = \frac{x_1 + x_2 + \cdots + x_N}{N}.$$

In the first example, for real numbers, the average is

$$\frac{3.95 + 3.20 + 3.10 + 5.55 + 6.93}{5} = 4.546.$$

In the second case, for integers, the average is  $32/5$ . In the third case, the average is  $385373/73656$ . In the fourth case, while we can add strings, we can't divide them by 5, so the average is undefined. Similarly for colors: the average is undefined.

This leads to an important definition. A sample space or population  $V$  is called a *vector space* if, roughly speaking, one can compute means or averages in  $V$ . In this case, we call the members of the population “vectors”, even though the members may be anything, as long as they satisfy the basic rules of a vector space.

In a vector space  $V$ , the rules are:

1. vectors can be added (and the sum  $v + w$  is back in  $V$ )
2. vector addition is commutative  $v + w = w + v$
3. vector addition is associative  $u + (v + w) = (u + v) + w$
4. there is a zero vector  $0$
5. vectors  $v$  have negatives  $-v$
6. vectors can be multiplied by real numbers (and the product  $rv$  is back in  $V$ )
7. multiplication is distributive over addition  $(r + s)v = rv + sv$  and  $r(u + v) = ru + rv$
8.  $1v = v$  and  $0v = 0$
9.  $r(sv) = (rs)v$ .



Let  $x_1, x_2, \dots, x_N$  be a dataset. Is the dataset a collection of points, or is the dataset a collection of vectors? In other words, what geometric picture of datasets should we have in our heads? Here's how it works.

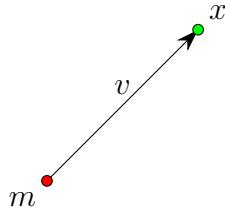


Figure 1.8: The vector  $v$  joining the points  $m$  and  $x$ .

A *vector* is an arrow joining two points (Figure 1.8). Given two points  $m = (a, b)$  and  $x = (c, d)$ , the vector joining them is

$$v = x - m = (c - a, d - b).$$

Then  $m$  is the *tail* of  $v$ , and  $x$  is the *head* of  $v$ . For example, the vector joining  $m = (1, 2)$  to  $x = (3, 4)$  is  $v = (2, 2)$ .

Given a point  $x$ , we would like to associate to it a vector  $v$  in a uniform manner. However, this cannot be done without a second point, a *reference point*. Given a dataset of *points*  $x_1, x_2, \dots, x_N$ , the most convenient choice for the reference point is the mean  $m$  of the dataset. This results in a dataset of *vectors*  $v_1, v_2, \dots, v_N$ , where  $v_k = x_k - m$ ,  $k = 1, 2, \dots, N$ .

The dataset  $v_1, v_2, \dots, v_N$  is *centered*, its mean is zero,

$$\frac{v_1 + v_2 + \cdots + v_N}{N} = 0.$$

So datasets can be points  $x_1, x_2, \dots, x_N$  with mean  $m$ , or vectors  $v_1, v_2, \dots, v_N$  with mean zero (Figure 1.9). This distinction makes a difference when measuring the dimension of a dataset (§2.8).

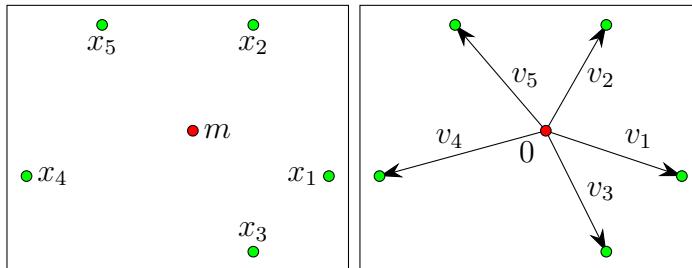


Figure 1.9: Datasets of points versus datasets of vectors.

### Centered Versus Non-Centered

If  $x_1, x_2, \dots, x_N$  is a dataset of points with mean  $m$  and

$$v_1 = x_1 - m, v_2 = x_2 - m, \dots, v_N = x_N - m,$$

then  $v_1, v_2, \dots, v_N$  is a centered dataset of vectors.



Let us go back to vector spaces. When we work with vector spaces, numbers are referred to as *scalars*, because  $2v, 3v, -v, \dots$  are scaled versions of  $v$ . When we multiply a vector  $v$  by a scalar  $r$  to get the scaled vector  $rv$ , we call it *scalar multiplication*. This is to distinguish this multiplication from the inner and outer products we see below.

For example, the samples in the list  $L_1$  form a vector space, the set of all real numbers  $\mathbf{R}$ . Even though one can add integers, the set  $\mathbf{Z}$  of all integers does not form a vector space because multiplying an integer by  $1/2$  does not result in an integer. The set  $\mathbf{Q}$  of all rational numbers (fractions) is a vector space, so  $L_3$  is a sampling from a vector space. The set of strings is not a vector space because even though one can add strings, addition is not commutative:

```
'alpha' + 'romeo' == 'romeo' + 'alpha'
```

returns `False`.

Usually, we can't take sample means from a population, we instead take the sample mean of a *statistic* associated to the population. A statistic is

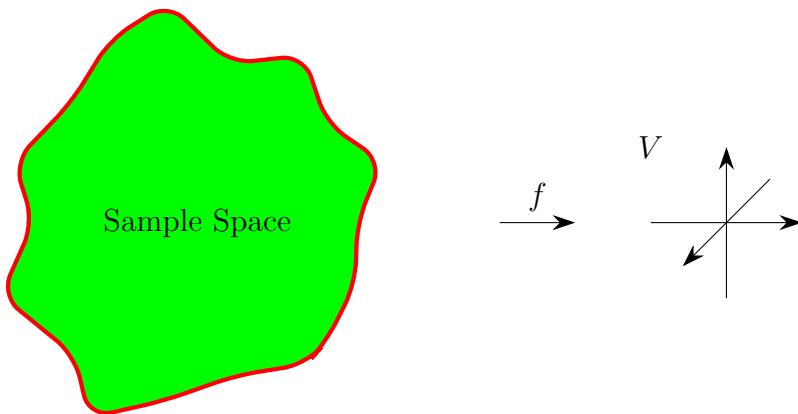


Figure 1.10: A statistic  $f$  valued in a vector space  $V$ .

an assignment of a number  $f(item)$  to each item in the population. For example, the human population on Earth is not a vector space (they can't be added), but their heights is a vector space (heights can be added). For the list  $L_4$ , a statistic might be the length of the string. For the HTML colors, a statistic is the HTML code of the color.

In general, a statistic need not be a number. A statistic can be anything that "behaves like a number". For example, we shall see below that  $f(item)$  can be a vector or a matrix. More generally, a statistic's values may be anything that lives in a vector space  $V$ , which we defined above.

For example, for the scalar dataset

$$x_1 = 1.23, x_2 = 4.29, x_3 = -3.3, x_4 = 555,$$

the average is

$$m = \frac{1.23 + 4.29 - 3.3 + 555}{4} = 139.305.$$

In Python, averages are computed using `numpy.mean`. For a scalar dataset, the code

```
from numpy import *

dataset = array([1.23, 4.29, -3.3, 555])
mean(dataset)
```

returns the average.

For the two-dimensional dataset

$$x_1 = (1, 2), x_2 = (3, 4), x_3 = (-2, 11), x_4 = (0, 66),$$

the average is

$$m = \frac{(1, 2) + (3, 4) + (-2, 11) + (0, 66)}{4} = (0.5, 20.75).$$

Note the  $x$ -components are summed, and the  $y$ -components are summed, leading to a two-dimensional mean. (This is *vector addition*, taken up in §1.4.)

In Python, a dataset in  $\mathbf{R}^2$  may be assembled as  $4 \times 2$  array

```
from numpy import *

dataset = array([[1,2], [3,4], [-2,11], [0,66]])
```

Then the code

```
mean(dataset, axis=0)
```

returns the mean  $(0.5, 20.75)$ .

To explain what `axis=0` does, we use matrix terminology. After arranging `dataset` into an array of four rows and two columns, to compute the mean, we sum over the row index.

This means summing the entries of the first column, then summing the entries of the second column, resulting in a mean with two components.

In Python, the default is to consider the row index  $i$  as index zero, and to consider the column index  $j$  as index one.

Summing over `index=1` is equivalent to thinking of the dataset as two points in  $\mathbf{R}^4$ , so

```
mean(dataset, axis=1)
```

returns  $(1.5, 3.5, 4.5, 33)$ .

Here is a more involved example of a dataset of random points and their mean:

```

from numpy import *
from numpy.random import random
from matplotlib.pyplot import scatter, grid, show

N = 20
dataset = array([ [random(), random()] for _ in range(N) ])
mean = mean(dataset, axis=0)

grid()
X = dataset[:,0]
Y = dataset[:,1]
scatter(X, Y)
scatter(*mean)
show()

```

This returns Figure 1.11.

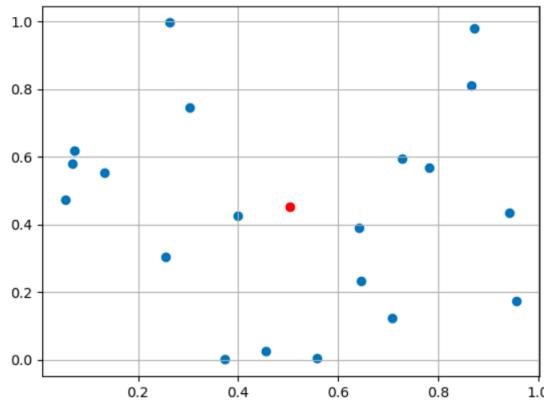


Figure 1.11: A dataset with its mean.

In this code, `scatter` expects two positional arguments, the  $x$  and the  $y$  components of a point, or two lists of  $x$  and  $y$  components separately. The *unpacking operator* `*` unpacks `mean` from one pair into its separate  $x$  and  $y$  components `*mean`. Also, for `scatter`, `dataset` is separated into its two columns.

## 1.4 Two Dimensions

We start with the geometry of vectors in two dimensions. This is the *cartesian plane*  $\mathbf{R}^2$ , also called 2-dimensional real space. The plane  $\mathbf{R}^2$  is a vector space, in the sense described in the previous section.

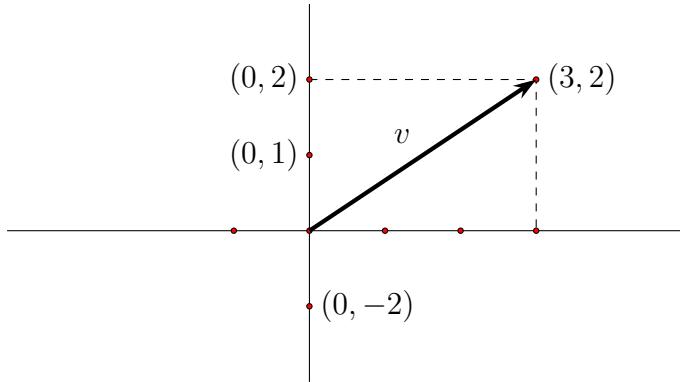


Figure 1.12: A vector  $v$ .

In the cartesian plane, a *vector* is an arrow  $v$  joining the origin to a point (Figure 1.12). In this way, points and vectors are almost interchangeable, as a point  $x$  in  $\mathbf{R}^d$  corresponds to the vector  $v$  starting at the origin 0 and ending at  $x$ .

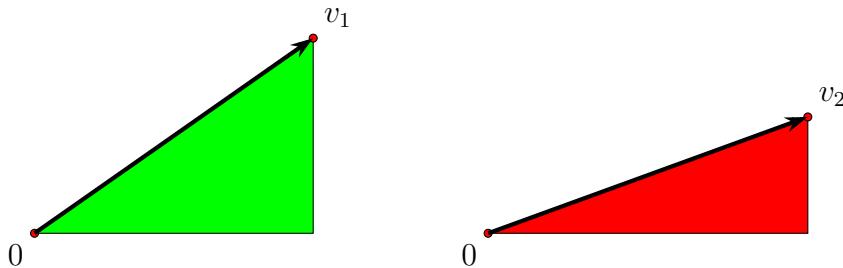


Figure 1.13: Vectors  $v_1$  and  $v_2$  and their shadows in the plane.

In the cartesian plane, each vector  $v$  has a *shadow*. This is the triangle constructed by dropping the perpendicular from the tip of  $v$  to the  $x$ -axis, as in Figure 1.13. This cannot be done unless one first draws a horizontal line (the  $x$ -axis), then a vertical line (the  $y$ -axis). In this manner, each vector  $v$

has *cartesian* coordinates  $v = (x, y)$ . In Figure 1.12, the coordinates of  $v$  are  $(3, 2)$ . In particular, the vector  $0 = (0, 0)$ , the *zero vector*, corresponds to the origin.

In the cartesian plane, vectors  $v_1 = (x_1, y_1)$  and  $v_2 = (x_2, y_2)$  are added by adding their coordinates,

### Addition of vectors

If  $v_1 = (x_1, y_1)$  and  $v_2 = (x_2, y_2)$ , then

$$v_1 + v_2 = (x_1 + x_2, y_1 + y_2). \quad (1.4.1)$$

Because points and vectors are interchangeable, the same formula is used for addition  $P + P'$  of points  $P$  and  $P'$ .

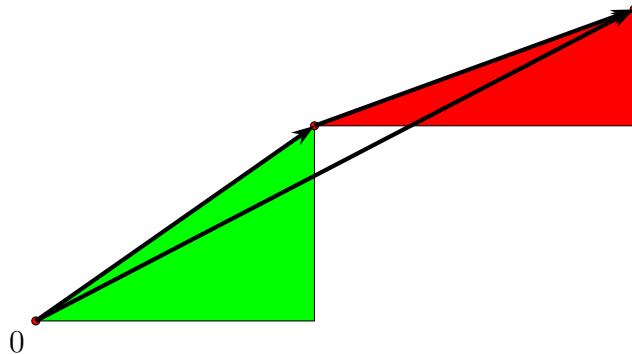


Figure 1.14: Adding  $v_1$  and  $v_2$

This is the same as combining their shadows as in Figure 1.14. In Python, lists and tuples do not add this way. Lists and tuples have to first be converted into `numpy arrays`.

```

v1 = (1,2)
v2 = (3,4)
v1 + v2 == (1+3,2+4) # returns False

v1 = [1,2]
v2 = [3,4]
v1 + v2 == [1+3,2+4] # returns False

```

```
from numpy import *
v1 = array([1,2])
v2 = array([3,4])
v1 + v2 == array([1+3,2+4]) # returns True
```

For example,  $v_1 = (-3, 1)$  and  $v_2 = (2, -2)$  returns

$$v_1 + v_2 = (-3, 1) + (2, -2) = (-3 + 2, 1 - 2) = (-1, -1).$$

A vector  $v = (x, y)$  in the plane may be scaled by scaling the shadow as in Figure 1.15. This is *vector scaling* by  $t$ . Note when  $t$  is negative, the shadow is also flipped. In Python, we write

```
from numpy import *
v = array([1,2])
3*v == array([3,6]) # returns True
```

Given a vector  $v$ , the scalings  $tv$  of  $v$  form a line passing through the origin 0 (Figure 1.17). This line is the *span* of  $v$  (see §2.4). Scalings  $tv$  of  $v$  are also called multiples of  $v$ .

### Scaling of vectors

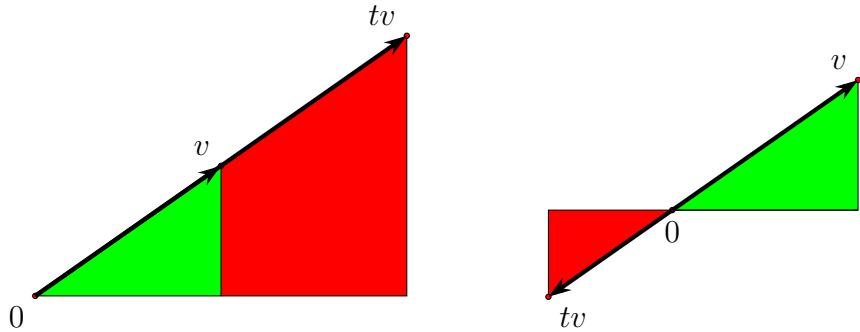
If  $v = (x, y)$ , then

$$tv = (tx, ty).$$

If  $t$  and  $s$  are real numbers, it is easy to check

$$t(v_1 + v_2) = tv_1 + tv_2 \quad \text{and} \quad t(sv) = (ts)v.$$

Thus multiplying  $v$  by  $s$ , and then multiplying the result by  $t$ , has the same effect as multiplying  $v$  by  $ts$ , in a single step. Because points and vectors are interchangeable, the same formula is used for scaling  $tP$  points  $P$  by  $t$ .

Figure 1.15: Scaling with  $t = 2$  and  $t = -2/3$ 

We set  $-v = (-1)v$ , and define subtraction of vectors by

$$v_1 - v_2 = v_1 + (-v_2).$$

This gives

### Subtraction of vectors

If  $v_1 = (x_1, y_1)$  and  $v_2 = (x_2, y_2)$ , then

$$v_1 - v_2 = (x_1 - x_2, y_1 - y_2) \quad (1.4.2)$$

```
from numpy import *
v1 = array([1,2])
v2 = array([3,4])
v1 - v2 == array([-2,-3]) # returns True
```



### Distance Formula

If  $v_1 = (x_1, y_1)$  and  $v_2 = (x_2, y_2)$ , then the *distance* between  $v_1$  and  $v_2$  is

$$|v_1 - v_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

The distance of  $v = (x, y)$  to the origin  $0 = (0, 0)$  is its *magnitude* or *norm* or *length*

$$r = |v| = |v - 0| = \sqrt{x^2 + y^2}.$$

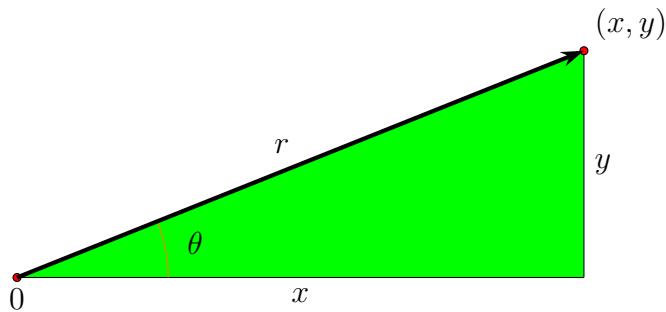


Figure 1.16: The polar representation of  $v = (x, y)$ .

In terms of  $r$  and  $\theta$  (Figure 1.16), the *polar* representation of  $(x, y)$  is

$$x = r \cos \theta, \quad y = r \sin \theta.$$

In Python,

```
from numpy import *
from numpy.linalg import norm

v = array([1,2])
norm(v) == sqrt(5)# returns True
```



The *unit circle* consists of the vectors which are distance 1 from the origin 0. When  $v$  is on the unit circle, the magnitude of  $v$  is 1, and we say  $v$  is a *unit vector*. In this case, the line formed by the scalings of  $v$  intersects the unit circle at  $\pm v$  (Figure 1.17).

When  $v$  is a unit vector,  $r = 1$ , and (Figure 1.16),

$$v = (x, y) = (\cos \theta, \sin \theta). \tag{1.4.3}$$

The unit circle intersects the horizontal axis at the vectors  $(1, 0)$ , and  $(-1, 0)$ , and intersects the vertical axis at the vectors  $(0, 1)$ , and  $(0, -1)$ . These four vectors are equally spaced on the unit circle (Figure 1.17).

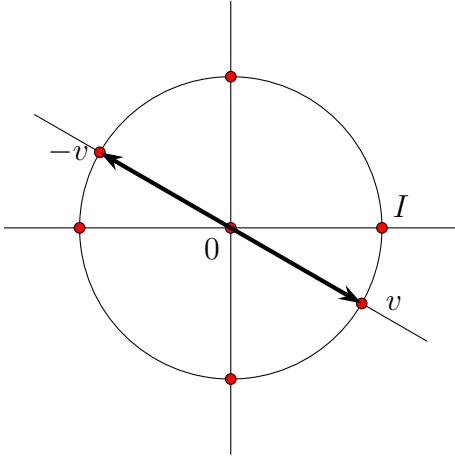


Figure 1.17:  $v$  and its antipode  $-v$

By the distance formula, a vector  $v = (x, y)$  is a unit vector when

$$x^2 + y^2 = 1.$$

More generally, any circle with *center*  $(a, b)$  and *radius*  $r$  consists of vectors  $v = (x, y)$  satisfying

$$(x - a)^2 + (y - b)^2 = r^2.$$

Let  $R$  be a point on the unit circle, and let  $t > 0$ . From this, we see the scaled point  $tR$  is on the circle with center  $(0, 0)$  and radius  $t$ . Moreover, if  $Q$  is any point,  $Q + tR$  is on the circle with center  $Q$  and radius  $r$ .

Given this, it is easy to check

$$|tv| = |t| |v|$$

for any real number  $t$  and vector  $v$ .

From this, if a vector  $v$  is unit and  $r > 0$ , then  $rv$  has magnitude  $r$ . If  $v$  is any vector not equal to the zero vector, then  $r = |v|$  is positive, and

$$\left| \frac{1}{r}v \right| = \frac{1}{r}|v| = \frac{1}{r}r = 1,$$

so  $v/r$  is a unit vector.

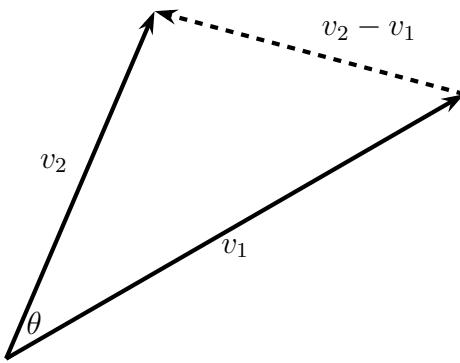


Figure 1.18: Two vectors  $v_1$  and  $v_2$ .



Now we discuss the dot product in two dimensions. We have two vectors  $v$  and  $v'$  in the plane  $\mathbf{R}^2$ , with  $v_1 = (x_1, y_1)$  and  $v_2 = (x_2, y_2)$ . The *dot product* of  $v_1$  and  $v_2$  is given algebraically as

$$v_1 \cdot v_2 = x_1 x_2 + y_1 y_2,$$

or geometrically as

$$v_1 \cdot v_2 = |v_1| |v_2| \cos \theta,$$

where  $\theta$  is the angle between  $v_1$  and  $v_2$ . To show that these are the same, below we derive the

### Dot Product Identity

$$x_1 x_2 + y_1 y_2 = v_1 \cdot v_2 = |v_1| |v_2| \cos \theta. \quad (1.4.4)$$

In Python, the dot product is given by `numpy.dot`,

```
from numpy import *
v1 = array([1,2])
v2 = array([3,4])
```

```
dot(v1,v2) == 1*3 + 2*4 # returns True
```

As a consequence of the dot product identity, we have code for the angle between two vectors,

```
from numpy import *

def angle(u,v):
    a = dot(u,v)
    b = dot(u,u)
    c = dot(v,v)
    theta = arccos(a / sqrt(b*c))
    return degrees(theta)
```

Recall that  $-1 \leq \cos \theta \leq 1$ . Using the dot product identity (1.4.4), we obtain the important

### Cauchy-Schwarz Inequality

If  $u$  and  $v$  are any two vectors, then

$$-|u||v| \leq u \cdot v \leq |u||v|. \quad (1.4.5)$$



To derive the dot product identity, we first derive Pythagoras' theorem for general triangles

$$c^2 = a^2 + b^2 - 2ab \cos \theta. \quad (1.4.6)$$

To derive (1.4.6), we drop a perpendicular to the base  $b$ , obtaining two right triangles (Figure 1.20). By Pythagoras applied to each triangle,

$$a^2 = d^2 + f^2 \quad \text{and} \quad c^2 = e^2 + f^2.$$

Also  $b = e + d$ , so

$$b^2 = (e + d)^2 = e^2 + 2ed + d^2.$$

By the definition of  $\cos \theta$ ,  $d = a \cos \theta$ . Putting this all together,

$$\begin{aligned} c^2 &= e^2 + f^2 = (b - d)^2 + f^2 \\ &= f^2 + d^2 + b^2 - 2db \\ &= a^2 + b^2 - 2ab \cos \theta, \end{aligned}$$

so we get (1.4.6).

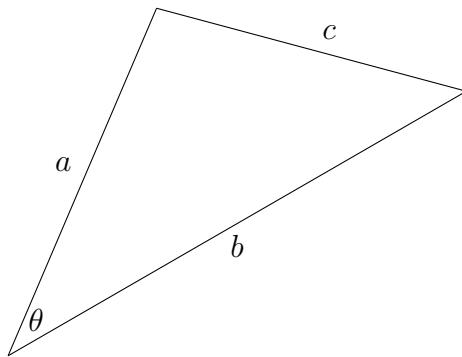


Figure 1.19: Pythagoras for general triangles.

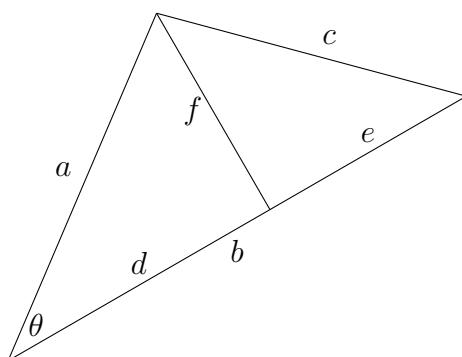


Figure 1.20: Proof of Pythagoras for general triangles.

Next, connect Figures 1.18 and 1.19 by noting  $a = |v_2|$  and  $b = |v_1|$  and  $c = |v_2 - v_1|$ .

Now go back to deriving (1.4.4). By vector addition, we have

$$v_2 - v_1 = (x_2 - x_1, y_2 - y_1),$$

and  $b^2 = |v_1|^2 = x_1^2 + y_1^2$ ,  $a^2 = |v_2|^2 = x_2^2 + y_2^2$ . By the binomial theorem,

$$\begin{aligned} c^2 &= |v_2 - v_1|^2 = |(x_2 - x_1, y_2 - y_1)|^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 \\ &= x_1^2 + y_1^2 - 2(x_1 x_2 + y_1 y_2) + x_2^2 + y_2^2 = a^2 + b^2 - 2(x_1 x_2 + y_1 y_2), \end{aligned}$$

thus

$$c^2 = a^2 + b^2 - 2(x_1 x_2 + y_1 y_2). \quad (1.4.7)$$

Comparing the terms in (1.4.6) and (1.4.7), we arrive at (1.4.4).



Let  $u$  and  $v$  be vectors. A basic property of dot product is

$$|u + v|^2 = |u|^2 + 2u \cdot v + |v|^2. \quad (1.4.8)$$

This is easily derived from the definition of  $u \cdot v$ .

If  $P = (x, y)$ , let  $P^\perp = (-y, x)$ , and let  $v = OP$  and  $v^\perp = OP'$  be the vectors emanating from the origin, and ending at  $P$  and  $P^\perp$ . Then

$$v \cdot v^\perp = (x, y) \cdot (-y, x) = 0.$$

This shows  $v$  and  $v^\perp$  are perpendicular (Figure 1.21).

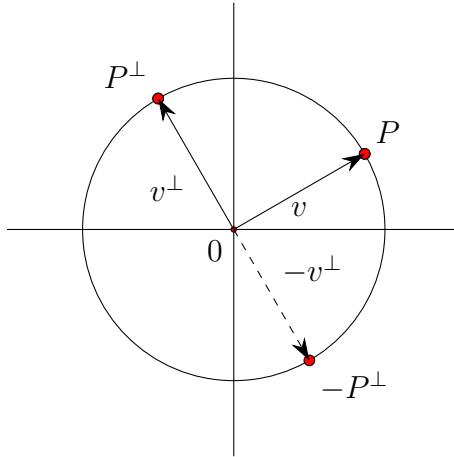


Figure 1.21:  $v$  and  $v^\perp$

From Figure 1.21, we see points  $P$  and  $P'$  on the unit circle satisfy  $P \cdot P' = 0$  iff  $P' = \pm P^\perp$ .



We now solve two linear equations in two unknowns  $x, y$ . We start with the homogeneous case

$$ax + by = 0, \quad cx + dy = 0. \quad (1.4.9)$$

Let  $A$  be the  $2 \times 2$  matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad (1.4.10)$$

It is easy to exhibit a solution of the first equation in (1.4.9): choose  $(x, y) = (-b, a)$ . If we want this to be a solution of the second equation as well, we must have  $cx + dy = ad - bc = 0$ . Based on this, we make the following definition. The *determinant* of  $A$  is

$$\det(A) = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

In (1.4.9), multiply the first equation by  $d$  and the second by  $b$  and subtract, obtaining

$$(ad - bc)x = d(ax + by) - b(cx + dy) = 0.$$

In (1.4.9), multiply the first equation by  $c$  and the second by  $a$  and subtract, obtaining

$$(bc - ad)y = c(ax + by) - a(cx + dy) = 0.$$

From here, we see there are two cases:  $\det(A) = 0$  and  $\det(A) \neq 0$ . When  $\det(A) \neq 0$ , the only solution of (1.4.9) is  $(x, y) = (0, 0)$ . When  $\det(A) = 0$ ,  $(x, y) = (-b, a)$  is a solution of both equations in (1.4.9). We have shown

### Homogeneous System

When  $\det(A) = 0$ , the homogeneous system (1.4.9) has a nonzero solution, and all solutions are scalar multiples of  $(x, y) = (-b, a)$ . When  $\det(A) \neq 0$ , the only solution is  $(x, y) = (0, 0)$ .  $\square$

This covers the homogeneous case. For the inhomogeneous case

$$ax + by = e, \quad cx + dy = f, \quad (1.4.11)$$

multiplying and subtracting as above, we obtain

$$(ad - bc)x = d(ax + by) - b(cx + dy) = de - bf,$$

$$(bc - d)y = c(ax + by) - a(cx + dy) = ce - af.$$

Dividing by  $\det(A)$ , we obtain

### Inhomogeneous System

When  $\det(A) \neq 0$ , the inhomogeneous system (1.4.11) has the unique solution

$$x = \frac{de - bf}{ad - bc}, \quad y = \frac{af - ce}{ad - bc}. \quad (1.4.12)$$

When  $\det(A) = 0$ , (1.4.11) has a solution iff  $ce = af$  and  $de = bf$ .

When  $a^2 + b^2 \neq 0$ , a solution is

$$x = \frac{ae}{a^2 + b^2}, \quad y = \frac{be}{a^2 + b^2}.$$

When  $c^2 + d^2 \neq 0$ , a solution is

$$x = \frac{cf}{c^2 + d^2}, \quad y = \frac{df}{c^2 + d^2}.$$

Any other solution differs from these solutions by a scalar multiple of the homogeneous solution  $(x, y) = (-b, a)$ .  $\square$



We now go over the basic properties of  $2 \times 2$  matrices. This we use in the next section. A  $2 \times 2$  matrix  $A$  is a block of four numbers as in (1.4.10).

The matrix (1.4.10) can be written in terms of the two vectors  $u = (a, b)$  and  $v = (c, d)$ , as follows

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}, \quad u = (a, b), v = (c, d).$$

In this case, we call  $u$  and  $v$  the *rows* of  $A$ . On the other hand,  $A$  may be written as

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} u & v \end{pmatrix}, \quad u = (a, c), v = (b, d).$$

In this case, we call  $u$  and  $v$  the *columns* of  $A$ . This shows there are at least three ways to think about a matrix: as rows, or as columns, or as a single block.

The simplest operations on matrices are addition and scalar multiplication. Addition is as follows,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad A' = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} \quad \Rightarrow \quad A + A' = \begin{pmatrix} a + a' & b + b' \\ c + c' & d + d' \end{pmatrix},$$

and scalar multiplication is as follows,

$$tA = \begin{pmatrix} ta & tb \\ tc & td \end{pmatrix}.$$

The *transpose*  $A^t$  of the matrix  $A$  is

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \Rightarrow \quad A^t = \begin{pmatrix} a & c \\ b & d \end{pmatrix}.$$

Then the rows of  $A^t$  are the columns of  $A$ .

Let  $w = (x, y)$  be a vector. We now explain how to multiply the matrix  $A$  by the vector  $w$ . The result is then another vector  $Aw$ . This is called right multiplication.

To do this, we write  $A$  as rows  $A = \begin{pmatrix} u \\ v \end{pmatrix}$ , then use the dot product,

$$Aw = (u \cdot w, v \cdot w) = (ax + by, cx + dy).$$

Notice  $Aw$  is a vector. When multiplying this way, one often writes

$$Aw = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix},$$

and we call  $w$  and  $Aw$  *column vectors*.

This terminology is introduced to keep things consistent: *It's always row-times-column with row on the left and column on the right*. Nevertheless, a vector, a row vector, and a column vector are all the same thing, just a vector.

Just like we can multiply matrices and vectors, we can also multiply two matrices  $A$  and  $A'$  and obtain a product  $AA'$ . Following the row-column rule

above, we write  $A = \begin{pmatrix} u \\ v \end{pmatrix}$  as rows and  $A' = (u', v')$  as columns to obtain

$$AA' = \begin{pmatrix} u \cdot u' & u \cdot v' \\ u' \cdot v & u' \cdot v' \end{pmatrix}.$$

If we do this the other way, we obtain

$$A'A = \begin{pmatrix} u' \cdot u & u' \cdot v \\ u \cdot v' & u \cdot v \end{pmatrix},$$

so

$$AA' \neq A'A.$$



A *rotation* in the plane is the matrix

$$U = U(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}.$$

Here  $\theta$  is the angle of rotation. By the trigonometric addition formulas (1.5.5),

$$\begin{aligned} U(\theta)U(\theta') &= \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} \cos \theta' & -\sin \theta' \\ \sin \theta' & \cos \theta' \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta + \theta') & -\sin(\theta + \theta') \\ \sin(\theta + \theta') & \cos(\theta + \theta') \end{pmatrix} = U(\theta + \theta'). \end{aligned}$$

This says rotating by  $\theta'$  followed by rotating by  $\theta$  is the same as rotating by  $\theta + \theta'$ .



There is a special matrix  $I$ , the identity matrix,

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

The matrix  $I$  satisfies

$$AI = IA = A$$

for any matrix  $A$ .

Also, for each matrix  $A$  with  $\det(A) \neq 0$ , the matrix

$$A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} = \begin{pmatrix} \frac{d}{ad - bc} & \frac{-b}{ad - bc} \\ \frac{-c}{ad - bc} & \frac{a}{ad - bc} \end{pmatrix}$$

is the *inverse* of  $A$ . The inverse matrix satisfies

$$AA^{-1} = A^{-1}A = I.$$

The inverse reverses the order of the product,

$$(AB)^{-1} = B^{-1}A^{-1}.$$

The transpose also reverses the order of a product,

$$(AB)^t = B^tA^t.$$



Using matrix-vector multiplication, we can rewrite (1.4.11) as

$$Av = w,$$

where

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad v = \begin{pmatrix} x \\ y \end{pmatrix}, \quad w = \begin{pmatrix} e \\ f \end{pmatrix}.$$

Then the solution (1.4.12) can be rewritten

$$v = A^{-1}w,$$

where  $A^{-1}$  is the inverse matrix. We study inverse matrices in depth in §2.3.

The matrix (1.4.10) is *symmetric* if  $b = c$ . A symmetric matrix looks like

$$Q = \begin{pmatrix} a & b \\ b & c \end{pmatrix}.$$

A general matrix  $A$  consists of four numbers  $a, b, c, d$ , and a symmetric matrix  $Q$  consists of three numbers  $a, b, c$ . A matrix  $Q$  is symmetric when

$$Q^t = Q.$$



Let  $A = (u, v)$  be a  $2 \times 2$  matrix with columns  $u, v$ . Then  $u, v$  are the rows of  $A^t = \begin{pmatrix} u \\ v \end{pmatrix}$ . Since matrix multiplication is *row*  $\times$  *column*,

$$A^t A = \begin{pmatrix} u \\ v \end{pmatrix} \begin{pmatrix} u & v \end{pmatrix} = \begin{pmatrix} u \cdot u & u \cdot v \\ v \cdot u & v \cdot v \end{pmatrix}.$$

Now suppose  $A^t A = I$ . Then  $u \cdot u = 1 = v \cdot v$  and  $u \cdot v = 0$ , so  $u$  and  $v$  are orthogonal unit vectors. Such vectors are called *orthonormal*. We have shown

### Orthogonal Matrices

Let  $A$  be a matrix. Then  $A^t A = I$  iff the columns of  $A$  are orthonormal, and  $AA^t = I$  iff the rows of  $A$  are orthonormal.

The second statement follows by applying the first to  $A^t$  instead of  $A$ . A matrix  $U$  is *orthogonal* if

$$U^t U = I = U U^t.$$

Thus a matrix is orthogonal iff its rows are orthonormal, and its columns are orthonormal.



Now we introduce the tensor product. If  $u = (a, b)$  and  $v = (c, d)$  are vectors, their *tensor product* is the matrix

$$u \otimes v = \begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix} = \begin{pmatrix} cu & du \\ bv & bv \end{pmatrix}.$$

Here we wrote  $u \otimes v$  as a single block, and also in terms of rows and columns.

If we do this the other way, we get

$$v \otimes u = \begin{pmatrix} ca & cb \\ da & db \end{pmatrix},$$

so

$$(u \otimes v)^t = v \otimes u.$$

When  $u = v$ ,  $u \otimes v = v \otimes v$  is a symmetric matrix.

Here is code for `tensor`.

```
from numpy import *

def tensor(u,v):
    return array([ [ a*b for b in v] for a in u ])
```

The *trace* of a matrix  $A$  is the sum of the diagonal entries,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \Rightarrow \quad \text{trace}(A) = a + d.$$

The determinant of  $u \otimes v$  is zero,

$$\det(u \otimes v) = 0.$$

This is true no matter what the vectors  $u$  and  $v$  are. Check this yourself.

Notice by definition of  $u \otimes v$ ,

$$\text{trace}(u \otimes v) = u \cdot v, \quad \text{and} \quad \text{trace}(v \otimes v) = |v|^2. \quad (1.4.13)$$

The basic property of tensor product is

### Tensor Product Identity

If  $A = u \otimes v$ , then

$$Aw = (u \otimes v)w = (v \cdot w)u. \quad (1.4.14)$$

This can be checked by writing out both sides in detail.

Now let

$$Q = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$$

be a symmetric matrix and let  $v = (x, y)$ . Then

$$Qv = (ax + by, bx + cy),$$

so

$$v \cdot Qv = (x, y) \cdot (ax + by, bx + cy) = ax^2 + 2bxy + cy^2.$$

This is the *quadratic form* associated to the matrix  $Q$ .

### Quadratic Form

If

$$Q = \begin{pmatrix} a & b \\ b & c \end{pmatrix} \quad \text{and} \quad v = (x, y),$$

then

$$v \cdot Qv = ax^2 + 2bxy + cy^2.$$

When  $Q$  is the identity

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

then the quadratic function is  $x^2 + y^2$ :

$$Q = I \implies v \cdot Qv = x^2 + y^2.$$

When  $Q$  is diagonal,

$$Q = \begin{pmatrix} a & 0 \\ 0 & c \end{pmatrix} \implies v \cdot Qv = ax^2 + cy^2.$$

An important case is when  $Q = u \otimes u$ . In this case, by (1.4.14),

### Quadratic Forms of Tensors

If  $Q = u \otimes u$ , then

$$v \cdot Qv = v \cdot (u \otimes u)v = (u \cdot v)^2. \quad (1.4.15)$$

## 1.5 Complex Numbers

This section is a brief review of complex numbers, for use in later sections. In §1.4, we studied points in two dimensions, and we saw how points can be added and subtracted. In §2.1, we study points in any number of dimensions, and there we also add and subtract points.

In two dimensions, each point has a shadow (Figure 1.13). By stacking shadows, points in the plane can be multiplied and divided (Figure 1.22). In this sense, points in the plane behave like numbers, because they follow the usual rules of arithmetic.

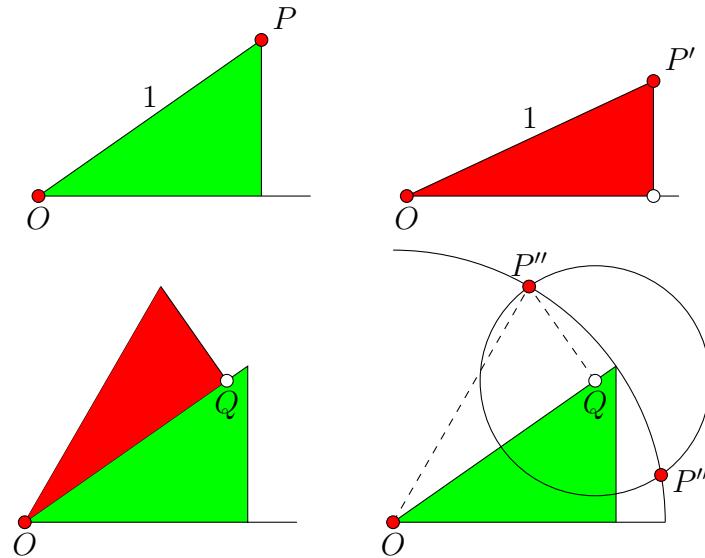


Figure 1.22: Multiplying and dividing points on the unit circle.

This ability of points in the plane to follow the usual rules of arithmetic is unique to one and two dimensions, and not present in any other dimension. When thought of in this manner, points in the plane are called *complex numbers*, and the plane is the *complex plane*.



To define multiplication of points, let  $P = (x, y)$  and  $P' = (x', y')$  be points on the unit circle. Stack the shadow of  $P'$  on top of the shadow of  $P$ , as in Figure 1.22. Here is how one does this without any angle measurement: Mark  $Q = x'P$  at distance  $x'$  along the vector  $OP$  joining  $O$  and  $P$ , and draw the circle with radius  $y'$  and center  $Q$ . Then this circle intersects the unit circle at two points, both called  $P''$ .

We think of the first point  $P''$  as the result of multiplying  $P$  and  $P'$ , and we write  $P'' = PP'$ , and we think of the second point  $P''$  as the result of dividing  $P$  by  $P'$ , and we write  $P'' = P/P'$ . Then we have

### Multiplication and Division of Points

For  $P = (x, y)$  and  $P' = (x', y')$  on the unit circle, when  $x'y' \neq 0$ ,

$$\begin{aligned} P'' &= PP' = (xx' - yy', x'y + xy'), \\ P'' &= P/P' = (xx' + yy', x'y - xy'). \end{aligned} \quad (1.5.1)$$

To derive (1.5.1), let  $P^\perp = (-y, x)$  ("P-perp"). Then

$$\begin{aligned} x'P + y'P^\perp &= (x'x, x'y) + (-y'y, y'x) = (xx' - yy', x'y + xy'), \\ x'P - y'P^\perp &= (x'x, x'y) - (-y'y, y'x) = (xx' + yy', x'y - xy'), \end{aligned}$$

so (1.5.1) is equivalent to

$$P'' = x'P \pm y'P^\perp. \quad (1.5.2)$$

To establish (1.5.2), since  $P''$  is on the circle of center  $Q$  and radius  $y'$ , we may write  $P'' = Q + y'R$ , for some point  $R$  on the unit circle (see §1.4).

Interpreting points as vectors, and using (1.4.8),  $P'' = x'P + y'R$  is on the unit circle iff

$$\begin{aligned} 1 &= |x'P + y'R|^2 = |x'P|^2 + 2x'P \cdot y'R + |y'R|^2 \\ &= x'^2|P|^2 + 2x'y'P \cdot R + y'^2|R|^2 \\ &= x'^2 + y'^2 + 2x'y'P \cdot R \\ &= 1 + 2x'y'P \cdot R. \end{aligned}$$

But this happens iff  $P \cdot R = 0$ , which happens iff  $R = \pm P^\perp$  (Figure 1.21). This establishes (1.5.2).



More generally, if  $r = |P|$  and  $r' = |P'|$ , let  $R$  be any point satisfying  $|R| = r$ . Then

$$P'' = Q + y'R = x'P + y'R$$

satisfies  $|P''| = rr'$  exactly when  $R = \pm P^\perp$ , leading to the two points in (1.5.1).

Let  $\bar{P}$  be the *conjugate*  $(x, -y)$  of  $P = (x, y)$ . The first  $P''$  is the product

$$PP' = (xx' - yy', x'y + xy'), \quad (1.5.3)$$

but the second  $P''$  is not division, it is the *hermitian product*  $P\bar{P}'$  of  $P$  and  $\bar{P}'$ .

The correct formula for division is given by

$$P/P' = \frac{1}{r'^2} P\bar{P}' = \frac{1}{x'^2 + y'^2} (xx' + yy', x'y - xy'). \quad (1.5.4)$$

When  $r' = 1$ , (1.5.4) reduces to the formula in (1.5.1).

With this understood, it is easily checked that division undoes multiplication,

$$(P/P')P' = P.$$

In fact, one can check that multiplication and division as defined by (1.5.3) and (1.5.4) follow the usual rules of arithmetic.



It is natural to identify points on the horizontal axis with real numbers, because, using (1.5.1),  $z = (x, 0)$  and  $z' = (x', 0)$  implies

$$z + z' = (x, 0) + (x', 0) = (x + x', 0), \quad zz' = (xx' - 00, x0 + x'0) = (xx', 0).$$

Because of this, we can write  $z = x$  instead of  $z = (x, 0)$ , this only for points in the plane, and we call the horizontal axis the *real axis*.

Similarly, let  $i = (0, 1)$ . Then the point  $i$  is on the vertical axis, and, using (1.5.1), one can check

$$ix = (0, 1)(x, 0) = (-0, x) = x^\perp.$$

Thus the vertical axis consists of all points of the form  $ix$ . These are called *imaginary numbers*, and the vertical axis is the *imaginary axis*.

Using  $i$ , any point  $P = (x, y)$  may be written

$$P = x + iy,$$

since  $x + iy = (x, 0) + (y, 0)(0, 1) = (x, 0) + (0, y) = (x, y)$ . This leads to Figure 1.23. In this way, real numbers  $x$  are considered complex numbers with zero imaginary part,  $x = x + 0i$ .

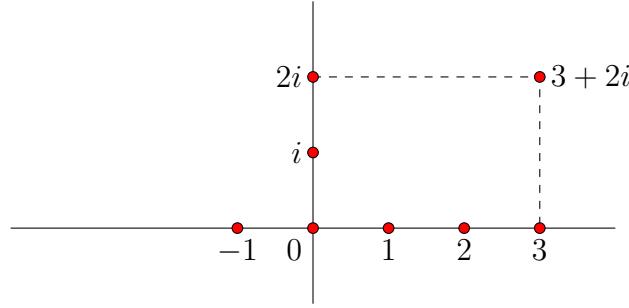


Figure 1.23: Complex numbers

Since by (1.5.1),  $i^2 = (0, 1)^2 = (-1, 0) = -1$ , we have

### Square Root of $-1$

The complex number  $i$  satisfies  $i^2 = -1$ .



When thinking of points in the plane as complex numbers, it is traditional to denote them by  $z$  instead of  $P$ . By (1.5.1), we have

$$z = x + iy, \quad z' = x' + iy' \quad \Rightarrow \quad zz' = (xx' - yy') + i(x'y + xy'),$$

and

$$\frac{z}{z'} = \frac{x + iy}{x' + iy'} = \frac{(xx' + yy') + i(x'y - xy')}{x'^2 + y'^2}.$$

In particular, one can always “move” the  $i$  from the denominator by the formula

$$\frac{1}{z} = \frac{1}{x + iy} = \frac{x - iy}{x^2 + y^2} = \frac{\bar{z}}{|z|^2}.$$

Here  $x^2 + y^2 = r^2 = |z|^2$  is the absolute value squared of  $z$ , and  $\bar{z}$  is the conjugate of  $z$ .



Let  $r, r', r''$  and  $\theta, \theta', \theta''$  be the polar coordinates of  $P, P', P'' = PP'$  (see Figure 1.16). Then Figure 1.22 suggests  $\theta'' = \theta + \theta'$ . Using multiplication

of points (1.5.1) together with his bisection method, Archimedes[11] defined angle measure  $\theta$  numerically and derived  $\theta'' = \theta + \theta'$ .

By elementary algebra,

$$(x^2 + y^2)(x'^2 + y'^2) = (xx' - yy')^2 + (x'y + xy')^2.$$

Since this implies  $r''^2 = r^2 r'^2$ , we conclude

### Polar Coordinates of Complex Numbers

If  $(r, \theta)$  and  $(r', \theta')$  are the polar coordinates of complex numbers  $P$  and  $P'$ , and  $(r'', \theta'')$  are the polar coordinates of the product  $P'' = PP'$ , then

$$r'' = rr' \quad \text{and} \quad \theta'' = \theta + \theta'.$$

From this and (1.5.1), using  $(x, y) = (\cos \theta, \sin \theta)$ ,  $(x', y') = (\cos \theta', \sin \theta')$ , we have the addition formulas

$$\begin{aligned} \sin(\theta + \theta') &= \sin \theta \cos \theta' + \cos \theta \sin \theta', \\ \cos(\theta + \theta') &= \cos \theta \cos \theta' - \sin \theta \sin \theta'. \end{aligned} \tag{1.5.5}$$

For example, if  $\omega = \cos \theta + i \sin \theta$ , then the polar coordinates of  $\omega$  are  $r = 1$  and  $\theta$ . It follows the polar coordinates of  $\omega^2$  are  $r = 1$  and  $2\theta$ , so  $\omega^2 = \cos(2\theta) + i \sin(2\theta)$ .

By the same logic, for any power  $k$ , the polar coordinates of  $\omega^k$  are  $r = 1$  and  $k\theta$ , so  $\omega^k = \cos(k\theta) + i \sin(k\theta)$ .

When  $P = (x, y) = x + iy$  is thought of as a complex number,  $r$  is called the *absolute value* of  $P$ , and written  $r = |P|$ . Then  $r = \sqrt{x^2 + y^2}$  and

$$P = x + iy = r \cos \theta + ir \sin \theta = r(\cos \theta + i \sin \theta).$$

for any complex number  $P$ .



We can reverse the logic in the previous paragraph to compute square roots. Let  $P$  be any point in the plane. We define the *square root* of  $P$  to be a point  $Q$  satisfying  $Q^2 = P$ . In this case, we write  $Q = \sqrt{P}$ . If  $Q$  is a square root, so is  $-Q$ , so there are two square roots  $\pm Q = \pm \sqrt{P}$ .

The formula for  $Q = \sqrt{P}$  is

$$P = (x, y) \implies \sqrt{P} = \pm \left( \frac{r+x}{\sqrt{2r+2x}}, \frac{y}{\sqrt{2r+2x}} \right).$$

This formula is valid as long as  $x \neq -r$ , and can be checked directly by checking  $Q^2 = P$ .

When  $P$  is on the unit circle,  $r = 1$ , so the formula reduces to

$$\sqrt{P} = \pm \left( \frac{1+x}{\sqrt{2+2x}}, \frac{y}{\sqrt{2+2x}} \right).$$



We will need the roots of unity in §3.2. This generalizes square roots, cube roots, etc.

A point  $\omega$  is a *root of unity* if  $\omega^d = 1$  for some power  $d$ . If  $d$  is the power, we say  $\omega$  is a  $d$ -th root of unity.

For example, the square roots of unity are  $\pm 1$ , since  $(\pm 1)^2 = 1$ . Here we have

$$1 = \cos 0 + i \sin 0, \quad -1 = \cos \pi + i \sin \pi.$$

The fourth roots of unity are  $\pm 1, \pm i$ , since  $(\pm 1)^4 = 1, (\pm i)^4 = 1$ . Here we have

$$\begin{aligned} 1 &= \cos 0 + i \sin 0, \\ i &= \cos(\pi/2) + i \sin(\pi/2), \\ -1 &= \cos \pi + i \sin \pi, \\ -i &= \cos(3\pi/2) + i \sin(3\pi/2). \end{aligned}$$

In general, the roots of unity are denoted by powers of  $\omega$ , so the square roots of unity are  $1$  and  $\omega = -1$ , and the fourth roots of unity are  $1, \omega = i, \omega^2 = -1, \omega^3 = -i$ .

Let  $\omega = \cos \theta + i \sin \theta$ . Since  $1 = \cos(2\pi) + i \sin(2\pi)$  and  $\omega^k = \cos(k\theta) + i \sin(k\theta)$ , a  $d$ -th root of unity  $\omega$  satisfies

$$\omega = \cos(2\pi/d) + i \sin(2\pi/d). \tag{1.5.6}$$

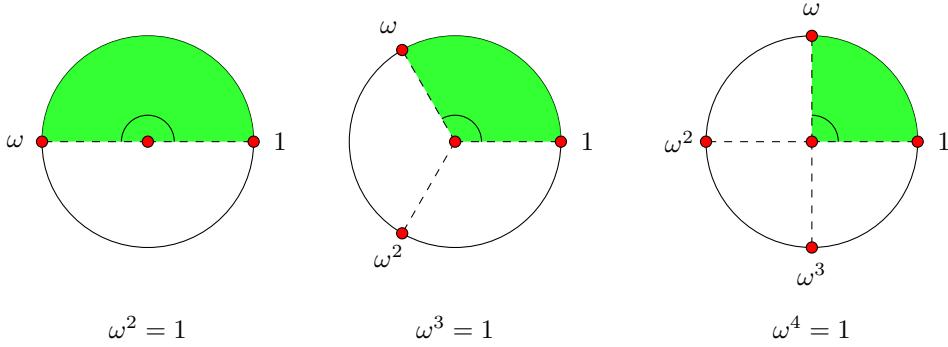


Figure 1.24: The second, third, and fourth roots of unity

If  $\omega^d = 1$ , then

$$(\omega^k)^d = (\omega^d)^k = 1^k = 1.$$

With  $\omega$  given by (1.5.6), this implies

$$1, \omega, \omega^2, \dots, \omega^{d-1}$$

are the  $d$ -th roots of unity.

If we set

$$\omega = -\frac{1}{2} + i\frac{\sqrt{3}}{2} = \cos(2\pi/3) + i\sin(2\pi/3),$$

then a calculation shows

$$1, \quad \omega, \quad \omega^2 = -\frac{1}{2} - i\frac{\sqrt{3}}{2}$$

are the cube roots of unity,

$$1^3 = 1, \quad \omega^3 = 1, \quad (\omega^2)^3 = 1.$$

Similarly, the fifth roots of unity are  $1, \omega, \omega^2, \omega^3, \omega^4$ , where

$$\omega = -\frac{1}{4} + \frac{\sqrt{5}}{4} + i\sqrt{\frac{\sqrt{5}}{8} + \frac{5}{8}} = \cos(2\pi/5) + i\sin(2\pi/5).$$

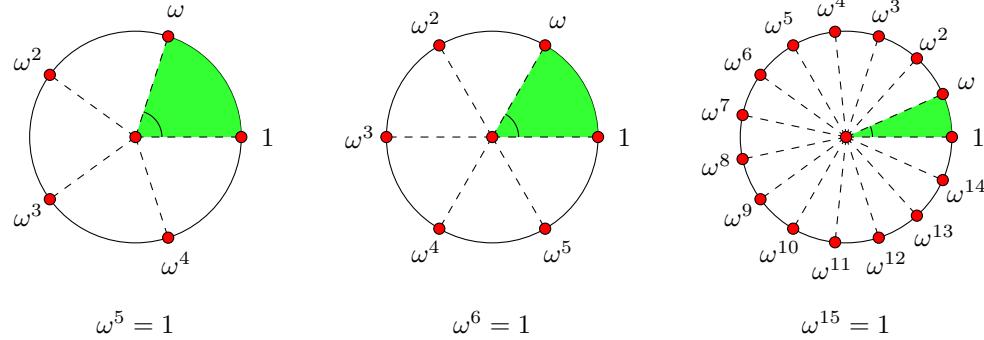


Figure 1.25: The fifth, sixth, and fifteenth roots of unity

Summarizing,

### Roots of Unity

If

$$\omega = \cos(2\pi/d) + i \sin(2\pi/d),$$

the  $d$ -th roots of unity are

$$1, \omega, \omega^2, \dots, \omega^{d-1}.$$

The roots satisfy

$$\omega^k = \cos(2\pi k/d) + i \sin(2\pi k/d), \quad k = 0, 1, 2, \dots, d-1.$$

Since  $\omega^d = 1$ , one has, from Figures 1.24 and 1.25,

$$\omega^k + \omega^{-k} = \omega^k + \omega^{d-k} = 2 \cos(2\pi k/d), \quad k = 0, 1, 2, \dots, d-1. \quad (1.5.7)$$

Here is `sympy` code for the roots of unity. We use `display` instead of `print` to pretty-print the output.

```
from sympy import RootOf, symbols, init_printing
init_printing()

x = symbols('x')
d = 5
```

```
for k in range(d): display(RootOf(x**d - 1,k))
```

The *fundamental theorem of algebra* states that a polynomial  $p(x)$  of degree  $d$  has  $d$  roots: There are  $d$  complex numbers  $x_0, x_1, \dots, x_{d-1}$  (not necessarily distinct) satisfying  $p(x) = 0$ . In `numpy`, the roots of the polynomial  $p(x) = ax^2 + bx + c$  are returned by

```
import numpy as np
np.roots([a,b,c])
```

Since the cube roots of unity are the roots of the polynomial  $p(x) = x^3 - 1$ , the code

```
import numpy as np
np.roots([1,0,0,-1])
```

returns the cube roots

```
array([-0.5+0.8660254j, -0.5-0.8660254j, 1. +0.j])
```

## 1.6 Mean and Covariance

Let  $x_1, x_2, \dots, x_N$  be a dataset in  $\mathbf{R}^d$ , and let  $x$  be any point in  $\mathbf{R}^d$ . The *mean-square distance* of  $x$  to  $D$  is

$$MSD(x) = \frac{1}{N} \sum_{k=1}^N |x_k - x|^2.$$

Above  $|x|$  stands for the length of the vector  $x$ , or the distance of the point  $x$  to the origin. When  $d = 2$  and we are in two dimensions, this was defined in §1.4. For general  $d$ , this is defined in §2.1. In this section we continue to focus on two dimensions  $d = 2$ .

The *mean* or *sample mean* is

$$m = \frac{1}{N} \sum_{k=1}^N x_k = \frac{x_1 + x_2 + \cdots + x_N}{N}.$$

The mean  $m$  is a point in feature space. The first result is

### Point of Best-fit

The mean is the point of best-fit: The mean minimizes the mean-square distance to the dataset (Figure 1.26).

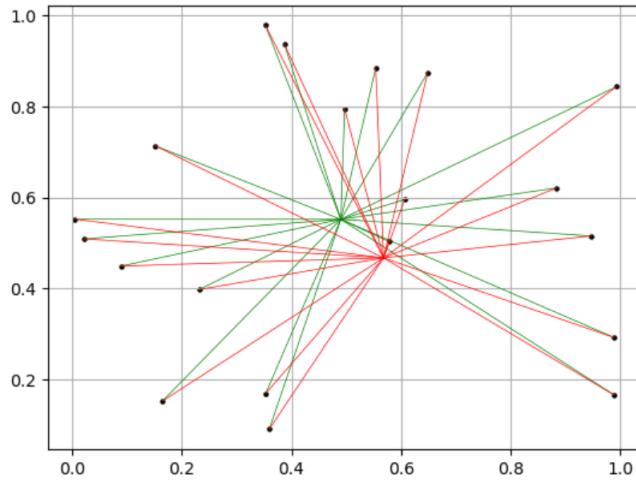


Figure 1.26: MSD for the mean (green) versus MSD for a random point (red).

Using (1.4.8),

$$|a + b|^2 = |a|^2 + 2a \cdot b + |b|^2$$

for vectors  $a$  and  $b$ , it is easy to derive the above result. Insert  $a = x_k - m$  and  $b = m - x$  to get

$$MSD(x) = MSD(m) + \frac{2}{N} \sum_{k=1}^N (x_k - m) \cdot (m - x) + |m - x|^2.$$

Now the middle term vanishes

$$\begin{aligned}\frac{2}{N} \sum_{k=1}^N (x_k - m) \cdot (m - x) &= \frac{2}{N} \left( \left( \sum_{k=1}^N x_k \right) - Nm \right) \cdot (m - x) \\ &= 2(m - m) \cdot (m - x) = 0,\end{aligned}$$

so we have

$$MSD(x) = MSD(m) + |m - x|^2 \geq MSD(m),$$

deriving the above result.

Here is the code for Figure 1.26.

```
from matplotlib.pyplot import *
from numpy import *
from numpy.random import random

N = 20
dataset = array([ [random(),random()] for _ in range(N) ] )

m = mean(dataset, axis=0)
p = array([random(),random()])

grid()
X = dataset[:,0]
Y = dataset[:,1]
scatter(X,Y)

for v in dataset:
    plot([m[0],v[0]],[m[1],v[1]],c='green')
    plot([p[0],v[0]],[p[1],v[1]],c='red')
show()
```



The covariance of a dataset is defined in any dimension  $d$ . When  $d = 1$ , the dataset consists of scalars  $x_1, x_2, \dots, x_N$ , and the mean  $m$  is a scalar.

In this case, the covariance  $q$  is also a scalar,

$$q = \frac{1}{N} \sum_{k=1}^N (x_k - m)^2.$$

In the scalar case, the covariance is called the *variance* of the scalar dataset. In general, the covariance is a symmetric  $d \times d$  matrix  $Q$ . When  $d = 1$ , a  $1 \times 1$  matrix is a scalar,  $Q = (q)$ , as above.

If the dataset  $x_1, x_2, \dots, x_N$  has mean  $m$ , we can center the dataset,

$$v_1 = x_1 - m, v_2 = x_2 - m, \dots, v_N = x_N - m.$$

Then the *covariance matrix* is (see §1.4 for tensor product)

$$Q = \frac{v_1 \otimes v_1 + v_2 \otimes v_2 + \cdots + v_N \otimes v_N}{N}. \quad (1.6.1)$$

The covariance is a symmetric  $d \times d$  matrix. Here is code from scratch for the covariance matrix of a dataset.

```
from numpy import *
from numpy.random import random

def tensor(u,v):
    return array([ [ a*b for b in v] for a in u ])

N = 20
dataset = array([ [random(),random()] for _ in range(N) ])
m = mean(dataset, axis=0)

# center dataset
vectors = dataset - m

Q = mean([ tensor(v,v) for v in vectors ], axis=0)
```

For example, suppose  $N = 5$  and

$$x_1 = (1, 2), \quad x_2 = (3, 4), \quad x_3 = (5, 6), \quad x_4 = (7, 8), \quad x_5 = (9, 10). \quad (1.6.2)$$

Then  $m = (5, 6)$  and

$$\begin{aligned} v_1 &= x_1 - m = (-4, -4), & v_2 &= x_2 - m = (-2, -2), & v_3 &= x_3 - m = (0, 0), \\ v_4 &= x_4 - m = (2, 2), & v_5 &= x_5 - m = (4, 4). \end{aligned}$$

Since

$$\begin{aligned} (\pm 4, \pm 4) \otimes (\pm 4, \pm 4) &= \begin{pmatrix} 16 & 16 \\ 16 & 16 \end{pmatrix}, \\ (\pm 2, \pm 2) \otimes (\pm 2, \pm 2) &= \begin{pmatrix} 4 & 4 \\ 4 & 4 \end{pmatrix}, \\ (0, 0) \otimes (0, 0) &= \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \end{aligned}$$

summing and dividing by  $N$  leads to the covariance

$$Q = \begin{pmatrix} 8 & 8 \\ 8 & 8 \end{pmatrix}.$$

Notice

$$Q = 8(1, 1) \otimes (1, 1),$$

which, as we see below (§2.5), reflects the fact that the points of this dataset lie on a line. Here the line is  $y = x + 1$ .

The covariance matrix as written in (1.6.1) is the *biased* covariance matrix. If the denominator is instead  $N - 1$ , the matrix is the *unbiased* covariance matrix.

For datasets with large  $N$ , it doesn't matter, since  $N$  and  $N - 1$  are almost equal. For simplicity, here we divide by  $N$ , and we only consider the biased covariance matrix.

In practice, datasets are *standardized* before computing their covariance. The covariance of standardized datasets — the correlation matrix — is the same whether one starts with bias or not (§2.2).

In numpy, the Python covariance constructor is

```
from numpy import *
from numpy.random import random

N = 20
dataset = array([ [random(), random()] for _ in range(N) ])
Q = cov(dataset, bias=True, rowvar=False)
```

This returns the same result as the previous code for  $Q$ . Notice here there is no need to compute the mean, this is taken care of automatically. The

option `bias=True` indicates division by  $N$ , returning the biased covariance. To return the unbiased covariance and divide by  $N - 1$ , change the option to `bias=False`, or remove it, since `bias=False` is the default.

The option `rowvar=False` indicates the vectors of the dataset are entered as rows; the dataset is then an  $N \times 2$  matrix. If the transpose `dataset.T` is entered, the  $X$  samples and  $Y$  samples of the dataset are entered separately, and we insert `rowvar=True`, or remove it, since `rowvar=True` is the default.

```
Q = cov(dataset.T,bias=True)
```

to return the same result.

From (1.4.13), if  $Q$  is the covariance matrix (1.6.1),

$$\text{trace}(Q) = \frac{1}{N} \sum_{k=1}^N |x_k - m|^2. \quad (1.6.3)$$

We call (1.6.3) the *total variance* of the dataset. Thus *the total variance equals MSD(m)*.

In Python, the total variance is

```
from numpy import *

Q = cov(dataset.T,bias=True)
Q.trace()
```



We now project a 2d dataset onto a line. Let  $u$  be a unit vector (a vector of length one,  $|u| = 1$ ), and let  $v_1, v_2, \dots, v_N$  be a 2d dataset, assumed for simplicity to be centered. We wish to project this dataset onto the line through  $u$ . This will result in a 1d dataset.

According to Figure 1.27, when a vector  $v$  is projected onto the line through  $u$ , the length of the projected vector  $\text{proj}_u v$  equals  $|v| \cos \theta$ , where  $\theta$  is the angle between the vectors  $v$  and  $u$ . Since  $|u| = 1$ , this length equals the dot product  $v \cdot u$ . Hence the projected vector is

$$\text{proj}_u v = (v \cdot u)u.$$

Applying this logic to each vector  $v_1, v_2, \dots, v_N$ , we conclude: *the projected dataset* onto the line through  $u$  is the dataset

$$(v_1 \cdot u)u, (v_2 \cdot u)u, \dots, (v_N \cdot u)u.$$

These vectors are all multiples of  $u$ , as they should be. The projected dataset is in  $\mathbf{R}^2$ .

Alternately, discarding  $u$  and retaining the scalar coefficients, we have the one-dimensional dataset

$$v_1 \cdot u, v_2 \cdot u, \dots, v_N \cdot u.$$

This is the *reduced dataset*. The reduced dataset consists of scalars.

Since the vector  $u$  is fixed, the reduced dataset and the projected dataset contain the same information.

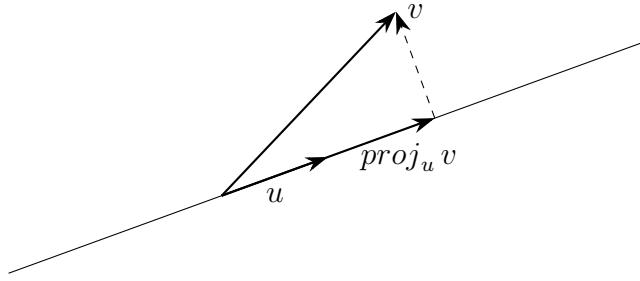


Figure 1.27: Projecting a vector  $v$  onto the line through  $u$ .

The mean of the reduced dataset is 0, since

$$\frac{v_1 \cdot u + v_2 \cdot u + \dots + v_N \cdot u}{N} = \left( \frac{v_1 + v_2 + \dots + v_N}{N} \right) \cdot u = 0 \cdot u = 0,$$

and the mean of the projected dataset is also 0.

Since the reduced dataset is one-dimensional, its variance is

$$q = \frac{1}{N} \sum_{k=1}^N (v_k \cdot u)^2.$$

But, according to (1.4.15), this equals

$$q = \frac{1}{N} \sum_{k=1}^N u \cdot (v_k \otimes v_k) u = u \cdot Qu.$$

Because the reduced dataset and projected dataset are essentially the same, we also refer to  $q$  as the *variance of the projected dataset*. Thus we conclude (see §1.4 for  $v \cdot Qv$ )

### Variance of Projected Dataset

Let  $Q$  be the covariance matrix of a dataset. Then the variance of the projected dataset onto the line through the vector  $u$  equals the quadratic function  $u \cdot Qu$ .

As a consequence, for any covariance  $Q$ , we see  $u \cdot Qu \geq 0$  for any vector  $u$ , as this is the variance of the projected dataset. Projections are studied further in §2.7.

Going back to the dataset (1.6.2),  $x_k - m$ ,  $k = 1, 2, 3, 4, 5$ , are all multiples of  $(1, 1)$ . If we select  $u = (1, -1)$ , then  $(x_k - m) \cdot u = 0$ , so the covariance  $Q$  satisfies  $u \cdot Qu = 0$ . This can also be seen by

$$Qu = 8((1, 1) \otimes (1, 1))u = 8(1, 1) \cdot u (1, 1) = 0.$$

This shows that the dataset lies on the line passing through  $m$  and perpendicular to  $(1, -1)$ .



Now we describe the covariance ellipses associated to a given dataset (Figure 1.28).

The contour of all points  $x$  satisfying  $x \cdot Qx = 1$  is the *covariance ellipsoid*. In two dimensions  $d = 2$ , this is the *covariance ellipse*. The contour of all points  $x$  satisfying  $x \cdot Q^{-1}x = 1$  is the *inverse covariance ellipsoid*. In two dimensions  $d = 2$ , this is the *inverse covariance ellipse*.

In two dimensions  $d = 2$ , a covariance matrix has the form

$$Q = \begin{pmatrix} a & b \\ b & c \end{pmatrix}.$$

If we write  $u = (x, y)$  for a vector in the plane, the covariance ellipse is

$$u \cdot Qu = ax^2 + 2bxy + cy^2 = 1.$$

Figure 1.28 displays examples of covariance ellipses (blue) and inverse covariance ellipses (red). When  $Q$  is diagonal, the lengths of the major and

minor axes of the inverse covariance ellipse equal  $2\sqrt{a}$  and  $2\sqrt{c}$ , and the lengths of the major and minor axes of the covariance ellipse equal  $2/\sqrt{a}$  and  $2/\sqrt{c}$ .

The covariance ellipse and inverse covariance ellipses described above are centered at the origin  $(0, 0)$ . When a dataset has mean  $m$  and covariance  $Q$ , the ellipses are drawn centered at  $m$ , as in Figures 1.30, 1.31, and 1.32.

In particular, when  $a = c$  and  $b = 0$ , then  $Q = aI$  is a multiple of the identity, the inverse covariance ellipse is the circle of radius  $\sqrt{a}$ , and the covariance ellipse is the circle of radius  $1/\sqrt{a}$ .

Here is the code for Figure 1.28. The ellipses drawn here are centered at the origin.

```
from matplotlib.pyplot import *
from numpy import *

L, delta = 4, .1
x = arange(-L,L,delta)
y = arange(-L,L,delta)
X,Y = meshgrid(x, y)

a, b, c = 9, 0, 4
det = a*c - b**2
A, B, C = c/det, -b/det, a/det

def ellipse(a,b,c,levels,color):
    contour(X,Y,a*X**2 + 2*b*X*Y + c*Y**2,levels,colors=color)

grid()
ellipse(a,b,c,[1],'blue')
ellipse(A,B,C,[1],'red')
show()
```

This completes the discussion of covariance ellipses.

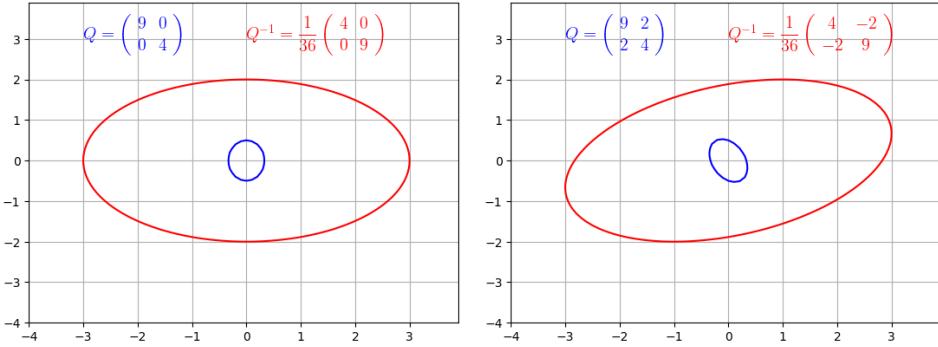


Figure 1.28: Covariance ellipses and inverse covariance ellipses.



Now we describe how to standardize datasets in  $\mathbf{R}^2$ . For datasets in  $\mathbf{R}^d$ , this is described in §2.2.

Remember, a dataset is a sequence of  $N$  points in a  $d$ -dimensional feature space. Restricting to the case  $d = 2$ , a dataset is a sequence of  $x$ -coordinates and  $y$ -coordinates

$$x_1, x_2, \dots, x_N, \quad \text{and} \quad y_1, y_2, \dots, y_N.$$

Suppose the mean of this dataset is  $m = (m_x, m_y)$ . Then, by the formula for tensor product, the covariance matrix is

$$Q = \begin{pmatrix} a & b \\ b & c \end{pmatrix},$$

where

$$a = \frac{1}{N} \sum_{k=1}^N (x_k - m_x)^2, \quad b = \frac{1}{N} \sum_{k=1}^N (x_k - m_x)(y_k - m_y), \quad c = \frac{1}{N} \sum_{k=1}^N (y_k - m_y)^2.$$

From this, we see  $a$  is the variance of the  $x$ -features, and  $c$  is the variance of  $y$ -features. We also see  $b$  is a measure of the correlation between the  $x$  and  $y$  features.

*Standardizing* the dataset means to center the dataset and to place the  $x$  and  $y$  features on the same scale. For example, the  $x$ -features may be close

to their mean  $m_x$ , resulting in a small  $x$  variance  $a$ , while the  $y$ -features may be spread far from their mean  $m_y$ , resulting in a large  $y$  variance  $c$ .

When this happens, the different scales of  $x$ 's and  $y$ 's distorts the relation between them, and  $b$  may not accurately reflect the correlation. To correct for this, we center and re-scale

$$x_1, x_2, \dots, x_N \quad \rightarrow \quad x'_1 = \frac{x_1 - m_x}{\sqrt{a}}, x'_2 = \frac{x_2 - m_x}{\sqrt{a}}, \dots, x'_N = \frac{x_N - m_x}{\sqrt{a}},$$

and

$$y_1, y_2, \dots, y_N \quad \rightarrow \quad y'_1 = \frac{y_1 - m_y}{\sqrt{c}}, y'_2 = \frac{y_2 - m_y}{\sqrt{c}}, \dots, y'_N = \frac{y_N - m_y}{\sqrt{c}}.$$

This results in a new dataset  $v_1 = (x'_1, y'_1), v_2 = (x'_2, y'_2), \dots, v_N = (x'_N, y'_N)$  that is centered,

$$\frac{v_1 + v_2 + \dots + v_N}{N} = 0,$$

with each feature standardized to have unit variance,

$$\frac{1}{N} \sum_{k=1}^N {x'_k}^2 = 1, \quad \frac{1}{N} \sum_{k=1}^N {y'_k}^2 = 1.$$

This is the *standardized dataset*.

Because of this, the covariance matrix of the standardized dataset has the form

$$Q' = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix},$$

where

$$\rho = \frac{1}{N} \sum_{k=1}^N x'_k y'_k = \frac{b}{\sqrt{ac}} = \frac{\sum_{k=1}^N (x_k - m_x)(y_k - m_y)}{\sqrt{\left( \sum_{k=1}^N (x_k - m_x)^2 \right) \left( \sum_{k=1}^N (y_k - m_y)^2 \right)}}$$

is the *Pearson correlation coefficient* of the dataset. The matrix  $Q'$  is the *correlation matrix*, or the standardized covariance matrix.

For example,

$$Q = \begin{pmatrix} 9 & 2 \\ 2 & 4 \end{pmatrix} \implies \rho = \frac{b}{\sqrt{ac}} = \frac{1}{3} \implies Q' = \begin{pmatrix} 1 & 1/3 \\ 1/3 & 1 \end{pmatrix}.$$

The correlation coefficient  $\rho$  (“rho”) is always between  $-1$  and  $1$  (this follows from the Cauchy-Schwarz inequality (1.4.5)).

When  $\rho = \pm 1$ , the dataset samples are perfectly correlated and lie on a line passing through the mean. When  $\rho = 1$ , the line has slope  $1$ , and when  $\rho = -1$ , the line has slope  $-1$ . When  $\rho = 0$ , the dataset samples are completely uncorrelated and are considered two independent one-dimensional datasets.

In Python `numpy`, the correlation matrix  $Q'$  is returned by

```
from numpy import *
corrcoef(dataset.T)
```

Here again, we input the transpose of the dataset if our default is vectors as rows. Notice the  $1/N$  cancels in the definition of  $\rho$ . Because of this, `corrcoef` is the same whether we deal with biased or unbiased covariance matrices.



We say a unit vector  $u$  is *best aligned* or *best-fit* with the dataset if  $u$  maximizes the variance  $v \cdot Qv$  over all unit vectors  $v$ ,

$$u \cdot Qu = \max_{|v|=1} v \cdot Qv.$$

We calculate the best-aligned unit vector. When a dataset is standardized, the variance of the dataset projected onto a vector  $v = (x, y)$  equals

$$v \cdot Qv = ax^2 + 2bxy + cy^2 = x^2 + 2\rho xy + y^2.$$

Since  $v = (x, y)$  is a unit vector, we have  $x^2 + y^2 = 1$ , so we can write  $(x, y) = (\cos \theta, \sin \theta)$ . Using the double-angle formula, we obtain

$$v \cdot Qv = x^2 + 2\rho xy + y^2 = 1 + 2\rho \sin \theta \cos \theta = 1 + \rho \sin(2\theta).$$

Since the sine function varies between  $+1$  and  $-1$ , we conclude the projected variance varies between

$$1 - \rho \leq v \cdot Qv \leq 1 + \rho,$$

and

$$\theta = \frac{\pi}{4}, \quad v_+ = \left( \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \right) \implies v_+ \cdot Qv_+ = 1 + \rho,$$

$$\theta = \frac{3\pi}{4}, \quad v_- = \left( \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \right) \implies v_- \cdot Qv_- = 1 - \rho.$$

Thus the best-aligned vector  $v_+$  is at  $45^\circ$ , and the worst-aligned vector is at  $135^\circ$  (Figure 1.29)

Actually, the above is correct only if  $\rho > 0$ . When  $\rho < 0$ , it's the other way. The correct answer is

$$1 - |\rho| \leq v \cdot Qv \leq 1 + |\rho|,$$

and  $v_\pm$  must be switched when  $\rho < 0$ . We study best-aligned vectors in  $\mathbf{R}^d$  in §3.2.

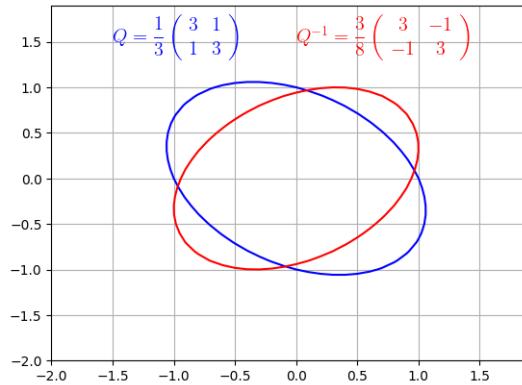


Figure 1.29: Covariance ellipse and incovariance ellipse.



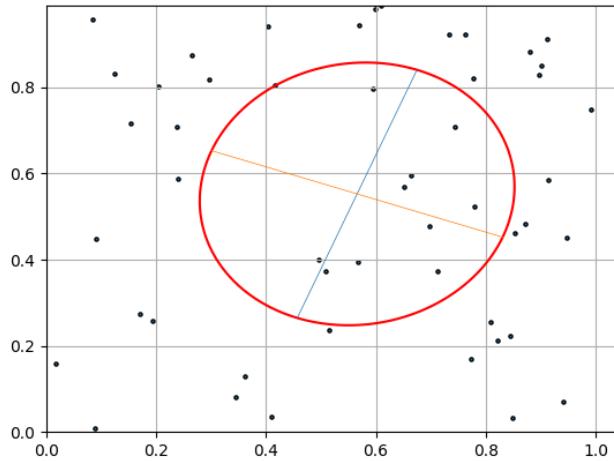


Figure 1.30: A positively correlated dataset  $\rho > 0$ .

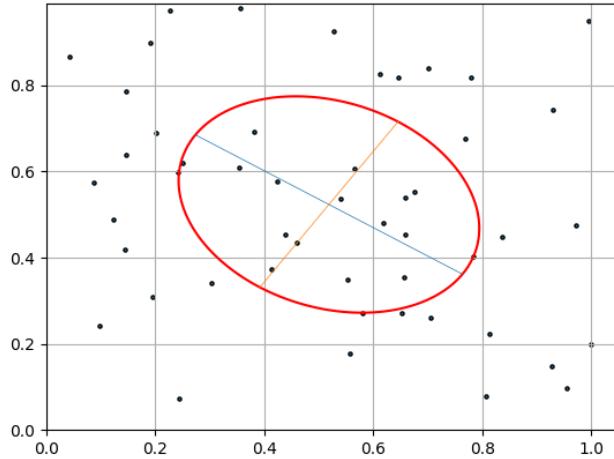


Figure 1.31: A negatively correlated dataset  $\rho < 0$ .

Here are two randomly generated datasets. For the dataset in Figure 1.30, the mean and covariance are

$$(0.46563359, 0.59153958) \quad \begin{pmatrix} 0.09652275 & 0.00939796 \\ 0.00939796 & 0.0674424 \end{pmatrix}.$$

For the dataset in Figure 1.31, the mean and covariance are

$$(0.48785572, 0.51945499) \quad \begin{pmatrix} 0.08266583 & -0.00976249 \\ -0.00976249 & 0.08298294 \end{pmatrix}.$$

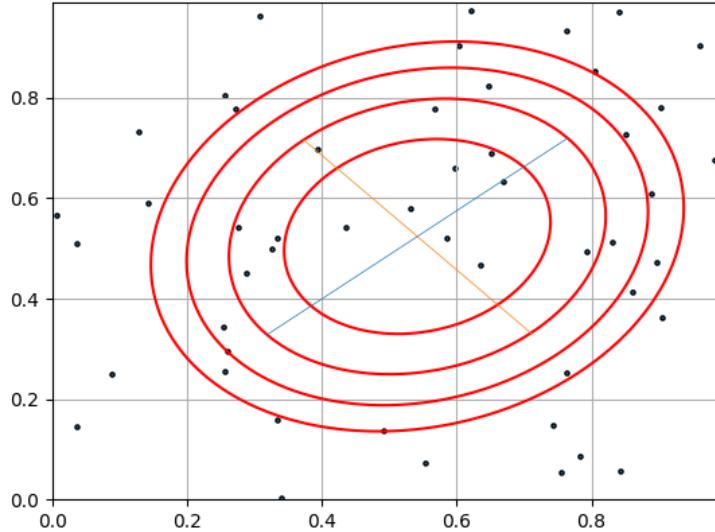


Figure 1.32: Level contours of  $v \cdot Q^{-1}v$ .



In general, for non-standardized datasets, the projected variance  $v \cdot Qv$  varies between two extremes  $\lambda_{\pm}$ ,

$$\lambda_- \leq v \cdot Qv \leq \lambda_+, \quad |v| = 1.$$

where  $\lambda_{\pm}$  are given by

$$\lambda_{\pm} = \frac{a+c}{2} \pm \sqrt{\left(\frac{a-c}{2}\right)^2 + b^2}. \quad (1.6.4)$$

When the dataset is standardized, as we saw above,  $\lambda_{\pm} = 1 \pm |\rho|$ .

The major axis of the inverse covariance ellipse  $v \cdot Q^{-1}v = 1$  has length  $2\sqrt{\lambda_+}$ , and the minor axis has length  $2\sqrt{\lambda_-}$ . These are the *principal axes* of the dataset.

When the dataset is not standardized, the best-aligned and worst-aligned vectors are

$$v_+ = (-b, a - \lambda_+), \quad v_- = (-b, a - \lambda_-). \quad (1.6.5)$$

All this will be discussed in detail in §3.2. In Figure 1.32, the level contours

$$v \cdot Q^{-1}v = k, \quad k = \frac{1}{2}, 1, \frac{3}{2}, 2,$$

are drawn.

In three dimensions, when  $d = 3$ , the ellipses are replaced by ellipsoids (Figure 1.33). In 3d, the inverse covariance ellipsoid and principal axes are displayed in Figure 1.33.

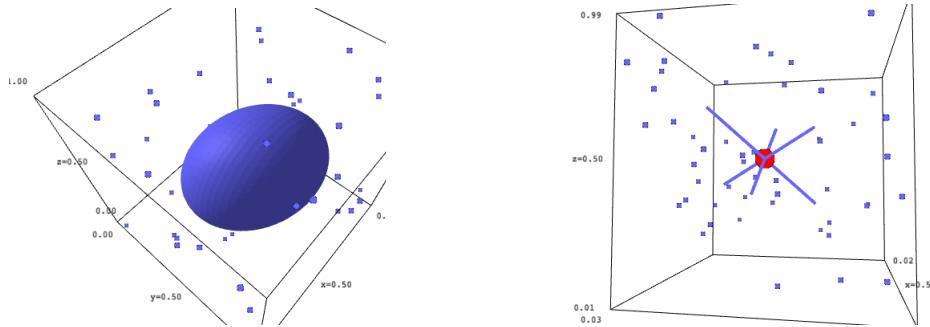


Figure 1.33: Ellipsoid and axes in 3d.



Here is code for Figures 1.30, 1.31, and 1.32. The code incorporates the formulas for  $\lambda_{\pm}$  and  $v_{\pm}$ .

```
from matplotlib.pyplot import *
from numpy import *
from numpy.random import *

N = 50
X = array([ random() for _ in range(N) ])
Y = array([ random() for _ in range(N) ])
scatter(X,Y,s=2)
```

```

m = mean([X,Y],axis=1)
Q = cov(X,Y,bias=True)
a, b, c = Q[0,0], Q[0,1], Q[1,1]

delta = .01
x = arange(0,1,delta)
y = arange(0,1,delta)
X,Y = meshgrid(x, y)

def ellipse(a,b,c,d,e,levels,color):
    det = a*c - b**2
    A, B, C = c/det, -b/det, a/det
    # inverse covariance ellipse centered at (d,e)
    Z = A*(X-d)**2 + 2*B*(X-d)*(Y-e) + C*(Y-e)**2
    contour(X,Y,Z,levels,colors=color)
    for pm in [+1,-1]:
        lamda = (a+c)/2 + pm * sqrt(b**2 + (a-c)**2/4)
        sigma = sqrt(lamda)
        len = sqrt(b**2 +(a-lamda)**2)
        axesX = [d+sigma*b/len,d-sigma*b/len]
        axesY = [e-sigma*(a-lamda)/len,e+sigma*(a-lamda)/len]
        plot(axesX,axesY,linewidth=.5)

    grid()
    levels = [.5,1,1.5,2]
    ellipse(a,b,c,*m,levels,'red')
    show()

```

## 1.7 High Dimensions

Although not directly used in later material, this section is here to boost intuition about high dimensions.

Draw four disks inside a square, as in Figure 1.34. In Figure 1.34, the edge-length of the square is 4, and the radius of each blue disk is 1. Since the length of the diagonal of the square is  $4\sqrt{2}$ , the radius of the red disk is

$$\frac{1}{4}(4\sqrt{2} - 4) = \sqrt{2} - 1.$$

Notice there are 4 blue disks.

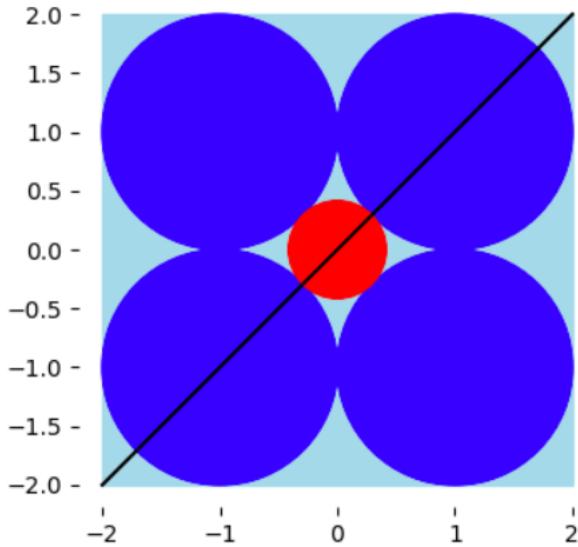


Figure 1.34: Disks inside the square

The following code returns Figure 1.34.

```
from matplotlib.pyplot import *
from matplotlib import patches
from numpy import *

fig, axes = subplots()

square = Rectangle((-2,-2), 4, 4,color='lightblue')
circle1 = Circle((1, 1), radius=1, color='blue')
circle2 = Circle((-1, 1), radius=1, color='blue')
circle3 = Circle((1, -1), radius=1, color='blue')
circle4 = Circle((-1, -1), radius=1, color='blue')
circle = patches.Circle((0, 0), radius=sqrt(2)-1,
    ↪ color='red')

plot([-2,2],[-2,2],color='black')
axes.add_patch(square)
```

```
axes.add_patch(circle1)
axes.add_patch(circle2)
axes.add_patch(circle3)
axes.add_patch(circle4)
axes.add_patch(circle)

for pos in ['right', 'top', 'bottom', 'left']:
    gca().spines[pos].set_visible(False)
axis('equal')
show()
```

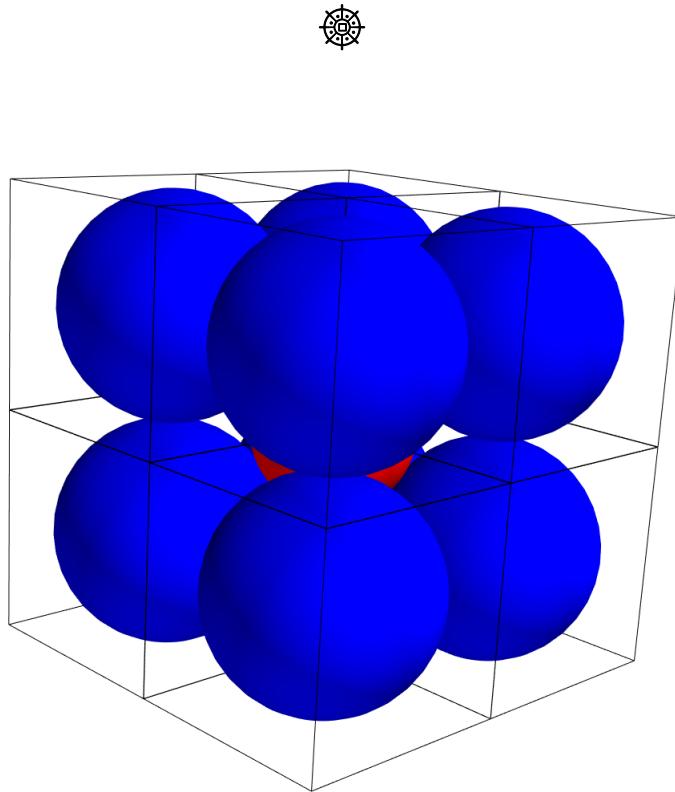


Figure 1.35: Balls inside the cube

Now we repeat this in three dimensions, obtaining Figure 1.35. Draw eight balls inside a cube, as in Figure 1.35.

Since the edge-length of the cube is 4, the radius of each blue ball is 1. Since the length of the diagonal of the cube is  $4\sqrt{3}$ , the radius of the red ball is

$$\frac{1}{4}(4\sqrt{3} - 4) = \sqrt{3} - 1.$$

Notice there are 8 blue balls.

In two dimensions, when a region is scaled by a factor  $t$ , its area increases by the factor  $t^2$ . In three dimensions, when a region is scaled by a factor  $t$ , its volume increases by the factor  $t^3$ . We conclude: In  $d$  dimensions, when a region is scaled by a factor  $t$ , its ( $d$ -dimensional) volume increases by the factor  $t^d$ . This is called the *scaling principle*.

In  $d$  dimensions, the edge-length of the cube remains 4, the radius of each blue ball remains 1, and there are  $2^d$  blue balls. Since the length of the diagonal of the cube is  $4\sqrt{d}$ , the same calculation results in the radius of the red ball equal to  $r = \sqrt{d} - 1$ .

By the scaling principle, the volume of the red ball equals  $r^d$  times the volume of the blue ball. We conclude the following:

- Since  $r = \sqrt{d} - 1 = 1$  exactly when  $d = 4$ , we have: *In four dimensions, the red ball and the blue balls are the same size.*
- Since there are  $2^d$  blue balls, the ratio of the volume of the red ball over the total volume of all the blue balls is  $r^d/2^d$ .
- Since  $r^d = 2^d$  exactly when  $r = 2$ , and since  $r = \sqrt{d} - 1 = 2$  exactly when  $d = 9$ , we have: *In nine dimensions, the volume of the red ball equals the sum total of the volumes of all blue balls.*
- Since  $r = \sqrt{d} - 1 > 2$  exactly when  $d > 9$ , we have: *In ten or more dimensions, the red ball sticks out of the cube.*
- Since the length of the semi-diagonal is  $2\sqrt{d}$ , for any dimension  $d$ , the radius of the red ball  $r = \sqrt{d} - 1$  is less than half the length of the semi-diagonal. *As the dimension grows without bound, the proportion of the diagonal covered by the red ball converges to 1/2.*

The code for Figure 1.35 is as follows. For 3d plotting, the module `mayavi` is better than `matplotlib`.

```
from mayavi.mlab import *
from numpy import *
from itertools import product

# run mayavi viewer inside notebook
init_notebook()

# clear any previously created
# mayavi scenes
clf()

# build sphere mesh
N = 40
theta = linspace(0,2*pi,N)
phi = linspace(0,pi,N)
theta,phi = meshgrid(theta,phi)
# spherical coordinates theta, phi
x = cos(theta)*sin(phi)
y = sin(theta)*sin(phi)
z = cos(phi)
# render ball
# here color is rgb triple of floats
def ball(a,b,c,r,color):
    return mesh(a + r*x,b + r*y, c + r*z,color=color)

pm1 = [-1,1]
for center in product(pm1,pm1,pm1):
    # blue balls: color (0,0,1)
    ball(*center,1,(0,0,1))
    # black wire cube: color (0,0,0)
    outline(color=(0,0,0))

    # red ball: color (1,0,0)
    ball(0,0,0,sqrt(3)-1,(1,0,0))
```



# Chapter 2

## Linear Geometry

In §1.4, we reviewed the geometry of vectors in the plane. Now we study linear geometry in any dimension  $d$ .

The material in this chapter is usually referred to as *Linear Algebra*. We prefer the term *Linear Geometry*, to emphasize that the material is, like much of data science, geometric.

### 2.1 Vectors and Matrices

A *vector* is a list of scalars

$$v = (t_1, t_2, \dots, t_d).$$

The scalars are the *components* or the *features* of  $v$ . If there are  $d$  features, we say the *dimension* of  $v$  is  $d$ . We call  $v$  a  $d$ -dimensional vector.

A *point*  $x$  is also a list of scalars,  $x = (t_1, t_2, \dots, t_d)$ . The relation between points  $x$  and vectors  $v$  is discussed in §1.3. The set of all  $d$ -dimensional vectors or points is  $d$ -dimensional space  $\mathbf{R}^d$ .

In Python, we use `numpy` or `sympy` for vectors and matrices. In Python, if `L` is a list, then `numpy.array(L)` or `sympy.Matrix(L)` return a vector or matrix.

```
from numpy import *
v = array([1,2,3])
v.shape
```

```
from sympy import *
v = Matrix([1,2,3])
v.shape
```

The first `v.shape` returns  $(3,)$ , and the second `v.shape` returns  $(3, 1)$ . In either case,  $v$  is a 3-dimensional vector.

Vectors are added component by component: With

$$v = (t_1, t_2, \dots) \quad \text{and} \quad v' = (t'_1, t'_2, \dots),$$

we have

$$v + v' = (t_1 + t'_1, t_2 + t'_2, \dots), \quad \text{and} \quad sv = (st_1, st_2, \dots).$$

Addition  $v + v'$  only works when  $v$  and  $v'$  have the same shape.

The *zero vector* is the vector  $0 = (0, 0, 0, \dots)$ . The zero vector is the only vector satisfying  $0 + v = v = v + 0$  for every vector  $v$ . Even though the zero scalar and the zero vector are distinct objects, we use 0 to denote both. A vector  $v$  is *nonzero* if  $v$  is not the zero vector.

In  $\mathbf{R}^4$ , the vectors

$$e_1 = (1, 0, 0, 0), \quad e_2 = (0, 1, 0, 0), \quad e_3 = (0, 0, 1, 0), \quad e_4 = (0, 0, 0, 1)$$

together are the *standard basis*. Similarly, in  $\mathbf{R}^d$ , we have the *standard basis*  $e_1, e_2, \dots, e_d$ .



A *matrix* is a listing arranged in a *rectangle of rows and columns*. Specifically, an  $N \times d$  matrix  $A$  has  $N$  rows and  $d$  columns,

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1d} \\ a_{21} & a_{22} & \dots & a_{2d} \\ \dots & \dots & \dots & \dots \\ a_{N1} & a_{N2} & \dots & a_{Nd} \end{pmatrix}.$$

In Python, if `L` is a list of lists, then both `array(L)` and `Matrix(L)` return a matrix. The code

```

from numpy import *

A = array([[1,6,11],[2,7,12],[3,8,13],[4,9,14],[5,10,15]])
A.shape

from sympy import *

A = Matrix([[1,6,11],[2,7,12],[3,8,13],[4,9,14],[5,10,15]])
A.shape

```

returns  $(5, 3)$ , so  $A$  is a  $5 \times 3$  matrix,

$$A = \begin{pmatrix} 1 & 6 & 11 \\ 2 & 7 & 12 \\ 3 & 8 & 13 \\ 4 & 9 & 14 \\ 5 & 10 & 15 \end{pmatrix}.$$

The *transpose* of a matrix  $A$  is the matrix  $B = A^t$  resulting from turning  $A$  on its side, so

$$B = A^t = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}.$$

Note the transpose operation interchanges rows and columns: the rows of  $A^t$  are the columns of  $A$ . In both `numpy` or `sympy`, the transpose of  $A$  is `A.T`.

A  $d$ -dimensional vector  $v$  may be written as a  $1 \times d$  matrix

$$v = (t_1 \ t_2 \ \dots \ t_d).$$

In this case, we call  $v$  a *row vector*.

An  $N$ -dimensional vector  $v$  may be written as a  $N \times 1$  matrix

$$v = \begin{pmatrix} t_1 \\ t_2 \\ \dots \\ t_N \end{pmatrix}.$$

In this case, we call  $v$  a *column vector*.



We will be considering matrices with different properties, and we use the following notation

- $A, B$ : any matrix
- $U, V$ : orthonormal rows or orthonormal columns
- $Q$ : symmetric matrix
- $P$ : projections



Vectors  $v_1, v_2, \dots, v_d$  with the same dimension may be horizontally stacked as columns of a matrix,

$$A = (v_1 \ v_2 \ \dots \ v_d).$$

Similarly, vectors  $v_1, v_2, \dots, v_N$  with the same dimension may be vertically stacked as rows of a matrix,

$$A = \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ v_N \end{pmatrix}.$$

By default, `sympy` creates column vectors. Because of this, it is easiest to build matrices as columns,

```
from sympy import *

# column vectors
u = Matrix([1,2,3,4,5])
v = Matrix([6,7,8,9,10])
w = Matrix([11,12,13,14,15])

# 5x3 matrix
```

```
A = Matrix.hstack(u,v,w)

# column vector
b = Matrix([1,1,1,1,1])

# 5x4 matrix
M = Matrix.hstack(A,b)
```

In general, for any `sympy` matrix `A`, column vectors can be `hstacked` and row vectors can be `vstacked`. For any matrix `A`, the code

```
from sympy import *

A == Matrix.hstack(*[A.col(j) for j in range(A.cols)])
```

returns `True`. Note we use the unpacking operator `*` to unpack the list, before applying `hstack`.

In `numpy`, there is `hstack` and `vstack`, but we prefer `column_stack` and `row_stack`, so the code

```
from numpy import *

A == row_stack([ row for row in A ])

A == column_stack([ col for col in A.T ])
```

returns `True`. In `numpy`, the input is a list, there is no unpacking.



In `numpy`, a matrix `A` is a list of rows, so

```
A == array([ row for row in A ])
A.T == array([ col for col in A.T ])
```

both return `True`. Here `col` refers to *rows* of  $A^t$ , hence refers to the columns of  $A$ .

The number of rows is `len(A)`, and the number of columns is `len(A.T)`. To access row  $i$ , use `A[i]`. To access column  $j$ , access row  $j$  of the transpose, `A.T[j]`. To access the  $j$ -th entry in row  $i$ , use `A[i,j]`.

In `sympy`, the number of rows in a matrix  $A$  is `A.rows`, and the number of columns is `A.cols`, so

```
A.shape == (A.rows,A.cols)
```

returns `True`. To access row  $i$ , use `A.row(i)`. Similarly, to access column  $j$ , use `A.col(j)`. So,

```
A == Matrix([ A.row(i) for i in range(A.rows) ])
A.T == Matrix([ A.col(j) for j in range(A.cols) ])
```

both return `True`.

A matrix is *square* if the number of rows equals the number of columns,  $N = d$ . A matrix is *diagonal* if it looks like one of these

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & d \end{pmatrix}, \quad \text{or} \quad \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \quad \text{or} \quad \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \\ 0 & 0 & 0 \end{pmatrix},$$

where some of the numbers on the diagonal  $a, b, c, d$  may be zero.



A dataset is a collection of points  $x_1, x_2, \dots, x_N$  in  $\mathbf{R}^d$ . After centering the mean to the origin (§1.3), the dataset becomes a collection of vectors  $v_1, v_2, \dots, v_N$ . Usually the vectors are presented as the rows of an  $N \times d$  matrix  $A$ . Corresponding to this, datasets are often provided as a *CSV file*.

The matrix  $A$  is the *dataset matrix*. In excel, this is called a *spreadsheet*. In SQL, this is called a *table*. In `numpy`, it's an *array*. In `pandas`, it's a *dataframe*. So, effectively,

*matrix = dataset = CSV file = spreadsheet = table = array = dataframe*



Matrices consisting of numbers are added and multiplied by scalars as follows. With

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1d} \\ a_{21} & a_{22} & \dots & a_{2d} \\ \dots & \dots & \dots & \dots \\ a_{N1} & a_{N2} & \dots & a_{Nd} \end{pmatrix} \quad \text{and} \quad A' = \begin{pmatrix} a'_{11} & a'_{12} & \dots & a'_{1d} \\ a'_{21} & a'_{22} & \dots & a'_{2d} \\ \dots & \dots & \dots & \dots \\ a'_{N1} & a'_{N2} & \dots & a'_{Nd} \end{pmatrix},$$

we have

$$A + A' = \begin{pmatrix} a_{11} + a'_{11} & a_{12} + a'_{12} & \dots & a_{1n} + a'_{1d} \\ a_{21} + a'_{21} & a_{22} + a'_{22} & \dots & a_{2n} + a'_{2d} \\ \dots & \dots & \dots & \dots \\ a_{N1} + a'_{N1} & a_{N2} + a'_{N2} & \dots & a_{Nd} + a'_{Nd} \end{pmatrix}$$

and

$$tA = \begin{pmatrix} ta_{11} & ta_{12} & \dots & ta_{1d} \\ ta_{21} & ta_{22} & \dots & ta_{2d} \\ \dots & \dots & \dots & \dots \\ ta_{N1} & ta_{N2} & \dots & ta_{Nd} \end{pmatrix}.$$

$A + A'$  is the result of *matrix addition*, and  $tA$  is the result of *matrix scaling*. Matrices may be added only if they have the same shape.

In Python, matrix scaling and matrix addition are `a*A` and `A + B`. The code

```
from sympy import *

A = zeros(2,3)
B = ones(2,2)
C = Matrix([[1,2],[3,4]])
D = B + C
E = 5 * C
F = eye(4)
A, B, C, D, E, F
```

returns

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}, \begin{pmatrix} 5 & 10 \\ 15 & 20 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Diagonal matrices are constructed using `diag`. The code

```
from sympy import *
A = diag(1,2,3,4)
B = diag(-1, ones(2, 2), Matrix([5, 7, 5]))
A, B
```

returns

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}, \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 5 \end{pmatrix}.$$

It is straightforward to convert back and forth between `numpy` and `sympy`. In the code

```
from sympy import *
A = diag(1,2,3,4)
from numpy import *
B = array(A)
C = Matrix(B)
```

$A$  and  $C$  are `sympy.Matrix`, and  $B$  is `numpy.array`. `numpy` is for numerical computations, and `sympy` is for algebraic/symbolic computations.



For the Iris dataset, the mean (§1.3) is given by the following code.

```
from sklearn import datasets
iris = datasets.load_iris()
```

```
dataset = iris["data"]
```

To center `dataset`, we compute the mean and subtract it,

```
m = mean(dataset, axis=0)
vectors = dataset - m
```

The mean is  $m = (5.84, 3.05, 3.76, 1.2)$ .

## 2.2 Products

Let  $t$  be a scalar,  $u, v, w$  be vectors, and let  $A, B$  be matrices. We already know how to compute  $tu$ ,  $tv$ , and  $tA, tB$ . In this section, we compute the dot product  $u \cdot v$ , the matrix-vector product  $Av$ , and the matrix-matrix product  $AB$ .

These products are not defined unless the dimensions “match”. In `numpy`, these products are written `dot`; in `sympy`, these products are written `*`.

In §1.4, we defined the dot product in two dimensions. We now generalize to any dimension  $d$ . Suppose  $u, v$  are vectors in  $\mathbf{R}^d$ . Then their *dot product*  $u \cdot v$  is the scalar obtained by multiplying corresponding features and then summing the products. This only works if the dimensions of  $u$  and  $v$  agree. In other words, if  $u = (s_1, s_2, \dots, s_d)$  and  $v = (t_1, t_2, \dots, t_d)$ , then

$$u \cdot v = s_1t_1 + s_2t_2 + \cdots + s_dt_d. \quad (2.2.1)$$

It’s best to think of this as “row-times-column” multiplication,

$$u \cdot v = \begin{pmatrix} s_1 & s_2 & s_3 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = s_1t_1 + s_2t_2 + s_3t_3.$$

As in §1.4, we always have *rows on the left, and columns on the right*.

In Python,

```
from numpy import *
u = array([1,2,3])
```

```
v = array([4, 5, 6])

dot(u,v) == 1*4 + 2*5 + 3*6

from sympy import *

u = Matrix([1,2,3])
v = Matrix([4, 5, 6])

u.T * v == 1*4 + 2*5 + 3*6
```

both return `True`.

For clarity, sometimes we write `(u.T)*v`; the parentheses don't change anything. Note in `sympy`, we take the transpose when multiplying, since vectors are by default column vectors, and it's always *row*  $\times$  *column*.



As in two dimensions, the *length* or *norm* or *magnitude* of a vector  $v$  is the square root of the dot product  $v \cdot v$ ,

$$|v| = \sqrt{v \cdot v}.$$

In Python, the length of a vector `v` is

```
from numpy import *

sqrt(dot(v,v))

from sympy import *

sqrt(v.T * v)
```

Notice `numpy` returns a scalar, while `sympy` returns a  $1 \times 1$  matrix.

A vector is a *unit vector* if its length equals 1. When  $|v| = 0$ , all the features of  $v$  equal zero. It follows the zero vector is the only vector with zero length. All other vectors have positive length.

Let  $v$  be any nonzero vector. By dividing  $v$  by its length  $|v|$ , we obtain a unit vector  $u = v/|v|$ .



As in §1.4,

### Dot Product

The dot product  $u \cdot v$  (2.2.1) satisfies

$$u \cdot v = |u| |v| \cos \theta, \quad (2.2.2)$$

where  $\theta$  is the angle between  $u$  and  $v$ .

In two dimensions, this was equation (1.4.4) in §1.4. Since any two vectors lie in a two-dimensional plane, this remains true in any dimension.

Based on this, we can compute the angle  $\theta$ ,

$$\cos \theta = \frac{u \cdot v}{\sqrt{|u| |v|}} = \frac{u \cdot v}{\sqrt{(u \cdot u)(v \cdot v)}}.$$

Here is code for the angle  $\theta$ ,

```
from numpy import *

def angle(u,v):
    a = dot(u,v)
    b = dot(u,u)
    c = dot(v,v)
    theta = arccos(a / sqrt(b*c))
    return degrees(theta)
```

Since  $|\cos \theta| \leq 1$ , we have the

### Cauchy-Schwarz Inequality

The dot product of two vectors is absolutely less or equal to the product of their lengths,

$$|u \cdot v| \leq |u| |v| \quad \text{or} \quad |u \cdot v|^2 \leq (u \cdot u)(v \cdot v). \quad (2.2.3)$$

Vectors  $u$  and  $v$  are said to be *perpendicular* or *orthogonal* if  $u \cdot v = 0$ . A collection of vectors is *orthogonal* if any pair of vectors in the collection

are orthogonal. With this understood, the zero vector is orthogonal to every vector. The converse is true as well: If  $u \cdot v = 0$  for every  $v$ , then in particular,  $u \cdot u = 0$ , which implies  $u = 0$ .

Vectors  $v_1, \dots, v_N$  are said to be *orthonormal* if they are both unit vectors and orthogonal. Orthogonal nonzero vectors can be made orthonormal by dividing each vector by its length.



An important application of the Cauchy-Schwarz inequality is the *triangle inequality*

$$|a + b| \leq |a| + |b|. \quad (2.2.4)$$

To see this, let  $v$  be any unit vector. Then

$$(a + b) \cdot v = a \cdot v + b \cdot v \leq |a||v| + |b||v| = |a| + |b|.$$

From this, selecting  $v = (a + b)/|a + b|$ ,

$$|a + b| = (a + b) \cdot v \leq |a| + |b|.$$



Suppose  $v$  is a vector and  $A$  is a matrix. If the rows of  $A$  have the same dimension as that of  $v$ , we can take the dot product of each row of  $A$  with  $v$ , obtaining the *matrix-vector product*  $Av$ : *Av is the vector whose features are the dot products of the rows of A with v.*

In other words,

```
dot(A,v) == array([ dot(row,v) for row in A ])
A*v == Matrix([ A.row(i) * v for i in range(A.rows) ])
```

both return True.

If  $u$  and  $v$  are vectors, we can think of  $u$  as a row vector, or a matrix consisting of a single row. With this interpretation, the matrix-vector product  $uv$  equals the dot product  $u \cdot v$ .

If  $u$  and  $v$  are vectors, we can think of  $u$  as a column vector, or a matrix consisting of a single column. With this interpretation,  $u^t$  is a single row, and the matrix-vector product  $u^t v$  equals the dot product  $u \cdot v$ .



Let  $A$  and  $B$  be two matrices. If the row dimension of  $A$  equals the column dimension of  $B$ , the *matrix-matrix product*  $AB$  is defined. When this condition holds, *the entries in the matrix  $AB$  are the dot products of the rows of  $A$  with the columns of  $B$* . In Python,

```
from numpy import *
C = array([ [ dot(row,col) for col in B.T ] for row in A ])
dot(A,B) == C

from sympy import *
C = Matrix([[ A.row(i)*B.col(j) for j in range(B.cols)] for i
    ↪ in range(A.rows) ])
A*B == C
```

both return `True`, and, with

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}, B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix},$$

the code

```
A,B,dot(A,B)
A,B,A*B
```

returns

$$AB = \begin{pmatrix} 70 & 80 & 90 \\ 158 & 184 & 210 \end{pmatrix}.$$



Let  $A$  and  $B$  be matrices. Since transpose interchanges rows and columns, we always have

$$(AB)^t = B^t A^t.$$

As a special case, if we think of  $v$  as a column vector, i.e. as a matrix with a single column, then the matrix-vector product  $Av$  is the same as the matrix-matrix product  $Av$ , so

$$(Av)^t = v^t A^t.$$

Here we are thinking of  $v$  as a matrix with one column, and  $v^t$  as a matrix with one row.

In Python,

```
dot(A,B).T == dot(B.T,A.T)

(A * B).T == B.T * A.T
```

both return `True`.

We also have

### Dot Product Transpose Identity

For any vectors  $u$ ,  $v$ , and matrices  $A$ , we have

$$(Au) \cdot v = u \cdot (A^t v) \quad \text{and} \quad (A^t u) \cdot v = u \cdot Av, \quad (2.2.5)$$

whenever the shapes of  $u$ ,  $v$ ,  $A$  match.

In terms of row vectors and column vectors, this is automatic. For example,

$$(Au) \cdot v = (Au)^t v = (u^t A^t)v = u^t (A^t v) = u \cdot (A^t v).$$

In Python,

```
dot(dot(A,u),v) == dot(u,dot(A.T,v))
dot(dot(A.T,u),v) == dot(u,dot(A,v))

(A*u).T * v == u.T * (A.T*v)
(A.T*u).T * v == u.T * (A*v)
```

all return True.



Let  $A$  be a matrix. We compute useful expressions for  $AA^t$  and  $A^tA$ .

Assume the rows of  $A$  are  $v_1, v_2, \dots, v_N$ . Since matrix-matrix multiplication is *row*  $\times$  *column*, we have

$$AA^t = \begin{pmatrix} v_1 \cdot v_1 & v_1 \cdot v_2 & \dots & v_1 \cdot v_N \\ v_2 \cdot v_1 & v_2 \cdot v_2 & \dots & v_2 \cdot v_N \\ \dots & \dots & \dots & \dots \\ v_N \cdot v_1 & v_N \cdot v_2 & \dots & v_N \cdot v_N \end{pmatrix}. \quad (2.2.6)$$

As a consequence,<sup>1</sup>

### Orthonormal Rows and Columns

Let  $U$  be a matrix.

- $U$  has orthonormal rows iff  $UU^t = I$ .
- $U$  has orthonormal columns iff  $U^tU = I$ .

The second statement follows from the first by substituting  $U^t$  for  $U$ .



To compute  $A^tA$ , we bring in the tensor product. If  $u$  and  $v$  are vectors, the *tensor product*  $u \otimes v$  is the matrix-matrix product  $u^t v$ , with  $u$  and  $v$  row vectors. If  $u$  is  $N$ -dimensional and  $v$  is  $d$ -dimensional, then  $u \otimes v$  is an  $N \times d$  matrix.

For example, if  $u = (a, b, c)$ ,  $v = (A, B)$ , then

$$u \otimes v = \begin{pmatrix} a \\ b \\ c \end{pmatrix} (A \quad B) = \begin{pmatrix} aA & aB \\ bA & bB \\ cA & cB \end{pmatrix}.$$

Then the identities (1.4.14) and (1.4.15) hold in general. Using the tensor product, we have

---

<sup>1</sup>Iff is short for if and only if.

### Tensor Identity

Let  $A$  be a matrix with rows  $v_1, v_2, \dots, v_N$ . Then

$$A^t A = v_1 \otimes v_1 + v_2 \otimes v_2 + \cdots + v_N \otimes v_N. \quad (2.2.7)$$

Multiplying (2.2.7) by  $x^t$  on the left and  $x$  on the right, and using (1.4.15), we see (2.2.7) is equivalent to

$$|Ax|^2 = x^t A^t Ax = (v_1 \cdot x)^2 + (v_2 \cdot x)^2 + \cdots + (v_N \cdot x)^2. \quad (2.2.8)$$

By matrix-vector multiplication,

$$Ax = (v_1 \cdot x, v_2 \cdot x, \dots, v_N \cdot x).$$

Since  $|Ax|^2$  is the sum of the squares of its components, this derives (2.2.8).



A matrix  $Q$  is *symmetric* if  $Q = Q^t$ . For any matrix  $A$ ,  $Q = AA^t$  and  $Q = A^t A$  are symmetric.

A symmetric matrix  $Q$  satisfying  $v \cdot Qv \geq 0$  for every vector  $v$  is *nonnegative*. A symmetric matrix  $Q$  satisfying  $v \cdot Qv > 0$  for every nonzero vector  $v$  is *positive*.

The most important example of a nonnegative matrix is the covariance matrix (§1.6) of a dataset. When a dataset in  $\mathbf{R}^d$  fills up all  $d$  dimensions, the covariance matrix is positive (see §2.5).

The *trace* of a square matrix

$$A = \begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix}$$

is the sum of its diagonal elements,

$$\text{trace}(A) = \text{trace} \begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix} = a + d + f.$$

Even though in general  $AB \neq BA$ , it is always true that

$$\text{trace}(AB) = \text{trace}(BA), \quad (2.2.9)$$

Trace and tensor product combine in the identity

$$u \cdot Qv = \text{trace}(Q(v \otimes u)). \quad (2.2.10)$$

The derivations of these identities are simple calculations that we skip.



If  $A = (a_{ij})$  is any matrix, then the *norm squared* of  $A$  is

$$\|A\|^2 = \sum_{i,j} a_{ij}^2.$$

By taking the trace in (2.2.7),

### Norm Squared of Matrix

Let  $A$  be a matrix with rows  $v_1, v_2, \dots, v_N$ . Then

$$\|A\|^2 = |v_1|^2 + |v_2|^2 + \cdots + |v_N|^2, \quad (2.2.11)$$

and

$$\|A\|^2 = \text{trace}(A^t A). \quad (2.2.12)$$

By replacing  $A$  by  $A^t$ , the same results hold for columns.



If  $x_1, x_2, \dots, x_N$  is a dataset of points in  $\mathbf{R}^d$  with mean  $m$ , and  $v_1, v_2, \dots, v_N$  is the corresponding centered dataset, then the covariance matrix  $Q$  is the average of tensor products (§1.6),

$$Q = \frac{v_1 \otimes v_1 + v_2 \otimes v_2 + \cdots + v_N \otimes v_N}{N}.$$

Let  $A$  be the matrix with rows  $v_1, v_2, \dots, v_N$ . By (2.2.7), the last equation is the same as

$$Q = \frac{1}{N} A^t A. \quad (2.2.13)$$

If we save the Iris dataset as a centered array `vectors`, as in §2.1, code from scratch for the covariance is

```
from numpy import *
Q = dot(vectors.T,vectors)/N
```

Of course, it is simpler to avoid centering and just do directly

```
Q = cov(dataset, rowvar=False)
```

or

```
Q = cov(dataset.T)
```

After downloading the Iris dataset as in §2.1, the mean, covariance, and total variance are

$$m = (5.84, 3.05, 3.76, 1.2), \quad Q = \begin{pmatrix} 0.68 & -0.04 & 1.27 & 0.51 \\ -0.04 & 0.19 & -0.32 & -0.12 \\ 1.27 & -0.32 & 3.09 & 1.29 \\ 0.51 & -0.12 & 1.29 & 0.58 \end{pmatrix}, \quad 4.54. \quad (2.2.14)$$



In §1.6, we discussed standardizing datasets in  $\mathbf{R}^2$ . This can be done in general.

Let  $x_1, x_2, \dots, x_N$  be a dataset in  $\mathbf{R}^d$ . Each sample point  $x$  has  $d$  features  $(t_1, t_2, \dots, t_d)$ . We compute the variance of each feature separately.

Let  $e_1, e_2, \dots, e_d$  be the standard basis in  $\mathbf{R}^d$ , and, for each  $j = 1, 2, \dots, d$ , project the dataset onto  $e_j$ , obtaining the scalar dataset  $x_1 \cdot e_j, x_2 \cdot e_j, \dots, x_N \cdot e_j$ , consisting of the  $j$ -th feature of the samples. If  $q_{jj}$  is the variance of this scalar dataset, then  $q_{11}, q_{22}, \dots, q_{dd}$  are the diagonal entries of the covariance matrix.

To *standardize* the dataset, we center it, and rescale the features to have variance one, as follows. Let  $m = (m_1, m_2, \dots, m_d)$  be the dataset mean. For each sample point  $x = (t_1, t_2, \dots, t_d)$ , the standardized vector is

$$v = \left( \frac{t_1 - m_1}{\sqrt{q_{11}}}, \frac{t_2 - m_2}{\sqrt{q_{22}}}, \dots, \frac{t_d - m_d}{\sqrt{q_{dd}}} \right).$$

Then the *standardized dataset* is  $v_1, v_2, \dots, v_N$ .

If  $Q = (q_{ij})$  is the covariance matrix, then the *correlation matrix* is the  $d \times d$  matrix  $Q' = (q'_{ij})$  with entries

$$q'_{ij} = \frac{q_{ij}}{\sqrt{q_{ii}q_{jj}}}, \quad i, j = 1, 2, \dots, d.$$

Then a straightforward calculation shows

### Standardized Covariance Equals Correlation

The covariance matrix of the standardized dataset equals the correlation matrix of the original dataset.

In Python,

```
from numpy import *
from sklearn.preprocessing import StandardScaler

# standardize dataset
vectors = StandardScaler().fit_transform(dataset)

Qcorr = corrcoef(dataset.T)
Qcov = cov(vectors.T, bias=True)

allclose(Qcov, Qcorr)
```

returns True.

## 2.3 Matrix Inverse

Let  $A$  be any matrix and  $b$  a vector. The goal is to solve the linear system

$$Ax = b. \quad (2.3.1)$$

In this section, we use the inverse  $A^{-1}$  and the pseudo-inverse  $A^+$  to solve (2.3.1).

However, it's very easy to construct matrices  $A$  and vectors  $b$  for which the linear system (2.3.1) has no solutions at all! For example, take  $A$  the

zero matrix and  $b$  any non-zero vector. Because of this, we must be careful when solving (2.3.1).



Given a square matrix  $A$ , the *inverse matrix* is the matrix  $B$  satisfying

$$AB = I = BA. \quad (2.3.2)$$

Here  $I$  is the identity matrix. Since  $I$  is a square matrix,  $A$  must also be a square matrix.

Only square matrices may have inverses. Moreover, not every square matrix has an inverse. For example, the zero matrix does not have an inverse. When  $A$  has an inverse, we say  $A$  is *invertible*.

If a matrix is  $d \times d$ , then the inverse is also  $d \times d$ . We write  $B = A^{-1}$  for the inverse matrix of  $A$ . For example, it is easy to check

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \implies A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}.$$

Since we can't divide by zero, a  $2 \times 2$  matrix is invertible only if  $ad - bc \neq 0$ .

Since

$$(AB)(B^{-1}A^{-1}) = A(BB^{-1})A^{-1} = AIA^{-1} = AA^{-1} = I,$$

we have

$$(AB)^{-1} = B^{-1}A^{-1}.$$



When  $A$  has an inverse  $A^{-1}$ , we can solve the linear system  $Ax = b$ .

### Solution of $Ax = b$ when $A$ invertible

If  $A$  is invertible, then

$$Ax = b \implies x = A^{-1}b. \quad (2.3.3)$$

This is easy to check, since

$$Ax = A(A^{-1}b) = (AA^{-1})b = Ib = b.$$

```

from sympy import *

# solving Ax=b
x = A.inv() * b

from numpy import *
from numpy.linalg import inv

# solving Ax=b
x = dot(inv(A) , b)

```



In general, a matrix  $A$  is not invertible, and  $Ax = b$  is solved using the pseudo-inverse  $x = A^+b$ . The definition and framework of the pseudo-inverse is in §2.6. The upshot is: *every (square or non-square) matrix  $A$  has a pseudo-inverse  $A^+$ .* Here is the general result.

### Solution of $Ax = b$ for General $A$

If  $Ax = b$  is solvable, then

$$x^+ = A^+b \quad \implies \quad Ax^+ = b.$$

If  $Ax = b$  is not solvable, then  $x^+$  minimizes the residual  $|Ax - b|^2$ .

This says if  $Ax = b$  has *some* solution, then  $x^+ = A^+b$  is also a solution. On the other hand,  $Ax = b$  may have no solution, in which case the error  $|Ax - b|^2$  is minimized. From this point of view, it's best to think of  $x^+$  as a *candidate* for a solution. It's a solution only after confirming equality of  $Ax^+$  and  $b$ . All this is worked out in §2.6.

To put this in context, there are three possibilities for a linear system (2.3.1). A linear system  $Ax = b$  can have

- no solutions, or
- exactly one solution, or

- infinitely many solutions.

As examples of these three possibilities, we have

- $A = 0$  and  $b \neq 0$ ,
- $A$  is invertible,
- $A = 0$  and  $b = 0$ .

The pseudo-inverse provides *a single systematic procedure* for deciding among these three possibilities. The pseudo-inverse is available in `numpy` and `sympy` as `pinv`. In this section, we focus on using Python to solve  $Ax = b$ , postponing concepts to §2.6.

How do we use the above result? Given  $A$  and  $b$ , using Python, we compute  $x = A^+b$ . Then we check, by multiplying in Python, equality of  $Ax$  and  $b$ .

The rest of the section consists of examples of solving linear systems. The reader is encouraged to work out the examples below in Python. However, because some linear systems have more than one solution, and the implementation of Python on your laptop may be different than on my laptop, our solutions may differ.

It can be shown that if the entries of  $A$  are integers, then the entries of  $A^+$  are fractions. This fact is reflected in `sympy`, but not in `numpy`, as the default in `numpy` is to work with floats.



Let

$$u = (1, 2, 3, 4, 5), v = (6, 7, 8, 9, 10), w = (11, 12, 13, 14, 15),$$

and let  $A$  be the matrix with columns  $u, v, w$ , and rows  $a, b, c, d, e$ ,

$$A = (u \ v \ w) = \begin{pmatrix} 1 & 6 & 11 \\ 2 & 7 & 12 \\ 3 & 8 & 13 \\ 4 & 9 & 14 \\ 5 & 10 & 15 \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix}. \quad (2.3.4)$$

```
from numpy import *

# vectors
u = array([1,2,3,4,5])
v = array([6,7,8,9,10])
w = array([11,12,13,14,15])

# arrange as columns
A = column_stack([u,v,w])
```

For this  $A$ , the code

```
from scipy.linalg import pinv

pinv(A)
```

returns

$$A^+ = \frac{1}{150} \begin{pmatrix} -37 & -20 & -3 & 14 & 31 \\ -10 & -5 & 0 & 5 & 10 \\ 17 & 10 & 3 & -4 & -11 \end{pmatrix}.$$

Alternatively, in `sympy`,

```
from sympy import *

# column vectors
u = Matrix([1,2,3,4,5])
v = Matrix([6,7,8,9,10])
w = Matrix([11,12,13,14,15])

A = Matrix.hstack(u,v,w)

A.pinv()
```

returns the same result.



Let  $A$  be as in (2.3.4) and let

$$b_1 = (8, 9, 10, 11, 12), \quad b_2 = (11, 6, 1, -4, -9).$$

We solve  $Ax = b_1$  and  $Ax = b_2$  by computing the candidates

$$x^+ = A^+b_1 = \frac{1}{15}(2, 5, 8),$$

and

$$x^+ = A^+b_2 = \frac{1}{30}(-173, -50, 73).$$

Then we check that the candidates are actually solutions, which they are, by comparing  $Ax^+$  and  $b_1$ , in the first case, and  $Ax^+$  and  $b_2$ , in the second case.



For

$$b_3 = (-9, -3, 3, 9, 10),$$

we have

$$x^+ = A^+b_3 = \frac{1}{15}(82, 25, -32).$$

However, for this  $x^+$ , we have

$$Ax^+ = (-8, -3, 2, 7, 12),$$

which is not equal to  $b_3$ . From this, not only do we conclude  $x^+$  is not a solution of  $Ax = b_3$ , but also, by the general result above, the system  $Ax = b_3$  is not solvable at all.



Let  $B$  be the matrix with columns  $b_1$  and  $b_2$ ,

$$B = (b_1, b_2) = \begin{pmatrix} 8 & 11 \\ 9 & 6 \\ 10 & 1 \\ 11 & -4 \\ 12 & -9 \end{pmatrix}.$$

We solve

$$Bx = u, \quad Bx = v, \quad Bx = w$$

by constructing the candidates

$$B^+u, \quad B^+v, \quad B^+w,$$

obtaining the solutions

$$x^+ = \frac{1}{51}(16, -7), \quad x^+ = \frac{1}{51}(41, -2), \quad x^+ = \frac{1}{51}(66, 3).$$



Let

$$C = A^t = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

and let  $f = (0, -5, -10)$ . Then

$$C^+ = (A^t)^+ = (A^+)^t = \frac{1}{150} \begin{pmatrix} -37 & -10 & 17 \\ -20 & -5 & 10 \\ -3 & 0 & 3 \\ 14 & 5 & -4 \\ 31 & 10 & -11 \end{pmatrix}$$

and

$$x^+ = C^+f = \frac{1}{50}(32, 35, 38, 41, 44).$$

Once we confirm equality of  $Cx^+$  and  $f$ , which is the case, we obtain a solution  $x^+$  of  $Cx = f$ .



Let  $D$  be the matrix with columns  $a$  and  $f$ ,

$$D = (a, f) = \begin{pmatrix} 1 & 0 \\ 6 & -5 \\ 11 & -10 \end{pmatrix},$$

and let  $a, b, c, d, e$  be the rows of  $A$ , or, equivalently, the columns of  $C$ . Then

$$D^+ = \frac{1}{30} \begin{pmatrix} 25 & 10 & -5 \\ 28 & 10 & -8 \end{pmatrix}.$$

We solve

$$Dx = a, \quad Dx = b, \quad Dx = c, \quad , Dx = d, \quad Dx = e,$$

by constructing the candidates

$$D^+a, \quad D^+b, \quad D^+c, \quad D^+d, \quad D^+e,$$

obtaining the solutions

$$x^+ = (1, 0), \quad x^+ = (2, 1), \quad x^+ = (3, 2), \quad x^+ = (4, 3), \quad x^+ = (5, 4).$$



## 2.4 Span and Linear Independence

Let  $u, v, w$  be three vectors. Then

$$3u - \frac{1}{6}v + 9w, \quad 5u + 0v - w, \quad 0u + 0v + 0w$$

are linear combinations of  $u, v, w$ .

In general, a *linear combination* of vectors  $v_1, v_2, \dots, v_d$  is

$$t_1v_1 + t_2v_2 + \cdots + t_dv_d. \tag{2.4.1}$$

Here the coefficients  $t_1, t_2, \dots, t_d$  are scalars.

In terms of matrices, let

$$u = (1, 2, 3, 4, 5), v = (6, 7, 8, 9, 10), w = (11, 12, 13, 14, 15),$$

and let  $A$  be the matrix with columns  $u, v, w$ , as in (2.3.4). Let  $x$  be the vector  $(r, s, t) = (1, 2, 3)$ . Then an explicit calculation shows (do this calculation!) the matrix-vector product  $Ax$  equals  $ru + sv + tw$ ,

$$Ax = ru + sv + tw.$$

The code

```
dot(A,x) == r*u + s*v + t*w
```

returns

```
array([ True,  True,  True,  True,  True])
```

To repeat, the linear combination  $ru + sv + tw$  is the same as the matrix-vector product  $Ax$ . This is a general fact on which everything depends:

### Column Linear Combination Same as Matrix-Vector Product

Let  $A$  be a matrix with columns  $v_1, v_2, \dots, v_d$ , and let

$$x = (t_1, t_2, \dots, t_d).$$

Then

$$Ax = t_1v_1 + t_2v_2 + \cdots + t_dv_d, \quad (2.4.2)$$

In other words,

$$Ax = b \quad \text{is the same as} \quad b = t_1v_1 + t_2v_2 + \cdots + t_dv_d. \quad (2.4.3)$$



The *span* of vectors  $v_1, v_2, \dots, v_d$  consists of *all linear combinations*

$$t_1v_1 + t_2v_2 + \cdots + t_dv_d$$

of the vectors. For example,  $\text{span}(b)$  of a single vector  $b$  is the line through  $b$ , and  $\text{span}(u, v, w)$  is the set of all linear combinations  $ru + sv + tw$ .

### Span Definition I

The span of  $v_1, v_2, \dots, v_d$  is the set  $S$  of all linear combinations of  $v_1, v_2, \dots, v_d$ , and we write

$$S = \text{span}(v_1, v_2, \dots, v_d).$$

When we don't want to specify the vectors  $v_1, v_2, v_3, \dots, v_d$ , we simply say  $S$  is a span.

From (2.4.2), we have

### Span Definition II

Let  $A$  be the matrix with columns  $v_1, v_2, v_3, \dots, v_d$ . Then  $\text{span}(v_1, v_2, \dots, v_d)$  is the set  $S$  of all vectors of the form  $Ax$ .

If each vector  $v_k$  is a linear combination of vectors  $w_1, w_2, \dots, w_N$ , then every vector  $v$  in  $\text{span}(v_1, v_2, \dots, v_d)$  is a linear combination of  $w_1, w_2, \dots, w_N$ , so  $\text{span}(v_1, v_2, \dots, v_d)$  is contained in  $\text{span}(w_1, w_2, \dots, w_N)$ .

If also each vector  $w_k$  is a linear combination of vectors  $v_1, v_2, \dots, v_d$ , then every vector  $w$  in  $\text{span}(w_1, w_2, \dots, w_N)$  is a linear combination of  $v_1, v_2, \dots, v_d$ , so  $\text{span}(w_1, w_2, \dots, w_N)$  is contained in  $\text{span}(v_1, v_2, \dots, v_d)$ .

When both conditions hold, it follows

$$\text{span}(v_1, v_2, \dots, v_d) = \text{span}(w_1, w_2, \dots, w_N).$$

Thus there are many choices of spanning vectors for a given span.

For example, let  $u, v, w$  be the columns of  $A$  in (2.3.4). Let  $\subset$  mean “is contained in”. Then

$$\text{span}(u, v) \subset \text{span}(u, v, w),$$

since adding a third vector can only increase the linear combination possibilities. On the other hand, since  $w = 2v - u$ , we also have

$$\text{span}(u, v, w) \subset \text{span}(u, v).$$

It follows that

$$\text{span}(u, v, w) = \text{span}(u, v).$$



Let  $A$  be a matrix. The *column space* of  $A$  is the span of its columns. For  $A$  as in (2.3.4), the column space of  $A$  is  $\text{span}(u, v, w)$ . The code

```
from sympy import *
# column vectors
```

```

u = Matrix([1,2,3,4,5])
v = Matrix([6,7,8,9,10])
w = Matrix([11,12,13,14,15])

A = Matrix.hstack(u,v,w)

# returns minimal spanning set for column space of A
A.columnspace()

```

returns a *minimal* set of vectors spanning the column space of  $A$ . The *column rank* of  $A$  is the *number* of vectors returned.

For example, for  $A$  as in (2.3.4), this code returns

$$u = (1, 2, 3, 4, 5), \quad v = (6, 7, 8, 9, 10).$$

Why is this? Because  $w = 2v - u$ , so

$$\text{span}(u, v, w) = \text{span}(u, v).$$

We conclude the column rank of  $A$  equals 2.



If the columns of  $A$  are  $v_1, v_2, \dots, v_d$ , and  $x = (t_1, t_2, \dots, t_d)$  is a vector, then by definition of matrix-vector multiplication,

$$Ax = t_1v_1 + t_2v_2 + \cdots + t_dv_d.$$

By (2.4.3),

### Column Space and $Ax = b$

The column space of a matrix  $A$  consists of all vectors of the form  $Ax$ . A vector  $b$  is in the column space of  $A$  when  $Ax = b$  has a solution.



The corresponding code in `numpy` is

```

from numpy import *
from scipy.linalg import orth

# returns minimal orthonormal spanning set
# for column space of A

orth(A)

```

This code returns two *orthonormal* vectors  $b_1/|b_1|$  and  $b_2/|b_2|$ , where

$$b_1 = (8, 9, 10, 11, 12), \quad b_2 = (11, 6, 1, -4, -9),$$

and  $|b_1| = \sqrt{510}$ ,  $|b_2| = \sqrt{255}$ .

We conclude the column space of  $A$  can be described in at least three ways,

$$\text{span}(b_1, b_2) = \text{span}(u, v, w) = \text{span}(u, v).$$

Explicitly,  $b_1$  and  $b_2$  are linear combinations of  $u$ ,  $v$ ,  $w$ ,

$$15b_1 = 2u + 5v + 8w, \quad 30b_2 = -173u - 50v + 73w, \quad (2.4.4)$$

and  $u$ ,  $v$ ,  $w$  are linear combinations of  $b_1$  and  $b_2$ ,

$$51u = 16b_1 - 7b_2, \quad 51v = 41b_1 - 2b_2, \quad w = 2v - u. \quad (2.4.5)$$

By (2.4.3), to derive (2.4.4), we solve  $Ax = b_1$  and  $Ax = b_2$  for  $x$ . But this was done in §2.3.

Similarly, let  $B$  be the matrix with columns  $b_1$  and  $b_2$ , and solve  $Bx = u$ ,  $Bx = v$ ,  $Bx = w$ , obtaining (2.4.5). This was also done in §2.3.

As a general rule, `sympy.columnspace` returns vectors in close to original form, and `scipy.linalg.orth` orthonormalizes the spanning vectors.



Let  $A$  be a matrix, and let  $b$  be a vector. Assume the columns of  $A$  and  $b$  all lie in  $\mathbf{R}^d$ . How can we tell if  $b$  is in the column space of  $A$ ? Given the above tools, here is an easy way to tell.

Write the *augmented matrix*  $\bar{A} = (A, b)$ ;  $\bar{A}$  obtained by adding  $b$  as an extra column next to the columns of  $A$ . If  $A$  is  $d \times N$ , then  $\bar{A}$  is  $d \times (N+1)$ .

Given  $A$  and  $\bar{A} = (A, b)$ , compute their column ranks. Let  $v_1, v_2, \dots, v_N$  be the columns of  $A$ . If these ranks are equal, then

$$\text{span}(v_1, v_2, \dots, v_N) = \text{span}(v_1, v_2, \dots, v_N, b),$$

so  $b$  is a linear combination of the columns, or  $b$  is in the column space of  $A$ .

### Column Space of Augmented Matrix

Let  $\bar{A}$  be the matrix  $A$  augmented by a vector  $b$ . Then  $b$  is in the column space of  $A$  iff

$$\text{column rank}(A) = \text{column rank}(\bar{A}). \quad (2.4.6)$$

For example, let  $b_3 = (-9, -3, 3, 9, 10)$  and let  $\bar{A} = (A, b_3)$ . Using Python, check the column rank of  $\bar{A}$  is 3. Since the column rank of  $A$  is 2, we conclude  $b_3$  is not in the column space of  $A$ :  $b_3$  is not a linear combination of  $u, v, w$ .

When (2.4.6) holds,  $b$  is a linear combination of the columns of  $A$ . However, (2.4.6) does not tell us which linear combination. According to (2.4.3), finding the linear combination is equivalent to solving  $Ax = b$ .



$\mathbf{R}^3$  consists of all vectors  $(r, s, t)$  in three dimensions. If

$$e_1 = (1, 0, 0), \quad e_2 = (0, 1, 0), \quad e_3 = (0, 0, 1),$$

then

$$(r, s, t) = re_1 + se_2 + te_3.$$

This shows the vectors  $e_1, e_2, e_3$  span  $\mathbf{R}^3$ , or

$$\mathbf{R}^3 = \text{span}(e_1, e_2, e_3).$$

As a consequence,  $\mathbf{R}^3$  is a span. Similarly, in dimension  $d$ , we can write

$$\begin{aligned} e_1 &= (1, 0, 0, \dots, 0, 0) \\ e_2 &= (0, 1, 0, \dots, 0, 0) \\ e_3 &= (0, 0, 1, \dots, 0, 0) \\ \dots &= \dots \\ e_d &= (0, 0, 0, \dots, 0, 1) \end{aligned} \quad (2.4.7)$$

Then  $e_1, e_2, \dots, e_d$  span  $\mathbf{R}^d$ , so

***d*-dimensional Space**

$\mathbf{R}^d$  is a span.

The set  $e_1, e_2, \dots, e_d$  is the *standard basis* for  $\mathbf{R}^d$ .



The *row space* of a matrix is the span of its rows.

```
from sympy import *

# returns minimal spanning set for row space of A
A.rowspace()
```

The *row rank* of a matrix is the number of vectors returned by `rowspace()`. This is the *minimal* number of vectors spanning the row space of  $A$ .

For example, call the rows of  $A$  in (2.3.4)  $a, b, c, d, e$ . Let

$$f = (0, -5, -10).$$

Then `rowspace` returns the vectors  $a$  and  $f$ , so

$$\text{span}(a, b, c, d, e) = \text{span}(a, f).$$

Explicitly, the linear combination

$$50f = 32a + 35b + 38c + 41d + 44e$$

is derived using  $C = A^t$  and solving  $Cx = f$ . The linear combinations

$$a = a + 0f, \quad b = 2a - 5f, \quad c = 3a - 10f, \quad d = 4a - 15f, \quad e = 5a - 20f$$

are derived using  $D = (a, f)$  and solving  $Dx = a$ ,  $Dx = b$ ,  $Dx = c$ ,  $Dx = d$ ,  $Dx = e$ . Again, these linear systems were solved in §2.3.

Since the transpose interchanges rows and columns, the row space of  $A$  equals the column space of  $A^t$ . Using this, we compute the row space in `numpy` by

```
from numpy import *
from scipy.linalg import orth

# returns minimal spanning set for row space of A
orth(A.T)
```

Numpy returns *orthonormal* vectors.

When  $Q$  is symmetric, the row space of  $Q$  equals the column space of  $Q$ .



A linear combination  $t_1v_1 + t_2v_2 + \cdots + t_dv_d$  is *trivial* if all the coefficients are zero,  $t_1 = t_2 = \cdots = t_d = 0$ . Otherwise it is *non-trivial*, if at least one coefficient is not zero. A linear combination  $t_1v_1 + t_2v_2 + \cdots + t_dv_d$  *vanishes* if it equals the zero vector,

$$t_1v_1 + t_2v_2 + \cdots + t_dv_d = 0.$$

For example, with  $u, v, w$  as above, we have  $w = 2v - u$ , so

$$ru + sv + tw = 1u - 2v + 1w = 0 \quad (2.4.8)$$

is a vanishing non-trivial linear combination of  $u, v, w$ .

We say  $v_1, v_2, \dots, v_d$  are *linearly dependent* if there is a non-trivial vanishing linear combination of  $v_1, v_2, \dots, v_d$ . Otherwise, if there is no non-trivial vanishing linear combination, we say  $v_1, v_2, \dots, v_d$  are *linearly independent*. For example,  $u, v, w$  above are linearly dependent.

Suppose  $u, v, w$  are any three vectors, and suppose  $u, v, w$  are linearly dependent. Then we have  $ru + sv + tw = 0$  for some scalars  $r, s, t$ , where at least one is not zero. If  $r \neq 0$ , then we may solve for  $u$ , obtaining

$$u = -(s/r)v - (t/r)w.$$

If  $s \neq 0$ , then we may solve for  $v$ , obtaining

$$v = -(r/s)u - (t/s)w.$$

If  $t \neq 0$ , then

$$w = -(r/t)u - (s/t)v.$$

Hence linear dependence of  $u, v, w$  means one of the three vectors is a multiple of the other two vectors.

In general, linear dependence of  $v_1, v_2, \dots, v_d$  is the same as saying at least one of the vectors is a linear combination of the remaining vectors.

In terms of matrices,

### Homogeneous Linear Systems

Let  $A$  be the matrix with columns  $v_1, v_2, \dots, v_d$ . Then

- $v_1, v_2, \dots, v_d$  are linearly dependent when  $Ax = 0$  has a nonzero solution  $x$ , and
- $v_1, v_2, \dots, v_d$  are linearly independent when  $Ax = 0$  has only the zero solution  $x = 0$ .



The set of vectors  $x$  satisfying  $Ax = 0$ , or the set of *solutions*  $x$  of  $Ax = 0$ , is the *null space* of the matrix  $A$ .

With this terminology,  $v_1, v_2, \dots, v_d$  are linearly dependent when there is a nonzero null space for the matrix  $A$ .

For example, with  $A$  as in (2.3.4), the `sympy` code

```
from sympy import *
A.nullspace()
```

returns a list with a single vector,

$$[x] = \begin{bmatrix} r \\ s \\ t \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}.$$

This says the null space of  $A$  consists of all multiples of  $(1, -2, 1)$ . Since the code

```
[r,s,t] = A.nullspace()[0]
r*u + s*v + t*w
```

returns the column vector

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

we have  $Ax = 0$ , in agreement with (2.4.8).



The corresponding numpy code is

```
from scipy.linalg import null_space
null_space(A)
```

This code returns the *unit* vector

$$\frac{-1}{\sqrt{6}} \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix},$$

which is a multiple of  $(1, -2, 1)$ . `scipy.linalg.null_space` always *returns orthonormal vectors*.



Here is a simple result that is used frequently.

### A Versus $A^t A$

Let  $A$  be any matrix. The null space of  $A$  equals the null space of  $A^t A$ .

If  $x$  is in the null space of  $A$ , then  $Ax = 0$ . Multiplying by  $A^t$  leads to  $A^t Ax = 0$ , so  $x$  is in the null space of  $A^t A$ .

Conversely, if  $x$  is in the null space of  $A^t A$ , then  $A^t A x = 0$ . By the dot-product-transpose identity, (2.2.5),

$$|Ax|^2 = Ax \cdot Ax = x \cdot A^t Ax = 0,$$

so  $Ax = 0$ , which means  $x$  is in the null space of  $A$ .



An important example of linearly independent vectors are orthonormal vectors.

### Orthonormal Implies Linearly Independent

If  $v_1, v_2, \dots, v_d$  are orthonormal, they are linearly independent.

To see this, suppose we have a vanishing linear combination

$$t_1 v_1 + t_2 v_2 + \cdots + t_d v_d = 0.$$

Take the dot product of both sides with  $v_1$ . Since the dot products of any two vectors is zero, and each vector has length one, we obtain

$$t_1 = t_1 v_1 \cdot v_1 = t_1 v_1 \cdot v_1 + t_2 v_2 \cdot v_1 + \cdots + t_d v_d \cdot v_1 = 0.$$

Similarly, all other coefficients  $t_k$  are zero. This shows  $v_1, v_2, \dots, v_d$  are linearly independent.



In general, `nullspace()` returns a *minimal* set of vectors spanning the null space of  $A$ . The *nullity* of  $A$  is the number of vectors returned by the method `nullspace()`.

For example, to compute the nullspace of the matrix

$$C = A^t = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix},$$

we solve  $Cx = 0$ . Since the code

```

from numpy import *
from scipy.linalg import null_space

u = array([1,2,3,4,5])
v = array([6,7,8,9,10])
w = array([11,12,13,14,15])

C = row_stack([u,v,w])
null_space(B)

```

returns the list of three vectors

$$[x_1, x_2, x_3] = \left[ \begin{pmatrix} 1 \\ -2 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ -3 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ -4 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right],$$

here we can make three conclusions: (1) the nullspace of  $C$  is spanned by three vectors, (2) this is the least number of vectors that spans the nullspace of  $C$ , and (3) the nullity of  $C$  is 3.



Let  $S$  and  $T$  be spans. We say  $S$  and  $T$  are *orthogonal complements* if every vector in  $S$  is orthogonal to every vector in  $T$ . In symbols, we write  $S = T^\perp$  and  $T = S^\perp$  (pronounced “ $T$ -perp” and “ $S$ -perp”).

Suppose  $S$  is the span of vectors  $a, b, c$ . How do we compute  $S^\perp$ ? The answer is by using `nullspace`: Let  $A$  be the matrix with rows  $a, b, c$ . By matrix-vector multiplication,

$$0 = Ax = \begin{pmatrix} a \\ b \\ c \end{pmatrix} x = \begin{pmatrix} a \cdot x \\ b \cdot x \\ c \cdot x \end{pmatrix}.$$

This shows  $x$  is orthogonal to  $a, b, c$  exactly when  $x$  is in the null space of  $A$ . Thus  $S^\perp$  equals the null space of  $A$ .

In general, if  $S = \text{span}(v_1, v_2, \dots, v_N)$ , let  $A$  be the matrix with rows  $v_1, v_2, \dots, v_N$ . Then  $S^\perp$  equals the null space of  $A$ .



An important example of orthogonality is the row space and the null space. Suppose  $A$  has rows  $v_1, v_2, \dots, v_N$ , and  $x$  is a vector, all of the same dimension. Then, by definition, the matrix-vector product is

$$Ax = (v_1 \cdot x, v_2 \cdot x, \dots, v_N \cdot x).$$

If  $x$  is in the null space,  $Ax = 0$ , then

$$v_1 \cdot x = 0, v_2 \cdot x = 0, \dots, v_N \cdot x = 0,$$

so  $x$  is orthogonal to the rows of  $A$ . Conversely, if  $x$  is orthogonal to the rows of  $A$ , then  $Ax = 0$ .

This shows *the null space of  $A$  and the row space of  $A$  are orthogonal complements*. Summarizing, we write

### Row Space and Null Space are Orthogonal

Every vector in the row space is orthogonal to every vector in the null space,

$$(\text{nullspace})^\perp = \text{rowspace}, \quad (\text{rowspace})^\perp = \text{nullspace}. \quad (2.4.9)$$



Since the row space is the orthogonal complement of the null space, and the null space of  $A$  equals the null space of  $A^t A$ , we conclude

### $A$ Versus $A^t A$

Let  $A$  be any matrix. Then the row space of  $A$  equals the row space of  $A^t A$ .

Now replace  $A$  by  $A^t$  in this last result. Since the row space of  $A^t$  equals the column space of  $A$ , and  $AA^t$  is symmetric, we also have

### A Versus $AA^t$

Let  $A$  be any matrix. Then the column space of  $A$  equals the column space of  $AA^t$ .



Let  $A$  be a matrix and  $b$  a vector. So far we've met four spaces,

- the null space: all  $x$ 's satisfying  $Ax = 0$ ,
- the row space: the span of the rows of  $A$ ,
- the column space: the span of the columns of  $A$ ,
- the solution space: the solutions  $x$  of  $Ax = b$ .

A set  $S$  of vectors is a *subspace* if  $x_1 + x_2$  is in  $S$  whenever  $x_1$  and  $x_2$  are in  $S$ , and  $tx$  is in  $S$  whenever  $x$  is in  $S$ . When this happens, we say  $S$  is closed under addition and scalar multiplication: A subspace is a set of vectors closed under addition and scalar multiplication.

Since a linear combination of linear combinations is a linear combination, every span is a subspace. In particular,  $\mathbf{R}^d$  is a subspace.

It's important to realize the first three are subspaces, but the fourth is not.

- If  $x_1$  and  $x_2$  are in the null space, and  $r_1$  and  $r_2$  are scalars, then so is  $r_1x_1 + r_2x_2$ , because

$$A(r_1x_1 + r_2x_2) = r_1Ax_1 + r_2Ax_2 = r_10 + r_20 = 0.$$

This shows the null space is a subspace.

- The row space is a span, so is a subspace.
- The column space is a span, so is a subspace.
- The solution space  $S$  of  $Ax = b$  is not a subspace, nor a span: If  $x$  is in  $S$ , then  $Ax = b$ , so  $A(5x) = 5Ax = 5b$ , so  $5x$  is not in  $S$ .

If  $x_1$  and  $x_2$  are solutions of  $Ax = b$ , then  $A(x_1 + x_2) = 2b$ , so the solution space is not a subspace. However

$$A(x_1 - x_2) = b - b = 0, \quad (2.4.10)$$

so the difference  $x_1 - x_2$  of any two solutions  $x_1$  and  $x_2$  is in the null space of  $A$ , which is a span.



Let  $A$  be an  $N \times d$  matrix. Then matrix multiplication by  $A$  transforms a vector  $x$  to the vector  $b = Ax$ . From this point of view, the set of vectors  $x$  is the *source space*  $\mathbf{R}^d$ , and the set of vectors  $b = Ax$  is the *target space*  $\mathbf{R}^N$ .

The null space and the row space are in the source space, and the column space is in the target space.



Let  $A$  be a  $d \times d$  invertible matrix. Then the source space is  $\mathbf{R}^d$  and the target space is  $\mathbf{R}^d$ . If  $Ax = 0$ , then

$$x = (A^{-1}A)x = A^{-1}(Ax) = A^{-1}0 = 0.$$

This shows the null space of an invertible matrix is zero, hence the nullity is zero.

Since the row space is the orthogonal complement of the null space, we conclude the row space is all of  $\mathbf{R}^d$ .

In §2.9, we see that the column rank and the row rank are equal. From this, we see also the column space is all of  $\mathbf{R}^d$ . In summary,

### Null Space of Invertible Matrix

Let  $A$  be a  $d \times d$  invertible matrix. Then the null space is zero, and the row space and column space are both  $\mathbf{R}^d$ . In particular, the nullity is 0, and the row rank rank and column rank are both  $d$ .

## 2.5 Zero Variance Directions

Let  $x_1, x_2, \dots, x_N$  be a dataset in  $\mathbf{R}^d$ . Then  $x_1, x_2, \dots, x_N$  are  $N$  points in  $\mathbf{R}^d$ , and each  $x$  has  $d$  features,  $x = (t_1, t_2, \dots, t_d)$ . From §1.6, the mean is

$$m = \frac{x_1 + x_2 + \cdots + x_N}{N}.$$

Center the dataset (see §1.3)

$$v_1 = x_1 - m, v_2 = x_2 - m, \dots, v_N = x_N - m,$$

and let  $A$  be the matrix with rows  $v_1, v_2, \dots, v_N$ . By (2.2.7), the covariance is

$$Q = \frac{v_1 \otimes v_1 + v_2 \otimes v_2 + \cdots + v_N \otimes v_N}{N} = \frac{1}{N} A^t A.$$

If  $b$  is a vector, the projection of the centered dataset onto the line through  $b$  results in the reduced dataset

$$v_1 \cdot b, v_2 \cdot b, \dots, v_N \cdot b.$$

The mean of this projected dataset is zero, and its variance is

$$\frac{(v_1 \cdot b)^2 + (v_2 \cdot b)^2 + \cdots + (v_N \cdot b)^2}{N} = \frac{1}{N} b^t A^t A b = b \cdot Q b. \quad (2.5.1)$$

We obtain this result, which was first stated in §1.6.

### Variance of Projected Dataset

Let  $Q$  be the covariance matrix of a dataset. Then the variance of the projected dataset onto the line through the vector  $b$  equals the quadratic function  $b \cdot Q b$ .

A vector  $b$  is a *zero variance direction* if the projected variance is zero,

$$b \cdot Q b = 0.$$

We investigate zero variance directions, but first we need a definition.

Let  $m$  be a point in  $\mathbf{R}^d$  and  $b$  a vector in  $\mathbf{R}^d$ . The *hyperplane* passing through  $m$  and orthogonal to  $b$  is the set of points  $x$  satisfying the equation

$$b \cdot (x - m) = 0.$$

In  $\mathbf{R}^3$ , a hyperplane is a plane, and in  $\mathbf{R}^2$ , a hyperplane is a line. In general, in  $\mathbf{R}^d$ , a hyperplane is  $(d - 1)$ -dimensional, always one less than the ambient dimension.

### Zero Variance Directions

Let  $m$  and  $Q$  be the mean and covariance of a dataset in  $\mathbf{R}^d$ . Then  $b \cdot Qb = 0$  is the same as saying every point in the dataset lies in the hyperplane passing through  $m$  and orthogonal to  $b$ ,

$$b \cdot (x - m) = 0.$$

This is easy to see. Let the dataset be  $x_1, x_2, \dots, x_N$ , and center it to  $v_1, v_2, \dots, v_N$ . If  $b \cdot Qb = 0$ , then, by (2.5.1),  $v_k \cdot b = 0$  for  $k = 1, 2, \dots, N$ . This shows  $b \cdot (x_k - m) = 0$ ,  $k = 1, 2, \dots, N$ , which means the points  $x_1, x_2, \dots, x_N$  lie on the hyperplane  $b \cdot (x - m) = 0$ . Here are some examples.

In two dimensions  $\mathbf{R}^2$ , a line is determined by a point on the line and a vector orthogonal to the line. If  $(a, b)$  is the vector orthogonal to the line and  $(x_0, y_0), (x, y)$  are points on the line, then  $(x, y) - (x_0, y_0)$  is orthogonal to  $(a, b)$ , or

$$(a, b) \cdot ((x, y) - (x_0, y_0)) = 0.$$

Writing this out, the equation of the line is

$$a(x - x_0) + b(y - y_0) = 0, \quad \text{or} \quad ax + by = c,$$

where  $c = ax_0 + by_0$ .

If the mean and covariance of a dataset are  $m = (2, 3)$  and

$$Q = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix},$$

and  $b = (1, 1)$ , then  $Qb = 0$ , so  $b \cdot Qb = 0$ . Since the line passes through the mean, the dataset lies on the line  $x + y = 5$ . We conclude this dataset is one-dimensional.

If

$$Q = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix},$$

and  $b = (x, y)$ , then

$$b \cdot Qb = 3x^2 + y^2,$$

so  $b \cdot Qb$  is never zero unless  $b = 0$ . In this case, we conclude the dataset is two-dimensional, because it does not lie on a line.

In three dimensions  $\mathbf{R}^3$ , a plane is determined by a point  $(x_0, y_0, z_0)$  in the plane, and a vector  $(a, b, c)$  orthogonal to the plane. If  $(x, y, z)$  is any

point in the plane, then  $(x, y, z) - (x_0, y_0, z_0)$  is orthogonal to  $(a, b, c)$ , so the equation of the plane is

$$(a, b, c) \cdot ((x, y, z) - (x_0, y_0, z_0)) = 0, \quad \text{or} \quad ax + by + cz = d,$$

where  $d = ax_0 + by_0 + cz_0$ .

Suppose we have a dataset in  $\mathbf{R}^3$  with mean  $m = (3, 2, 1)$ , and covariance

$$Q = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}. \quad (2.5.2)$$

Let  $b = (2, -1, -1)$ . Then  $Qb = 0$ , so  $b \cdot Qb = 0$ . We conclude the dataset lies in the plane

$$(2, -1, -1) \cdot ((x, y, z) - (x_0, y_0, z_0)) = 0, \quad \text{or} \quad 2x - y - z = 3.$$

In this case, the dataset is two-dimensional, as it lies in a plane.

If a dataset has covariance the  $3 \times 3$  identity matrix  $I$ , then  $b \cdot Ib$  is never zero unless  $b = 0$ . Such a dataset is three-dimensional, it does not lie in a plane.

Sometimes there may be several zero variance directions. For example, for the covariance (2.5.2) and  $u = (2, -1, -1)$ ,  $v = (0, 1, -1)$ , we have both

$$u \cdot Qu = 0 \quad \text{and} \quad v \cdot Qv = 0.$$

From this we see the dataset corresponding to this  $Q$  lies in two planes: The plane orthogonal to  $u$ , and the plane orthogonal  $v$ . But the intersection of two planes is a line, so this dataset lies in a line, which means it is one-dimensional.

Which line does this dataset lie in? Well, the line has to pass through the mean, and is orthogonal to  $u$  and  $v$ . If we find a vector  $b$  satisfying  $b \cdot u = 0$  and  $b \cdot v = 0$ , then the line will pass through  $m$  and will be parallel to  $b$ . But we know how to find such a vector. Let  $A$  be the matrix with rows  $u, v$ . Then  $b$  in the nullspace of  $A$  fullfills the requirements. We obtain  $b = (1, 1, 1)$ .



Let  $v_1, v_2, \dots, v_N$  be a centered dataset of vectors in  $\mathbf{R}^d$ , and let  $Q$  be the covariance matrix of the dataset. Then  $v \cdot Qv$  is the variance of the projected dataset onto  $v$ . Being a variance, we know  $v \cdot Qv \geq 0$ . A zero variance direction is a vector  $v$  satisfying  $v \cdot Qv = 0$ . We show the zero variance directions are the same as the nullspace of  $Q$ ,

### Zero Variance Directions and Nullspace I

Let  $Q$  be a covariance matrix. Then the null space of  $Q$  equals the zero variance directions of  $Q$ .

To see this, we use the quadratic equation from high school. If  $Q$  is symmetric, then  $u \cdot Qv = v \cdot Qu$ . For  $t$  scalar and  $u, v$  vectors, it follows the quadratic function

$$(v + tu) \cdot Q(v + tu) = t^2 u \cdot Qu + 2tu \cdot Qv + v \cdot Qv = at^2 + 2bt + c$$

is nonnegative for all  $t$  scalar. Thus the parabola  $at^2 + 2bt + c$  intersects the horizontal axis in at most one root. This implies the discriminant  $b^2 - ac$  is not positive,  $b^2 - ac \leq 0$ , which yields

$$(u \cdot Qv)^2 \leq (u \cdot Qu)(v \cdot Qv).$$

Inserting  $u = Qv$ , we have

$$|Qv|^4 = (Qv \cdot Qv)^2 \leq (Qv \cdot QQv)(v \cdot Qv). \quad (2.5.3)$$

Now we can derive the result. If  $v$  is in the null space of  $Q$ , then  $Qv = 0$ . Taking the dot product with  $v$ , we get  $v \cdot Qv = v \cdot 0 = 0$ , so  $v$  is a zero variance direction. Conversely, if  $v$  is a zero variance direction, then  $v \cdot Qv = 0$ . By (2.5.3), this implies  $Qv = 0$ , so  $v$  is in the null space of  $Q$ .



Based on the above result, here is code that returns zero variance directions.

```
from numpy import *

def zero_variance(dataset):
    Q = cov(dataset.T)
    return null_space(Q)
```

Let  $A$  be an  $N \times d$  dataset matrix, and let  $Q$  be the covariance of the dataset. By (2.2.13),  $Q = A^t A / N$  if the dataset is centered. Then the null space of  $Q$  equals the null space of  $A^t A$ , which equals the null space of  $A$ . We conclude

### Zero Variance Directions and Nullspace II

Let  $Q$  be a covariance matrix of a centered dataset  $A$ . Then the null space of  $A$  equals the zero variance directions of  $Q$ .

Suppose the dataset is

$$(1, 2, 3, 4, 5), (6, 7, 8, 9, 10), (11, 12, 13, 14, 15), (16, 17, 18, 19, 20).$$

This is four vectors in  $\mathbf{R}^5$ . Since it is only four vectors, it is at most a four-dimensional dataset. The code returns three vectors

$$(1, -2, 1, 0, 0), (2, -3, 0, 1, 0), (3, -4, 0, 0, 1).$$

Thus this dataset is orthogonal to three directions. Each hyperplane is one condition, so each cuts the dimension down by one, so the dimension of this dataset is  $5 - 3 = 2$ . Dimension of a dataset is discussed further in §2.9.

## 2.6 Pseudo-Inverse

What exactly is the pseudo-inverse? It turns out the answer is best understood geometrically.

Think of  $b$  and  $Ax$  as points, and measure the distance between them, and think of  $x$  and the origin  $0$  as points, and measure the distance between them (Figure 2.1).

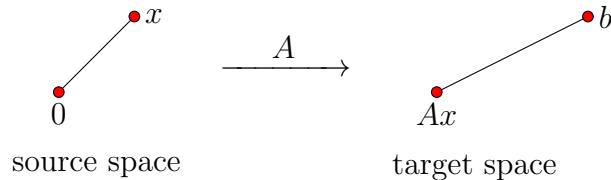


Figure 2.1: The points  $0$ ,  $x$ ,  $Ax$ , and  $b$ .

If  $Ax = b$  is solvable, then, among all solutions  $x^*$ , select the solution  $x^+$  closest to  $0$ .

More generally, if  $Ax = b$  is not solvable, select the points  $x^*$  so that  $Ax^*$  is closest to  $b$ , then, among all such  $x^*$ , select the point  $x^+$  closest to the origin.

Even though the point  $x^+$  may not solve  $Ax = b$ , this procedure (Figure 2.2) results in a uniquely determined  $x^+$ : While there may be several points  $x^*$ , there is only one  $x^+$ .

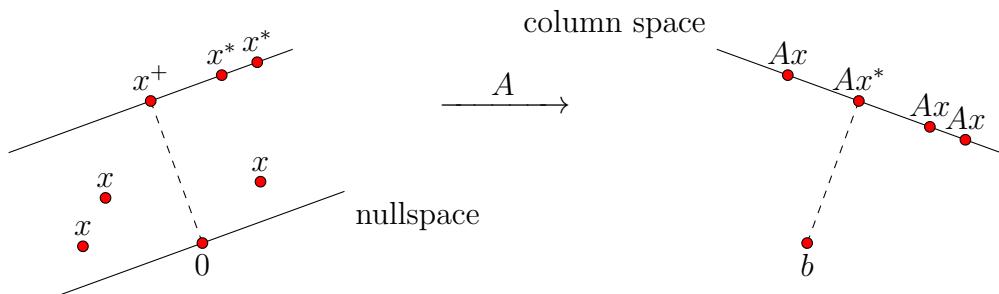


Figure 2.2: The points  $x$ ,  $Ax$ , the points  $x^*$ ,  $Ax^*$ , and the point  $x^+$ .

The results in this section are as follows. Let  $A$  be any matrix. There is a unique matrix  $A^+$  — the *pseudo-inverse of  $A$*  — with the following properties.

- the linear system  $Ax = b$  is solvable, when  $b = AA^+b$ .
- $x^+ = A^+b$  is a solution of
  1. the linear system  $Ax = b$ , if  $Ax = b$  is solvable.
  2. the regression equation  $A^tAx = A^tb$ , always.
- In either case,
  1. there is exactly one solution with minimum norm.
  2. Among all solutions,  $x^+$  has minimum norm.
  3. Every other solution is  $x = x^+ + v$  for  $v$  in the null space of  $A$ .



Key concepts in this section are the *residual*

$$|Ax - b|^2 \quad (2.6.1)$$

and the *regression equation*

$$A^t Ax = A^t b. \quad (2.6.2)$$

The following is clear.

### Zero Residual

$x$  is a solution of (2.3.1) iff the residual is zero.



For  $A$  as in (2.3.4) and  $b = (-9, -3, 3, 9, 10)$ , the linear system  $Ax = b$  is

$$\begin{aligned} x + 6y + 11z &= -9 \\ 2x + 7y + 12z &= -3 \\ 3x + 8y + 13z &= 3 \\ 4x + 9y + 14z &= 9 \\ 5x + 10y + 15z &= 10 \end{aligned} \quad (2.6.3)$$

and the regression equation  $A^t Ax = A^t b$  is

$$\begin{aligned} 11x + 26y + 41z &= 16 \\ 13x + 33y + 53z &= 13 \\ 41x + 106y + 171z &= 36. \end{aligned} \quad (2.6.4)$$



Let  $b$  be any vector, not necessarily in the column space of  $A$ . To see how close we can get to solving (2.3.1), we minimize the residual (2.6.1). We say  $x^*$  is a *residual minimizer* if

$$|Ax^* - b|^2 = \min_x |Ax - b|^2. \quad (2.6.5)$$

A residual minimizer always exists.

### Existence of Residual Minimizer

There is a residual minimizer  $x^*$  in the row space of  $A$ .

The derivation of this technical result is in §7.5, see (7.5.12), (7.5.13).

### Regression Equation

$x^*$  is a residual minimizer iff  $x^*$  solves the regression equation.

To see this, let  $v$  be any vector, and  $t$  a scalar. Insert  $x = x^* + tv$  into the residual and expand in powers of  $t$  to obtain

$$|Ax - b|^2 = |Ax^* - b|^2 + 2t(Ax^* - b) \cdot Av + t^2|Av|^2 = f(t).$$

If  $x^*$  is a residual minimizer, then  $f(t)$  is minimized when  $t = 0$ . But a parabola

$$f(t) = a + 2bt + ct^2$$

is minimized at  $t = 0$  only when  $b = 0$ . Thus the linear coefficient vanishes,  $(Ax^* - b) \cdot Av = 0$ . This implies

$$A^t(Ax^* - b) \cdot v = (Ax^* - b) \cdot Av = 0.$$

Since  $v$  is any vector, this implies

$$A^t(Ax^* - b) = 0,$$

which is the regression equation. Conversely, if the regression equation holds, then the linear coefficient in the parabola  $f(t)$  vanishes, so  $t = 0$  is a minimum, establishing that  $x^*$  is a residual minimizer.



If  $x_1$  and  $x_2$  are solutions of the regression equation, then

$$A^t A(x_1 - x_2) = A^t A x_1 - A^t A x_2 = A^t b - A^t b = 0,$$

so  $x_1 - x_2$  is in the null space of  $A^t A$ . But from §2.4, the nullspace of  $A^t A$  equals the nullspace of  $A$ . We conclude  $x_1 - x_2$  is in the null space of  $A$ . This establishes

### Multiple Solutions

Any two residual minimizers differ by a vector in the nullspace of  $A$ .



We say  $x^+$  is a *minimum norm* residual minimizer if  $x^+$  is a residual minimizer and

$$|x^+|^2 \leq |x^*|^2$$

for any residual minimizer  $x^*$ .

Since any two residual minimizers differ by a vector in the null space of  $A$ ,  $x^+$  is a minimum norm residual minimizer if  $x^+$  is a residual minimizer and

$$|x^+|^2 \leq |x^+ + v|^2$$

for any  $v$  in the null space of  $A$ .

### Minimum Norm Residual Minimizer

Let  $x^*$  be a residual minimizer. Then  $x^*$  is a minimum norm residual minimizer iff  $x^*$  is in the row space of  $A$ .

Since we know from above there is a residual minimizer in the row space of  $A$ , we always have a minimum norm residual minimizer.

Let  $v$  be in the null space of  $A$ , and write

$$|x^* + v|^2 = |x^*|^2 + 2x^* \cdot v + |v|^2.$$

This shows  $x^*$  is a minimum norm solution of the regression equation iff

$$2x^* \cdot v + |v|^2 \geq 0. \quad (2.6.6)$$

If  $x^*$  is in the row space of  $A$ , then  $x^* \cdot v = 0$ , so (2.6.6) is valid.

Conversely, if (2.6.6) is valid for every  $v$  in the null space of  $A$ , replacing  $v$  by  $tv$  yields

$$2tx^* \cdot v + t^2|v|^2 \geq 0.$$

Dividing by  $t$  and inserting  $t = 0$  yields

$$x^* \cdot v \geq 0.$$

Since both  $\pm v$  are in the null space of  $A$ , this implies  $\pm x^* \cdot v \geq 0$ , hence  $x^* \cdot v = 0$ . Since the row space is the orthogonal complement of the null space, the result follows.



Now we use this to show

### Uniqueness

There is exactly one minimum norm residual minimizer  $x^+$ .

If  $x_1^+$  and  $x_2^+$  are minimum norm residual minimizers, then  $v = x_1^+ - x_2^+$  is both in the row space and in the null space of  $A$ , so  $x_1^+ - x_2^+ = 0$ . Hence  $x_1^+ = x_2^+$ .

Putting the above all together, each vector  $b$  leads to a unique  $x^+$ . Defining  $A^+$  by setting

$$x^+ = A^+b,$$

we obtain  $A^+$ , the *pseudo-inverse* of  $A$ .

Notice if  $A$  is, for example,  $5 \times 4$ , then  $Ax = b$  implies  $x$  is a 4-vector and  $b$  is a 5-vector. Then from  $x = A^+b$ , it follows  $A^+$  is  $4 \times 5$ . Thus *the shape of  $A^+$  equals the shape of  $A^t$* .

Summarizing what we have so far,

### Regression Equation is Always Solvable

The regression equation (2.6.2) is always solvable. The solution of minimum norm is  $x^+ = A^+b$ . Any other solution differs by a vector in the null space of  $A$ .

For  $A$  as in (2.3.4) and  $b = (-9, -3, 3, 9, 10)$ ,

$$x^+ = A^+b = \frac{1}{15} \begin{pmatrix} 82 \\ 25 \\ -32 \end{pmatrix}$$

is the minimum norm solution of the regression equation (2.6.4).



Returning to the linear system (2.3.1), we show

### Linear System Versus Regression Equation

If the linear system is solvable, then every solution of the regression equation is a solution of the linear system, and vice-versa.

We know any two solutions of the linear system (2.3.1) differ by a vector in the null space of  $A$  (2.4.10), and any two solutions of the regression equation (2.6.2) differ by a vector in the null space of  $A$  (above).

If  $x$  is a solution of (2.3.1), then, by multiplying by  $A^t$ ,  $x$  is a solution of the regression equation (2.6.2). Since  $x^+ = A^+b$  is a solution of the regression equation,  $x^+ = x + v$  for some  $v$  in the null space of  $A$ , so

$$Ax^+ = A(x + v) = Ax + Av = b + 0 = b.$$

This shows  $x^+$  is a solution of the linear system. Since all other solutions differ by a vector  $v$  in the null space of  $A$ , this establishes the result.

Now we can state when  $Ax = b$  is solvable,

### Solvability of $Ax = b$

The linear system  $Ax = b$  is solvable iff  $b = AA^+b$ . When this happens,  $x^+ = A^+b$  is the solution of minimum norm.

If (2.3.1) is solvable, then from above,  $x^+$  is a solution, so

$$AA^+b = A(A^+b) = Ax^+ = b.$$

Conversely, if  $AA^+b = b$ , then clearly  $x^+ = A^+b$  is a solution of (2.3.1).

When (2.3.1) is solvable, (2.3.1) and (2.6.2) have the same solutions, so  $x^+$  is the minimum norm solution of (2.3.1).

For example, let  $b = (-9, -3, 3, 9, 10)$ , and let  $A$  be as in (2.3.4). Since

$$AA^+b = \begin{pmatrix} -8 \\ -3 \\ 2 \\ 7 \\ 12 \end{pmatrix} \quad (2.6.7)$$

is not equal to  $b$ , the linear system (2.6.3) is not solvable.



Suppose  $A$  is invertible. Then (2.3.1) has only the solution  $x = A^{-1}b$ , so  $A^{-1}b$  is the minimum norm residual minimizer. We conclude

### Inverse Equals Pseudo-Inverse

If  $A$  is invertible, then  $A^+ = A^{-1}$ .



The key properties [20] of  $A^+$  are

### Properties of Pseudo-Inverse

- A.  $AA^+A = A$
  - B.  $A^+AA^+ = A^+$
  - C.  $AA^+$  is symmetric
  - D.  $A^+A$  is symmetric
- (2.6.8)

The verification of these properties is very enlightening, so we do it carefully. Let  $u$  be a vector and set  $b = Au$ . Then the residual

$$|Ax - b|^2 = |Ax - Au|^2$$

is minimized at  $x = u$ . Since  $A^+b = A^+Au$  is the minimum norm residual minimizer,  $u$  and  $A^+Au$  differ by a vector  $v$  in the null space of  $A$ ,

$$u = A^+Au + v. \quad (2.6.9)$$

Since  $Av = 0$ , multiplying by  $A$  leads to

$$Au = AA^+Au.$$

Since  $u$  was any vector, this yields A.

Now let  $w$  be a vector and set  $u = A^+w$ . Inserting into (2.6.9) yields

$$A^+w = A^+AA^+w + v$$

for some  $v$  in the null space of  $A$ . But both  $A^+w$  and  $A^+AA^+w$  are in the row space of  $A$ , hence so is  $v$ . Since  $v$  is in both the null space and the row space,  $v$  is orthogonal to itself, so  $v = 0$ . This implies  $A^+AA^+w = A^+w$ . Since  $w$  was any vector, we obtain **B**.

Since  $A^+b$  solves the regression equation,  $A^tAA^+b = A^tb$  for any vector  $b$ . Hence  $A^tAA^+ = A^t$ . Let  $P = AA^+$ . Now

$$P^tP = (AA^+)^t(AA^+) = (A^+)^tA^tAA^+ = (A^+)^tA^t = P^t.$$

Since the left side is symmetric, so is  $P^t$ . Hence  $P$  is symmetric, obtaining **C**.

For any vector  $x$ ,

$$A(x - A^+Ax) = Ax - AA^+Ax = 0,$$

so  $x - A^+Ax$  is in the null space of  $A$ . For any  $y$ ,  $A^+Ay$  is in the row space of  $A$ . Since the row space and the null space are orthogonal,

$$(x - A^+Ax) \cdot A^+Ay = 0.$$

Let  $P = A^+A$ . This implies

$$x \cdot Py = Px \cdot Py = x \cdot P^tPy$$

Since this is true for any vectors  $x$  and  $y$ ,  $P = P^tP$ . This shows  $P = A^+A$  is symmetric, obtaining **D**.

Having arrived at **A**, **B**, **C**, **D**, the reasoning is reversible: It can be shown any matrix  $A^+$  satisfying **A**, **B**, **C**, **D** must equal the pseudo-inverse.



Also we have

### Pseudo-Inverse and Transpose

If  $U$  has orthonormal columns or orthonormal rows, then  $U^+ = U^t$ .

From (2.2.6), such a matrix  $U$  satisfies  $UU^t = I$  or  $U^tU = I$ . In either case, **A**, **B**, **C**, **D** are immediate consequences.

## 2.7 Projections

In this section, we study projection matrices  $P$ , and we show

- $P = AA^+$  is the projection matrix onto the column space of  $A$ ,
- $P = A^+A$  is the projection matrix onto the row space of  $A$ ,
- $P = I - A^+A$  is the projection matrix onto the null space of  $A$ ,

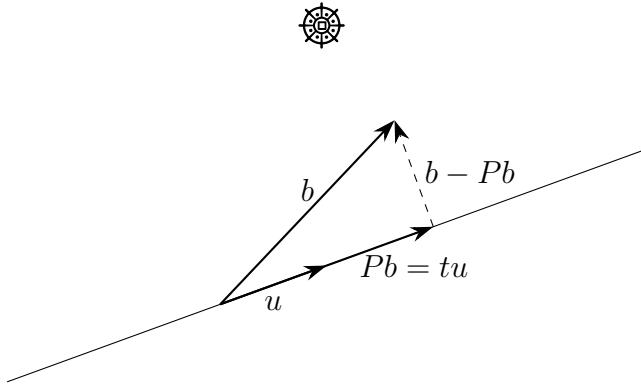


Figure 2.3: Projecting onto a line.

Let  $u$  be a *unit* vector, and let  $b$  be any vector. Let  $\text{span}(u)$  be the line through  $u$  (Figure 2.3). The *projection* of  $b$  onto  $\text{span}(u)$  is the vector  $v$  in  $\text{span}(u)$  that is closest to  $b$ .

It turns out this closest vector  $v$  equals  $Pb$  for some matrix  $P$ , the *projection matrix*. Since  $\text{span}(u)$  is a line, the projected vector  $Pb$  is a multiple  $tu$  of  $u$ .

From Figure 2.3,  $b - Pb$  is orthogonal to  $u$ , so

$$0 = (b - Pb) \cdot u = b \cdot u - Pb \cdot u = b \cdot u - t u \cdot u = b \cdot u - t.$$

Solving for  $t$ , this implies  $t = b \cdot u$ . Thus

$$Pb = (b \cdot u)u = (u \otimes u)b. \quad (2.7.1)$$

Notice  $Pb = b$  when  $b$  is already on the line through  $u$ . In other words, *the projection of a vector onto a line equals the vector itself when the vector*

is already on the line. If  $U$  is the matrix with the single column  $u$ , we obtain  $P = UU^t$ .

To summarize, the *projected vector* is the vector  $(b \cdot u)u$ , and the *reduced vector* is the scalar  $b \cdot u$ . If  $U$  is the matrix with the single column  $u$ , then the reduced vector is  $U^tb$  and the projected vector is  $UU^tb$ .

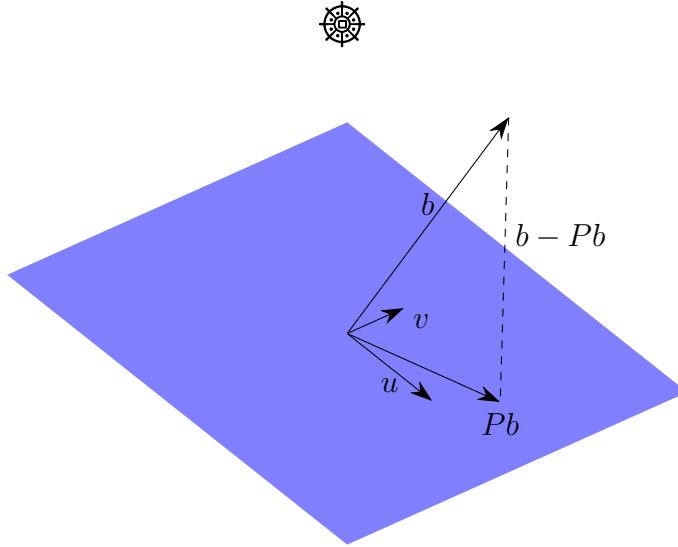


Figure 2.4: Projecting onto a plane,  $Pb = ru + sv$ .

Now we project onto a plane. Let  $u, v$  be an *orthonormal* pair of vectors, so  $u \cdot v = 0$ ,  $u \cdot u = 1 = v \cdot v$ . We project a vector  $b$  onto  $\text{span}(u, v)$ . As before, there is a matrix  $P$ , the *projection matrix*, such that the projection of  $b$  onto the plane equals  $Pb$ . Then  $b - Pb$  is orthogonal to the plane (Figure 2.4), which means  $b - Pb$  satisfies

$$(b - Pb) \cdot u = 0 \quad \text{and} \quad (b - Pb) \cdot v = 0.$$

Since  $Pb$  lies in the plane,  $Pb = ru + sv$  is a linear combination of  $u$  and  $v$ . Inserting  $Pb = ru + sv$ , we obtain

$$r = b \cdot u, \quad s = b \cdot v.$$

If  $U$  is the matrix with columns  $u, v$ , by (2.2.7), this yields,

$$Pb = (b \cdot u)u + (b \cdot v)v = (u \otimes u + v \otimes v)b = UU^tb,$$

As before, here also the projection matrix is  $P = UU^t$ .

Notice  $Pb = b$  when  $b$  is already in the plane. In other words, *the projection of a vector onto a plane equals the vector itself when the vector is already in the plane.*

To summarize, here the *projected vector* is the vector  $UU^t b = (b \cdot u)u + (b \cdot v)v$ , and the *reduced vector* is the vector  $U^t b = (b \cdot u, b \cdot v)$ . The projected vector has the same dimension as the original vector, and the reduced vector has only two components.



We define projection matrices in general. Let  $S$  be a span. A matrix  $P$  is a *projection matrix onto  $S$*  if

1.  $Pb$  is in  $S$  for any vector  $b$ ,
2.  $Pb = b$  if  $b$  is in  $S$ ,
3.  $b - Pb$  is orthogonal to  $S$  for any vector  $b$ .

Let  $A$  be any matrix and let  $S$  be the column space of  $A$ . We show

### Projection Onto a Column Space

The projection matrix onto the column space of  $A$  is

$$P = AA^+. \quad (2.7.2)$$

By definition, the column space  $S$  of  $A$  consists of vectors of the form  $Ax$ . If  $b$  is any vector, then

$$Pb = AA^+b = Ax, \quad \text{with} \quad x = A^+b,$$

so  $Pb$  is in  $S$ . This establishes 1. If  $b$  is in  $S$ , then  $b = Ax$ , so  $Pb = AA^+b = AA^+Ax = Ax = b$ , establishing 2. For 3., let  $x^+ = A^+b$ . Then  $x^+$  is a solution of the regression equation, so

$$(b - Pb) \cdot Av = A^t(b - AA^+b) \cdot v = (A^t b - A^t A x^+) \cdot v = 0,$$

establishing 3.

Now let  $x = A^+b$ . Then  $Ax = AA^+b = Pb$  is the projection of  $v$  onto the column space of  $A$ . If the columns are  $v_1, v_2, \dots, v_d$ , and  $x = (t_1, t_2, \dots, t_d)$ , then by matrix-vector multiplication,

$$Pv = t_1v_1 + t_2v_2 + \cdots + t_dv_d.$$

So the reduced vector  $x$  consists of the coefficients when writing  $Pv$  as a linear combination of the columns.

```
from numpy import *
from numpy.linalg import pinv

# projection of column vector b
# onto column space of A

def project(A,b):
    Aplus = pinv(A)
    x = dot(Aplus,b)      # reduced
    return dot(A,x)       # projected
```

### Projected and Reduced Vectors

Let  $A$  be a matrix and  $v$  a vector. Then the projected vector is  $Pv = AA^+v$  and the reduced vector is  $x = A^+v$ .

For  $A$  as in (2.3.4) and  $b = (-9, -3, 3, 9, 10)$  the reduced vector onto the column space of  $A$  is

$$x = A^+b = \frac{1}{15}(82, 25, -32),$$

and the projected vector onto the column space of  $A$  is

$$Pb = Ax = AA^+b = (-8, -3, 2, 7, 12).$$

The projection matrix onto the column space of  $A$  is

$$P = AA^+ = \frac{1}{10} \begin{pmatrix} 6 & 4 & 2 & 0 & -2 \\ 4 & 3 & 2 & 1 & 0 \\ 2 & 2 & 2 & 2 & 2 \\ 0 & 1 & 2 & 3 & 4 \\ -2 & 0 & 2 & 4 & 6 \end{pmatrix}.$$



In the same way, one can show

### Projection Onto a Row Space

The projection matrix onto the row space of  $A$  is

$$P = A^+ A. \quad (2.7.3)$$

For  $A$  as in (2.3.4), the projection matrix onto the row space is

$$P = A^+ A = \frac{1}{6} \begin{pmatrix} 5 & 2 & -1 \\ 2 & 2 & 2 \\ -1 & 2 & 5 \end{pmatrix}$$



When the columns of a matrix  $U$  are orthonormal, in the previous section we saw  $U^+ = U^t$ , so we have

### Projection onto Orthonormal Vectors

If the columns of  $U$  are orthonormal, the projection matrix onto the column space of  $U$  is

$$P = UU^t \quad (2.7.4)$$

Here the projected vector is  $UU^t b$ , and the reduced vector is  $U^t b$ . The code here is

```
from numpy import *

# projection of column vector b
# onto column space of U
# with orthonormal columns

def project_to_ortho(U,b):
    x = dot(U.T,b)      # reduced
    return dot(U,x)     # projected
```



Let  $v_1, v_2, \dots, v_N$  be a dataset in  $\mathbf{R}^d$ , and let  $U$  be a  $d \times n$  matrix with orthonormal columns. Then the projection matrix onto the column space of  $U$  is  $P = UU^t$ , and  $P$  is the projection onto an orthonormal span.

In this case, the dataset  $U^t v_1, U^t v_2, \dots, U^t v_N$  is the *reduced dataset*, and  $UU^t v_1, UU^t v_2, \dots, UU^t v_N$  is the *projected dataset*.

The projected dataset is in  $\mathbf{R}^d$ , and the reduced dataset is in  $\mathbf{R}^n$ . Table 2.5 summarizes the relationships.

dataset	$v_k$ in $\mathbf{R}^d$ , $k = 1, 2, \dots, N$
reduced	$U^t v_k$ in $\mathbf{R}^n$ , $k = 1, 2, \dots, N$
projected	$UU^t v_k$ in $\mathbf{R}^d$ , $k = 1, 2, \dots, N$

Table 2.5: Dataset, reduced dataset, and projected dataset,  $n < d$ .



Let  $S$  and  $T$  be spans. Let  $S + T$  consist of all sums of vectors  $u + v$  with  $u$  in  $S$  and  $v$  in  $T$ . Then a moment's thought shows  $S + T$  is itself a span. When the intersection of  $S$  and  $T$  is the zero vector, we write  $S \oplus T$ , and we say  $S \oplus T$  is the *direct sum* of  $S$  and  $T$ .

Let  $S$  be a span and let  $S^\perp$  consist of all vectors orthogonal to  $S$ . We call  $S^\perp$  the *orthogonal complement*. This is pronounced “ $S$ -perp”. If  $v$  is in both  $S$  and in  $S^\perp$ , then  $v$  is orthogonal to itself, hence  $v = 0$ . From this, we see  $S + S^\perp$  is a direct sum  $S \oplus S^\perp$ .

### Direct Sum and Orthogonal Complement

If  $S$  is a span in  $\mathbf{R}^d$ , then

$$\mathbf{R}^d = S \oplus S^\perp. \quad (2.7.5)$$

This is an immediate consequence of what we already know. Let  $P$  be the projection matrix onto  $S$ . Since any vector  $v$  in  $\mathbf{R}^d$  may be written

$$v = Pv + (v - Pv),$$

we see any vector is a sum of a vector in  $S$  and a vector in  $S^\perp$ .



An important example of (2.7.5) is the relation between the row space and the null space of a matrix. In §2.4, we saw that, for any matrix  $A$ , the row space and the null space are orthogonal complements.

Taking  $S = \text{nullspace}$  in (2.7.5), we have the important

### Null space plus Row Space Equals Source Space

If  $A$  is an  $N \times d$  matrix,

$$\text{nullspace} \oplus \text{rowspace} = \mathbf{R}^d, \quad (2.7.6)$$

and the null space and row space are orthogonal to each other.

From this, the projection onto the null space of  $A$  is

$$P = I - A^+ A. \quad (2.7.7)$$

For  $A$  as in (2.3.4), the projection matrix onto the null space is

$$P = I - A^+ A = \frac{1}{6} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$



Let  $S$  be the column space of a matrix  $A$ , and let  $P$  be the projection matrix onto  $S$ . We end the section by establishing the claim made at the start of the section, that  $Pb$  is the point in  $S$  that is closest to  $b$ .

Since every point in  $S$  is of the form  $Ax$ , we need to check

$$|Pb - b|^2 = \min_x |Ax - b|^2.$$

But this was already done in §2.3, since  $Pb = AA^+b = Ax^+$  where  $x^+ = A^+b$  is a residual minimizer.

### Projection is the Nearest Point in the Span

Let  $Pb = AA^+b$  be the projection of  $b$  onto the column space of  $A$ , and let  $x^+ = A^+b$  be the reduced vector. Then

$$|Ax^+ - b|^2 = \min_x |Ax - b|^2. \quad (2.7.8)$$

## 2.8 Basis

Let  $S$  be the span of vectors  $v_1, v_2, \dots, v_N$ . Then there are many other choices of spanning vectors for  $S$ . For example,  $v_1 + v_2, v_2, v_3, \dots, v_N$  also spans  $S$ .

If  $S$  cannot be spanned by fewer than  $N$  vectors, then we say  $v_1, v_2, \dots, v_N$  is a *basis* for  $S$ , and we call  $N$  is the *dimension* of  $S$ .

In other words, when  $N$  is the smallest number of spanning vectors, we say  $N$  is the dimension  $\dim S$  of  $S$ , and  $v_1, v_2, \dots, v_N$  is a *minimal spanning set* for  $S$ . This definition is important enough to repeat,

### Basis and Dimension Definition

A basis for a span  $S$  is a minimal spanning set of vectors. The dimension of  $S$  is the number of vectors in *any* basis for  $S$ .

To clarify this definition, suppose someone asks “Who is the shortest person in the room?” There may be several shortest people in the room, but, no matter how many shortest people there are, there is only one shortest height. In other words, a span may have several bases, but a span’s dimension is uniquely determined.

When a basis  $v_1, v_2, \dots, v_N$  consists of orthogonal vectors, we say  $v_1, v_2, \dots, v_N$  is an *orthogonal basis*. When  $v_1, v_2, \dots, v_N$  are also unit vectors, we say  $v_1, v_2, \dots, v_N$  is an *orthonormal basis*.

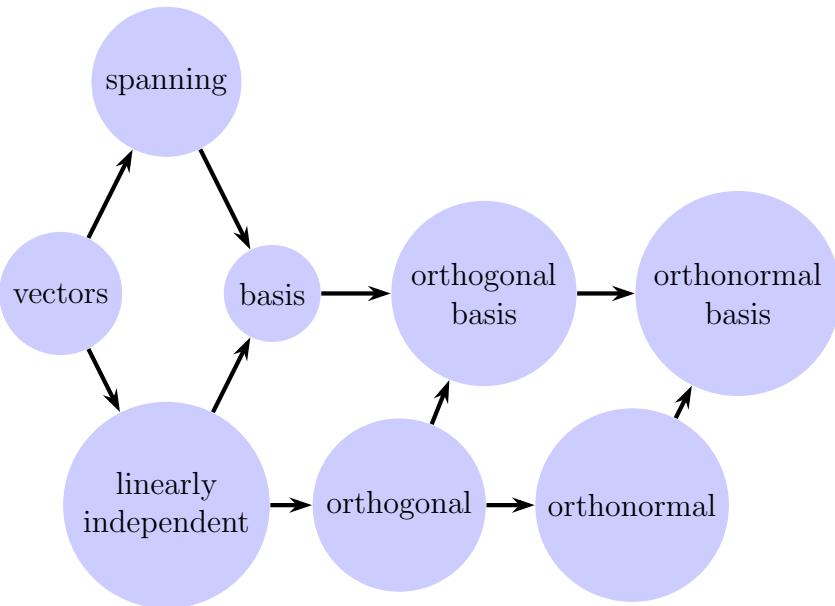


Figure 2.6: Relations between vector classes.

Here are two immediate consequences of this terminology.

### Span of $N$ Vectors

If  $S = \text{span}(v_1, v_2, \dots, v_N)$ , then  $\dim S \leq N$ .

### Larger Span has Larger Dimension

If a span  $S_1$  is contained in a span  $S_2$ , then  $\dim S_1 \leq \dim S_2$ .

With this terminology,

- `rowspace()` returns a basis of the row space,
- `columnspace()` returns a basis of the column space,
- `nullspace()` returns a basis for the null space,
- row rank equals the dimension of the row space,
- column rank equals the dimension of the column space,

- nullity equals the dimension of the null space.



Let  $S$  be the span of vectors  $v_1, v_2, \dots, v_N$ . How can we check if these vectors constitute a basis for  $S$ ? The answer is the main result of the section.

### Spanning Plus Linearly Independent Equals Basis

Let  $S$  be the span of vectors  $v_1, v_2, \dots, v_N$ . Then the vectors are a basis for  $S$  if and only if they are linearly independent.

Remember, to check for linear independence of given vectors, assemble the vectors as columns of a matrix  $A$ , and check whether `A.nullspace()` equals zero. If that is the case, the vectors are a basis for their span. If not, the vectors are not a basis for their span. The proof of the main result is at the end of the section.



Here is an example. Let  $e_1 = (1, 0, 0)$ ,  $e_2 = (0, 1, 0)$ ,  $e_3 = (0, 0, 1)$ . We have just seen that  $e_1, e_2, e_3$  are linearly independent. Hence  $e_1, e_2, e_3$  is a basis for  $\mathbf{R}^3$ , which means  $e_1, e_2, e_3$  is a minimal spanning set of vectors for  $\mathbf{R}^3$ . From this, we conclude  $\dim \mathbf{R}^3 = 3$ .

The statement  $\dim \mathbf{R}^3 = 3$  may at first seem trivial or obvious. But, if we flesh this out following our terminology above, the statement is saying that any minimal spanning set of vectors in  $\mathbf{R}^3$  must have exactly 3 vectors. Stated in this manner, the statement has content.

Since we can do the same calculation with the standard basis

$$\begin{aligned} e_1 &= (1, 0, \dots, 0), \\ e_2 &= (0, 1, 0, \dots, 0), \\ \dots &= \dots \\ e_d &= (0, 0, \dots, 0, 1), \end{aligned}$$

in  $\mathbf{R}^d$ , we conclude  $e_1, e_2, \dots, e_d$  are linearly independent, so

### Dimension of Euclidean Space

The dimension of  $\mathbf{R}^d$  is  $d$ .



The MNIST dataset consists of vectors  $v_1, v_2, \dots, v_N$  in  $\mathbf{R}^d$ , where  $N = 60000$  and  $d = 784$ . For the MNIST dataset, the dimension is 712, as returned by the code

```
from numpy.linalg import matrix_rank
matrix_rank(vectors)
```

In particular, since  $712 < 784$ , approximately 10% of pixels are never touched by any image. For example, a likely pixel to remain untouched is at the top left corner  $(0, 0)$ . For this dataset, there are  $72 = 784 - 712$  zero variance directions.

We pose the following question: What is the least  $n$  for which the first  $n$  images are linearly dependent? Since the dimension of the feature space is 784, we must have  $n \leq 784$ . To answer the question, we compute the rank of the first  $n$  vectors for  $n = 1, 2, 3, \dots$ , and continue until we have linear dependence of  $v_1, v_2, \dots, v_n$ .

If we save the MNIST dataset as a centered array `vectors`, as in §2.1, and run the code below, we obtain  $n = 560$  (Figure 2.7). `matrix_rank` is discussed in §2.9.

```
from numpy import *
from numpy.linalg import matrix_rank

# vectors as Nxd array

def find_first_defect(vectors):
    d = len(vectors[0])
    previous = 0
    for n in range(len(vectors)):
        r = matrix_rank(vectors[:n+1, :])
```

```

print((r,n+1),end=",")
if r == previous: break
if r == d: break
previous = r

```

```

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 4
2, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 11
4, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 14
3, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 17
2, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 20
1, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 23
0, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 25
9, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 28
8, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 31
7, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 34
6, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 37
5, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 40
4, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 43
3, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 46
2, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 49
1, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 52
0, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 54
9, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 560

```

Figure 2.7: First defect for MNIST.



Let  $v_1, v_2, \dots, v_N$  be a dataset. We want to compute the dimensions of the first  $k$  vectors,

$$d_1 = \dim(v_1), \quad d_2 = \dim(v_1, v_2), \quad d_3 = \dim(v_1, v_2, v_3), \quad \text{and so on}$$

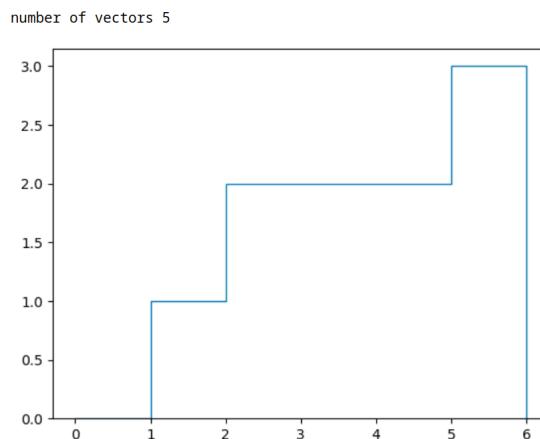


Figure 2.8: The dimension staircase with defects.

This we call the *dimension staircase*. For example, Figure 2.8 is the dimension staircase for

$$v_1 = (1, 0, 0), v_2 = (0, 1, 0), v_3 = (1, 1, 0), v_4 = (3, 4, 0), v_5 = (0, 0, 1).$$

In Figure 2.8, we call the points  $(3, 2)$  and  $(4, 2)$  *defects*.

In the code, the staircase is drawn by `stairs(X, Y)`, where the horizontal points `X` and the vertical values `Y` satisfy `len(X) == len(Y)+1`. In Figure 2.8, `X = [1, 2, 3, 4, 5, 6]`, and `Y = [1, 2, 2, 2, 3]`.

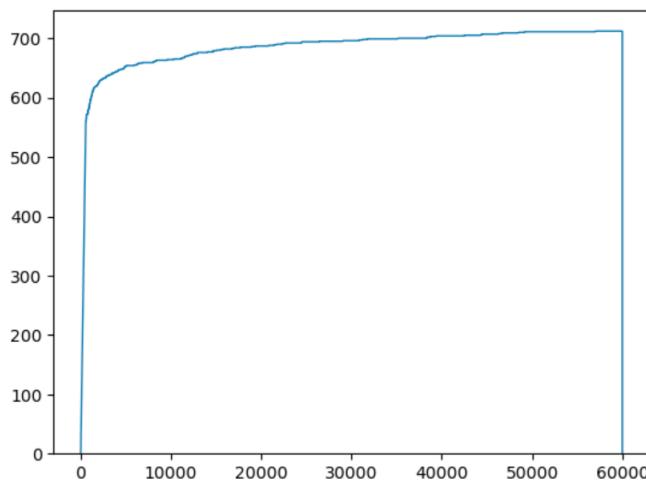


Figure 2.9: The dimension staircase for the MNIST dataset.

With the MNIST dataset loaded as `vectors`, here is code returning Figure 2.9. This code is not efficient, but it works. It takes 57041 vectors in the dataset to fill up 712 dimensions.

```
from matplotlib.pyplot import *
from numpy.linalg import matrix_rank

# vectors as Nxd array

def dimension_staircase(vectors):
    d = vectors[0].size
    N = len(vectors)
    rmax = matrix_rank(vectors)
```

```

dimensions = [ ]
basis = []
for n in range(1,N):
    r = matrix_rank(vectors[:n,:])
    print((r,n),end=",")
    dimensions.append(r)
    if r == rmax: break
stairs(dimensions, range(n+1))

```



*Proof of main result.* Here we derive: *Let  $S$  be the span of  $v_1, v_2, \dots, v_N$ . Then  $v_1, v_2, \dots, v_N$  is a basis for  $S$  if and only if  $v_1, v_2, \dots, v_N$  are linearly independent.*

Suppose  $v_1, v_2, \dots, v_N$  are not linearly independent. Then  $v_1, v_2, \dots, v_N$  are linearly dependent, which means one of the vectors, say  $v_1$ , is a linear combination of the other vectors  $v_2, v_3, \dots, v_N$ . Then any linear combination of  $v_1, v_2, \dots, v_N$  is necessarily a linear combination of  $v_2, v_3, \dots, v_N$ , thus

$$\text{span}(v_1, v_2, \dots, v_N) = \text{span}(v_2, v_3, \dots, v_N).$$

This shows  $v_1, v_2, \dots, v_N$  is not a minimal spanning set, and completes the derivation in one direction.

In the other direction, suppose  $v_1, v_2, \dots, v_N$  are linearly independent, and suppose  $b_1, b_2, \dots, b_d$  is a minimal spanning set. Since  $b_1, b_2, \dots, b_d$  is minimal, we must have  $d \leq N$ . Once we establish  $d = N$ , it follows  $v_1, v_2, \dots, v_N$  is minimal, and the proof will be complete.

Since by assumption,

$$\text{span}(v_1, v_2, \dots, v_N) = \text{span}(b_1, b_2, \dots, b_d),$$

$v_1$  is a linear combination of  $b_1, b_2, \dots, b_d$ ,

$$v_1 = t_1 b_1 + t_2 b_2 + \cdots + t_d b_d.$$

Since  $v_1 \neq 0$ , at least one of the coefficients, say  $t_1$ , is not zero, so we can solve

$$b_1 = \frac{1}{t_1} (v_1 - t_2 b_2 - t_3 b_3 - \cdots - t_d b_d).$$

This shows

$$\text{span}(v_1, v_2, \dots, v_N) = \text{span}(v_1, b_2, b_3, \dots, b_d).$$

Repeating the same logic,  $v_2$  is a linear combination of  $v_1, b_2, b_3, \dots, b_d$ ,

$$v_2 = s_1 v_1 + t_2 b_2 + t_3 b_3 + \cdots + t_d b_d.$$

If all the coefficients of  $b_2, b_3, \dots, b_d$  are zero, then  $v_2$  is a multiple of  $v_1$ , contradicting linear independence of  $v_1, v_2, \dots, v_N$ . Thus there is at least one coefficient, say  $t_2$ , which is not zero. Solving for  $b_2$ , we obtain

$$b_2 = \frac{1}{t_2} (v_2 - s_1 v_1 - t_3 b_3 - \cdots - t_d b_d).$$

This shows

$$\text{span}(v_1, v_2, \dots, v_N) = \text{span}(v_1, v_2, b_3, b_4, \dots, b_d).$$

Repeating the same logic,  $v_3$  is a linear combination of  $v_1, v_2, b_3, b_4, \dots, b_d$ ,

$$v_3 = s_1 v_1 + s_2 v_2 + t_3 b_3 + t_4 b_4 + \cdots + t_d b_d.$$

If all the coefficients of  $b_3, b_4, \dots, b_d$  are zero, then  $v_3$  is a linear combination of  $v_1, v_2$ , contradicting linear independence of  $v_1, v_2, \dots, v_N$ . Thus there is at least one coefficient, say  $t_3$ , which is not zero. Solving for  $b_3$ , we obtain

$$b_3 = \frac{1}{t_3} (v_3 - s_1 v_1 - s_2 v_2 - t_4 b_4 - \cdots - t_d b_d).$$

This shows

$$\text{span}(v_1, v_2, \dots, v_N) = \text{span}(v_1, v_2, v_3, b_4, b_5, \dots, b_d).$$

Continuing in this manner, we eventually arrive at

$$\text{span}(v_1, v_2, \dots, v_N) = \cdots = \text{span}(v_1, v_2, \dots, v_d).$$

This shows  $v_N$  is a linear combination of  $v_1, v_2, \dots, v_d$ . This shows  $N = d$ , because  $N > d$  contradicts linear independence. Since  $d$  is the minimal spanning number, this shows  $v_1, v_2, \dots, v_N$  is a minimal spanning set for  $S$ .

## 2.9 Rank

If  $A$  is an  $N \times d$  matrix, then (Figure 2.10)  $x \mapsto Ax$  is a linear transformation that sends a vector  $x$  in  $\mathbf{R}^d$  (the source space) to the vector  $Ax$  in  $\mathbf{R}^N$  (the target space). The transpose  $A^t$  goes in the reverse direction: The linear transformation  $b \mapsto A^t b$  sends a vector  $b$  in  $\mathbf{R}^N$  (the target space) to the vector  $A^t b$  in  $\mathbf{R}^d$  (the source space).

It follows that for an  $N \times d$  matrix, the dimension of the source space is  $d$ , and the dimension of the target space is  $N$ ,

$$\dim(\text{source space}) = d, \quad \dim(\text{target space}) = N.$$

```
from sympy import *

d = A.cols # source space dimension
N = A.rows # target space dimension
```

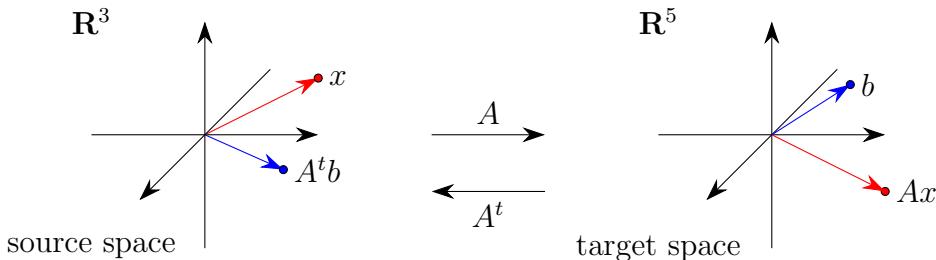


Figure 2.10: A  $5 \times 3$  matrix  $A$  is a linear transformation from  $\mathbf{R}^3$  to  $\mathbf{R}^5$ .

By (2.4.2), the column space is in the target space, and the row space is in the source space. Thus we always have

$$0 \leq \text{row rank} \leq d \quad \text{and} \quad 0 \leq \text{column rank} \leq N.$$

For  $A$  as in (2.3.4), the column rank is 2, the row rank is 2, and the nullity is 1. Thus the column space is a 2-d plane in  $\mathbf{R}^5$ , the row space is a 2-d plane in  $\mathbf{R}^3$ , and the null space is a 1-d line in  $\mathbf{R}^3$ .



The main result in this section is

### Rank Theorem

Let  $A$  be any matrix. Then

$$\text{row rank}(A) = \text{column rank}(A). \quad (2.9.1)$$

This is established at the end of the section.

Because the row rank and the column rank are equal, below we just say *rank* of a matrix, and we write  $\text{rank}(A)$ . In Python,

```
from sympy import *
A.rank()
from numpy.linalg import matrix_rank
matrix_rank(A)
```

returns the rank of a matrix. The main result implies  $\text{rank}(A) = \text{rank}(A^t)$ , so

### Upper bound for Rank

For any  $N \times d$  matrix, the rank is never greater than  $\min(N, d)$ .



An  $N \times d$  matrix  $A$  is *full-rank* if its rank is the highest it can be,  $\text{rank}(A) = \min(N, d)$ . Here are some consequences of the main result.

- When  $N \geq d$ , full-rank is the same as  $\text{rank}(A) = d$ , which is the same as saying the columns are linearly independent and the rows span  $\mathbf{R}^d$ .
- When  $N \leq d$ , full-rank is the same as  $\text{rank}(A) = N$ , which is the same as saying the rows are linearly independent and the columns span  $\mathbf{R}^N$ .
- When  $N = d$ , full-rank is the same as saying the rows are a basis of  $\mathbf{R}^d$ , and the columns are a basis of  $\mathbf{R}^N$ .

When  $A$  is a square matrix, we can say more:

### Full Rank Square Equals Invertible

Let  $A$  be a *square* matrix. Then  $A$  is full-rank iff  $A$  is invertible.

Suppose  $A$  is  $d \times d$ . If  $A$  is invertible and  $B$  is its inverse, then  $AB = I$ . Since  $ABx = A(Bx) = Ay$  with  $y = Bx$ , the column space of  $AB$  is contained in the column space of  $A$ . Since the column space of  $AB = I$  is  $\mathbf{R}^d$ , we conclude the column space of  $A$  is  $\mathbf{R}^d$ , thus  $\text{rank}(A) = d$ .

Conversely, suppose  $A$  is full-rank. This means the columns of  $A$  span  $\mathbf{R}^d$ . By (2.4.3), this implies

$$Ax = b$$

is solvable for any  $b$ . Let  $e_1, e_2, \dots, e_d$  be the standard basis. If we set successively  $b = e_1, b = e_2, \dots, b = e_d$ , we then get solutions  $x_1, x_2, \dots, x_d$ . If  $B$  is the matrix with columns  $x_1, x_2, \dots, x_d$ , then

$$AB = A[x_1, x_2, \dots, x_d] = [Ax_1, Ax_2, \dots, Ax_d] = [e_1, e_2, \dots, e_d] = I.$$

Thus we found a matrix  $B$  satisfying  $AB = I$ .

Repeating the same argument with rows instead of columns, we find a matrix  $C$  satisfying  $CA = I$ . Then

$$C = CI = CAB = IB = B,$$

so  $B = C$  is the inverse of  $A$ .



### Orthonormal Rows and Columns

Let  $U$  be a matrix.

- $U$  has orthonormal rows iff  $UU^t = I$ .
- $U$  has orthonormal columns iff  $U^tU = I$ .

If  $U$  is square and either holds, then they both hold.

The first two assertions are in §2.2. For the last assertion, assume  $U$  is a square matrix. From §2.4, orthonormality of the rows implies linear

independence of the rows, so  $U$  is full-rank. If  $U$  also is a square matrix, then  $U$  is invertible. Multiply by  $U^{-1}$ ,

$$U^{-1} = U^{-1}I = U^{-1}UU^t = U^t.$$

Since we have  $U^{-1}U = I$ , we also have  $U^tU = I$ .



A square matrix  $U$  satisfying

$$UU^t = I = U^tU \quad (2.9.2)$$

is an *orthogonal matrix*.

Equivalently, we can say

### Orthogonal Matrix

A matrix  $U$  is orthogonal iff its rows are an orthonormal basis iff its columns are an orthonormal basis.

Since

$$Uu \cdot Uv = u \cdot U^tUv = u \cdot v,$$

$U$  preserves dot products. Since lengths are dot products,  $U$  also preserves lengths. Since angles are computed from dot products,  $U$  also preserves angles. Summarizing,

### Angles, Lengths, and Dot Products

Orthogonal Matrices Preserve Angles, Lengths, and Dot Products:

As a consequence,

### Orthogonal Matrix sends ON Vectors to ON Vectors

Let  $U$  be an orthogonal matrix. If  $v_1, v_2, \dots, v_d$  are orthonormal, then  $Uv_1, Uv_2, \dots, Uv_d$  are orthonormal.

In two dimensions,  $d = 2$ , an orthogonal matrix must have two orthonormal columns, so must be of the form

$$U = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \quad \text{or} \quad U = \begin{pmatrix} \cos \theta & \sin \theta \\ \sin \theta & -\cos \theta \end{pmatrix}.$$

In the first case,  $U$  is a rotation, while in the second,  $U$  is a rotation followed by a reflection.



If  $v_1, v_2, \dots, v_d$  is an orthonormal basis of  $\mathbf{R}^d$ , and  $U$  has columns  $v_1, v_2, \dots, v_d$ , then  $U$  is square and  $UU^t = I = U^tU$ . By (2.2.7), we have

$$I = v_1 \otimes v_1 + v_2 \otimes v_2 + \cdots + v_d \otimes v_d.$$

Multiplying both sides by  $v$ , by (1.4.14), we obtain

### Orthonormal Basis Expansion

If  $v_1, v_2, \dots, v_d$  is an orthonormal basis, and  $v$  is any vector, then

$$v = (v \cdot v_1)v_1 + (v \cdot v_2)v_2 + \cdots + (v \cdot v_d)v_d \quad (2.9.3)$$

and

$$|v|^2 = |v \cdot v_1|^2 + |v \cdot v_2|^2 + \cdots + |v \cdot v_d|^2. \quad (2.9.4)$$



Let  $x_1, x_2, \dots, x_N$  be a dataset, and let  $A$  be the dataset matrix with rows  $x_1, x_2, \dots, x_N$ . The dataset is *full-rank* if  $A$  is full-rank. This is the same as saying the span of  $x_1, x_2, \dots, x_N$  is the whole feature space.

The *dimension* of the dataset is the rank of  $A$ . Hence the dimension of the dataset equals the rank of  $A^t A$ . When the dataset is centered, the covariance is the matrix  $Q = A^t A / N$ . Since scaling a matrix has no effect on the rank, we conclude *the dimension of a dataset equals the rank of its covariance*.



To derive the main result, first we recall (2.7.6). From the definition of dimension, we can rewrite (2.7.6) as

### Row Rank plus Nullity equals Source Space Dimension

For any matrix, the row rank plus the nullity equals the dimension of the source space. If the matrix is  $N \times d$ ,  $r$  is the rank, and  $n$  is the nullity, then

$$r + n = d.$$

Assume  $A$  has  $N$  rows and  $d$  columns. By (2.7.6), every vector  $x$  in the source space  $\mathbf{R}^d$  can be written as a sum  $x = u + v$  with  $u$  in the null space, and  $v$  in the row space. In other words, each vector  $x$  may be written as a sum  $x = u + v$  with  $Au = 0$  and  $v$  in the row space.

From this, we have

$$Ax = A(u + v) = Au + Av = Av.$$

This shows the column space consists of vectors of the form  $Av$  with  $v$  in the row space.

Let  $v_1, v_2, \dots, v_r$  be a basis for the row space. From the previous paragraph, it follows  $Av_1, Av_2, \dots, Av_r$  spans the column space of  $A$ . We claim  $Av_1, Av_2, \dots, Av_r$  are linearly independent. To check this, we write

$$0 = t_1Av_1 + t_2Av_2 + \cdots + t_rAv_r = A(t_1v_1 + t_2v_2 + \cdots + t_rv_r).$$

If  $v$  is the vector  $t_1v_1 + t_2v_2 + \cdots + t_rv_r$ , this shows  $v$  is in the null space. But  $v$  is a linear combination of basis vectors of the row space, so  $v$  is also in the row space. Since the row space is the orthogonal complement of the null space, we must have  $v$  orthogonal to itself. Thus  $v = 0$ , or  $t_1v_1 + t_2v_2 + \cdots + t_rv_r = 0$ .

But  $v_1, v_2, \dots, v_r$  is a basis. By linear independence of  $v_1, v_2, \dots, v_r$ , we conclude  $t_1 = 0, \dots, t_r = 0$ . This establishes the claim, hence  $Av_1, Av_2, \dots, Av_r$  is a basis for the column space. This shows  $r$  is the dimension of the column space, which is by definition the column rank. Since by construction,  $r$  is also the row rank, this establishes the rank theorem.

# Chapter 3

## Principal Components

In this chapter, we look at the two fundamental methods of breaking or decomposing a matrix into elementary components, the eigenvalue decomposition and the singular value decomposition, then we apply this to principal component analysis.

We begin by looking at the geometry of a matrix as a linear transformation.

### 3.1 Geometry of Matrices

Matrix multiplication by an  $N \times d$  matrix  $A$  sends a point  $x$  in the source space  $\mathbf{R}^d$  to a point  $b = Ax$  in the target space  $\mathbf{R}^N$  (Figure 2.10).

Equivalently, since points in  $\mathbf{R}^d$  are essentially the same as vectors in  $\mathbf{R}^d$  (see §1.3), an  $N \times d$  matrix  $A$  sends a vector  $v$  in  $\mathbf{R}^d$  to a vector  $Av$  in  $\mathbf{R}^N$ .

Looked at this way, a matrix  $A$  induces a *linear transformation*: Matrix multiplication by  $A$  satisfies

$$A(v_1 + v_2) = Av_1 + Av_2, \quad A(tv) = tAv.$$

One way to understand what the transformation does is to see how it distorts distances between vectors. If  $v_1$  and  $v_2$  are in  $\mathbf{R}^d$ , then the distance between them is  $d = |v_1 - v_2|$  (recall  $|v|$  denotes the euclidean length of  $v$ ). How does this compare with the distance between  $Av_1$  and  $Av_2$ , or  $|Av_1 - Av_2|$ ?

If we let

$$u = \frac{v_1 - v_2}{|v_1 - v_2|},$$

then  $u$  is a unit vector,  $|u| = 1$ , and by linearity

$$|Au| = \frac{|Av_1 - Av_2|}{|v_1 - v_2|}.$$

This ratio is a scaling factor of the linear transformation. Of course this scaling factor depends on the given vectors  $v_1, v_2$ .

From this, to understand the scaling distortions, it is enough to understand what multiplication by  $A$  does to unit vectors  $u$ .

The first step in understanding this is to compute

$$\sigma_1 = \max |Au| \quad \text{and} \quad \sigma_2 = \min |Au|.$$

Here the maximum and minimum are taken over all unit vectors  $u$ .

Then  $\sigma_1$  is the distance of the furthest image from the origin, and  $\sigma_2$  is the distance of the nearest image to the origin. It turns out  $\sigma_1$  and  $\sigma_2$  are the top and bottom singular values of  $A$ .



To keep things simple, assume both the source space and the target space are  $\mathbf{R}^2$ ; then  $A$  is  $2 \times 2$ .

The *unit circle* (in red in Figure 3.1) is the set of vectors  $u$  satisfying  $|u| = 1$ . The *image* of the unit circle (also in red in Figure 3.1) is the set of vectors of the form

$$\{Au : |u| = 1\}.$$

The *annulus* is the set (the region between the dashed circles in Figure 3.1) of vectors  $b$  satisfying

$$\{b : \sigma_2 < |b| < \sigma_1\}.$$

It turns out the image is an ellipse, and this ellipse lies in the annulus.

Thus the numbers  $\sigma_1$  and  $\sigma_2$  constrain how far the image of the unit circle is from the origin, and how near the image is to the origin.

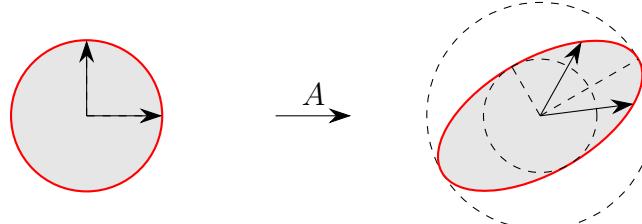


Figure 3.1: Image of unit circle with  $\sigma_1 = 1.5$  and  $\sigma_2 = .75$ .

To relate  $\sigma_1$  and  $\sigma_2$  to what we've seen before, let  $Q = A^t A$ . Then,

$$\sigma_1^2 = \max |Au|^2 = \max(Au) \cdot (Au) = \max u \cdot A^t Au = \max u \cdot Qu.$$

Thus  $\sigma_1^2$  is the maximum projected variance corresponding to the covariance  $Q$ . Similarly,  $\sigma_2^2$  is the minimum projected variance corresponding to the covariance  $Q$ .

Now let  $Q = AA^t$ , and let  $b$  be in the image. Then  $b = Au$  for some unit vector  $u$ , and

$$b \cdot Q^{-1} b = (Au) \cdot Q^{-1} Au = u \cdot A^t (AA^t)^{-1} Au = u \cdot I u = |u|^2 = 1.$$

This shows the image of the unit circle is the inverse covariance ellipse (§1.6) corresponding to the covariance  $Q$ , with major axis length  $2\sigma_1$  and minor axis length  $2\sigma_2$ .



Let us look at some special cases.

The first example is

$$V = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}. \quad (3.1.1)$$

If  $e_1 = (1, 0)$ ,  $e_2 = (0, 1)$  is the standard basis in  $\mathbf{R}^2$ . then the columns of  $V$  are

$$Ve_1 = (\cos \theta, \sin \theta), \quad \text{and} \quad Ve_2 = (-\sin \theta, \cos \theta).$$

Since  $V^t V = I$ , the columns of  $V$  are orthonormal. Thus  $V$  transforms the orthonormal basis  $e_1, e_2$  into the orthonormal basis  $Ve_1, Ve_2$  (see §2.9). By (1.4.3),  $V$  is a rotation by the angle  $\theta$ .

The second example is

$$S = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}.$$

Then  $S$  scales the horizontal direction by the factor  $\sigma_1$ , and  $S$  scales the vertical direction by  $\sigma_2$ .

The third example are the reflections

$$R = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

These reflect vectors across the horizontal axis, and across the vertical axis.

Recall an orthogonal matrix is a matrix  $U$  satisfying  $U^t U = I = U U^t$  (2.9.2). Every orthogonal matrix  $U$  is a rotation  $V$  or a rotation times a reflection  $VR$ .



The SVD decomposition (§3.3) states that every matrix  $A$  can be written as a product

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = USV.$$

Here  $S$  is a diagonal matrix as above, and  $U, V$  are orthogonal and rotation matrices as above.

In more detail, apart from a possible reflection, there are scalings  $\sigma_1$  and  $\sigma_2$  and angles  $\alpha$  and  $\beta$ , so that  $A$  transforms vectors by first rotating by  $\alpha$ , then scaling by  $(\sigma_1, \sigma_2)$ , then by rotating by  $\beta$  (Figure 3.2).

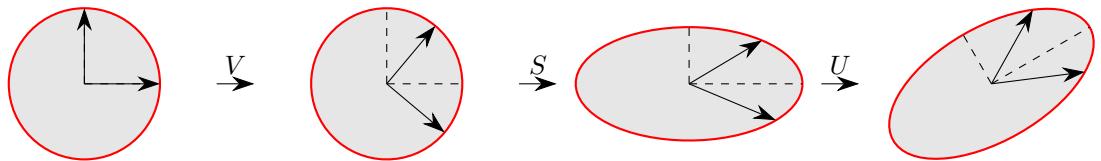


Figure 3.2: SVD decomposition  $A = USV$ .

In other words, each  $2 \times 2$  matrix  $A$ , consisting of four numbers  $a, b, c, d$ , may be described by four other numbers. These other numbers present a much clearer picture of the geometry of  $A$ : two angles  $\alpha, \beta$ , and two scalings  $\sigma_1, \sigma_2$ .

Everything in this section generalizes to any  $N \times d$  matrix, as we see in the coming sections.

## 3.2 Eigenvalue Decomposition

Let  $A$  be a matrix. An *eigenvector* for  $A$  is a *nonzero* vector  $v$  such that  $Av$  is aligned with  $v$ . This means

$$Av = \lambda v \tag{3.2.1}$$

for some scalar  $\lambda$ , the corresponding *eigenvalue*.

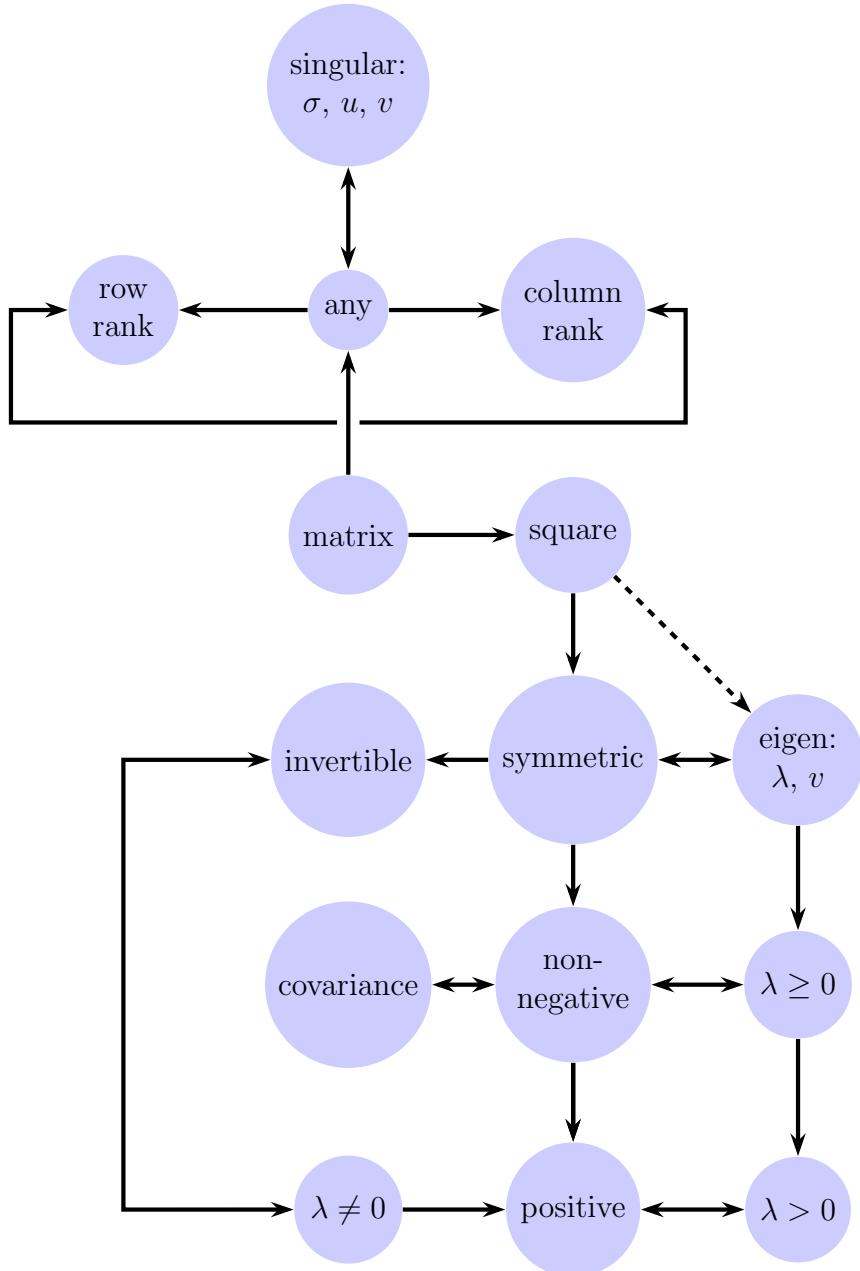


Figure 3.3: Relations between matrix classes.

Because the solution  $v = 0$  of (3.2.1) is not useful, we insist eigenvectors be nonzero. If  $v$  is an eigenvector, then the dimension of  $v$  equals the dimension of  $Av$ , which can only happen when  $A$  is a square matrix.

If  $v$  is an eigenvector corresponding to eigenvalue  $\lambda$ , then any scalar multiple  $u = tv$  is also an eigenvector corresponding to eigenvalue  $\lambda$ , since

$$Av = \lambda v \quad \implies \quad Au = A(tv) = t(Av) = t(\lambda v) = \lambda(tv) = \lambda u.$$

Because of this, we usually take eigenvectors to be unit vectors, by normalizing them. Even then, this does not determine  $v$  uniquely, since both  $\pm v$  are unit eigenvectors.

Let

$$Q = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Then  $Q$  has eigenvalues 3 and 1, with corresponding eigenvectors  $(1, 1)$  and  $(1, -1)$ . These are not unit vectors, but the corresponding unit eigenvectors are  $(1/\sqrt{2}, 1/\sqrt{2})$  and  $(1/\sqrt{2}, -1/\sqrt{2})$ .

The code

```
from numpy import *
from numpy.linalg import eig

# lambda is a keyword in Python
# so we use lamda instead

A = array([[2,1],[1,2]])
lamda, U = eig(A)
lamda
```

returns the eigenvalues  $[3, 1]$  as an array, and returns the eigenvectors  $v_1$ ,  $v_2$  of  $Q$ , as the *columns* of the matrix  $U$ . The matrix  $U$  is discussed further below.

The method `eig(A)` works on any square matrix  $A$ , but may return complex eigenvalues. When `eig(A)` returns real eigenvalues, they are not necessarily ordered in any predetermined fashion.

If the matrix  $Q$  is known to be symmetric, then the eigenvalues are guaranteed real. In this case, the method `eigh(Q)` returns these eigenvalues in increasing order. If `eigh` is used on a non-symmetric matrix, it will return erroneous data.

```

from numpy import *
from numpy.linalg import eigh

Q = array([[2,1],[1,2]])
lamda, U = eigh(Q)
lamda

```

returns the array [1,3].



Let  $A$  be a square  $d \times d$  matrix. The ideal situation is when there is a basis  $v_1, v_2, \dots, v_d$  in  $\mathbf{R}^d$  of eigenvectors of  $A$ . However, this is not always the case. For example, if

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad (3.2.2)$$

and  $Av = \lambda v$ , then  $v = (x, y)$  satisfies  $x + y = \lambda x$ ,  $y = \lambda y$ . This system has only the nonzero solution  $(x, y) = (1, 0)$  (or its multiples) and  $\lambda = 1$ . Thus  $A$  has only one eigenvector  $e_1 = (1, 0)$ , and the corresponding eigenvalue is  $\lambda = 1$ .

Let  $A$  be any square matrix.

### Eigenvalues of $A$ Versus Eigenvalues of $A^t$

The eigenvalues of  $A$  and the eigenvalues of  $A^t$  are the same.

This result is a consequence of the rank theorem in §2.9. To see why, suppose  $\lambda$  is an eigenvalue of  $A$  with corresponding eigenvector  $v$ . Then  $Av = \lambda v$ , which implies

$$(A - \lambda I)v = Av - \lambda v = 0.$$

As a consequence, if we let  $B = A - \lambda I$ , then  $\lambda$  is an eigenvalue of  $A$  iff<sup>1</sup>  $B$  has a nonzero null space. If we show  $B^t = A^t - \lambda I$  has a nonzero null space, by the same logic, we will conclude  $\lambda$  is an eigenvalue of  $A^t$ . Now  $B$  has a nonzero null space iff  $B$  is not full-rank. Since  $B$  is square, by the rank theorem, this happens iff  $B^t$  is not full-rank, which happens iff  $B^t$  has

---

<sup>1</sup>Iff is short for *if and only if*.

a nonzero null space. Thus  $\lambda$  is an eigenvalue of  $A$  iff  $\lambda$  is an eigenvalue of  $A^t$ .



Let  $v$  be a unit vector. From §2.5, when  $Q$  is the covariance matrix of a dataset,  $v \cdot Qv$  is the variance of the dataset projected onto the line through  $v$ . When  $v$  is an eigenvector,  $Qv = \lambda v$ , the variance equals

$$v \cdot Qv = v \cdot \lambda v = \lambda v \cdot v = \lambda.$$

More generally, this holds for any symmetric matrix  $Q$ . We conclude

### Projected Variance along Eigenvector Direction

If  $v$  is a unit eigenvector of a symmetric matrix  $Q$ , then  $v \cdot Qv$  equals the corresponding eigenvalue. In particular, the eigenvalues of a covariance matrix are nonnegative.

In general, when  $Q$  is symmetric but not a covariance matrix, some eigenvalues of  $Q$  may be negative.



Suppose  $\lambda$  and  $\mu$  are eigenvalues of a symmetric matrix  $Q$  with corresponding eigenvectors  $u, v$ . Since  $Q$  is symmetric,  $u \cdot Qv = v \cdot Qu$ . Using  $Qu = \lambda u$ ,  $Qv = \mu v$ , we compute  $u \cdot Qv$  in two ways:

$$\mu u \cdot v = u \cdot (\mu v) = u \cdot Qv = v \cdot Qu = v \cdot (\lambda u) = \lambda u \cdot v.$$

This implies

$$(\mu - \lambda)u \cdot v = 0.$$

If  $\lambda \neq \mu$ , we must have  $u \cdot v = 0$ . We conclude:

### Distinct Eigenvalues Have Orthogonal Eigenvectors

For a symmetric matrix  $Q$ , eigenvectors corresponding to distinct eigenvalues are orthogonal.



Suppose there is a basis  $v_1, v_2, \dots, v_d$  of eigenvectors of  $Q$ , with corresponding eigenvalues  $\lambda_1, \lambda_2, \dots, \lambda_d$ . Let  $E$  be the diagonal matrix with  $\lambda_1, \lambda_2, \dots, \lambda_d$  on the diagonal,

$$E = \begin{pmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \lambda_{d-1} & 0 \\ 0 & 0 & \dots & 0 & \lambda_d \end{pmatrix}.$$

Let  $U$  be the matrix with columns  $v_1, v_2, \dots, v_d$ . By matrix multiplication and  $Qv_j = \lambda_j v_j$ ,  $j = 1, 2, \dots, d$ , we obtain

$$QU = UE. \quad (3.2.3)$$

When this happens, we say  $Q$  is *diagonalizable*. Thus  $A$  in (3.2.2) is not diagonalizable. On the other hand, we will show *every symmetric matrix  $Q$  is diagonalizable*. In fact,  $Q$  symmetric leads to an orthonormal basis of eigenvectors.

In Python, given  $Q$ , we compute the third eigenvector  $v$  and third eigenvalue  $\lambda$ , and verify  $Qv = \lambda v$ . The code

```
from numpy import *
from numpy.linalg import eigh

# Q is any symmetric matrix
lamda, U = eigh(Q)
lamda = lamda[2]
v = U[:,2]

allclose(dot(Q,v), lamda*v)
```

returns True.



The main result in this section is

### Eigenvalue Decomposition (EVD)

Let  $Q$  be a *symmetric*  $d \times d$  matrix. There is an orthonormal basis  $v_1, v_2, \dots, v_d$  in  $\mathbf{R}^d$  of eigenvectors of  $Q$ , with corresponding eigenvalues

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d.$$

Here are some consequences of the eigenvalue decomposition.

If  $V$  is the matrix with rows  $v_1, v_2, \dots, v_d$  then  $U = V^t$  is the matrix with columns  $v_1, v_2, \dots, v_d$ . Since  $v_1, v_2, \dots, v_d$  are orthonormal,  $U$  is orthogonal (see (2.9.2)), so  $U^t U = I = U U^t$ . By (3.2.3),  $QU = UE$ . Multiplying on the right by  $V = U^t$ ,

$$Q = QUV = UEV.$$

Thus the eigenvalue decomposition states

### Diagonalization (EVD)

There is an orthogonal matrix  $V$  and a diagonal matrix  $E$  such that with  $U = V^t$ , we have

$$Q = UEV. \quad (3.2.4)$$

When this happens, the rows of  $V$  are the eigenvectors of  $Q$ , and the diagonal entries of  $E$  are the eigenvalues of  $Q$ .

In other words, with the correct choice of orthonormal basis, the matrix  $Q$  becomes a diagonal matrix  $E$ .

The orthonormal basis eigenvectors  $v_1, v_2, \dots, v_d$  are the *principal components* of the matrix  $Q$ . The eigenvalues and eigenvectors of  $Q$ , taken together, are the *eigendata* of  $Q$ . The code

```
from numpy import *
from numpy.linalg import eigh

Q = array([[2,1],[1,2]])
lamda, U = eigh(Q)
lamda, U
```

returns the eigenvectors  $[1, 3]$  and the matrix  $U = [u, v]$  with columns

$$u = (1/\sqrt{2}, -1/\sqrt{2}), \quad v = (1/\sqrt{2}, 1/\sqrt{2}).$$

These columns are the orthonormal eigenvectors  $Qv = 3v$ ,  $Qu = 1u$ . By (3.2.3),  $QU = UE$ , where  $E$  is the diagonal matrix with the eigenvalues on the diagonal,

```
from numpy import *
from numpy.linalg import eigh

Q = array([[2,1],[1,2]])
lamda, U = eigh(Q)
V = U.T
E = diag(lamda)

allclose(Q,dot(U,dot(E,V)))
```

returns True.



In sympy, the corresponding commands are

```
from sympy import *
from sympy import init_printing

init_printing()

# eigenvalues
Q.eigenvals()

# eigenvectors
Q.eigenvects()

U, E = Q.diagonalize()
```

The command `init_printing` pretty-prints the output.



Let  $\lambda_1, \lambda_2, \dots, \lambda_r$  be the nonzero eigenvalues of  $Q$ . Then the diagonal matrix  $E$  has  $r$  nonzero entries on the diagonal, so  $\text{rank}(E) = r$ . Since  $U$  and  $V = U^t$  are invertible,  $\text{rank}(E) = \text{rank}(UEV)$ . Since  $Q = UEV$ ,

$$\text{rank}(Q) = \text{rank}(E) = r.$$

### Rank Equals Number of Nonzero Eigenvalues

The rank of a diagonal matrix equals the number of nonzero entries. The rank of a square symmetric matrix  $Q$  equals the number of nonzero eigenvalues of  $Q$ .



Using `sympy`,

```
from sympy import *

Q = Matrix([[2,1],[1,2]])
U, E = Q.diagonalize()
display(U,E)
```

returns

$$U = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \quad E = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}.$$

Also,

```
from sympy import *

a,b,c = symbols("a b c")

Q = Matrix([[a,b],[b,c]])
U, E = Q.diagonalize()
display(Q,U,E)
```

returns

$$Q = \begin{pmatrix} a & b \\ b & c \end{pmatrix}, \quad U = \frac{1}{2b} \begin{pmatrix} a - c - \sqrt{D} & a - c + \sqrt{D} \\ 2b & 2b \end{pmatrix}$$

and

$$E = \frac{1}{2} \begin{pmatrix} a+c-\sqrt{D} & 0 \\ 0 & a+c+\sqrt{D} \end{pmatrix}, \quad D = (a-c)^2 + 4b^2.$$

`display` is used to pretty-print the output.



When all the eigenvalues are nonzero, we can write

$$E^{-1} = \begin{pmatrix} 1/\lambda_1 & 0 & 0 & \dots & 0 \\ 0 & 1/\lambda_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 & 1/\lambda_d \end{pmatrix}.$$

Then a straightforward calculation using (3.2.4) shows

### Nonzero Eigenvalues Equals Invertible

Let  $Q = UEV$  be the EVD of a symmetric matrix  $Q$ . Then  $Q$  is invertible iff all its eigenvalues are nonzero. When this happens, we have

$$Q^{-1} = UE^{-1}V$$

More generally, using (2.6.8), one can check

### Pseudo-Inverse (EVD)

If  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_r$  are the nonzero eigenvalues of  $Q$ , then  $1/\lambda_1 \leq 1/\lambda_2 \leq \dots \leq 1/\lambda_r$  are the nonzero eigenvalues of  $Q^+$ . Moreover, if  $Q = UEV$  as in (3.2.4), then  $Q^+ = UE^+V$ .

Similarly, eigendata may be used to solve linear systems.

### Nonzero Eigenvalues Equals Solvable

Let  $v_1, v_2, \dots, v_d$  be the orthonormal basis of eigenvectors of  $Q$  corresponding to eigenvalues  $\lambda_1, \lambda_2, \dots, \lambda_d$ . Then the linear system

$$Qx = b$$

has a solution  $x$  for every vector  $b$  iff all eigenvalues are nonzero, in which case

$$x = \frac{1}{\lambda_1}(b \cdot v_1)v_1 + \frac{1}{\lambda_2}(b \cdot v_2)v_2 + \cdots + \frac{1}{\lambda_d}(b \cdot v_d)v_d. \quad (3.2.5)$$

The proof is straightforward using (2.9.3), multiply by  $Q$  to verify,

$$\begin{aligned} Qx &= Q \left( \frac{1}{\lambda_1}(b \cdot v_1)v_1 + \frac{1}{\lambda_2}(b \cdot v_2)v_2 + \dots \right) \\ &= \frac{1}{\lambda_1}(b \cdot v_1)Qv_1 + \frac{1}{\lambda_2}(b \cdot v_2)Qv_2 + \dots \\ &= \frac{1}{\lambda_1}(b \cdot v_1)v_1 + \frac{1}{\lambda_2}(b \cdot v_2)v_2 + \dots \\ &= (b \cdot v_1)v_1 + (b \cdot v_2)v_2 + \cdots = b. \end{aligned}$$

Another consequence of the eigenvalue decomposition is

### Trace is the Sum of Eigenvalues

Let  $Q$  be a symmetric matrix with eigenvalues  $\lambda_1, \lambda_2, \dots, \lambda_d$ . Then

$$\text{trace}(Q) = \lambda_1 + \lambda_2 + \cdots + \lambda_d. \quad (3.2.6)$$

To derive this, use (3.2.3): Since  $U$  is orthogonal,  $UV = UU^t = I$ . By (2.2.9),  $\text{trace}(AB) = \text{trace}(BA)$ , so

$$\text{trace}(Q) = \text{trace}(QUV) = \text{trace}(VQU) = \text{trace}(VUEVU) = \text{trace}(E).$$

Since  $E = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_d)$ ,  $\text{trace}(E) = \lambda_1 + \lambda_2 + \cdots + \lambda_d$ , and the result follows.

Let  $Q$  be symmetric with eigenvalues  $\lambda_1, \lambda_2, \dots, \lambda_d$ . Since

$$Qv = \lambda v \implies Q^2v = QQv = Q(\lambda v) = \lambda Qv = \lambda^2 v,$$

$Q^2$  is symmetric with eigenvalues  $\lambda_1^2, \lambda_2^2, \dots, \lambda_d^2$ . Applying the last result to  $Q^2$ , we have

$$\text{trace}(Q^2) = \text{trace}(QQ^t) = \text{trace}(Q^2) = \lambda_1^2 + \lambda_2^2 + \cdots + \lambda_d^2.$$

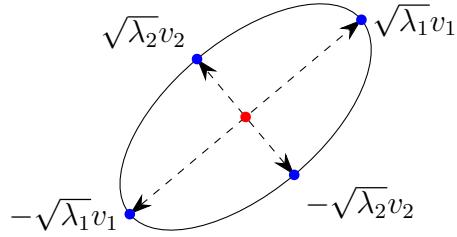


Figure 3.4: Inverse covariance ellipse and centered dataset.



It turns out every symmetric nonnegative matrix is the covariance of a simple dataset (Figure 3.4).

### Sum of Tensor Products

Let  $Q$  be a symmetric  $d \times d$  matrix with eigenvalues  $\lambda_1, \lambda_2, \dots, \lambda_d$  and orthonormal eigenvectors  $v_1, v_2, \dots, v_d$ . Then

$$Q = \lambda_1 v_1 \otimes v_1 + \lambda_2 v_2 \otimes v_2 + \cdots + \lambda_d v_d \otimes v_d. \quad (3.2.7)$$

In particular, when  $Q$  is nonnegative, the dataset consisting of the  $2d$  points

$$\pm\sqrt{\lambda_1}v_1, \pm\sqrt{\lambda_2}v_2, \dots, \pm\sqrt{\lambda_d}v_d$$

is centered and has covariance  $Q/d$

Since  $v_1, v_2, \dots, v_d$  is an orthonormal basis, by (2.9.3), every vector  $v$  can be written

$$v = (v \cdot v_1) v_1 + (v \cdot v_2) v_2 + \cdots + (v \cdot v_d) v_d.$$

Multiply by  $Q$ . Since  $Qv_k = \lambda_k v_k$ ,

$$\begin{aligned} Qv &= (v \cdot v_1) Qv_1 + (v \cdot v_2) Qv_2 + \cdots + (v \cdot v_d) Qv_d \\ &= \lambda_1(v \cdot v_1) v_1 + \lambda_2(v \cdot v_2) v_2 + \cdots + \lambda_d(v \cdot v_d) v_d \\ &= (\lambda_1 v_1 \otimes v_1 + \lambda_2 v_2 \otimes v_2 + \cdots + \lambda_d v_d \otimes v_d) v \end{aligned}$$

This proves the first part. For the second part, let  $b_k = \sqrt{\lambda_k}v_k$ . Then the mean of the  $2d$  vectors  $\pm b_1, \pm b_2, \dots, \pm b_d$  is clearly zero, and by (3.2.7), the

covariance matrix

$$\frac{2}{2d} (b_1 \otimes b_1 + b_2 \otimes b_2 + \cdots + b_d \otimes b_d)$$

equals  $Q/d$ .



Now we approach the eigenvalues of  $Q$  from a different angle. In §2.5, we studied zero variance directions. Since the eigenvalues of a covariance matrix are nonnegative, for a covariance matrix, they may also be called minimum variance directions. Now we study maximum variance directions.

Let

$$\lambda_1 = \max_{|v|=1} v \cdot Qv,$$

where the maximum is over all unit vectors  $v$ . We say a *unit vector*  $b$  is *best-fit for  $Q$*  or *best-aligned with  $Q$*  if the maximum is achieved at  $v = b$ :  $\lambda_1 = b \cdot Qb$ . When  $Q$  is a covariance matrix, this means the unit vector  $b$  is chosen so that the variance  $b \cdot Qb$  of the dataset projected onto  $b$  is maximized.

An eigenvalue  $\lambda_1$  of  $Q$  is the *top eigenvalue* if  $\lambda_1 \geq \lambda$  for any other eigenvalue. An eigenvalue  $\lambda_1$  of  $Q$  is the *bottom eigenvalue* if  $\lambda_1 \leq \lambda$  for any other eigenvalue.

### Maximum Projected Variance is an Eigenvalue

Let  $Q$  be a symmetric matrix. Then

$$\lambda_1 = \max_{|v|=1} v \cdot Qv, \tag{3.2.8}$$

is the top eigenvalue of  $Q$ .

### Best-aligned vector is an eigenvector

Let  $Q$  be a symmetric matrix. Then a best-aligned vector  $b$  is an eigenvector of  $Q$  corresponding to the top eigenvalue  $\lambda_1$ .



To prove these results, we begin with a simple calculation, whose derivation we skip.

### A Calculation

Suppose  $\lambda, a, b, c, d$  are real numbers and suppose we know

$$\frac{\lambda + at + bt^2}{1 + ct + dt^2} \leq \lambda, \quad \text{for all } t \text{ real.}$$

Then  $a = \lambda c$ .

Let  $\lambda$  be any eigenvalue of  $Q$ , with eigenvector  $v$ :  $Qv = \lambda v$ . Dividing  $v$  by its length, we may assume  $|v| = 1$ . Then

$$\lambda_1 \geq v \cdot Qv = v \cdot (\lambda v) = \lambda v \cdot v = \lambda.$$

This shows  $\lambda_1 \geq \lambda$  for any eigenvalue  $\lambda$ .

Now we show  $\lambda_1$  itself is an eigenvalue. Let  $v_1$  be a unit vector maximizing  $v \cdot Qv$ , so  $v_1$  is best-fit for  $Q$ . Then

$$\lambda_1 = v_1 \cdot Qv_1 \geq v \cdot Qv \tag{3.2.9}$$

for all unit vectors  $v$ . Let  $u$  be any vector. Then for any real  $t$ ,

$$v = \frac{v_1 + tu}{|v_1 + tu|}$$

is a unit vector. Insert this  $v$  into (3.2.9) to obtain

$$\lambda_1 \geq \frac{(v_1 + tu) \cdot Q(v_1 + tu)}{|v_1 + tu|^2}.$$

Since  $Q$  is symmetric,  $u \cdot Qv_1 = v_1 \cdot Qu$ . Expanding with  $|v_1|^2 = 1$ , we obtain

$$\lambda_1 \geq \frac{\lambda_1 + 2tu \cdot Qv_1 + t^2 u \cdot Qu}{1 + 2tu \cdot v_1 + t^2 |u|^2} = \frac{\lambda_1 + at + bt^2}{1 + ct + dt^2}.$$

Applying the calculation with  $\lambda = \lambda_1$ ,  $a = 2u \cdot Qv_1$ ,  $b = u \cdot Qu$ ,  $c = 2u \cdot v_1$ , and  $d = |u|^2$ , we conclude

$$u \cdot Qv_1 = \lambda_1 u \cdot v_1$$

for all vectors  $u$ . But this implies

$$u \cdot (Qv_1 - \lambda_1 v_1) = 0$$

for all  $u$ . Thus  $Qv_1 - \lambda_1 v_1$  is orthogonal to all vectors, hence orthogonal to itself. Since this can only happen if  $Qv_1 - \lambda_1 v_1 = 0$ , we conclude  $Qv_1 = \lambda_1 v_1$ . Hence  $\lambda_1$  is itself an eigenvalue. This completes the proof of the two results.



Just as the maximum variance (3.2.8) is the top eigenvalue  $\lambda_1$ , the minimum variance

$$\lambda_d = \min_{|v|=1} v \cdot Qv, \quad (3.2.10)$$

is the bottom eigenvalue, and the corresponding eigenvector  $v_d$  is the worst-aligned vector.

By the eigenvalue decomposition, the eigenvalues  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_d$  of a symmetric matrix  $Q$  may be arranged in decreasing order, and may be positive, zero, or negative scalars. When  $Q$  is a covariance, the eigenvalues are nonnegative, and the bottom eigenvalue is at least zero. When the bottom eigenvalue is zero, the corresponding eigenvectors are zero variance directions.



Now we can complete the proof the eigenvalue decomposition. Having found the top eigenvalue  $\lambda_1$  with its corresponding unit eigenvector  $v_1$ , we let  $S = \text{span}(v_1)$  and  $T = S^\perp$  be the orthogonal complement of  $v_1$  (Figure 3.5). Then  $\dim(T) = d - 1$ , and we can repeat the process and maximize  $v \cdot Qv$  over all unit  $v$  in  $T$ , i.e. over all unit  $v$  orthogonal to  $v_1$ . This leads to another eigenvalue  $\lambda_2$  with corresponding eigenvector  $v_2$  orthogonal to  $v_1$ .

Since  $\lambda_1$  is the maximum of  $v \cdot Qv$  over all vectors in  $\mathbf{R}^d$ , and  $\lambda_2$  is the maximum of  $v \cdot Qv$  over the restricted space  $T$  of vectors orthogonal to  $v_1$ , we must have  $\lambda_1 \geq \lambda_2$ .

Having found the top two eigenvalues  $\lambda_1 \geq \lambda_2$  and their orthonormal eigenvectors  $v_1, v_2$ , we let  $S = \text{span}(v_1, v_2)$  and  $T = S^\perp$  be the orthogonal complement of  $S$ . Then  $\dim(T) = d - 2$ , and we can repeat the process to obtain  $\lambda_3$  and  $v_3$  in  $T$ . Continuing in this manner, we obtain eigenvalues

$$\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \dots \geq \lambda_d.$$

with corresponding orthonormal eigenvectors

$$v_1, v_2, v_3, \dots, v_d.$$

This proves the eigenvalue decomposition.

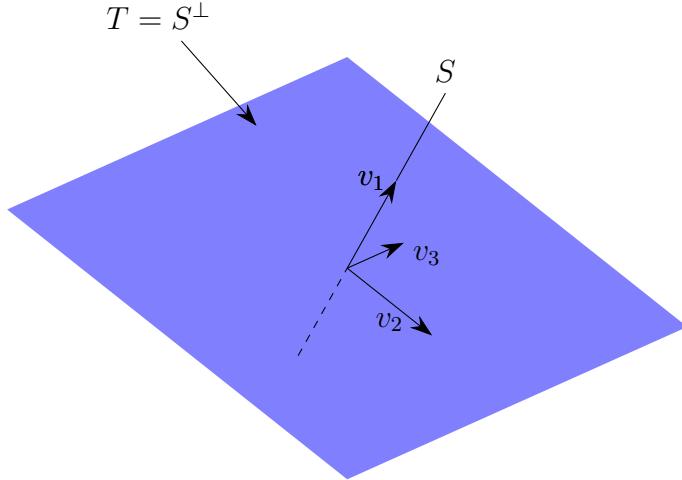


Figure 3.5:  $S = \text{span}(v_1)$  and  $T = S^\perp$ .



Let  $Q$  be a positive covariance matrix and let  $b \cdot Q^{-1}b = 1$  be the inverse covariance ellipsoid. If  $v$  is a unit eigenvector corresponding to an eigenvalue  $\lambda$ , then  $\lambda \geq 0$ , and the vector  $b = \sqrt{\lambda}v$  has length  $\sqrt{\lambda}$ . Moreover  $b$  satisfies

$$b \cdot Q^{-1}b = (\sqrt{\lambda}v) \cdot Q^{-1}(\sqrt{\lambda}v) = \lambda v \cdot Q^{-1}v = \lambda v \cdot (\lambda^{-1}v) = v \cdot v = 1.$$

Hence the line segment joining the vectors  $\pm\sqrt{\lambda}v$  is an axis of the inverse covariance ellipsoid, with length  $2\sqrt{\lambda}$  (Figure 3.4).

When  $\lambda = \lambda_1$  is the top eigenvalue, the axis is the *principal axis* of the inverse covariance ellipsoid. When  $\lambda = \lambda_2$  is the next highest eigenvalue, the axis is orthogonal to the principal axis, and is the second principal axis. Continuing in this manner, we obtain all the principal axes of the inverse covariance ellipsoid.

### Principal Axes of Inverse Covariance Ellipsoid

Let  $v$  be a unit eigenvector of a covariance matrix  $Q$  with eigenvalue  $\lambda$ . Then the line segment joining  $-\sqrt{\lambda}v$  and  $+\sqrt{\lambda}v$  is a principal axis of the inverse covariance ellipsoid, with length  $2\sqrt{\lambda}$ .

Together with Figure 1.32, this result provides a geometric interpretation of eigenvalues: They control the variances of a dataset's points, in the principal directions.

Sometimes, several eigenvalues are equal, leading to several eigenvectors, say  $m$  of them, corresponding to a given eigenvalue  $\lambda$ . In this case, we say the eigenvalue  $\lambda$  has *multiplicity*  $m$ , and we call the span

$$S_\lambda = \{v : Qv = \lambda v\}$$

the *eigenspace* corresponding to  $\lambda$ . For example, suppose the top three eigenvalues are equal:  $\lambda_1 = \lambda_2 = \lambda_3$ , with  $b_1, b_2, b_3$  the corresponding eigenvectors. Calling this common value  $\lambda$ , the eigenspace is  $S_\lambda = \text{span}(b_1, b_2, b_3)$ . Since  $b_1, b_2, b_3$  are orthonormal,  $\dim(V_\lambda) = 3$ . In Python, the eigenspaces  $V_\lambda$  are obtained by the matrix  $U$  above: The columns of  $U$  are an orthonormal basis for the entire space, so selecting the columns corresponding to a specific  $\lambda$  yields an orthonormal basis for  $S_\lambda$ .



Let `(evs,U)` be the list of eigenvalues and matrix  $U$  whose columns are the eigenvectors. Then the eigenvectors are the rows of  $U^t$ . Here is code for selecting just the eigenvectors corresponding to eigenvalue  $s$ .

```
from numpy import *
from numpy.linalg import eigh

lamda, U = eigh(Q)
V = U.T
V[isclose(lamda,s)]
```

The function `isclose(a,b)` returns `True` when  $a$  and  $b$  are numerically close. Using this boolean, we extract only those rows of  $V$  whose corresponding eigenvalue is close to  $s$ .

The subspace  $S_\lambda$  is defined for any  $\lambda$ . However,  $\dim(S_\lambda) = 0$  unless  $\lambda$  is an eigenvalue, in which case  $\dim(S_\lambda) = m$ , where  $m$  is the multiplicity of  $\lambda$ .

The proof of the eigenvalue decomposition is a systematic procedure for finding eigenvalues  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_d$ . Now we show there are no other eigenvalues.

### The Eigenvalue Decomposition is Complete

If  $\lambda$  is an eigenvalue for  $Q$ ,  $Qv = \lambda v$ , then  $\lambda$  equals one of the eigenvalues in the eigenvalue decomposition of  $Q$ .

To see this, suppose  $Qv = \lambda v$  with  $\lambda \neq \lambda_j$  for  $j = 1, \dots, d$ . Since  $\lambda \neq \lambda_j$  for  $j = 1, \dots, d$ , the vector  $v$  must be orthogonal to every  $v_j$ ,  $j = 1, \dots, d$ . Since  $\text{span}(v_1, \dots, v_d) = \mathbf{R}^d$ , it follows  $v$  is orthogonal to every vector, hence  $v$  is orthogonal to itself, hence  $v = 0$ . We conclude  $\lambda$  cannot be an eigenvalue.



All this can be readily computed in Python. For the Iris dataset, we have the covariance matrix in (2.2.14). The eigenvalues are

$$4.2 > 0.24 > 0.08 > 0.02,$$

and the orthonormal eigenvectors are the *columns* of the matrix

$$U = \begin{pmatrix} 0.36 & -0.66 & -0.58 & 0.32 \\ -0.08 & -0.73 & 0.6 & -0.32 \\ 0.86 & 0.18 & 0.07 & -0.48 \\ 0.36 & 0.07 & 0.55 & 0.75 \end{pmatrix}$$

Since the eigenvalues are distinct, the multiplicity of each eigenvalue is 1.

From (2.2.14), the total variance of the Iris dataset is

$$4.54 = \text{trace}(Q) = \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4.$$

For the Iris dataset, the top eigenvalue is  $\lambda_1 = 4.2$ , it has multiplicity 1, and its corresponding list of eigenvectors contains only one eigenvector,

$$v_1 = (0.36, -0.08, 0.86, 0.36).$$

The top eigenvalue accounts for 92.5% of the total variance.

The second eigenvalue is  $\lambda_2 = 0.24$  with eigenvector

$$v_2 = (-0.66, -0.73, 0.18, 0.07).$$

The top two eigenvalues account for 97.8% of the total variance.

The third eigenvalue is  $\lambda_3 = 0.08$  with eigenvector

$$v_3 = (-0.58, 0.60, 0.07, 0.55).$$

The top three eigenvalues account for 99.5% of the total variance.

The fourth eigenvalue is  $\lambda_4 = 0.02$  with eigenvector

$$v_4 = (0.32, -0.32, -0.48, 0.75).$$

The top four eigenvalues account for 100% of the total variance. Here each eigenvalue has multiplicity 1, since there are four distinct eigenvalues.



An important class of symmetric matrices are of the form

$$\begin{aligned} & \left( \begin{array}{cc} 2 & -2 \\ -2 & 2 \end{array} \right) \left( \begin{array}{ccc} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{array} \right) \left( \begin{array}{cccc} 2 & -1 & 0 & -1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{array} \right) \\ & \left( \begin{array}{ccccc} 2 & -1 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ -1 & 0 & 0 & -1 & 2 \end{array} \right) \left( \begin{array}{cccccc} 2 & -1 & 0 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ -1 & 0 & 0 & 0 & -1 & 2 \end{array} \right) \\ & \left( \begin{array}{cccccc} 2 & -1 & 0 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ -1 & 0 & 0 & 0 & -1 & 2 \end{array} \right). \end{aligned}$$

We denote these matrices  $Q(2), Q(3), Q(4), Q(5), Q(6), Q(7)$ . The following code generates these symmetric  $d \times d$  matrices  $Q(d)$ ,

```

def row(i,d):
    v = [0]*d
    v[i] = 2
    if i > 0: v[i-1] = -1
    if i < d-1: v[i+1] = -1
    if i == 0: v[d-1] += -1
    if i == d-1: v[0] += -1
    return v

# using sympy
from sympy import Matrix

def Q(d): return Matrix([ row(i,d) for i in range(d) ])

# using numpy
from numpy import *

def Q(d): return array([ row(i,d) for i in range(d) ])

```



The eigenvalues of these symmetric matrices follow interesting patterns that are best explored using Python.

Below we will see, the eigenvalues of  $Q(d)$  are between 4 and 0, and each eigenvalue other than 4 and 0 has multiplicity 2.

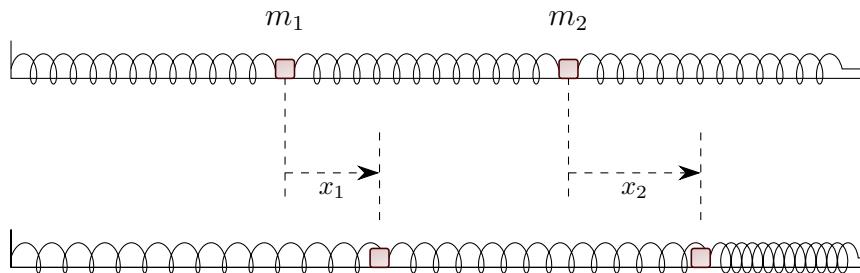


Figure 3.6: Three springs at rest and perturbed.

To explain where these matrices come from, look at the mass-spring systems in Figures 3.6 and 3.7. Here we have springs attached to masses and

walls on either side. At rest, the springs are the same length. When perturbed, some springs are compressed and some stretched. In Figure 3.6, let  $x_1$  and  $x_2$  denote the displacement of each mass from its rest position.

When extended by  $x$ , each spring fights back by exerting a force  $kx$  proportional to the displacement  $x$ . For example, look at the mass  $m_1$ . The spring to its left is extended by  $x_1$ , so exerts a force of  $-kx_1$ . Here the minus indicates pulling to the left. On the other hand, the spring to its right is extended by  $x_2 - x_1$ , so it exerts a force  $+k(x_2 - x_1)$ . Here the plus indicates pulling to the right. Adding the forces from either side, the total force on  $m_1$  is  $-k(2x_1 - x_2)$ . For  $m_2$ , the spring to its left exerts a force  $-k(x_2 - x_1)$ , and the spring to its right exerts a force  $-kx_2$ , so the total force on  $m_2$  is  $-k(2x_2 - 2x_1)$ . We obtain the force vector

$$-k \begin{pmatrix} 2x_1 - x_2 \\ -x_1 + 2x_2 \end{pmatrix} = -k \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

However, as you can see, the matrix here is not exactly  $Q(2)$ .

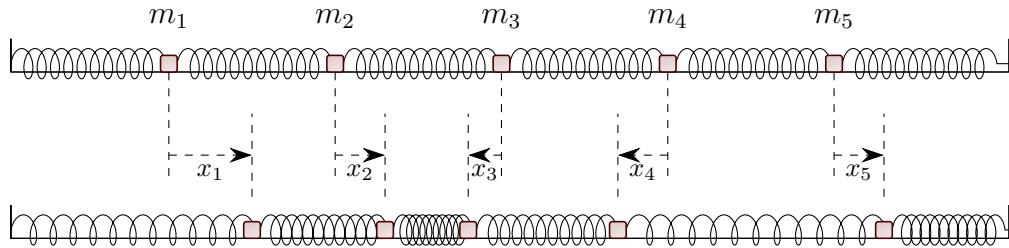


Figure 3.7: Six springs at rest and perturbed.

For five masses, let  $x_1, x_2, x_3, x_4, x_5$  denote the displacement of each mass from its rest position. In Figure 3.7,  $x_1, x_2, x_5$  are positive, and  $x_3, x_4$  are negative.

As before, the total force on  $m_1$  is  $-k(2x_1 - x_2)$ , and the total force on  $m_5$  is  $-k(2x_5 - x_4)$ . For  $m_2$ , the spring to its left exerts a force  $-k(x_2 - x_1)$ , and the spring to its right exerts a force  $+k(x_3 - x_2)$ . Hence, the total force on  $m_2$  is  $-k(-x_1 + 2x_2 - x_3)$ . Similarly for  $m_3, m_4$ . We obtain the force

vector

$$-k \begin{pmatrix} 2x_1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 - x_4 \\ -x_3 + 2x_4 - x_5 \\ -x_4 + 2x_5 \end{pmatrix} = -k \begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}.$$

But, again, the matrix here is not  $Q(5)$ . Notice, if we place one mass and two springs in Figure 3.6, we obtain the  $1 \times 1$  matrix 2.



To obtain  $Q(2)$  and  $Q(5)$ , we place the springs along a circle, as in Figures 3.8 and 3.9. Now we have as many springs as masses. Repeating the same logic, this time we obtain  $Q(2)$  and  $Q(5)$ . Notice if we place one mass and one spring in Figure 3.8,  $d = 1$ , we obtain the  $1 \times 1$  matrix  $Q(1) = 0$ : There is no force if we move a single mass around the circle, because the spring is not being stretched.

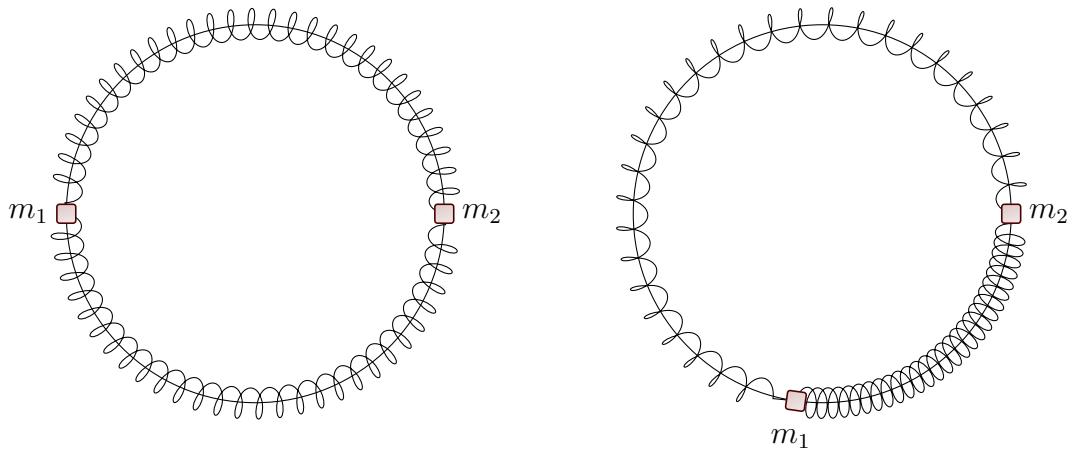


Figure 3.8: Two springs along a circle leading to  $Q(2)$ .

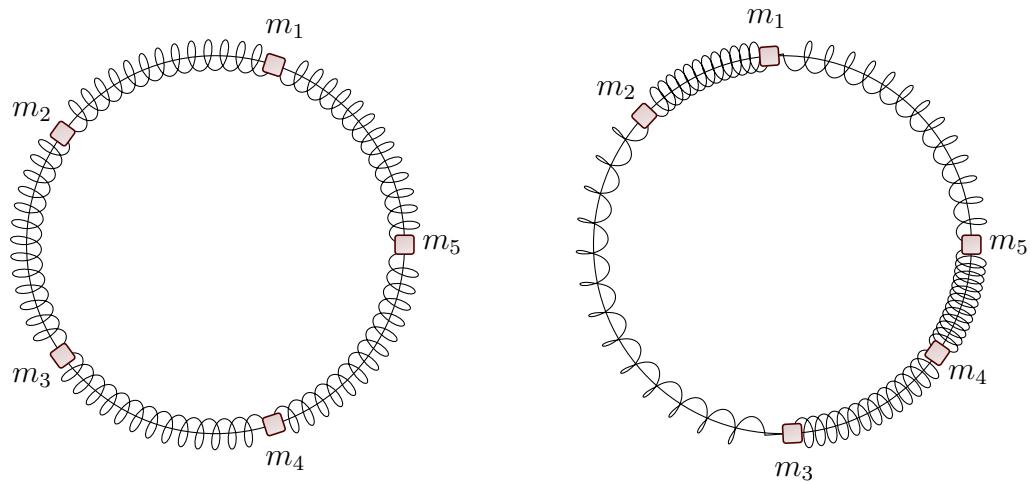


Figure 3.9: Five springs along a circle leading to  $Q(5)$ .

Thus the matrices  $Q(d)$  arise from mass-spring systems arranged on a circle. From Newton's law (force equals mass times acceleration), one shows the frequencies of the vibrating springs equal  $\sqrt{\lambda k/m}$ , where  $k$  is above,  $m$  is the mass of each of the masses, and  $\lambda$  is an eigenvalue of  $Q(d)$ . This is the physical meaning of the eigenvalues of  $Q(d)$ .



Let  $v$  have features  $(x_1, x_2, \dots, x_d)$ , and let  $Q = Q(d)$ . By elementary algebra, check that

$$v \cdot Qv = (x_1 - x_2)^2 + (x_2 - x_3)^2 + \dots + (x_{d-1} - x_d)^2 + (x_d - x_1)^2. \quad (3.2.11)$$

As a consequence of (3.2.11), show also the following.

- For any vector  $v$ ,  $0 \leq v \cdot Qv \leq 4|v|^2$ . Conclude every eigenvalue  $\lambda$  satisfies  $0 \leq \lambda \leq 4$ .
- $\lambda = 0$  is an eigenvalue, with multiplicity 1.
- When  $d$  is even,  $\lambda = 4$  is an eigenvalue with multiplicity 1.
- When  $d$  is odd,  $\lambda = 4$  is not an eigenvalue.



To compute the eigenvalues, we use complex numbers, specifically the  $d$ -th root of unity  $\omega$  (§1.5). Let

$$p(t) = 2 - t - t^{d-1},$$

and let

$$v_1 = \begin{pmatrix} 1 \\ \omega \\ \omega^2 \\ \omega^3 \\ \vdots \\ \omega^{d-1} \end{pmatrix}.$$

Then  $Qv_1$  is

$$Qv_1 = \begin{pmatrix} 2 - \omega - \omega^{d-1} \\ -1 + 2\omega - \omega^2 \\ -\omega + 2\omega^2 - \omega^3 \\ \vdots \\ -\omega^{d-2} + 2\omega^{d-1} - 1 \end{pmatrix} = p(\omega) \begin{pmatrix} 1 \\ \omega \\ \omega^2 \\ \omega^3 \\ \vdots \\ \omega^{d-1} \end{pmatrix} = p(\omega)v_1.$$

Thus  $v_1$  is an eigenvector corresponding to eigenvalue  $p(\omega)$ .

For each  $k = 0, 1, 2, \dots, d-1$ , define

$$v_k = (1, \omega^k, \omega^{2k}, \omega^{3k}, \dots, \omega^{(d-1)k}).$$

By the same calculation, we have

$$Qv_k = p(\omega^k)v_k, \quad k = 0, 1, 2, \dots, d-1.$$

By (1.5.7),

$$p(\omega^k) = 2 - \omega^k - \omega^{(d-1)k} = 2 - \omega^k - \omega^{-k} = 2 - 2 \cos(2\pi k/d).$$

### Eigenvalues of $Q(d)$

The (unsorted) eigenvalues of  $Q(d)$  are

$$\lambda_k = p(\omega^k) = 2 - 2 \cos\left(\frac{2\pi k}{d}\right), \quad k = 0, 1, 2, \dots, d-1. \quad (3.2.12)$$

Corresponding to each eigenvalue  $\lambda_k$ , there is the complex eigenvector  $v_k$ . Separating  $v_k$  into its real and imaginary parts yields two real eigenvectors

$$\begin{aligned} \Re(v_k) &= \left(1, \cos\left(\frac{2\pi k}{d}\right), \cos\left(\frac{4\pi k}{d}\right), \cos\left(\frac{6\pi k}{d}\right), \dots, \cos\left(\frac{2(d-1)\pi k}{d}\right)\right), \\ \Im(v_k) &= \left(0, \sin\left(\frac{2\pi k}{d}\right), \sin\left(\frac{4\pi k}{d}\right), \sin\left(\frac{6\pi k}{d}\right), \dots, \sin\left(\frac{2(d-1)\pi k}{d}\right)\right). \end{aligned}$$

When  $k = 0$  or when  $k = d/2$ ,  $d$  even, we have  $\Im(v_k) = 0$ . This explains the double multiplicity in Figure 3.10, except when  $k = 0$  or  $k = d/2$ ,  $d$  even.



Applying this formula, we obtain eigenvalues

$$Q(2) = (4, 0)$$

$$Q(3) = (3, 3, 0)$$

$$Q(4) = (4, 2, 2, 0)$$

$$Q(5) = \left(\frac{5}{2} + \frac{\sqrt{5}}{2}, \frac{5}{2} + \frac{\sqrt{5}}{2}, \frac{5}{2} - \frac{\sqrt{5}}{2}, \frac{5}{2} - \frac{\sqrt{5}}{2}, 0\right)$$

$$Q(6) = (4, 3, 3, 1, 1, 0)$$

$$Q(8) = (4, 2 + \sqrt{2}, 2 + \sqrt{2}, 2, 2, 2 - \sqrt{2}, 2 - \sqrt{2}, 0)$$

$$\begin{aligned} Q(10) &= \left(4, \frac{5}{2} + \frac{\sqrt{5}}{2}, \frac{5}{2} + \frac{\sqrt{5}}{2}, \frac{3}{2} + \frac{\sqrt{5}}{2}, \frac{3}{2} + \frac{\sqrt{5}}{2}, \right. \\ &\quad \left. \frac{5}{2} - \frac{\sqrt{5}}{2}, \frac{5}{2} - \frac{\sqrt{5}}{2}, \frac{3}{2} - \frac{\sqrt{5}}{2}, \frac{3}{2} - \frac{\sqrt{5}}{2}, 0\right) \end{aligned}$$

$$Q(12) = (4, 2 + \sqrt{3}, 2 + \sqrt{3}, 3, 3, 2, 2, 1, 1, 2 - \sqrt{3}, 2 - \sqrt{3}, 0).$$

The matrices  $Q(d)$  are *circulant matrices*. Each row in  $Q(d)$  is obtained from the row above by shifting the entries to the right. The trick of using the roots of unity to compute the eigenvalues and eigenvectors works for any circulant matrix.



Our last topic is the distribution of the eigenvalues for large  $d$ . How are the eigenvalues scattered? Figure 3.10 plots the eigenvalues for  $Q(50)$  using the code below.

```
from numpy.linalg import eigh
from matplotlib.pyplot import stairs, show, scatter, legend

d = 50
lamda = eigh(Q(d))[0]
stairs(lamda, range(d+1), label="numpy")

k = arange(d)
lamda = 2 - 2*cos(2*pi*k/d)
sorted = sort(lamda)

scatter(k, lamda, s=5, label="unordered")
scatter(k, sorted, c="red", s=5, label="increasing order")

legend()
show()
```

Figure 3.10 shows the eigenvalues tend to cluster near the top  $\lambda_1 \sim 4$  and the bottom  $\lambda_d = 0$  for  $d$  large. Using the double-angle formula,

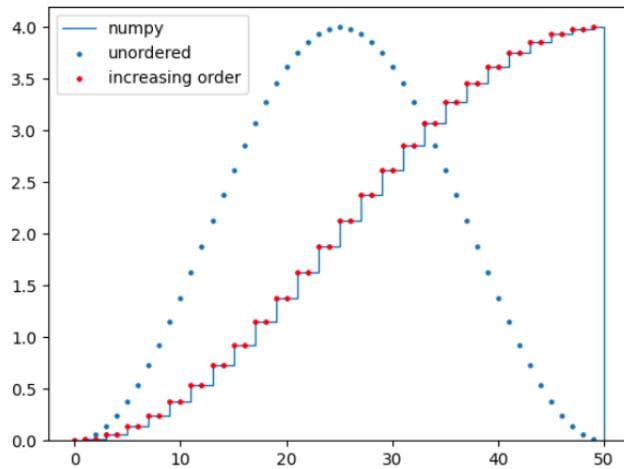
$$\lambda_k = 4 \sin^2 \left( \frac{\pi k}{d} \right), \quad k = 0, 1, 2, \dots, d-1.$$

Solving for  $k/d$  in terms of  $\lambda$ , and multiplying by two to account for the double multiplicity, we obtain<sup>2</sup> the proportion of eigenvalues below threshold  $\lambda$ ,

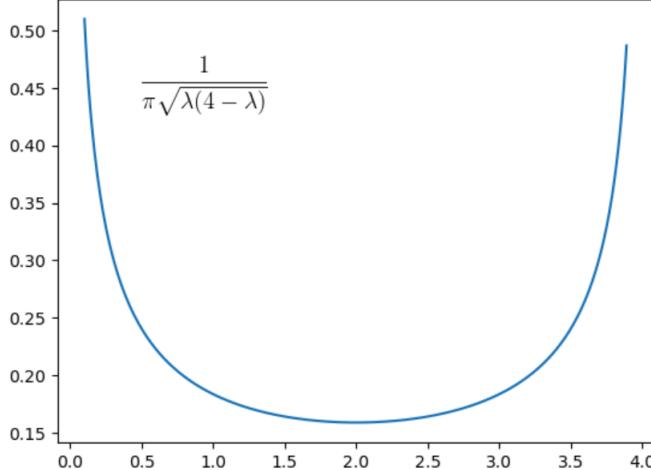
$$\frac{\#\{k : \lambda_k \leq \lambda\}}{d} \approx \frac{2}{\pi} \arcsin \left( \frac{1}{2} \sqrt{\lambda} \right), \quad 0 \leq \lambda \leq 4. \quad (3.2.13)$$

---

<sup>2</sup>This is an approximate equality: The ratio of the two sides approaches 1 as  $d \rightarrow \infty$ .

Figure 3.10: Plot of eigenvalues of  $Q(50)$ .

Equivalently, the derivative of the arcsine law (3.2.13) exhibits (see (7.1.9)) the eigenvalue clustering near the ends (Figure 3.11).

Figure 3.11: Density of eigenvalues of  $Q(d)$  for  $d$  large.

```
from numpy import *
from matplotlib.pyplot import *
```

```

lamda = arange(0.1,3.9,.01)
density = 1/(pi*sqrt(lamda*(4-lamda)))
plot(lamda,density)
# r"..." means raw string
f = r"\frac{1}{\pi \sqrt{\lambda(4-\lambda)}}
text(.5,.45,f,usetex=True,fontsize="x-large")
show()

```

The matrices  $Q(d)$  are prototypes of matrices that are fundamental in many areas of physics and engineering, including time series analysis and information theory, see [10]. This clustering of eigenvalues near the top and bottom is valid for a wide class of matrices, not just  $Q(d)$ , as the matrix size  $d$  grows without bound,  $d \rightarrow \infty$ .

### 3.3 Singular Value Decomposition

In this section, we discuss the singular value decomposition  $(U, S, V)$  of a matrix  $A$ .

Let  $A$  be a matrix. We say a *positive* number  $\sigma > 0$  is a *singular value* of  $A$  if there are nonzero vectors  $v$  and  $u$  satisfying

$$Av = \sigma u \quad \text{and} \quad A^t u = \sigma v. \quad (3.3.1)$$

When this happens,  $v$  is a *right singular vector* and  $u$  is a *left singular vector* associated to  $\sigma$ .

Some books allow singular values to be zero. Here we insist that singular values be positive. Contrast singular values with eigenvalues: While eigenvalues may be negative or zero, for us singular values are positive.

The definition immediately implies

#### Singular Values of $A$ Versus $A^t$

The singular values of  $A$  are the same as the singular values of  $A^t$ .

We work out our first example. Let

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Then  $Av = \lambda v$  implies  $\lambda = 1$  and  $v = (1, 0)$ . Thus  $A$  has only one eigenvalue equal to 1. Set

$$Q = A^t A = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}.$$

Since  $Q$  is symmetric,  $Q$  has two eigenvalues  $\lambda_1, \lambda_2$  and corresponding eigenvectors  $v_1, v_2$ . Moreover, as we saw in an earlier section,  $v_1, v_2$  may be chosen orthonormal.

The eigenvalues of  $Q$  are given by

$$0 = \det(Q - \lambda I) = \lambda^2 - 3\lambda + 1.$$

By the quadratic formula,

$$\lambda_1 = \frac{3}{2} + \frac{\sqrt{5}}{2} = 2.62, \quad \lambda_2 = \frac{3}{2} - \frac{\sqrt{5}}{2} = .38.$$

Now we turn to singular values. If  $v$  and  $u$  and  $\sigma > 0$  satisfy (3.3.1), then

$$Qv = A^t Av = A^t(\sigma u) = \sigma^2 v. \quad (3.3.2)$$

Hence  $\sigma^2 = \lambda$ , and we obtain

$$\sigma_1 = \sqrt{\frac{3}{2} + \frac{\sqrt{5}}{2}} = 1.62, \quad \sigma_2 = \sqrt{\frac{3}{2} - \frac{\sqrt{5}}{2}} = 0.62.$$

To make (3.3.1) work, we set  $u_1 = Av_1/\sigma_1$ . Then  $Av_1 = \sigma_1 u_1$ , and

$$A^t u_1 = A^t Av_1/\sigma_1 = Qv_1/\sigma_1 = \lambda_1 v_1/\sigma_1 = \sigma_1 v_1.$$

Thus  $v_1, u_1$  are right and left singular vectors corresponding to the singular value  $\sigma_1$  of  $A$ . Similarly, if we set  $u_2 = Av_2/\sigma_2$ , then  $v_2, u_2$  are right and left singular vectors corresponding to the singular value  $\sigma_2$  of  $A$ .

We show  $v_1, v_2$  are orthonormal, and  $u_1, u_2$  are orthonormal. We already know  $v_1, v_2$  are orthonormal, because we chose them that way, as eigenvectors of the symmetric matrix  $Q$ . Also

$$0 = \lambda_1 v_1 \cdot v_2 = Qv_1 \cdot v_2 = (A^t Av_1) \cdot v_2 = (Av_1) \cdot (Av_2) = \sigma_1 u_1 \cdot \sigma_2 u_2 = \sigma_1 \sigma_2 u_1 \cdot u_2.$$

Since  $\sigma_1 \neq 0, \sigma_2 \neq 0$ , it follows  $u_1, u_2$  are orthogonal. Also

$$\lambda_1 = \lambda_1 v_1 \cdot v_1 = Qv_1 \cdot v_1 = (A^t Av_1) \cdot v_1 = (Av_1) \cdot (Av_1) = \sigma_1^2 u_1 \cdot u_1.$$

Since  $\lambda_1 = \sigma_1^2$ ,  $u_1 \cdot u_1 = 1$ . Similarly,  $u_2 \cdot u_2 = 1$ . This shows  $u_1, u_2$  are orthonormal, and completes the first example.



Let  $A$  be an  $N \times d$  matrix, and suppose  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$  are singular values with corresponding left singular vectors  $u_1, u_2, \dots, u_r$  and right singular vectors  $v_1, v_2, \dots, v_r$ . Then, since  $u_k = Av_k/\sigma_k$ , the vectors  $u_1, u_2, \dots, u_r$  are in the column space of  $A$ . If  $u_1, u_2, \dots, u_r$  are linearly independent, it follows  $r$  is no larger than  $\text{rank}(A)$ , hence  $r$  is no larger than  $\min(N, d)$ .

We seek the largest value of  $r$ . Many books include zero as a possible singular value, allowing the largest  $r$  to equal  $\min(N, d)$ . Here we insist singular values are positive. Because of this, we will see the largest  $r$  is  $\text{rank}(A)$ .



The close connection between singular values  $\sigma$  of  $A$  and positive eigenvalues  $\lambda$  of  $Q = A^t A$  carries over in the general case.

### A Versus $Q$

Let  $A$  be any matrix. Then

- the rank of  $A$  equals the rank of  $Q$ ,
- $\sigma$  is a singular value of  $A$  iff  $\lambda = \sigma^2$  is a positive eigenvalue of  $Q$ .

Since the rank equals the dimension of the row space, the first part follows from §2.4.

If  $Av = \sigma u$  and  $A^t u = \sigma v$ , then

$$Qv = A^t Av = A^t(\sigma u) = \sigma A^t u = \sigma^2 v,$$

so  $v$  is an eigenvector of  $A^t A$  corresponding to  $\lambda = \sigma^2 > 0$ . Conversely, If  $Qv = \lambda v$  and  $\lambda > 0$ , then set  $\sigma = \sqrt{\lambda}$  and  $u = Av/\sigma$ . Then

$$Av = \sigma u, \quad A^t u = A^t Av/\sigma = Qv/\sigma = \lambda v/\sigma = \sigma v.$$

From §3.2, the number of positive eigenvalues (possibly repeated) of  $Q$  equals the rank of  $Q$ . By the above, we conclude *the rank of  $A$  equals the number of singular values (possibly repeated) of  $A$ .*



### Singular Value Decomposition (SVD)

Let  $A$  be any matrix and let  $r$  be the rank of  $A$ . Then there is an orthonormal basis  $u_1, u_2, \dots, u_N$  of the target space and an orthonormal basis  $v_1, v_2, \dots, v_d$  of the source space and positive scalars  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ , such that

$$Av_k = \sigma_k u_k, \quad A^t u_k = \sigma_k v_k, \quad k = 1, 2, \dots, r, \quad (3.3.3)$$

and

$$Av_k = 0, \quad A^t u_k = 0 \quad \text{for } k > r.$$

The proof is very simple once we remember the rank of  $Q$  equals the number of positive eigenvalues of  $Q$ . By the eigenvalue decomposition, there is an orthonormal basis of the source space  $v_1, v_2, \dots$  and  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_r > 0$  such that  $Qv_k = \lambda_k v_k$ ,  $k = 1, \dots, r$ , and  $Qv_k = 0$ ,  $k > r$ .

Setting  $\sigma_k = \sqrt{\lambda_k}$  and  $u_k = Av_k/\sigma_k$ ,  $k = 1, \dots, r$ , as in our first example, we have (3.3.3), and, again as in our first example,  $u_1, u_2, \dots, u_r$  are orthonormal.

Assume  $A$  is  $N \times d$ . Then the source space is  $\mathbf{R}^d$ , and the target space is  $\mathbf{R}^N$ . By construction,  $v_{r+1}, v_{r+2}, \dots, v_d$  is an orthonormal basis for the null space of  $A$ . Set  $u_1 = Av_1/\sigma_1$ ,  $u_2 = Av_2/\sigma_2$ ,  $\dots$ ,  $u_r = Av_r/\sigma_r$ . Since  $Av_{r+1} = 0, \dots, Av_d = 0$ ,  $u_1, u_2, \dots, u_r$  is an orthonormal basis for the column space of  $A$ .

Since the column space of  $A$  is the row space of  $A^t$ , the column space of  $A$  is the orthogonal complement of the nullspace of  $A^t$  (2.7.6). Choose  $u_{r+1}, u_{r+2}, \dots, u_N$  any orthonormal basis for the nullspace of  $A^t$ . Then  $\{u_1, u_2, \dots, u_r\}$  and  $\{u_{r+1}, u_{r+2}, \dots, u_N\}$  are orthogonal. From this,  $u_1, u_2, \dots, u_N$  is an orthonormal basis for the target.



For our second example, let  $a$  and  $b$  be nonzero vectors, possibly of different sizes, and let  $A$  be the matrix

$$A = a \otimes b, \quad A^t = b \otimes a.$$

Then

$$Av = (v \cdot b)a = \sigma u \quad \text{and} \quad A^t u = (u \cdot a)b = \sigma v.$$

Since the range of  $A$  equals  $\text{span}(a)$ , the rank of  $A$  equals one.

Since  $\sigma > 0$ ,  $v$  is a multiple of  $b$  and  $u$  is a multiple of  $a$ . If we write  $v = tb$  and  $u = sa$  and plug in, we get

$$v = |a|b, \quad u = |b|a, \quad \sigma = |a||b|.$$

Thus there is only one singular value of  $A$ , equal to  $|a||b|$ . This is not surprising since the rank of  $A$  is one.

In a similar manner, one sees the only singular value of the  $1 \times n$  matrix  $A = a$  equals  $\sigma = |a|$ .

Our third example is

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \quad (3.3.4)$$

Then

$$A^t = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \quad Q = A^t A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Since  $Q$  is diagonal symmetric, its rank is 3 and its eigenvalues are  $\lambda_1 = 1$ ,  $\lambda_2 = 1$ ,  $\lambda_3 = 1$ ,  $\lambda_4 = 0$ , and its eigenvectors are

$$v_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, v_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, v_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, v_4 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Clearly  $v_1, v_2, v_3, v_4$  are orthonormal. By (3.3.2),  $\sigma_1 = 1$ ,  $\sigma_2 = 1$ ,  $\sigma_3 = 1$ .

Since we must have  $Av = \sigma u$ , we can check that

$$u_1 = Av_1 = v_2, \quad u_2 = Av_2 = v_3, \quad u_3 = Av_3 = v_4, \quad u_4 = v_1$$

satisfies (3.3.1). This completes our third example.



Let's look at the SVD decomposition in more detail. Suppose  $A$  is  $N \times d$  and  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$  are the singular values of  $A$ .

If  $N \leq d$ , then  $r \leq N$ , we set

$$S = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Here we have  $(N, d) = (4, 6)$ ,  $r = 3$ .

If  $N \geq d$ , then  $r \leq d$ , we set

$$S = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Here we have  $(N, d) = (6, 4)$ ,  $r = 3$ . In either case,  $S$  has the same shape  $N \times d$  as  $A$ .

Let  $U$  be the matrix with columns  $u_1, u_2, \dots, u_N$ , and let  $V$  be the matrix with rows  $v_1, v_2, \dots, v_d$ . Then  $V^t$  has columns  $v_1, v_2, \dots, v_d$ .

Then  $U$  and  $V$  are orthogonal  $N \times N$  and  $d \times d$  matrices. By (3.3.1),

$$AV^t = US.$$

Right-multiplying by  $V$  and using  $V^t V = I$  implies

$$A = USV.$$

Summarizing,

### Diagonalization (SVD)

If  $A$  is any matrix, there is a diagonal matrix  $S$  with nonnegative diagonal entries, with the same shape as  $A$ , and orthogonal matrices  $U$  and  $V$ , satisfying

$$A = USV.$$

The rows of  $V$  are an orthonormal basis of right singular vectors, and the columns of  $U$  are an orthonormal basis of left singular vectors.

From this, if  $A$  is  $N \times d$ , then  $U$  is  $N \times N$ ,  $S$  is  $N \times d$ , and  $V$  is  $d \times d$ .

When  $A = Q$  is a covariance matrix,  $Q \geq 0$ , then the eigenvalues are nonnegative, and, from (3.2.4), we have  $UEU^t = Q$ . If we choose  $V = U^t$ , we see EVD is a special case of SVD, with the singular values corresponding to the positive eigenvalues.

In general, however, if  $Q$  has negative eigenvalues,  $V$  is not equal to  $U^t$ ; instead  $V$  is obtained from  $U^t$  by re-sorting the rows of  $U$ .



In Python, `svd` returns the orthogonal matrices  $U$  and  $V$  and a vector `sigma` of singular values. The singular values are arranged in decreasing order. To recover the diagonal matrix  $S$ , we use `diag`.

```
from numpy import *
from numpy.linalg import svd

U, sigma, V = svd(A)
# sigma is a vector

# build diag matrix S
p = min(A.shape)
S = zeros(A.shape)
S[:p,:p] = diag(sigma)

print(U.shape,S.shape,V.shape)
print(U,S,V)

allclose(A, dot(U, dot(S, V)))
```

This code returns True.



Given the relation between the singular values of  $A$  and the eigenvalues of  $Q = A^t A$ , we also can conclude

### Right Singular Vectors Are the Same as Eigenvectors

Let  $A$  be any matrix and let  $Q = A^t A$ .

$$v \text{ is an eigenvector of } Q \iff v \text{ is a right singular vector of } A. \quad (3.3.5)$$

(This statement ignores zero eigenvalues.)

For example, if `dataset` is the Iris dataset (ignoring the classes), the code

```
from numpy import *
from numpy.linalg import svd,eigh

# center dataset
m = mean(dataset, axis=0)
A = dataset - m
# rows of V are right
# singular vectors of A
V = svd(A)[2]

# any of these will work
Q = dot(A.T,A)
Q = cov(dataset.T,bias=False)
Q = cov(dataset.T,bias=True)

# columns of U are
# eigenvectors of Q
U = eigh(Q)[1]

# compare columns of U
# and rows of V
```

**U, V**

returns

$$U = \begin{pmatrix} 0.36 & -0.66 & -0.58 & 0.32 \\ -0.08 & -0.73 & 0.6 & -0.32 \\ 0.86 & 0.18 & 0.07 & -0.48 \\ 0.36 & 0.07 & 0.55 & 0.75 \end{pmatrix}, V = \begin{pmatrix} 0.36 & -0.08 & 0.86 & 0.36 \\ -0.66 & -0.73 & 0.18 & 0.07 \\ 0.58 & -0.6 & -0.07 & -0.55 \\ 0.32 & -0.32 & -0.48 & 0.75 \end{pmatrix}$$

This shows the columns of  $U$  are identical to the rows of  $V$ , except for the third column of  $U$ , which is the negative of the third row of  $V$ .



Now we turn to the pseudo-inverse.

### To get the Pseudo-Inverse, Invert the Singular Values

The pseudo-inverse  $A^+$  is obtained by replacing singular values of  $A$  by their reciprocals, and taking the transpose.

More explicitly, we can write

### Inverse Singular Values, and Flipped Singular Vectors

Let  $A$  have rank  $r$ , and let  $\sigma_k, v_k, u_k$  be the singular data as above. Then

$$A^+ u_k = \frac{1}{\sigma_k} v_k, \quad (A^+)^t v_k = \frac{1}{\sigma_k} u_k, \quad k = 1, 2, \dots, r,$$

and

$$A^+ u_k = 0, \quad (A^+)^t v_k = 0 \quad \text{for } k > r.$$

We illustrate these results in the case of a diagonal matrix

$$S = \begin{pmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \left( \begin{array}{c|cc} Q & \begin{matrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{matrix} \\ \hline 0 & 0 & 0 \end{array} \right).$$

Since  $S$  is  $4 \times 5$  and  $SS^+S = S$ ,  $S^+$  must be  $5 \times 4$ . Writing

$$S^+ = \left( \begin{array}{ccc|c} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right) = \left( \begin{array}{c|c} A & B \\ \hline C & D \end{array} \right),$$

and applying the four properties of the pseudo-inverse  $S^+$ , leads to

$$A = Q^{-1} = \begin{pmatrix} 1/a & 0 & 0 \\ 0 & 1/b & 0 \\ 0 & 0 & 1/c \end{pmatrix}, \quad B = C = D = 0.$$

From SVD, one then deduces the above results.

### 3.4 Principal Component Analysis

Let  $Q$  be the covariance matrix of a dataset in  $\mathbf{R}^d$ . Then  $Q$  is a  $d \times d$  symmetric matrix, and the eigenvalue decomposition guarantees an orthonormal basis  $v_1, v_2, \dots, v_d$  in  $\mathbf{R}^d$  consisting of eigenvectors of  $Q$ ,

$$Qv_k = \lambda_k v_k, \quad k = 1, \dots, d.$$

These eigenvectors are the *principal components* of the dataset. Principal Component Analysis (PCA) consists of projecting the dataset onto lower-dimensional subspaces spanned by some of the eigenvectors.

Let  $Q$  be a symmetric matrix with eigenvalue  $\lambda$  and corresponding eigenvector  $v$ ,  $Qv = \lambda v$ . If  $t$  is a scalar, then the matrix  $tQ$  has eigenvalue  $t\lambda$  and corresponding eigenvector  $v$ , since

$$(tQ)v = tQv = t\lambda v = (t\lambda)v.$$

Hence *multiplying  $Q$  by a scalar does not change the eigenvectors*.

Let  $A$  be the dataset matrix of a given dataset with  $N$  samples, and  $d$  features. If the samples are the rows of  $A$ , then  $A$  is  $N \times d$ . If we assume the dataset is centered, then, by (2.2.13), the covariance is  $Q = A^t A / N$ . From the previous paragraph, the eigenvectors of the covariance  $Q$  equal the eigenvectors of  $A^t A$ . From (3.3.5), these are the same as the right singular vectors of  $A$ .

Thus *the principal components of a dataset are the right singular vectors of the centered dataset matrix.* This shows there are two approaches to the principal components of a dataset: Either through EVD and eigenvectors of the covariance matrix, or through SVD and right singular vectors of the centered dataset matrix. We shall do both.

Assuming the eigenvalues are ordered top to bottom,

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d,$$

in PCA one takes the most significant components, those components whose eigenvalues are near the top eigenvalue. For example, one can take the top two eigenvalues  $\lambda_1 \geq \lambda_2$  and their eigenvectors  $v_1, v_2$ , and project the dataset onto the plane  $\text{span}(v_1, v_2)$ . The projected dataset can then be visualized as points in the plane. Similarly, one can take the top three eigenvalues  $\lambda_1 \geq \lambda_2 \geq \lambda_3$  and their eigenvectors  $v_1, v_2, v_3$  and project the dataset onto the space  $\text{span}(v_1, v_2, v_3)$ . This can then be visualized as points in three dimensions.

Recall the MNIST dataset consists of  $N = 60000$  points in  $d = 784$  dimensions. After we download the dataset,

```
from numpy import *
from keras.datasets import mnist

(train_X, train_y), (test_X, test_y) = mnist.load_data()

dataset = train_X.reshape((60000,784))
labels = train_y
```

we compute  $Q$ , the total variance, and the eigenvalues, as percentages of the total variance. We also name the targets as `labels` for later use.

```
array([[ 9.705,  9.705],
       [ 7.096, 16.801],
       [ 6.169, 22.97 ],
       [ 5.389, 28.359],
       [ 4.869, 33.228],
       [ 4.312, 37.54 ],
       [ 3.272, 40.812],
       [ 2.884, 43.696],
       [ 2.762, 46.458],
       [ 2.357, 48.815],
       [ 2.109, 50.924],
       [ 2.023, 52.947],
       [ 1.716, 54.663],
       [ 1.692, 56.355],
       [ 1.579, 57.934],
       [ 1.483, 59.417],
       [ 1.325, 60.741],
       [ 1.277, 62.018],
       [ 1.187, 63.205],
       [ 1.153, 64.358]]))
```

Figure 3.12: MNIST eigenvalues as a percentage of the total variance.

```
Q = cov(dataset.T)
totvar = Q.trace()

from numpy.linalg import eigh

# use eigh for symmetric matrices
lamda, U = eigh(Q)

# sort in ascending order then reverse
sorted = sort(lamda)[::-1]
percent = sorted*100/totvar

# cumulative sums
sums = cumsum(percent)

data = array([percent,sums])
print(data.T[:20].round(decimals=3))

d = len(lamda)
```

```
from matplotlib.pyplot import stairs
stairs(percent,range(d+1))
```

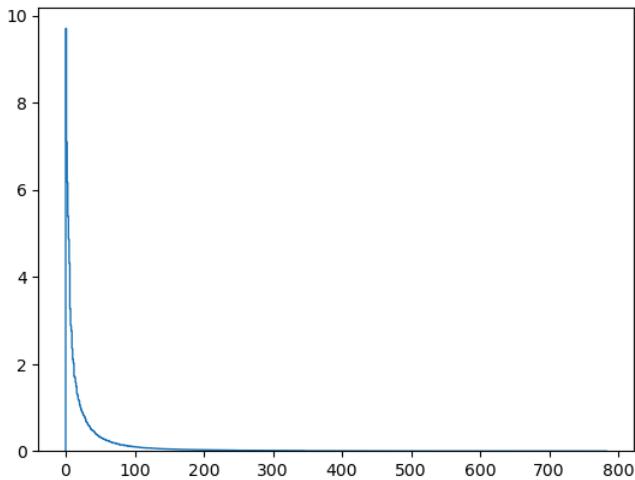


Figure 3.13: MNIST eigenvalue percentage plot.

The left column in Figure 3.12 lists the top twenty eigenvalues as a percentage of their sum. For example, the top eigenvalue  $\lambda_1$  is around 10% of the total variance. The right column lists the cumulative sums of the eigenvalues, so the third entry in the right column is the sum of the top three eigenvalues,  $\lambda_1 + \lambda_2 + \lambda_3 = 22.97\%$ .

This results in Figures 3.12 and 3.13. Here we `sort` the array `eig` in decreasing order, then we `cumsum` the array to obtain the cumulative sums.

Because the rank of the MNIST dataset is 712 (§2.9), the bottom  $72 = 784 - 712$  eigenvalues are exactly zero. A full listing shows that many more eigenvalues are near zero, and the second column in Figure 3.12 shows the top ten eigenvalues alone sum to almost 50% of the total variance.



A MNIST image is a point in  $\mathbf{R}^{784}$ . Now we turn to projecting the image from 784 dimensions down to  $n$  dimensions, where  $n$  is 784, 600, 350, 150, 50, 10, 1. Let  $Q$  be any  $d \times d$  covariance matrix, and let  $v$  be in  $\mathbf{R}^d$ . Let

$v_1, v_2, \dots, v_d$  be the orthonormal basis of eigenvectors corresponding to the eigenvalues of  $Q$ , arranged in decreasing order. Here is code that returns the projection matrix  $P$  (§2.7) onto the span of the eigenvectors  $v_1, v_2, \dots, v_n$  corresponding to the top  $n$  eigenvalues of  $Q$ .

```
from numpy import *
from numpy.linalg import eigh

# projection matrix onto top n
# eigenvectors of covariance
# of dataset

def pca(dataset,n):
    Q = cov(dataset.T)
    # columns of V are
    # eigenvectors of Q
    lamda, U = eigh(Q)
    # decreasing eigenvalue sort
    order = lamda.argsort()[:-1]
    # sorted top n columns of U
    # are cols of U
    V = U[:,order[:n]]
    P = dot(V,V.T)
    return P
```

In the code, `lamda` is sorted in decreasing order, and the sorting order is saved as `order`. To obtain the top  $n$  eigenvectors, we sort the first  $n$  columns  $U[:,order[:n]]$  in the same order, resulting in the  $d \times n$  matrix  $V$ . The code then returns the projection matrix  $P = VV^t$  (2.7.4).

Instead of working with the covariance  $Q$ , as discussed at the start of the section, we can work directly with the dataset, using `svd`, to obtain the eigenvectors.

```
from numpy import *
from numpy.linalg import svd

# projection matrix onto top n
# eigenvectors of covariance
```

```
# of dataset

def pca_with_svd(dataset,n):
    # center dataset
    m = mean(dataset, axis=0)
    vectors = dataset - m
    # rows of V are
    # right singular vectors
    V = svd(vectors)[2]
    # no need to sort, already decreasing order
    U = V[:n].T # top n rows as columns
    P = dot(U,U.T)
    return P
```

Let  $v = \text{dataset}[1]$  be the second image in the MNIST dataset, and let  $Q$  be the covariance of the dataset. Then the code below returns the image compressed down to  $n = 784, 600, 350, 150, 50, 10, 1$  dimensions, returning Figure 1.4.

```
from matplotlib.pyplot import *

figure(figsize=(10,5))
# eight subplots
rows, cols = 2, 4

v = dataset[1] # second image
display_image(v, row, col, 1)

for i, n in enumerate([784, 600, 350, 150, 50, 10, 1], start=2):
    # either will work
    P = pca_with_svd(dataset, n)
    P = pca(dataset, n)
    projv = dot(P, v)
    A = reshape(projv, (28, 28))
    subplot(rows, cols, i)
    imshow(A, cmap="gray_r")
```

If you run out of memory trying this code, cut down the dataset from 60,000 points to 10,000 points or fewer. The code works with `pca` or with

pca\_with\_svd.



We now show how to project a vector  $v$  in the dataset using `sklearn`. The following code sets up the PCA engine using `sklearn`.

```
from sklearn.decomposition import PCA

N = len(dataset)
n = 10
engine = PCA(n_components = n)
```

The following code computes the reduced dataset (§2.7)

```
reduced = engine.fit_transform(dataset)
reduced.shape
```

and returns  $(N, n) = (60000, 10)$ . The following code computes the projected dataset

```
projected = engine.inverse_transform(reduced)
projected.shape
```

and returns  $(N, d) = (60000, 784)$ .

Let  $U$  be the  $d \times n$  matrix with columns the top  $n$  eigenvectors. Then the projection matrix onto the column space of  $U$  (`project_to_ortho` in §2.7) is  $P = UU^t$ . In the above code, `reduced` equals  $U^tv$  for each image  $v$ , and `projected` is  $UU^tv$  for each image  $v$ .

Then the code

```
from matplotlib.pyplot import *

figure(figsize=(10,5))
# eight subplots
rows, cols = 2, 4

v = dataset[1] # second image
```

```

display_image(v,rows,cols,1)

for i,n in enumerate([784,600,350,150,50,10,1],start=2):
    engine = PCA(n_components = n)
    reduced = engine.fit_transform(dataset)
    projected = engine.inverse_transform(reduced)
    projv = projected[1] # second image
    A = reshape(projv,(28,28))
    subplot(rows, cols,i)
    imshow(A,cmap="gray_r")

```

returns Figure 3.14.

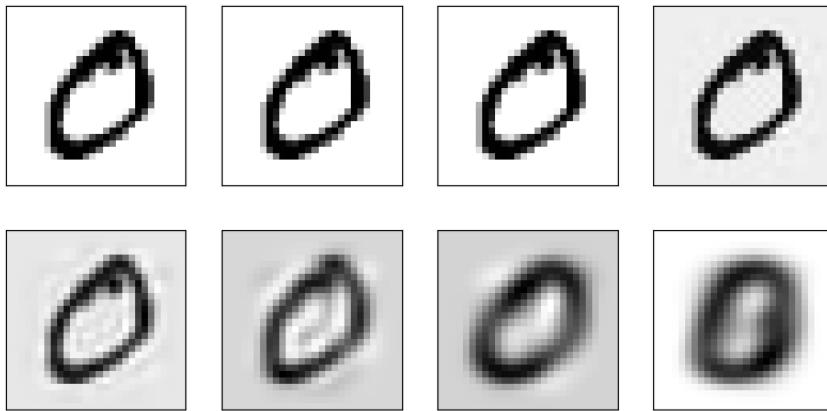


Figure 3.14: Original and projections:  $n = 784, 600, 350, 150, 50, 10, 1$ .



Now we project all vectors of the MNIST dataset onto two and three dimensions, those corresponding to the top two or three eigenvalues. To start, we compute `reduced` as above with  $n = 3$ , the top three components.

In the two-dimensional plotting code below, `reduced` is an array of shape  $(60000, 3)$ , but we use only the top two components 0 and 1. When the rows are plotted as a scatterplot, we obtain Figure 3.15. Note the rows are plotted grouped by color, to match the legend, and each plot point's color is determined by the value of its label.

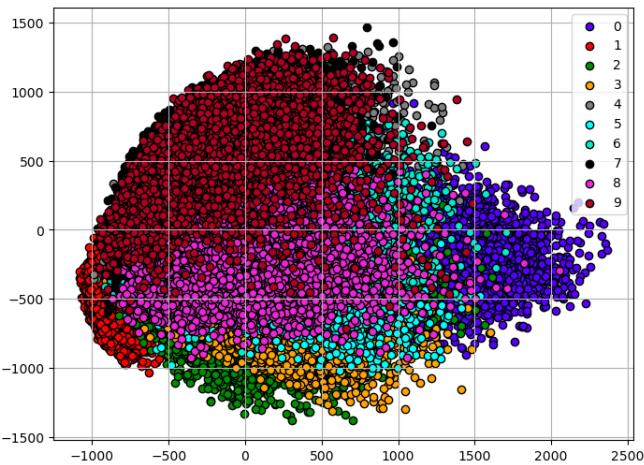


Figure 3.15: The full MNIST dataset (2d projection).

```
from matplotlib.pyplot import *

Colors = ('blue', 'red', 'green', 'orange', 'gray','cyan',
          → 'turquoise', 'black', 'orchid', 'brown')
for i,color in enumerate(Colors):
    scatter(reduced[labels==i,0], reduced[labels==i,1],
            → label=i, c=color, edgecolor='black')

grid()
legend(loc='upper right')
show()
```

Code for the 2d plot (Figure 3.16) of the Iris dataset is

```
from matplotlib.pyplot import *

Colors = ['blue', 'red', 'green']
Classes = ["Iris-setosa", "Iris-virginica", "Iris-versicolor"]

for a,b in zip(Classes,Colors):
    scatter(reduced[labels==a,0], reduced[labels==a,1],
            → label=a, c=b, edgecolor='black')
```

```
grid()
legend(loc='upper right')
show()
```

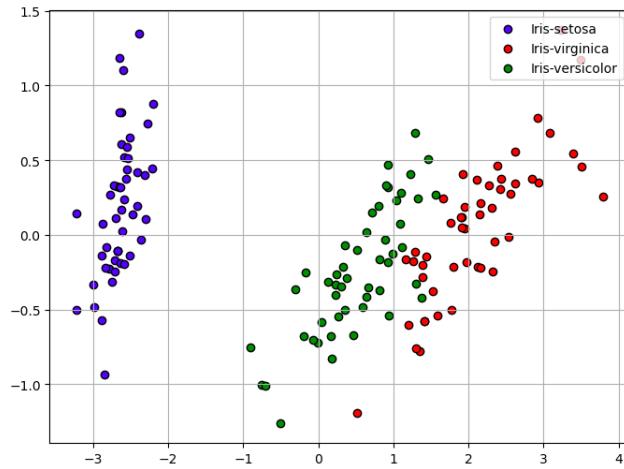


Figure 3.16: The Iris dataset (2d projection).



Now we turn to three dimensional plotting. Here is the code

```
%matplotlib notebook
from matplotlib.pyplot import *
from mpl_toolkits import mplot3d

P = axes(projection='3d')
P.set_axis_off()

Colors = ('blue', 'red', 'green', 'orange', 'gray', 'cyan' ,
          'turquoise', 'black', 'orchid', 'brown')

for i,color in enumerate(Colors):
    P.scatter(reduced[labels==i,0], reduced[labels==i,1],
```

```

    ↪ reduced[labels==i,2], label=i, c=color,
    ↪ edgecolor='black')

legend(loc='upper right')
show()

```

The three dimensional plot of the complete MNIST dataset is Figure 1.5 in §1.2. The command `%matplotlib notebook` allows the figure to rotated and scaled.

## 3.5 Cluster Analysis

★ under construction ★

Cluster analysis seeks to partition a dataset into groups or clusters based on selected criteria, such as proximity in distance.

Let  $x_1, x_2, \dots, x_N$  be a dataset in  $\mathbf{R}^d$ . The simplest algorithm is *k-means clustering*. The algorithm is iterative: We start with  $k$  means  $m_1, m_2, \dots, m_k$ , not necessarily part of the dataset, and we divide the dataset into  $k$  clusters, where the  $i$ -th cluster consists of the points  $x$  in the dataset for which mean  $m_i$  is nearest to  $x$ .

The algorithm is in two parts, the *assignment step* and the *update step*. Initially the means  $m_1, m_2, \dots, m_k$  are chosen at random, or by an educated guess, then clusters  $C_1, C_2, \dots, C_k$  are assigned, then each mean is recomputed as the mean of each cluster.

The `sklearn` package contains clustering routines, but here we write the code from scratch to illustrate the ideas. [Here](#) is an animated gif illustrating the convergence of the algorithm.

Assume the means are given as a list of length  $k$ ,

```
means = [ means[0], means[1], ... ]
```

and each cluster is a list of points (so `clusters` is a list of lists)

```
clusters = [ clusters[0], clusters[1], ... ]
```

such that

```
N == sum([ len(cluster) for cluster in clusters] )
```

Given a point  $x$ , we first select the mean closest to  $x$ :

```
from numpy import *
from numpy.linalg import norm

def nearest_index(x,means):
    i = 0
    for j,m in enumerate(means):
        n = means[i]
        if norm(x - m) < norm(x - n): i = j
    return i
```

Starting with empty clusters ( $k$  is the number of clusters), we iterate the assign/update steps until the means no longer change. If any clusters remain empty, we discard them. Here is the assignment step.

```
def assign_clusters(dataset,means):
    clusters = [ [] for m in means ]
    for x in dataset:
        i = nearest_index(x,means)
        clusters[i].append(x)
    return [ c for c in clusters if len(c) > 0 ]
```

Here is the update step.

```
def update_means(clusters):
    return [ mean(c, axis=0) for c in clusters ]
```

Here is the iteration.

```
from numpy.random import random

d = 2
k,N = 7,100
```

```

def random_vector(d):
    return array([ random() for _ in range(d) ])

dataset = [ random_vector(d) for _ in range(N) ]
means = [ random_vector(d) for _ in range(k) ]

close_enough = False

while not close_enough:
    clusters = assign_clusters(dataset,means)
    print([len(c) for c in clusters])
    newmeans = update_means(clusters)
    # only check closeness if number of means unchanged
    if len(newmeans) == len(means):
        close_enough = all([ allclose(m,n) for m,n in
            ↪ zip(means,newmeans) ])
    means = newmeans

```

This code returns the size the clusters after each iteration. Here is code that plots a cluster.

```

def plot_cluster(mean,cluster,color,marker):
    for v in cluster:
        scatter(v[0],v[1], s=50, c=color, marker=marker)
    scatter(mean[0], mean[1], s=100, c=color, marker='*')

```

Here is code for the entire iteration. hexcolor is in §1.3.

```

from matplotlib.pyplot import *

d = 2
k,N = 7,100

def random_vector(d):
    return array([ random() for _ in range(d) ])

dataset = [ random_vector(d) for _ in range(N) ]
means = [ random_vector(d) for _ in range(k) ]

```

```
colors = [ hexcolor() for _ in range(k) ]  
  
close_enough = False  
  
figure(figsize=(4,4))  
grid()  
for v in dataset: scatter(v[0],v[1],s=20,c='black')  
show()  
  
while not close_enough:  
    clusters = assign_clusters(dataset,means)  
    newmeans = update_means(clusters)  
    # only check closeness if number of means unchanged  
    if len(newmeans) == len(means):  
        close_enough = all([ allclose(m,n) for m,n in  
        → zip(means,newmeans) ])  
    figure(figsize=(4,4))  
    grid()  
    for i,c in enumerate(clusters):  
        plot_cluster(newmeans[i], c, colors[i],  
        '$' + str(i) + '$')  
    show()  
    means = newmeans
```



# Chapter 4

## Counting

Some of the material in this chapter is first seen in high school. Because repeating the exposure leads to a deeper understanding, we review it in a manner useful to the later chapters.

### 4.1 Permutations and Combinations

Suppose we have three balls in a bag, colored red, green, and blue. Suppose they are pulled out of the bag and arranged in a line. We then obtain six possibilities, listed in Figure 4.1.

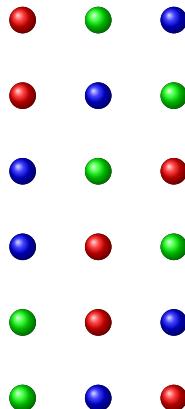


Figure 4.1:  $6 = 3!$  permutations of 3 balls.

Why are there six possibilities? Because they are three ways of choosing

the first ball, then two ways of choosing the second ball, then one way of choosing the third ball, so the total number of ways is

$$6 = 3 \times 2 \times 1.$$

In particular, we see that the number of ways *multiply*,  $6 = 3 \times 2 \times 1$ .

Similarly, there are  $5 \times 4 \times 3 \times 2 \times 1 = 120$  ways of selecting five distinct balls. Since this pattern appears frequently, it has a name.

If  $n$  is a positive integer, then  $n$ -factorial is

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1.$$

The factorial function grows large rapidly, for example,

$$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3,628,800.$$

Notice also

$$(n + 1)! = (n + 1) \times n \times (n - 1) \times \cdots \times 2 \times 1 = (n + 1) \times n!,$$

and  $(n + 2)! = (n + 2) \times (n + 1)!$ , and so on.

### Permutations of $n$ Objects

The number of ways of selecting  $n$  objects from a collection of  $n$  distinct objects is  $n!$ .

We also have

$$1! = 1, \quad 0! = 1.$$

It's clear that  $1! = 1$ . It's less clear that  $0! = 1$ , but it's reasonable if you think about it: The number of ways of selecting from zero balls results in only one possibility — no balls.



More generally, we can consider the selection of  $k$  balls from a bag containing  $n$  distinct balls. There are two varieties of selections that can be made: *Ordered selections* and *unordered selections*. An ordered selection is a *permutation*, and an unordered selection is a *combination*. In particular, when  $k = n$ ,  $n!$  is the *number of ways of permuting  $n$  objects*.

The number of permutations of  $k$  objects from  $n$  objects is written as  $P(n, k)$ , and the number of combinations of  $k$  objects from  $n$  objects is written as  $C(n, k)$ .

For ordered selections, there are  $n$  choices for the first ball,  $n - 1$  choices for the second ball, and so on, until we have  $n - k + 1$  choices for the  $k$ -th ball. Thus

$$P(n, k) = n \times (n - 1) \times \cdots \times (n - k + 1).$$

For example, there are  $5 \times 4 = 20$  ordered selections of two balls from five distinct balls. Because ordering is taken into account, selecting ball #2 then ball #3 is considered distinct from selecting ball #3 then ball #2.

### Permutation of $k$ Objects from $n$ Objects

The number of permutations of  $k$  objects from  $n$  objects is

$$P(n, k) = n(n - 1)(n - 2) \dots (n - k + 1) = \frac{n!}{(n - k)!}.$$

The last formula follows by canceling,

$$\frac{n!}{(n - k)!} = \frac{n(n - 1) \dots (n - k + 1)(n - k)!}{(n - k)!} = n(n - 1) \dots (n - k + 1).$$

Notice  $P(x, k)$  is defined for any real number  $x$  by the same formula,

$$P(x, k) = x(x - 1)(x - 2) \dots (x - k + 1).$$



When a selection of  $k$  objects is made, and the  $k$  objects are permuted, that is considered the same unordered selection, but a different ordered selection. Since the number of permutations of  $k$  objects is  $k!$ , the number  $P(n, k)$  of ordered selections is  $k!$  times the number  $C(n, k)$  of unordered selections. This leads to

### Combinations of $k$ Objects from $n$ Objects

The number of combinations of  $k$  objects from  $n$  objects is

$$C(n, k) = \frac{P(n, k)}{k!} = \frac{n!}{(n - k)!k!}.$$

For example,

$$P(5, 2) = 5 \times 4 = 20, \quad C(5, 2) = \frac{5 \times 4}{2 \times 1} = 10,$$

so we have twenty ordered pairs

$$(1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), \\ (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 5), (5, 1), (5, 2), (5, 3), (5, 4)$$

and ten unordered pairs

$$\{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 5\}.$$

The number  $C(n, k)$  is also called *n-choose-k*. Because it appears in the binomial theorem,  $C(n, k)$  is also called the *binomial coefficient* (§4.3).

Since  $P(x, k)$  is defined for any real number  $x$ , so is  $C(n, k)$ :

$$C(x, k) = \frac{P(x, k)}{k!} = \frac{x(x - 1)(x - 2) \dots (x - k + 1)}{1 \cdot 2 \cdot 3 \dots k}.$$



An important question is the *rate of growth* of the factorial function  $n!$ . Attempting to answer this question leads to the exponential (§4.4) and to the entropy (§7.2). Here is how this happens.

Since  $n!$  is a product of the  $n$  factors

$$1, 2, 3, \dots, n - 1, n,$$

each no larger than  $n$ , it is clear that

$$n! < n^n.$$

However, because half of the factors are less than  $n/2$ , we expect an approximation smaller than  $n^n$ , maybe something like  $(n/2)^n$  or  $(n/3)^n$ .

To be systematic about it, assume an approximation of the form<sup>1</sup>

$$n! \sim e \left( \frac{n}{e} \right)^n, \quad \text{for } n \text{ large,} \quad (4.1.1)$$

for some *constant*  $e$ . We seek the best constant  $e$  that fits here. In this approximation, we multiply by  $e$  so that (4.1.1) is an equality when  $n = 1$ .

Using the binomial theorem, in §4.4 we show

$$3 \left( \frac{n}{3} \right)^n \leq n! \leq 2 \left( \frac{n}{2} \right)^n, \quad n \geq 1. \quad (4.1.2)$$

Based on this, a constant  $e$  satisfying (4.1.1) must lie between 2 and 3,

$$2 \leq e \leq 3.$$

To figure out the best constant  $e$  to pick, we see how much both sides of (4.1.1) increase when we replace  $n$  by  $n + 1$ . Write (4.1.1) with  $n + 1$  replacing  $n$ , obtaining

$$(n + 1)! \sim e \left( \frac{n + 1}{e} \right)^{n+1}. \quad (4.1.3)$$

Dividing the left sides of (4.1.1), (4.1.3) yields

$$\frac{(n + 1)!}{n!} = (n + 1).$$

Dividing the right sides yields

$$\frac{e((n + 1)/e)^{n+1}}{e(n/e)^n} = (n + 1) \cdot \frac{1}{e} \cdot \left( 1 + \frac{1}{n} \right)^n. \quad (4.1.4)$$

To make these quotients match as closely as possible, we should choose

$$e \sim \left( 1 + \frac{1}{n} \right)^n. \quad (4.1.5)$$

Choosing  $n = 1, 2, 3, \dots, 100, \dots$  results in

$$e \sim 2, e \sim 2.25, e \sim 2.37, \dots, e \sim 2.705, \dots$$

---

<sup>1</sup>~means *approximately equal*.

As  $n \rightarrow \infty$ , we obtain *Euler's constant*  $e$  (§4.4).

Equation (4.1.1) can be improved to *Stirling's approximation*

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n, \quad \text{for } n \text{ large.} \quad (4.1.6)$$

This is an *asymptotic equality*: the ratio of the two sides approaches 1 as  $n$  grows to infinity.

## 4.2 Graphs

A *graph* consists of *nodes* and *edges*. For example, the graphs in Figure 4.2 each have four nodes and three edges. The left graph is *directed*, in that a direction is specified for each edge. The graph on the right is *undirected*, no direction is specified.



Figure 4.2: Directed and undirected graphs.

In a directed graph, if there is an edge pointing from node  $i$  to node  $j$ , we say  $(i, j)$  is an edge. For undirected graphs, we say  $i$  and  $j$  are *adjacent*.

An edge  $(i, j)$  is *weighed* if a scalar  $w_{ij}$  is attached to it. If every edge in a graph is weighed, then the graph is a *weighed graph*. Any two nodes may be considered adjacent by assigning the weight zero to the edge between them.

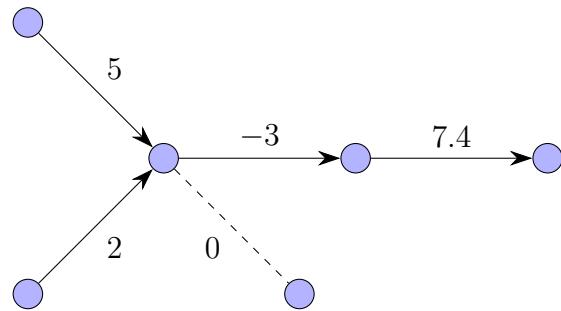


Figure 4.3: A weighed directed graph.

In §7.4, back propagation on weighed directed graphs is used to calculate derivatives.



Let  $w_{ij}$  be the weight on the edge  $(i, j)$  in a weighed directed graph. The *weight matrix* of a weighed directed graph is the matrix  $W = (w_{ij})$ .

If the graph is unweighted, then we set  $A = (a_{ij})$ , where

$$a_{ij} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ adjacent,} \\ 0, & \text{if not.} \end{cases} .$$

In this case,  $A$  consists of ones and zeros, and is called the *adjacency matrix*. If the graph is also undirected, then the adjacency matrix is symmetric,

$$a_{ij} = a_{ji}.$$

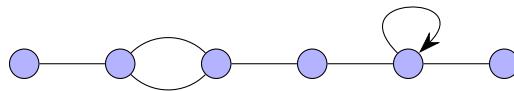


Figure 4.4: A double edge and a loop.

Sometimes graphs may have multiple edges between nodes, or loops, which are edges starting and ending at the same node. A graph is *simple* if it has no loops and no multiple edges. In this section, we deal only with *simple undirected unweighted graphs*.

To summarize, a simple undirected graph  $G = (V, E)$  is a collection  $V$  of nodes, and a collection of edges  $E$ , each edge corresponding to a pair of nodes.

The number of nodes is the *order*  $n$  of the graph, and the number of edges is the *size*  $m$  of the graph. In a (simple undirected) graph of order  $n$ , the number of pairs of nodes is  $n$ -choose-2, so the number of edges satisfies

$$0 \leq m \leq \binom{n}{2} = \frac{1}{2}n(n - 1).$$

How many graphs of order  $n$  are there? Since graphs are built out of edges, the answer depends on how many subsets of edges you can grab from

a maximum of  $n(n - 1)/2$  edges. The number of subsets of a set with  $m$  elements is  $2^m$ , so the number  $G_n$  of graphs with  $n$  nodes is

$$G_n = 2^{\binom{n}{2}} = 2^{n(n-1)/2}.$$

For example, the number of graphs with  $n = 5$  is  $2^{5(5-1)/2} = 2^{10} = 1,024$ , and the number of graphs with  $n = 10$  is

$$n = 10 \quad \implies G_n = 2^{45} = 35,184,372,088,832.$$

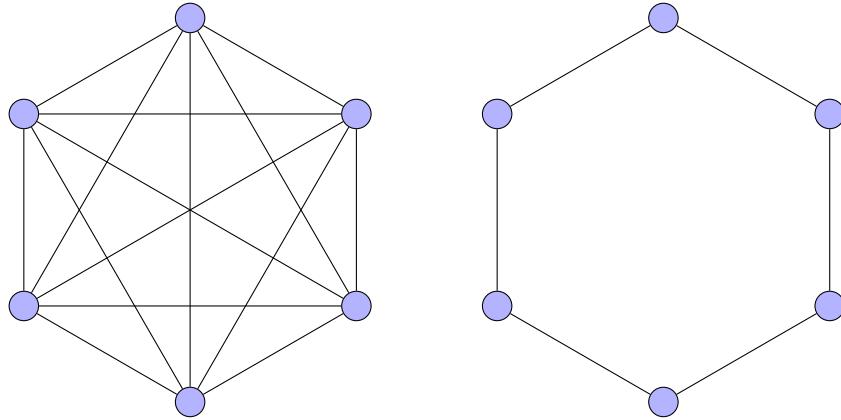


Figure 4.5: The complete graph  $K_6$  and the cycle graph  $C_6$ .

When  $m = 0$ , there are no edges, and we say the graph is *empty*. When  $m = n(n - 1)/2$ , there are the maximum number of edges, and we say the graph is *complete*. The *complete graph* with  $n$  nodes is written  $K_n$  (Figure 4.5).

The *cycle graph*  $C_n$  with  $n$  nodes is as in Figure 4.5. The graph  $C_n$  has  $n$  edges. The cycle graph  $C_3$  is a triangle.



A graph  $G'$  is a *subgraph* of a graph  $G$  if every node of  $G'$  is a node of  $G$ , and every edge of  $G'$  is an edge of  $G$ . For example, a *triangle* in  $G$  is a graph triangle that is a subgraph of  $G$ . Below we see the graph  $K_6$  in Figure 4.5 contains twenty triangles.

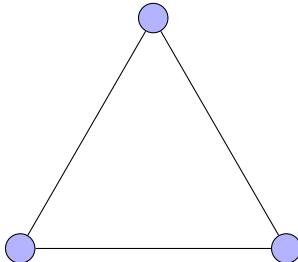


Figure 4.6: The triangle  $K_3 = C_3$ .

Let  $v$  be a node in a (simple, undirected) graph  $G$ . The *degree* of  $v$  is the number  $d_v$  of edges containing  $v$ . If the nodes are labelled  $1, 2, \dots, n$ , with the degrees in decreasing order, then

$$d_1 \geq d_2 \geq d_3 \geq \cdots \geq d_n$$

is the *degree sequence* of the graph. We write

$$(d_1, d_2, d_3, \dots, d_n)$$

for the degree sequence.

If we add the degrees over all nodes, we obtain the number of edges counted twice, because each edge contains two nodes. Thus we have

### Handshaking Lemma

If the order is  $n$ , the size is  $m$ , and the degrees are  $d_1, d_2, \dots, d_n$ , then

$$d_1 + d_2 + \cdots + d_n = \sum_{k=1}^n d_k = 2m.$$

A node is *isolated* if its degree is zero. A node is *dominating* if it has the highest degree. Notice the highest degree is  $\leq n - 1$ , because there are no loops. We show

### Nodes with Equal Degree

In any graph, there are at least two nodes with the same degree.

To see this, we consider two cases. First case, assume there are no isolated nodes. Then the degree sequence is

$$n - 1 \geq d_1 \geq d_2 \geq \dots \geq d_n \geq 1.$$

So we have  $n$  integers spread between 1 and  $n - 1$ . This can't happen unless at least two of these integers are equal. This completes the first case. In the second case, we have at least one isolated node, so  $d_n = 0$ . If  $d_{n-1} = 0$  also, then we have found two nodes with the same degree. If not, then the maximum degree is  $n - 2$  (because node  $n$  is isolated), and

$$n - 2 \geq d_1 \geq d_2 \geq \dots \geq d_{n-1} \geq 1.$$

So now we have  $n - 1$  integers spread between 1 and  $n - 2$ . This can't happen unless at least two of these integers are equal. This completes the second case.



A graph is *regular* if all the node degrees are equal. If the node degrees are all equal to  $k$ , we say the graph is  $k$ -regular. From the handshaking lemma, for a  $k$ -regular graph, we have  $kn = 2m$ , so

$$m = \frac{1}{2}kn.$$

For example, because  $2m$  is even, there are no 3-regular graphs with 11 nodes. Both  $K_n$  and  $C_n$  are regular, with  $K_n$  being  $(n - 1)$ -regular, and  $C_n$  being 2-regular.

A *walk* on a graph is a sequence of nodes  $v_1, v_2, v_3, \dots$  where each consecutive pair  $v_i, v_{i+1}$  of nodes are adjacent. For example, if  $v_1, v_2, v_3, v_4, v_5, v_6$  are the nodes (in any order) of the complete graph  $K_6$ , then  $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_2$  is a walk. A *path* is a walk with no backtracking: A path visits each node at most once. A *closed walk* is a walk that ends where it starts. A *cycle* is a closed walk with no backtracking.

Two nodes  $a$  and  $b$  are *connected* if there is a walk starting at  $a$  and ending at  $b$ . If  $a$  and  $b$  are connected, then there is a path starting at  $a$  and ending at  $b$ , since we can cut out the cycles of the walk. A graph is *connected* if every two nodes are connected. A graph is *disconnected* if it is not connected. For

example, Figure 4.5 may be viewed as two connected graphs  $K_6$  and  $C_6$ , or a single disconnected graph  $K_6 \cup C_6$ .



Consider a graph with order  $n$ . The *adjacency matrix* is the  $n \times n$  matrix  $A = (a_{ij})$  given by

$$a_{ij} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ are adjacent,} \\ 0, & \text{if not.} \end{cases}$$

For example, the empty graph has adjacency matrix given by the zero matrix. Since our graphs are undirected, the adjacency matrix is symmetric.

Let  $\mathbf{1}$  be the vector  $\mathbf{1} = (1, 1, 1, \dots, 1)$ . The adjacency matrix of the complete graph  $K_n$  is the  $n \times n$  matrix  $A$  with all ones except on the diagonal. If  $I$  is the  $n \times n$  identity matrix, then this adjacency matrix is

$$A = \mathbf{1} \otimes \mathbf{1} - I$$

For example, for the triangle  $K_3$ ,

$$A = (1 \ 1 \ 1) \otimes \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

If we label the nodes of the cycle graph  $C_n$  consecutively, then node  $i$  shares an edge with  $i - 1$  and  $i + 1$ , except when  $i = 1$  and  $i = n$ . Node 1 shares an edge with 2 and  $n$ , and node  $n$  shares an edge with  $n - 1$  and 1. So for  $C_6$  the adjacency matrix is

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Notice there are ones on the sub-diagonal, and ones on the super-diagonal, and ones in the upper-right and lower-left corners.

For any adjacency matrix  $A$ , the sum of each row is equal to the degree of the node corresponding to that row. This is the same as saying

$$A\mathbf{1} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix}.$$

In particular, for a  $k$ -regular graph, we have

$$A\mathbf{1} = k\mathbf{1},$$

so for a  $k$ -regular graph,  $k$  is an eigenvalue of  $A$ .

What is the connection between degrees and eigenvalues in general? To explain this, let  $\lambda$  be an eigenvalue of  $A$  with eigenvector  $v = (v_1, v_2, \dots, v_n)$ , so  $Av = \lambda v$ . Since a multiple  $tv$  of  $v$  is also an eigenvector, we may assume the biggest component of  $v$  equals 1. Suppose the nodes are labelled so that  $v = (1, v_2, v_3, \dots, v_n)$ , with

$$v_1 = 1 \geq |v_j|, \quad j = 2, 3, \dots, n.$$

Taking the first component of  $Av = \lambda v$ , we have

$$(Av)_1 = a_{11}v_1 + a_{12}v_2 + a_{13}v_3 + \dots + a_{1n}v_n.$$

Since the sum  $a_{11} + a_{12} + \dots + a_{1n}$  equals the degree  $d_1$  of node 1, this implies

$$d_1 = a_{11} + a_{12} + \dots + a_{1n} \geq a_{11}v_1 + a_{12}v_2 + a_{13}v_3 + \dots + a_{1n}v_n = (Av)_1 = \lambda v_1 = \lambda.$$

Since  $d_1$  is one of the degrees,  $d_1$  is no greater than the maximum degree. This explains

### Maximum Degree of Graph

If  $\lambda$  is any eigenvalue of the adjacency matrix  $A$ , then  $\lambda$  is less or equal to the maximum degree.

In particular, for a  $k$ -regular graph, the maximum degree equals  $k$ , and we already saw  $k$  is an eigenvalue, so

### Top Eigenvalue

For a  $k$ -regular graph,  $k$  is the top eigenvalue of the adjacency matrix  $A$ .

Let  $A = \mathbf{1} \otimes \mathbf{1} - I$  be the adjacency matrix of complete graph  $K_n$ . Then for any vector  $v$  orthogonal to  $\mathbf{1}$ ,

$$Av = (\mathbf{1} \otimes \mathbf{1} - I)v = (\mathbf{1} \cdot v)\mathbf{1} - v = 0 - v = -v,$$

so  $\lambda = -1$  is an eigenvalue with multiplicity  $n - 1$ . Since

$$A\mathbf{1} = (\mathbf{1} \cdot \mathbf{1})\mathbf{1} - \mathbf{1} = n\mathbf{1} - \mathbf{1} = (n - 1)\mathbf{1},$$

$n - 1$  is an eigenvalue. Hence the eigenvalues of  $A$  are  $n - 1$  with multiplicity 1 and  $-1$  with multiplicity  $n - 1$ .



Let  $A$  be the adjacency matrix of the cycle graph  $C_n$ . Since  $C_n$  is 2-regular, the top eigenvalue of  $A$  is 2. Since  $A$  is a circulant matrix, the method used to find the eigenvalues of  $Q(d)$  in §3.2 works here. However, it is immediate that

$$A = 2I - Q(n).$$

From this and by (3.2.12), the eigenvalues of  $A$  are

$$2 \cos(2\pi k/n), \quad k = 0, 1, 2, \dots, n - 1,$$

and the eigenvectors of  $A$  are the eigenvectors of  $Q(n)$ .



The *complement* of graph  $G$  is the graph  $\bar{G}$  obtained by switching 1's and 0's, so the adjacency matrix  $\bar{A}$  of  $\bar{G}$  is

$$\bar{A} = A(\bar{G}) = \mathbf{1} \otimes \mathbf{1} - I - A(G).$$

Let  $G$  be a  $k$ -regular graph, and suppose  $k = \lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$  are the eigenvalues of  $A = A(G)$ . Since  $A$  is symmetric, we have an orthogonal basis of eigenvectors  $v_1, v_2, \dots, v_n$ , with  $v_1 = \mathbf{1}$ . Then  $\bar{G}$  is an  $(n - 1 - k)$ -regular

graph, so the top eigenvalue of  $\bar{A} = A(\bar{G})$  is  $n - 1 - k$ , with eigenvector  $v_1 = \mathbf{1}$ . If  $v_k$  is any eigenvector of  $A$  other than  $\mathbf{1}$ , then  $v_k$  is orthogonal to  $\mathbf{1}$ , hence

$$\bar{A}v = (\mathbf{1} \otimes \mathbf{1} - I - A)v_k = -v - \lambda_k v_k = (-1 - \lambda_k)v_k.$$

Hence the eigenvalues of  $\bar{A}$  are  $n - 1 - k$  and  $-1 - \lambda_k$ ,  $k = 2, \dots, n$ , with the same eigenbasis.



Now we look at powers of the adjacency matrix  $A$ . By definition of matrix multiplication,

$$(A^2)_{ij} = i\text{-th row} \times j\text{-th column} = \sum_{k=1}^n a_{ik}a_{kj}.$$

Now  $a_{ik}a_{kj}$  is either 0 or 1, and equals 1 exactly if there is a 2-step path from  $i$  to  $j$ . Hence

$$(A^2)_{ij} = \text{number of 2-step walks connecting } i \text{ and } j.$$

Notice a 2-step walk between  $i$  and  $j$  is the same as a 2-step path between  $i$  and  $j$ .

When  $i = j$ ,  $(A^2)_{ii}$  is the number of 2-step paths connecting  $i$  and  $i$ , which means number of edges. Since this counts edges twice, we have

$$\frac{1}{2} \text{trace}(A^2) = m = \text{number of edges.}$$

Similarly,  $(A^3)_{ij}$  is the number of 3-step walks connecting  $i$  and  $j$ . Since a 3-step walk from  $i$  to  $i$  is the same as a triangle,  $(A^3)_{ii}$  is the number of triangles in the graph passing through  $i$ . Since the trace is the sum of the diagonal elements,  $\text{trace}(A^3)$  counts the number of triangles. But this overcounts by a factor of  $3! = 6$ , since three labels may be rearranged in six ways. Hence

$$\frac{1}{6} \text{trace}(A^3) = \text{number of triangles.}$$

### Loops, Edges, Triangles

Let  $A$  be the adjacency matrix. Then

- $\text{trace}(A) = \text{number of loops} = 0$ ,
- $\text{trace}(A^2) = 2 \times \text{number of edges}$ ,
- $\text{trace}(A^3) = 6 \times \text{number of triangles}$ .

Let us compute these for the complete graph  $K_n$ . Since

$$(u \otimes v)^2 = (u \otimes v)(u \otimes v) = (u \cdot v)(u \otimes v),$$

and  $\mathbf{1} \cdot \mathbf{1} = n$ , we have  $(\mathbf{1} \otimes \mathbf{1})^2 = n\mathbf{1} \otimes \mathbf{1}$ . So

$$A^2 = (\mathbf{1} \otimes \mathbf{1} - I)^2 = (\mathbf{1} \otimes \mathbf{1})^2 - 2\mathbf{1} \otimes \mathbf{1} + I = (n-2)\mathbf{1} \otimes \mathbf{1} + I.$$

Since  $\text{trace}(u \otimes v) = u \cdot v$ , we have  $\text{trace}(\mathbf{1} \otimes \mathbf{1}) = n$ . Hence

$$\text{trace}(A^2) = \text{trace}((n-2)\mathbf{1} \otimes \mathbf{1} + I) = n(n-2) + n = n(n-1).$$

This is correct because for a complete graph,  $n(n-1)/2$  is the number of edges.

Continuing,

$$\begin{aligned} A^3 &= A^2 A = ((n-2)\mathbf{1} \otimes \mathbf{1} + I)(\mathbf{1} \otimes \mathbf{1} - I) \\ &= n(n-2)\mathbf{1} \otimes \mathbf{1} - (n-2)\mathbf{1} \otimes \mathbf{1} + \mathbf{1} \otimes \mathbf{1} - I \\ &= (n^2 - 3n + 3)\mathbf{1} \otimes \mathbf{1} - I. \end{aligned}$$

From this, we get

$$\text{trace}(A^3) = n(n^2 - 3n + 3) - n = n(n^2 - 3n + 2) = n(n-1)(n-2).$$

This is correct because for a complete graph, we have a triangle whenever we have a triple of nodes, and there are  $n$ -choose-3 triples, which equals  $n(n-1)(n-2)/6$ .

Remember, a graph is connected if there is a walk connecting any two nodes. Since there is a 4-step walk between  $i$  and  $j$  exactly when there are  $r$ ,  $s$ , and  $t$  satisfying

$$a_{ir}a_{rs}a_{st}a_{tj} = 1,$$

we see there is a 4-step walk connecting  $i$  and  $j$  if  $(A^4)_{ij} > 0$ . Hence

### Connected Graph

Let  $A$  be the adjacency matrix. Then the graph is connected if for every  $i \neq j$ , there is a  $k$  with  $(A^k)_{ij} > 0$ .



Two graphs are *isomorphic* if a re-labelling of the nodes in one makes it identical to the other. To explain this, we need permutations.

A *permutation* on  $n$  letters is a re-arrangement of  $1, 2, 3, \dots, n$ . Here are two permutations of  $(1, 2, 3, 4)$ ,

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \end{pmatrix}.$$

There are  $n!$  permutations of  $(1, 2, \dots, n)$ . If a permutation sends  $i$  to  $j$ , we write  $i \rightarrow j$ . Since a permutation is just a re-labelling, if  $i \rightarrow k$  and  $j \rightarrow k$ , then we must have  $i = j$ .

Each permutation leads to a permutation matrix. A *permutation matrix* is a matrix of zeros and ones, with only one 1 in any column or row. For example, the above permutations correspond to the  $4 \times 4$  matrices

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

In general, the permutation matrix  $P$  has  $P_{ij} = 1$  if  $i \rightarrow j$ , and  $P_{ij} = 0$  if not. If  $P$  is any permutation matrix, then  $P_{ik}P_{jk}$  equals 1 if both  $i \rightarrow k$  and  $j \rightarrow k$ . In other words,  $P_{ik}P_{jk} = 1$  if  $i = j$  and  $i \rightarrow k$ , and  $P_{ik}P_{jk} = 0$  otherwise. Since  $i \rightarrow k$  for exactly one  $k$ ,

$$(PP^t)_{ij} = \sum_{k=1}^n P_{ik}P_{kj}^t = \sum_{k=1}^n P_{ik}P_{jk} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Hence  $P$  is orthogonal,

$$PP^t = I, \quad P^{-1} = P^t.$$

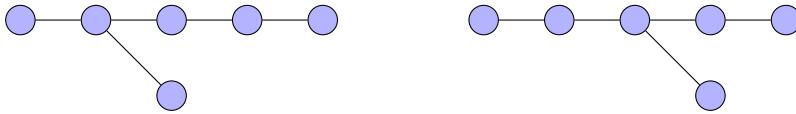


Figure 4.7: Non-isomorphic graphs with degree sequence  $(3, 2, 2, 1, 1)$ .

Using permutation matrices, we can say two graphs are isomorphic if their adjacency matrices  $A$ ,  $A'$  satisfy

$$A' = PAP^{-1} = PAP^t$$

for some permutation matrix  $P$ .

If two graphs are isomorphic, then it is easy to check their degree sequences are equal. However, the converse is not true. Figure 4.7 displays two non-isomorphic graphs with degree sequences  $(3, 2, 2, 1, 1)$ . These graphs are non-isomorphic because in one graph, there are two degree-one nodes adjacent to a degree-three node, while in the other graph, there is only one degree-one node adjacent to a degree-three node.

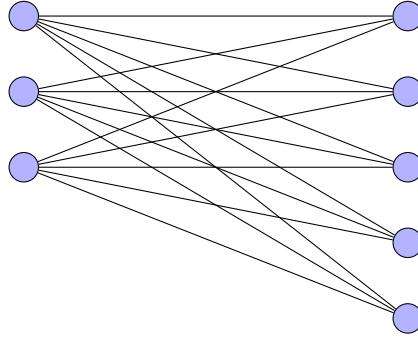


A graph is *bipartite* if the nodes can be divided into two groups, with adjacency only between nodes across groups. If we call the two groups even and odd, then odd nodes are never adjacent to odd nodes, and even nodes are never adjacent to even nodes.

The *complete bipartite* graph is the bipartite graph with maximum number of edges: Every odd node is adjacent to every even node. The complete bipartite graph with  $n$  odd nodes with  $m$  even nodes is written  $K_{nm}$ . Then the order of  $K_{mn}$  is  $n + m$ .

Let  $a = (1, 1, \dots, 1, 0, 0, \dots, 0)$  be the vector with  $n$  ones and  $m$  zeros, and let  $b = \mathbf{1} - a$ . Then  $b$  has  $n$  zeros and  $m$  ones, and the adjacency matrix of  $K_{nm}$  is

$$A = A(K_{nm}) = a \otimes b + b \otimes a.$$

Figure 4.8: Complete bipartite graph  $K_{5,3}$ .

For example, the adjacency matrix of  $K_{5,3}$  is  $A = A(K_{nm})$  which equals

$$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Recall we have

$$(a \otimes b)v = (b \cdot v)a.$$

From this, we see the column space of  $A = a \otimes b + b \otimes a$  is  $\text{span}(a, b)$ . Thus the rank of  $A$  is 2, and the nullspace of  $A$  consists of the orthogonal complement  $\text{span}(a, b)^\perp$  of  $\text{span}(a, b)$ . Using this, we compute the eigenvalues of  $A$ .

Since the nullspace is  $\text{span}(a, b)^\perp$ , any vector orthogonal to  $a$  and to  $b$  is an eigenvector for  $\lambda = 0$ . Hence the eigenvalue  $\lambda = 0$  has multiplicity  $n + m - 2$ . Since  $\text{trace}(A) = 0$ , the sum of the eigenvalues is zero, and the remaining two eigenvalues are  $\pm\lambda \neq 0$ .

Let  $v$  be an eigenvector for  $\lambda \neq 0$ . Then  $v$  is orthogonal to the nullspace of  $A$ , so  $v$  must be a linear combination of  $a$  and  $b$ ,  $v = ra + sb$ . Since  $a \cdot b = 0$ ,

$$Aa = nb, \quad Ab = ma.$$

Hence

$$\lambda v = Av = A(ra + sb) = rnb + sma.$$

Applying  $A$  again,

$$\lambda^2 v = A^2 v = A(rnb + sma) = rnma + smnb = nm(ra + sb) = nmv.$$

Hence  $\lambda = \sqrt{nm}$ . We conclude the eigenvalues of  $K_{nm}$  are

$$\sqrt{nm}, 0, 0, \dots, 0, -\sqrt{nm}, \quad (\text{with } 0 \text{ repeated } n+m-2 \text{ times}).$$

For example, for the graph in Figure 4.8, the nonzero eigenvalues are  $\lambda = \pm\sqrt{3 \times 5} = \pm\sqrt{15}$ .



Let  $G$  be a graph with  $n$  nodes and  $m$  edges. The *incidence matrix* of  $G$  is a matrix whose rows are indexed by the edges, and whose columns are indexed by the nodes. Therefore, the incidence matrix has shape  $m \times n$ .

By placing arrows along the edges, we can make  $G$  into a directed graph. In a directed graph, each edge has a *tail node* and a *head node*. Then the incidence matrix is given by

$$B_{ij} = \begin{cases} 1, & \text{if node } j \text{ is the head of edge } i, \\ -1, & \text{if node } j \text{ is the tail of edge } i, \\ 0, & \text{if node } j \text{ is not on edge } i. \end{cases}$$

The *laplacian* of a graph  $G$  is the symmetric  $n \times n$  matrix

$$L = B^t B.$$

Both the laplacian matrix and the adjacency matrix are  $n \times n$ . What is the connection between them?

### Laplacian

The laplacian satisfies

$$L = D - A,$$

where  $D = \text{diag}(d_1, d_2, \dots, d_n)$  is the diagonal degree matrix.

For example, for the cycle graph  $C_6$ , the degree matrix is  $2I$ , and the laplacian is the matrix we saw in §3.2,

$$L = Q(6) = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ -1 & 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

### 4.3 Binomial Theorem

Let  $x$  and  $a$  be two variables. A *binomial* is an expression of the form

$$(a + x)^2, \quad (a + x)^3, \quad (a + x)^4, \quad \dots$$

The *degree* of each of these binomials is 2, 3, and 4.

When binomials are expanded by multiplying out, one obtains a sum of terms. The *binomial theorem* specifies the exact pattern or form of the resulting sum.

Recall that

$$(a + b)(c + d) = a(c + d) + b(c + d) = ac + ad + bc + bd.$$

Similarly,

$$(a + b)(c + d + e) = a(c + d + e) + b(c + d + e) = ac + ad + ae + bc + bd + be.$$

Using this algebra, we can expand each binomial.

Expanding  $(a + x)^2$  yields

$$(a + x)^2 = (a + x)(a + x) = a^2 + xa + ax + x^2 = a^2 + 2ax + x^2. \quad (4.3.1)$$

Similarly, for  $(a + x)^3$ , we have

$$\begin{aligned} (a + x)^3 &= (a + x)(a + x)^2 = (a + x)(a^2 + 2ax + x^2) \\ &= a^3 + 2a^2x + ax^2 + xa^2 + 2xax + x^3 \\ &= a^3 + 3a^2x + 3ax^2 + x^3. \end{aligned} \quad (4.3.2)$$

For  $(a + x)^4$ , we have

$$\begin{aligned}(a + x)^4 &= (a + x)(a + x)^3 = (a + x)(a^3 + 3a^2x + 3ax^2 + x^3) \\&= a^4 + 3a^3x + 3a^2x^2 + ax^3 + a^3x + 3a^2x^2 + 3ax^3 + x^4 \quad (4.3.3) \\&= a^4 + 4a^3x + 6a^2x^2 + 4ax^3 + x^4.\end{aligned}$$

Thus

$$\begin{aligned}(a + x)^2 &= a^2 + 2ax + x^2 \\(a + x)^3 &= a^3 + 3a^2x + 3ax^2 + x^3 \\(a + x)^4 &= a^4 + 4a^3x + 6a^2x^2 + 4ax^3 + x^4 \quad (4.3.4) \\(a + x)^5 &= \star a^5 + \star a^4x + \star a^3x^2 + \star a^2x^3 + \star ax^4 + \star x^5.\end{aligned}$$

Here  $\star$  means we haven't found the coefficient yet.



There is a pattern in (4.3.4). In the first line, the powers of  $a$  are in decreasing order, 2, 1, 0, while the powers of  $x$  are in increasing order, 0, 1, 2. In the second line, the powers of  $a$  decrease from 3 to 0, while the powers of  $x$  increase from 0 to 3. In the third line, the powers of  $a$  decrease from 4 to 0, while the powers of  $x$  increase from 0 to 4.

This pattern of powers is simple and clear. Now we want to find the pattern for the coefficients in front of each term. In (4.3.4), these coefficients are  $(1, 2, 1)$ ,  $(1, 3, 3, 1)$ ,  $(1, 4, 6, 4, 1)$ , and  $(\star, \star, \star, \star, \star, \star)$ . These coefficients are the *binomial coefficients*.

Before we determine the pattern, we introduce a useful notation for these coefficients by writing

$$\binom{2}{0} = 1, \quad \binom{2}{1} = 2, \quad \binom{2}{2} = 1$$

and

$$\binom{3}{0} = 1, \quad \binom{3}{1} = 3, \quad \binom{3}{2} = 3, \quad \binom{3}{3} = 1$$

and

$$\binom{4}{0} = 1, \quad \binom{4}{1} = 4, \quad \binom{4}{2} = 6, \quad \binom{4}{3} = 4, \quad \binom{4}{4} = 1$$

and

$$\binom{5}{0} = \star, \quad \binom{5}{1} = \star, \quad \binom{5}{2} = \star, \quad \binom{5}{3} = \star, \quad \binom{5}{4} = \star, \quad \binom{5}{5} = \star.$$

With this notation, the number

$$\binom{n}{k} \tag{4.3.5}$$

is the *coefficient of  $a^{n-k}x^k$  when you multiply out  $(a+x)^n$* . This is the binomial coefficient. Here  $n$  is the degree of the binomial, and  $k$ , which specifies the term in the resulting sum, varies from 0 to  $n$  (not 1 to  $n$ ).

It is important to remember that, in this notation, the binomial  $(a+x)^2$  expands into the sum of *three* terms  $a^2$ ,  $2ax$ ,  $x^2$ . These are term 0, term 1, and term 2. Alternatively, one says these are the *zeroth term*, the *first term*, and the *second term*. Thus the second term in the expansion of the binomial  $(a+x)^4$  is  $6a^2x^2$ , and the binomial coefficient  $\binom{4}{2} = 6$ . In general, the binomial  $(a+x)^n$  of degree  $n$  expands into a sum of  $n+1$  terms.

Since the binomial coefficient  $\binom{n}{k}$  is the coefficient of  $a^{n-k}x^k$  when you multiply out  $(a+x)^n$ , we have the binomial theorem.

### Binomial Theorem

The binomial  $(a+x)^n$  equals

$$\binom{n}{0}a^n + \binom{n}{1}a^{n-1}x + \binom{n}{2}a^{n-2}x^2 + \cdots + \binom{n}{n-1}ax^{n-1} + \binom{n}{n}x^n. \tag{4.3.6}$$

Using summation notation, the binomial theorem states

$$(a+x)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} x^k. \tag{4.3.7}$$



The binomial coefficient  $\binom{n}{k}$  is called “*n-choose-k*”, because it is the coefficient of the term corresponding to choosing  $k$   $x$ ’s when multiplying the  $n$  factors in the product

$$(a+x)^n = (a+x)(a+x)(a+x) \dots (a+x).$$

For example, the term  $\binom{4}{2}a^2x^2$  corresponds to choosing two  $a$ 's, and two  $x$ 's, when multiplying the four factors in the product

$$(a+x)^4 = (a+x)(a+x)(a+x)(a+x).$$

The binomial coefficients may be arranged in a triangle, *Pascal's triangle* (Figure 4.9). Can you figure out the numbers  $\star$  in this triangle before peeking ahead?

$n = 0:$	1					
$n = 1:$	1      1					
$n = 2:$	1      2      1					
$n = 3:$	1      3      3      1					
$n = 4:$	1      4      6      4      1					
$n = 5:$	1      5      10     10     5      1					
$n = 6:$	$\star$ 6      15     20     15     6 $\star$					
$n = 7:$	1 $\star$ 21     35     35     21 $\star$ 1					
$n = 8:$	1      8 $\star$ 56     70     56 $\star$ 8      1					
$n = 9:$	1      9      36 $\star$ 126     126 $\star$ 36     9      1					
$n = 10:$	1      10     45     120 $\star$ 252 $\star$ 120     45     10     1					

Figure 4.9: Pascal's triangle.

In Pascal's triangle, the very top row has one number in it: This is the *zeroth row* corresponding to  $n = 0$  and the binomial expansion of  $(a+x)^0 = 1$ . The *first row* corresponds to  $n = 1$ ; it contains the numbers  $(1, 1)$ , which correspond to the binomial expansion of  $(a+x)^1 = 1a + 1x$ . We say the *zeroth entry* ( $k = 0$ ) in the *first row* ( $n = 1$ ) is 1 and the *first entry* ( $k = 1$ ) in the first row is 1. Similarly, the *zeroth entry* ( $k = 0$ ) in the *second row* ( $n = 2$ ) is 1, and the *second entry* ( $k = 2$ ) in the *second row* ( $n = 2$ ) is 1. The *second entry* ( $k = 2$ ) in the *fourth row* ( $n = 4$ ) is 6. For every row, the entries are counted starting from  $k = 0$ , and end with  $k = n$ , so there are  $n+1$  entries in row  $n$ . With this understood, the  $k$ -th entry in the  $n$ -th row is the binomial coefficient  $n$ -choose- $k$ . So 10-choose-2 is

$$\binom{10}{2} = 45.$$



We can learn a lot about the binomial coefficients from this triangle. First, we have 1's all along the left edge. Next, we have 1's all along the right edge. Similarly, one step in from the left or right edge, we have the row number. Thus we have

$$\binom{n}{0} = 1 = \binom{n}{n}, \quad \binom{n}{1} = n = \binom{n}{n-1}, \quad n \geq 1.$$

Note also Pascal's triangle has a left-to-right symmetry: If you read off the coefficients in a particular row, you can't tell if you're reading them from left to right, or from right to left. It's the same either way: The fifth row is  $(1, 5, 10, 10, 5, 1)$ . In terms of our notation, this is written

$$\binom{n}{k} = \binom{n}{n-k}, \quad 0 \leq k \leq n;$$

the binomial coefficients remain unchanged when  $k$  is replaced by  $n - k$ .

The key step in finding a formula for  $n$ -choose- $k$  is to notice

$$(a + x)^{n+1} = (a + x)(a + x)^n.$$

Let's work this out when  $n = 3$ . Then the left side is  $(a + x)^4$ . From (4.3.4), we get

$$\begin{aligned} \binom{4}{0}a^4 + \binom{4}{1}a^3x + \binom{4}{2}a^2x^2 + \binom{4}{3}ax^3 + \binom{4}{4}x^4 \\ = (a + x) \left( \binom{3}{0}a^3 + \binom{3}{1}a^2x + \binom{3}{2}ax^2 + \binom{3}{3}x^3 \right) \\ = \binom{3}{0}a^4 + \binom{3}{1}a^3x + \binom{3}{2}a^2x^2 + \binom{3}{3}ax^3 \\ \quad + \binom{3}{0}a^3x + \binom{3}{1}a^2x^2 + \binom{3}{2}ax^3 + \binom{3}{3}x^4 \\ = \binom{3}{0}a^4 + \left( \binom{3}{1} + \binom{3}{0} \right) a^3x + \left( \binom{3}{2} + \binom{3}{1} \right) a^2x^2 \\ \quad + \left( \binom{3}{3} + \binom{3}{2} \right) ax^3 + \binom{3}{3}x^4. \end{aligned}$$

Equating corresponding coefficients of  $x$ , we get,

$$\binom{4}{1} = \binom{3}{1} + \binom{3}{0}, \quad \binom{4}{2} = \binom{3}{2} + \binom{3}{1}, \quad \binom{4}{3} = \binom{3}{3} + \binom{3}{2}.$$

In general, a similar calculation establishes

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}, \quad 1 \leq k \leq n. \quad (4.3.8)$$

This allows us to build Pascal's triangle (Figure 4.9), where, apart from the ones on either end, each term ("the child") in a given row is the sum of the two terms ("the parents") located directly above in the previous row.



Insert  $x = 1$  and  $a = 1$  in the binomial theorem to get

$$2^n = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n-1} + \binom{n}{n}. \quad (4.3.9)$$

We conclude *the sum of the binomial coefficients along the  $n$ -th row of Pascal's triangle is  $2^n$*  (remember  $n$  starts from 0).

Now insert  $x = 1$  and  $a = -1$ . You get

$$0 = \binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \cdots \pm \binom{n}{n-1} \pm \binom{n}{n}.$$

Hence: *the alternating<sup>2</sup> sum of the binomial coefficients along the  $n$ -th row of Pascal's triangle is zero.*



We now show

---

<sup>2</sup>Alternating means the plus-minus pattern  $+ - + - + - \dots$

### Binomial Coefficient

The binomial coefficient  $\binom{n}{k}$  equals  $C(n, k)$ ,

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{1 \cdot 2 \cdots k} = \frac{n!}{k!(n-k)!}, \quad 1 \leq k \leq n. \quad (4.3.10)$$

To establish (4.3.10), because

$$C(0, 0) = 1 = \binom{0}{0},$$

it is enough to show  $C(n, k)$  also satisfies (4.3.8),

$$C(n+1, k) = C(n, k) + C(n, k-1), \quad 1 \leq k \leq n. \quad (4.3.11)$$

To establish (4.3.11), we simplify

$$\begin{aligned} C(n, k) + C(n, k-1) &= \frac{n!}{k!(n-k)!} + \frac{n!}{(k-1)!(n-k+1)!} \\ &= \frac{n!}{(k-1)!(n-k)!} \left( \frac{1}{k} + \frac{1}{n-k+1} \right) \\ &= \frac{n!(n+1)}{(k-1)!(n-k)!k(n-k+1)} \\ &= \frac{(n+1)!}{k!(n+1-k)!} = C(n+1, k). \end{aligned}$$

This establishes (4.3.11), and, consequently, (4.3.10).

For example,

$$\binom{7}{3} = \frac{7 \cdot 6 \cdot 5}{1 \cdot 2 \cdot 3} = 35 = \binom{7}{4} \quad \text{and} \quad \binom{10}{2} = \frac{10 \cdot 9}{1 \cdot 2} = 45 = \binom{10}{8}.$$

The formula (4.3.10) is easy to remember: There are  $k$  terms in the numerator as well as the denominator, the factors in the denominator increase starting from 1, and the factors in the numerator decrease starting from  $n$ .

In Python, the code

```
from scipy.special import comb
comb(n,k)
comb(n,k,exact=True)
```

returns the binomial coefficient.



The binomial coefficient  $\binom{n}{k}$  makes sense even for fractional  $n$ . This can be seen from (4.3.10). For example, for  $n = 1/2$  and  $k = 3$ ,

$$\binom{1/2}{3} = \frac{\frac{1}{2} \left(\frac{1}{2} - 1\right) \left(\frac{1}{2} - 2\right)}{1 \cdot 2 \cdot 3} = \frac{(1/2)(-1/2)(-3/2)}{6} = \frac{3}{48}. \quad (4.3.12)$$

This works also for  $n$  negative,

$$\binom{-1/2}{3} = \frac{\left(-\frac{1}{2}\right) \left(-\frac{1}{2} - 1\right) \left(-\frac{1}{2} - 2\right)}{1 \cdot 2 \cdot 3} = \frac{(-1/2)(-3/2)(-5/2)}{6} = \frac{15}{48}. \quad (4.3.13)$$

In fact, in (4.3.10),  $n$  may be any real number, for example  $n = \sqrt{2}$ .

## 4.4 Exponential Function

In this section, our first goal is to derive (4.1.2), as promised in §4.1.

To begin, use the binomial theorem (4.3.7) with  $a = 1$  and  $x = 1/n$ , obtaining

$$\left(1 + \frac{1}{n}\right)^n = \sum_{k=0}^n \binom{n}{k} 1^{n-k} \left(\frac{1}{n}\right)^k = \sum_{k=0}^n \frac{1}{k!} \frac{n(n-1)(n-2)\dots(n-k+1)}{n \cdot n \cdot n \cdots n}.$$

Rewriting this by pulling out the first two terms  $k = 0$  and  $k = 1$  leads to

$$\left(1 + \frac{1}{n}\right)^n = 1 + 1 + \sum_{k=2}^n \frac{1}{k!} \left(1 - \frac{1}{n}\right) \left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right). \quad (4.4.1)$$

From (4.4.1), we can tell a lot. First, since all terms are positive, we see

$$\left(1 + \frac{1}{n}\right)^n \geq 2, \quad n \geq 1.$$

Second, each factor in (4.4.1) is of the form

$$\left(1 - \frac{j}{n}\right), \quad 1 \leq j \leq k-1. \quad (4.4.2)$$

Since  $n$  is in the denominator, each such factor *increases* with  $n$ . Moreover, as  $n$  increases, the *number of terms* in (4.4.1) increases, hence so does the sum. We conclude

$$\left(1 + \frac{1}{n}\right)^n \quad \text{increases as } n \text{ increases.}$$

Therefore, as  $n$  increases without bound, there is a definite limit

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n.$$

From above, we have  $e \geq 2$ .

Third, when  $k \geq 2$ , we know

$$k! = k(k-1)(k-2)\dots 3 \cdot 2 \geq 2^{k-1}.$$

Since each factor in (4.4.2) is no greater than 1, by (4.4.1),

$$\left(1 + \frac{1}{n}\right)^n \leq 1 + 1 + \sum_{k=2}^n \frac{1}{k!} \leq 2 + \sum_{k=2}^n \frac{1}{2^{k-1}}. \quad (4.4.3)$$

But the sum

$$s_n = \sum_{k=2}^n \frac{1}{2^{k-1}} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^{n-1}}$$

may be easily estimated as follows.

Multiplying  $s_n$  by 2 doubles each term, and results in almost the same sum, so

$$2s_n = 1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^{n-2}} = 1 + s_n - \frac{1}{2^{n-1}}.$$

Solving for  $s_n$ , we conclude

$$\sum_{k=2}^n \frac{1}{2^{k-1}} = s_n = 1 - \frac{1}{2^{n-1}} \leq 1, \quad n \geq 2.$$

By (4.4.3), we arrive at

$$2 \leq \left(1 + \frac{1}{n}\right)^n \leq 3, \quad n \geq 1. \quad (4.4.4)$$

Summarizing, we established the following strengthening of (4.1.5).

### Euler's Constant

The limit

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n \quad (4.4.5)$$

exists and satisfies  $2 \leq e \leq 3$ .



We use (4.4.4) to establish (4.1.2). Write (4.1.2) as  $a_n \leq b_n \leq c_n$ . When  $n = 1$ ,

$$a_1 = b_1 = c_1.$$

Moreover, as  $n$  increases,  $a_n$ ,  $b_n$ ,  $c_n$  all increase. Therefore, to establish (4.1.2), it is enough to show  $b_n$  increases faster than  $a_n$ , and  $c_n$  increases faster than  $b_n$ , both as  $n$  increases.

To measure how  $a_n$ ,  $b_n$ ,  $c_n$  increase with  $n$ , divide the  $(n+1)$ -st term by the  $n$ -th term: It is enough to show

$$\frac{a_{n+1}}{a_n} \leq \frac{b_{n+1}}{b_n} \leq \frac{c_{n+1}}{c_n}.$$

But we already know

$$\frac{b_{n+1}}{b_n} = n + 1,$$

and, from (4.4.4),

$$\frac{a_{n+1}}{a_n} = \frac{3((n+1)/3)^{n+1}}{3(n/3)^n} = (n+1) \cdot \frac{1}{3} \cdot \left(1 + \frac{1}{n}\right)^n \leq n + 1 = \frac{b_{n+1}}{b_n},$$

and, from (4.4.4) again,

$$\frac{b_{n+1}}{b_n} = n + 1 \leq (n + 1) \cdot \frac{1}{2} \cdot \left(1 + \frac{1}{n}\right)^n = \frac{2((n + 1)/2)^{n+1}}{2(n/2)^n} = \frac{c_{n+1}}{c_n}.$$

Since we've shown  $b_n$  increases faster than  $a_n$ , and  $c_n$  increases faster than  $b_n$ , we have derived (4.1.2).



By definition, Euler's constant  $e$  satisfies (4.4.5). To obtain a second formula for  $e$ , insert  $n = \infty$  in (4.4.1), which means let  $n$  grow to infinity without bound in (4.4.1). Using  $1/\infty = 0$ , since the  $k$ -th term approaches  $1/k!$ , and since the number of terms increases with  $n$ , we obtain the second formula

$$e = 1 + 1 + \sum_{k=2}^{\infty} \frac{1}{k!} \left(1 - \frac{1}{\infty}\right) \left(1 - \frac{2}{\infty}\right) \dots \left(1 - \frac{k-1}{\infty}\right) = \sum_{k=0}^{\infty} \frac{1}{k!}.$$

To summarize,

### Euler's Constant

Euler's constant satisfies

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \frac{1}{720} + \dots$$



Depositing one dollar in a bank offering 100% interest returns two dollars after one year. Depositing one dollar in a bank offering the same annual interest compounded at mid-year returns

$$\left(1 + \frac{1}{2}\right)^2 = 2.25$$

dollars after one year.

Depositing one dollar in a bank offering the same annual interest compounded at  $n$  intermediate time points returns  $(1 + 1/n)^n$  dollars after one year.

Passing to the limit, depositing one dollar in a bank and continuously compounding at an annual interest rate of 100% returns  $e$  dollars after one year. Because of this, (4.4.5) is often called the *compound-interest formula*.



Now we derive the result of continuously compounding at any specified annual interest rate  $x$ . Note here  $x$  is a proportion, not a percent. An interest rate of 30% corresponds to  $x = .3$  in the exponential function.

### Exponential Function

For any real number  $x$ , the limit

$$\exp x = \lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n \quad (4.4.6)$$

exists. In particular,  $\exp 0 = 1$  and  $\exp 1 = e$ .

Note, in the compound-interest interpretation, when  $x > 0$ , the bank is giving you interest, while, if  $x < 0$ , the bank is taking away interest, leading to a continual loss.

To derive this, assume first  $x > 0$  is a positive real number. Then, exactly as before, using the binomial theorem,

$$\left(1 + \frac{x}{n}\right)^n, \quad n \geq 1,$$

is increasing with  $n$ , so the limit in (4.4.6) is well-defined.

To establish the existence of the limit when  $x < 0$ , we first show

$$(1 - x)^n \geq 1 - nx, \quad 0 < x < 1, n \geq 1. \quad (4.4.7)$$

This follows inductively: Each of the following inequalities is implied by the

preceding one,

$$\begin{aligned}
 (1-x) &= 1-x \\
 (1-x)^2 &= 1-2x+x^2 \geq 1-2x \\
 (1-x)^3 &= (1-x)(1-x)^2 \geq (1-x)(1-2x) = 1-3x+2x^2 \geq 1-3x \\
 (1-x)^4 &= (1-x)(1-x)^3 \geq (1-x)(1-3x) = 1-4x+3x^3 \geq 1-4x \\
 &\dots &&\dots
 \end{aligned}$$

This establishes (4.4.7) for all  $n \geq 1$ .

Now let  $x$  be any real number. Then, for  $n$  large enough,  $x^2/n^2$  lies between 0 and 1. Replacing  $x$  by  $x^2/n^2$  in (4.4.7), we obtain

$$1 \geq \left(1 - \frac{x^2}{n^2}\right)^n \geq 1 - \frac{x^2}{n}.$$

As  $n \rightarrow \infty$ , both sides of this last equation approach 1, so

$$\lim_{n \rightarrow \infty} \left(1 - \frac{x^2}{n^2}\right)^n = 1. \quad (4.4.8)$$

Now let  $n$  grow without bound in

$$\left(1 + \frac{x}{n}\right)^n \left(1 - \frac{x}{n}\right)^n = \left(1 - \frac{x^2}{n^2}\right)^n.$$

Since the limit  $\exp x$  is well-defined when  $x > 0$ , by (4.4.8), we obtain

$$\exp x \cdot \lim_{n \rightarrow \infty} \left(1 - \frac{x}{n}\right)^n = 1, \quad x > 0.$$

This shows the limit  $\exp x$  in (4.4.6) is well-defined when  $x < 0$ , and

$$\exp(-x) = \frac{1}{\exp x}, \quad \text{for all } x.$$



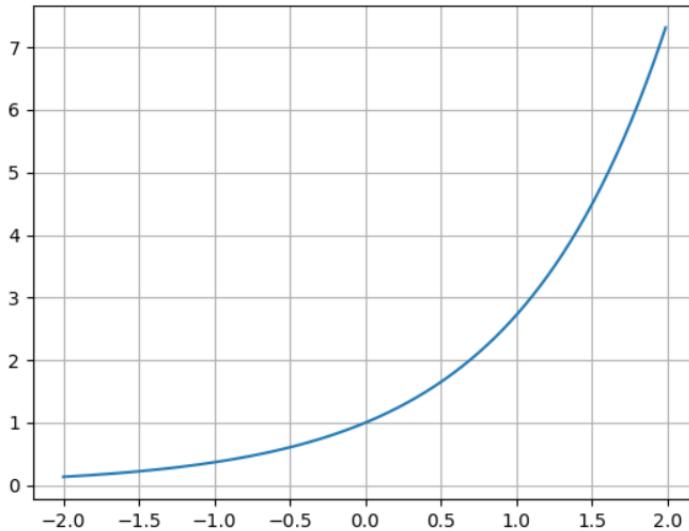


Figure 4.10: The exponential function  $\exp x$ .

Repeating the logic yielding (4.4.1), we have

$$\left(1 + \frac{x}{n}\right)^n = 1 + x + \sum_{k=2}^n \frac{x^k}{k!} \left(1 - \frac{1}{n}\right) \left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right). \quad (4.4.9)$$

Letting  $n \rightarrow \infty$  in (4.4.9) as before, results in the following.

### Exponential Series

The exponential function is always positive and satisfies, for every real number  $x$ ,

$$\exp x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \dots \quad (4.4.10)$$

The graph of  $\exp x$  is in Figure 4.10.



We use the binomial theorem one more time to show

### Law of Exponents

For real numbers  $x$  and  $y$ ,

$$\exp(x + y) = \exp x \cdot \exp y.$$

To see this, multiply out the sums

$$(a_0 + a_1 + a_2 + a_3 + \dots)(b_0 + b_1 + b_2 + b_3 + \dots)$$

in a “symmetric” manner, obtaining

$$a_0b_0 + (a_0b_1 + a_1b_0) + (a_0b_2 + a_1b_1 + a_2b_0) + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0) + \dots$$

Using summation notation, the  $n$ -th term in this last sum is

$$\sum_{k=0}^n a_k b_{n-k} = a_0 b_n + a_1 b_{n-1} + \dots + a_{n-1} b_1 + a_n b_0.$$

Thus

$$\left( \sum_{k=0}^{\infty} a_k \right) \left( \sum_{m=0}^{\infty} b_m \right) = \sum_{n=0}^{\infty} \left( \sum_{k=0}^n a_k b_{n-k} \right).$$

Now insert

$$a_k = \frac{x^k}{k!}, \quad b_{n-k} = \frac{y^{n-k}}{(n-k)!}.$$

Then the  $n$ -th term in the resulting sum equals, by the binomial theorem,

$$\sum_{k=0}^n a_k b_{n-k} = \sum_{k=0}^n \frac{x^k}{k!} \frac{y^{n-k}}{(n-k)!} = \frac{1}{n!} \sum_{k=0}^n \binom{n}{k} x^k y^{n-k} = \frac{1}{n!} (x+y)^n.$$

Thus

$$\exp x \cdot \exp y = \left( \sum_{k=0}^{\infty} \frac{x^k}{k!} \right) \left( \sum_{m=0}^{\infty} \frac{y^m}{m!} \right) = \sum_{n=0}^{\infty} \frac{(x+y)^n}{n!} = \exp(x+y).$$

This derives the law of exponents.

Repeating the law of exponents  $n$  times implies

$$\exp(nx) = \exp(x + x + \dots + x) = \exp x \cdot \exp x \cdot \dots \cdot \exp x = (\exp x)^n.$$

If we write  $\sqrt[n]{x} = x^{1/n}$ , replacing  $x$  by  $x/n$  yields

$$\exp(x/n) = (\exp x)^{1/n}.$$

Combining the last two equations yields

$$\exp(nx/m) = ((\exp x)^n)^{1/m} = (\exp x)^{n/m}.$$

Inserting  $x = 1$  in this last equation, it follows, for any rational number  $x = n/m$ ,

$$\exp x = \exp(1 \cdot x) = (\exp 1)^x = e^x.$$

Because of this, as a matter of convenience, we write the exponential function either way,  $\exp x$  or  $e^x$ , even when  $x$  is not rational.

### Exponential Notation

For any real number  $x$ ,

$$e^x = \exp x.$$



Suppose  $0 < r < 1$ . Then  $r^2 < r$ ,  $r^3 < r$ , and so on. Replacing  $x$  by  $rx$  in the exponential series (4.4.10),

$$\begin{aligned} e^{rx} &= 1 + rx + \frac{1}{2!}r^2x^2 + \frac{1}{3!}r^3x^3 + \dots \\ &< 1 + rx + \frac{1}{2!}rx^2 + \frac{1}{3!}rx^3 + \dots \\ &= 1 - r + re^x. \end{aligned} \tag{4.4.11}$$

From this we can show

### Convexity of the Exponential Function

For  $0 < r < 1$ ,

$$\exp((1 - r)x + ry) < (1 - r)\exp x + r\exp y. \tag{4.4.12}$$

To derive (4.4.12), replace  $x$  by  $y - x$  in (4.4.11), obtaining

$$e^{r(y-x)} < 1 - r + re^{y-x}.$$

Now multiply both sides by  $e^x$ , obtaining (4.4.12).

Graphically, the convexity of the exponential functions is the fact that the line segment joining two points on the graph lies above the graph (Figure 4.11).

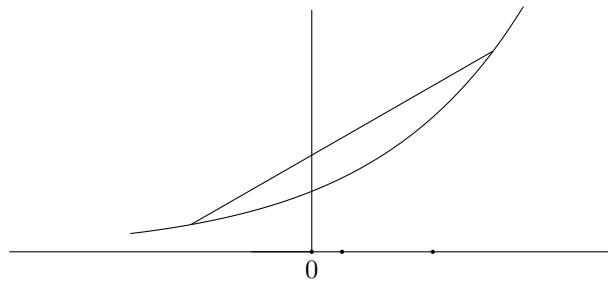


Figure 4.11: Convexity of the exponential function.

Convexity is discussed further in §7.5.

# Chapter 5

## Probability

### 5.1 Binomial Probability

Suppose a coin is tossed repeatedly, landing heads or tails each time. After tossing the coin 100 times, we obtain 53 heads. What can we say about this coin? Can we claim the coin is fair? Can we claim the probability of obtaining heads is .53?

Whatever claims we make about the coin, they should be *reliable*, in that they should more or less hold up to *repeated* verification.

To obtain reliable claims, we therefore repeat the above experiment 20 times, obtaining for example the following count of heads

$$[57, 49, 55, 44, 55, 50, 49, 50, 53, 49, 53, 50, 51, 53, 53, 54, 48, 51, 50, 53].$$

On the other hand, suppose someone else repeats the same experiment 20 times with a different coin, and obtains

$$[69, 70, 79, 74, 63, 70, 68, 71, 71, 73, 65, 63, 68, 71, 71, 64, 73, 70, 78, 67].$$

In this case, one suspects the two coins are statistically distinct, and have different probabilities of obtaining heads.

In this section, we study how the probabilities of coin-tossing behave, with the goal of answering the question: *Is a given coin fair?*



Assume we are tossing a coin. If we let  $p = \text{Prob}(H)$  and  $q = \text{Prob}(T)$  be the probabilities of obtaining heads and tails after a single toss, then

$$p + q = 1.$$

In particular, we see  $q = 1 - p$ , and  $p$  may be any real number between 0 and 1, depending on the particular coin being tossed.

If we toss the coin twice, we obtain one of four possibilities,  $HH$ ,  $HT$ ,  $TH$ , or  $TT$ . If we make the natural assumption that the coin has no memory, that the result of the first toss has no bearing on the result of the second toss, then the probabilities are

$$\text{Prob}(HH) = p^2, \text{Prob}(HT) = pq, \text{Prob}(TH) = qp, \text{Prob}(TT) = q^2. \quad (5.1.1)$$

These are valid probabilities since their sum equals 1,

$$p^2 + pq + qp + q^2 = (p + q)^2 = 1^2 = 1.$$

To see why these are the correct probabilities, we use the conditional probability definition,

$$\text{Prob}(A | B) = \frac{\text{Prob}(A \text{ and } B)}{\text{Prob}(B)}. \quad (5.1.2)$$

We use this formula to compute the probability that we obtain heads on the second toss given that we obtain tails on the first toss. The conditional probability definition (5.1.2) is equivalent to the *chain rule*

$$\text{Prob}(A \text{ and } B) = \text{Prob}(A | B) \text{Prob}(B).$$

To compute this, we introduce the convenient notation

$$X_n = \begin{cases} 1, & \text{if the } n\text{-th toss is heads,} \\ 0, & \text{if the } n\text{-th toss is tails.} \end{cases}$$

Then  $X_n$  is a random variable (§5.3) and represents a numerical reward function of the outcome (heads or tails) at the  $n$ -th toss.

With this notation, (5.1.1) may be rewritten

$$\begin{aligned} \text{Prob}(X_1 = 1 \text{ and } X_2 = 1) &= p^2, \\ \text{Prob}(X_1 = 1 \text{ and } X_2 = 0) &= pq, \\ \text{Prob}(X_1 = 0 \text{ and } X_2 = 1) &= qp, \\ \text{Prob}(X_1 = 0 \text{ and } X_2 = 0) &= q^2. \end{aligned}$$

In particular, this implies

$$\begin{aligned} \text{Prob}(X_1 = 1) &= \text{Prob}(X_1 = 1 \text{ and } X_2 = 0) + \text{Prob}(X_1 = 1 \text{ and } X_2 = 1) \\ &= pq + p^2 = \text{Prob}(p + q) = p. \end{aligned}$$

Similarly,  $\text{Prob}(X_2 = 1) = p$ . Computing,

$$\text{Prob}(X_2 = 1 \mid X_1 = 0) = \frac{\text{Prob}(X_1 = 0 \text{ and } X_2 = 1)}{\text{Prob}(X_1 = 0)} = \frac{qp}{q} = p = \text{Prob}(X_2 = 1),$$

so

$$\text{Prob}(X_2 = 1 \mid X_1 = 0) = \text{Prob}(X_2 = 1).$$

Thus  $X_1 = 0$  has no effect on the probability that  $X_2 = 1$ , and similarly for the other possibilities. This is often referred to as the *independence* of the coin tosses. We conclude

### Independent Coin-Tossing

With the conditional probability definition (5.1.2), a coin has no memory between successive tosses if and only if the probabilities at distinct tosses multiply,

$$\text{Prob}(X_1 = a_1, X_2 = a_2, \dots) = \text{Prob}(X_1 = a_1) \text{Prob}(X_2 = a_2) \dots \quad (5.1.3)$$

Here  $a_1, a_2, \dots$  are 0 or 1.

Since we are tossing the same coin, we can set

$$\text{Prob}(X_n = 1) = p, \quad \text{Prob}(X_n = 0) = q = 1 - p, \quad n \geq 1.$$

Thus all probabilities in (5.1.3) are determined by the parameter  $p$ , which may be any number between 0 and 1.



Suppose  $X$  is a random variable taking on three values  $a, b, c$  with probabilities  $p, q, r$ ,

$$P(X = a) = p, \quad P(X = b) = q, \quad P(X = c) = r.$$

Then the *mean* or *average* or *expected value* of  $X$  is

$$E(X) = ap + bq + cr.$$

Since  $p + q + r = 1$ , the expected value of  $X$  lies between the greatest of  $a$ ,  $b$ ,  $c$ , and the least,

$$\min(a, b, c) \leq E(X) \leq \max(a, b, c).$$

The *variance* of  $X$  is a measure of how far  $X$  deviates from its mean,

$$Var(X) = E((X - m)^2), \quad m = E(X).$$

For example,

$$Var(X) = (a - m)^2 \cdot p + (b - m)^2 \cdot q + (c - m)^2 \cdot r.$$

By expanding the squares, one has the identity

$$Var(X) = E(X^2) - m^2, \quad m = E(X).$$

A random variable  $Z$  is *standard* if its mean is zero and its variance is one. If  $X$  is any random variable with mean  $m$  and variance  $\sigma^2$ , the random variable

$$Z = \frac{X - m}{\sigma}$$

is standard.

For  $X = X_n$ , the mean is

$$E(X_n) = 1 \cdot p + 0 \cdot (1 - p) = p,$$

and the variance is

$$Var(X_n) = E(X_n^2) - m^2 = 1^2 \cdot p + 0^2 \cdot (1 - p) - p^2 = p - p^2 = p(1 - p).$$



Let

$$S_n = X_1 + X_2 + \cdots + X_n.$$

Since  $X_k = 1$  when the  $k$ -th toss is heads, and  $X_k = 0$  when the  $k$ -th toss is tails,  $S_n$  is the number of heads in  $n$  tosses.

The mean of  $S_n$  is

$$E(S_n) = E(X_1) + E(X_2) + \cdots + E(X_n) = p + p + \cdots + p = np.$$

The second moment of  $S_n$ , or the mean of  $S_n^2$ , is

$$E(S_n^2) = E\left(\left(\sum_{k=1}^n X_k\right)^2\right) = \sum_{k=1}^n E(X_k^2) + \sum_{k \neq j} E(X_k X_j).$$

By independence and  $X_k^2 = X_k$ ,

$$E(S_n^2) = \sum_{k=1}^n E(X_k) + \sum_{k \neq j} E(X_k)E(X_j) = np + n(n-1)p^2 = np(1-p) + n^2p^2.$$

Hence the variance of  $S_n$  is

$$\text{Var}(S_n) = E(S_n^2) - (np)^2 = np(1-p) + n^2p^2 - n^2p^2 = np(1-p).$$

It is natural to ask for the probability of obtaining  $k$  heads in  $n$  tosses,  $\text{Prob}(S_n = k)$ . Here  $k$  varies between 0 and  $n$ , corresponding to all tails or all heads respectively.

There are  $n+1$  possibilities  $S_n = 0, S_n = 1, S_n = 2, \dots, S_n = n$  for the number of heads in  $n$  tosses. If we have no idea what the parameter  $p$  is, then all possibilities are equally likely, so one expects

$$\text{Prob}(S_n = k) = \frac{1}{n+1}, \quad 0 \leq k \leq n. \quad (5.1.4)$$

Notice

$$\sum_{k=0}^n \text{Prob}(S_n = k) = \sum_{k=0}^n \frac{1}{n+1} = 1,$$

as it should be.

Now suppose we are given  $p$ , so we know  $p = \text{Prob}(X_n = 1)$ . Since the number of ways of choosing  $k$  heads from  $n$  tosses is  $\binom{n}{k}$ , and the probabilities of distinct tosses multiply, the probability of  $k$  heads in  $n$  tosses is as follows.

### Binomial Distribution With Parameters $n, p$

If a coin has heads-probability  $p$ , the probability of obtaining  $k$  heads in  $n$  tosses is

$$\text{Prob}(S_n = k) = \binom{n}{k} p^k (1-p)^{n-k}. \quad (5.1.5)$$

Moreover the mean and variance of the binomial distribution is  $np$  and  $np(1-p)$ .

By the binomial theorem,

$$\sum_{k=0}^n \text{Prob}(S_n = k) = \sum_{k=0}^n \binom{n}{k} p^k (1-p)^{n-k} = (p+1-p)^n = 1,$$

again as it should be.

The binomial distribution with  $n = 1$  corresponds to a single coin toss, and is called the *bernoulli distribution*. The corresponding random variable  $X$ ,

$$\text{Prob}(X = 1) = p, \quad \text{Prob}(X = 0) = 1 - p,$$

is a *bernoulli* or *bernoulli* random variable. The values of a bernoulli random variable need not be 0, 1, they may be any two values  $a$  and  $b$ .



Now we assume the coin parameter  $p$  is unknown, and we interpret (5.1.5) as the conditional probability that  $S_n = k$  given knowledge of  $p$ , which we rewrite as

$$\text{Prob}(S_n = k \mid p = r) = \binom{n}{k} r^k (1-r)^{n-k}, \quad 0 \leq k \leq n. \quad (5.1.6)$$

Now  $\text{Prob}(S_n = k)$  is the sum of the probabilities  $\text{Prob}(S_n = k \text{ and } p = r)$  over  $0 \leq r \leq 1$ . By the definition of conditional probability (5.1.2),

$$\text{Prob}(S_n = k \text{ and } p = r) = \text{Prob}(S_n = k \mid p = r) \text{Prob}(p = r).$$

Thus  $\text{Prob}(S_n = k)$  is the sum of  $\text{Prob}(S_n = k \mid p = r) \text{Prob}(p = r)$  over  $0 \leq r \leq 1$ . Since  $p$  varies continuously over  $0 \leq r \leq 1$ , the sum is replaced by the integral, so

$$\text{Prob}(S_n = k) = \int_0^1 \text{Prob}(S_n = k \mid p = r) \text{Prob}(r < p < r + dr).$$

Since we don't know anything about  $p$ , it's simplest to assume a *uniform* a priori probability

$$\text{Prob}(a < p < b) = b - a, \quad 0 \leq a < b \leq 1,$$

which is the same as saying  $\text{Prob}(r < p < r + dr) = dr$ . By (5.1.6), we obtain

$$\text{Prob}(S_n = k) = \int_0^1 \binom{n}{k} r^k (1-r)^{n-k} dr.$$

But by integration by parts,

$$\int_0^1 r^k (1-r)^{n-k} dr = \frac{k!(n-k)!}{(n+1)!}.$$

From this, we conclude

$$\text{Prob}(S_n = k) = \int_0^1 \binom{n}{k} r^k (1-r)^{n-k} dr = \binom{n}{k} \frac{k!(n-k)!}{(n+1)!} = \frac{1}{n+1}, \quad (5.1.7)$$

agreeing with our intuitive result earlier.

Notice the difference: In (5.1.5), we know the coin's heads probability  $p$ , and obtain the binomial distribution, while in (5.1.7), since we don't know  $p$ , and there are  $n+1$  possibilities  $0 \leq k \leq n$ , we obtain the uniform distribution  $1/(n+1)$ .



We now turn things around: Suppose we toss the coin  $n$  times, and obtain  $k$  heads. How can we use this data to estimate the coin's probability of heads  $p$ ?

To this end, we introduce the fundamental

### Bayes Theorem

$$\text{Prob}(A | B) = \frac{\text{Prob}(B | A) \cdot \text{Prob}(A)}{\text{Prob}(B)}. \quad (5.1.8)$$

The proof of Bayes Theorem is straightforward:

$$\begin{aligned} \text{Prob}(A | B) &= \frac{\text{Prob}(A \text{ and } B)}{\text{Prob}(B)} \\ &= \frac{\text{Prob}(A \text{ and } B)}{\text{Prob}(A)} \cdot \frac{\text{Prob}(A)}{\text{Prob}(B)} \\ &= \text{Prob}(B | A) \cdot \frac{\text{Prob}(A)}{\text{Prob}(B)}. \end{aligned}$$

The depth of the result lies in its widespread usefulness.

We now write Bayes Theorem to compute

$$\text{Prob}(p = r | S_n = k) = \text{Prob}(S_n = k | p = r) \cdot \frac{\text{Prob}(p = r)}{\text{Prob}(S_n = k)}. \quad (5.1.9)$$

But  $\text{Prob}(S_n = k | p = r)$  is as in (5.1.6),  $\text{Prob}(S_n = k)$  is as in (5.1.7), and  $\text{Prob}(p = r) = 1$ . Inserting these quantities into (5.1.9) leads to

### A Posteriori probability Given $k$ heads in $n$ tosses

Assume the unknown heads probability  $p$  of a coin is uniformly distributed on  $0 \leq r \leq 1$ . Then the probability that  $p = r$  given  $k$  heads in  $n$  tosses equals

$$\text{Prob}(p = r | S_n = k) = (n + 1) \cdot \binom{n}{k} r^k (1 - r)^{n-k}. \quad (5.1.10)$$

Notice because of the extra factor  $(n + 1)$ , this is not equal to (5.1.6). In (5.1.6),  $p$  is fixed, and  $k$  is the variable. In (5.1.10),  $k$  is fixed, and  $r$  is the variable. This a posteriori distribution for  $(n, k) = (10, 7)$  is plotted in Figure 5.1. Notice this distribution is concentrated about  $k/n = 7/10 = .7$ .

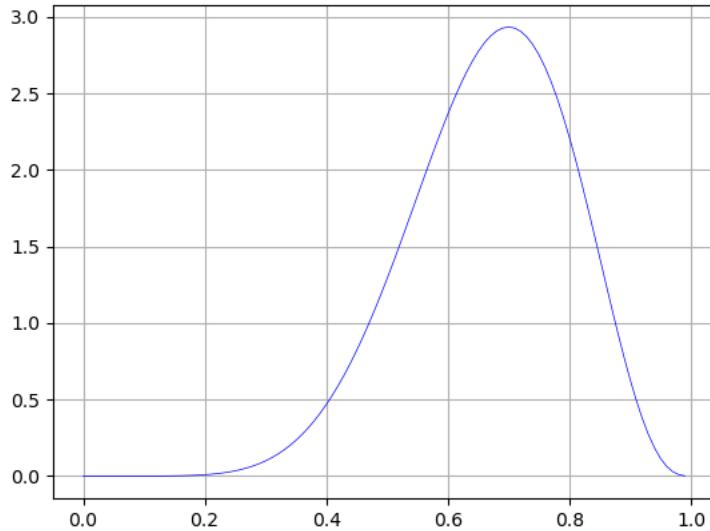


Figure 5.1: The distribution of  $p$  given 7 heads in 10 tosses.

The code generating this figure is

```
from matplotlib.pyplot import *
from numpy import arange

def f(x): return 1320 * x^7*(1-x)^3

grid()
X = arange(0,1,.01)
plot(X,f(X),color="blue",linewidth=.5)
show()
```



Because Bayes Theorem is so useful, here are two alternate forms. First, since

$$\begin{aligned} \text{Prob}(B) &= \text{Prob}(B \text{ and } A) + \text{Prob}(B \text{ and } A^c) \\ &= \text{Prob}(B | A) \text{Prob}(A) + \text{Prob}(B | A^c) \text{Prob}(A^c), \end{aligned}$$

Bayes rule also states

$$\text{Prob}(A | B) = \frac{\text{Prob}(B | A) \text{Prob}(A)}{\text{Prob}(B | A) \text{Prob}(A) + \text{Prob}(B | A^c) \text{Prob}(A^c)}. \quad (5.1.11)$$

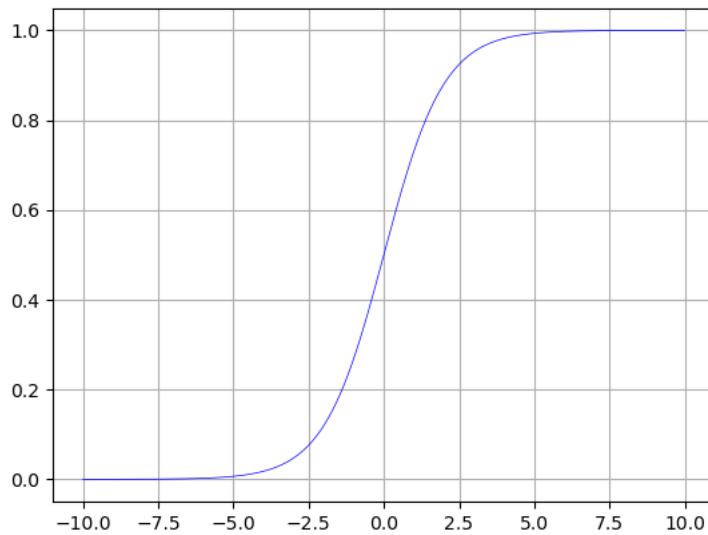


Figure 5.2: The logistic function.

Let

$$p = \sigma(z) = \frac{1}{1 + e^{-z}}. \quad (5.1.12)$$

This is the *logistic function* or *sigmoid function* (Figure 5.2). The logistic function takes as inputs real numbers  $y$ , and returns as outputs probabilities  $p$  (Figure 5.3). Think of the input  $z$  as an activation energy, and the output  $p$  as the probability of activation. In Python,  $\sigma$  is the `expit` function.

```
from scipy.special import expit
p = expit(z)
```

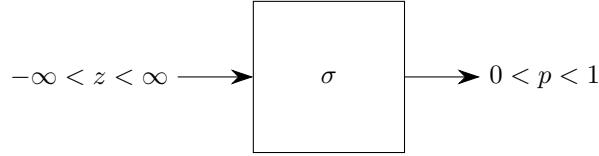


Figure 5.3: The logistic function takes real numbers to probabilities.

Dividing the numerator and denominator of (5.1.11) by the numerator, we also obtain in terms of log-probabilities,

$$\text{Prob}(A | B) = \sigma \left( \log \left( \frac{\text{Prob}(B | A) \text{Prob}(A)}{\text{Prob}(B | A^c) \text{Prob}(A^c)} \right) \right). \quad (5.1.13)$$

For example, suppose we have two groups of points in  $\mathbf{R}^d$ , selected as follows. A fair coin is tossed. If the result is heads, select a point  $x$  in  $\mathbf{R}^d$  at random with normal probability (§5.4) with mean  $m_H$ , or

$$\text{Prob}(x | H) \sim e^{-|x - m_H|^2/2}.$$

If the result is tails, select a point  $x$  at random with normal probability with mean  $m_T$ , or

$$\text{Prob}(x | T) \sim e^{-|x - m_T|^2/2}.$$

This says the the groups are centered around the points  $m_H$  and  $m_T$  respectively.

Given a point  $x$ , what is the probability  $x$  is in the heads group? In other words, what is

$$\text{Prob}(H | x)?$$

This question is begging for Bayes theorem.

Let

$$w = m_H - m_T, \quad w_0 = -\frac{1}{2}|m_H|^2 + \frac{1}{2}|m_T|^2.$$

Since  $\text{Prob}(H) = \text{Prob}(T)$ , here we have  $\text{Prob}(A) = \text{Prob}(A^c)$ . Inserting the probabilities and simplifying leads to

$$\log \left( \frac{\text{Prob}(x | H) \text{Prob}(H)}{\text{Prob}(x | T) \text{Prob}(T)} \right) = w \cdot x + w_0. \quad (5.1.14)$$

By (5.1.13), this leads to

$$\text{Prob}(H | x) = \sigma(w \cdot x + w_0).$$

Thus the hyperplane

$$z = w \cdot x + w_0$$

is the cut-off between the two groups. Written this way, the probability is a single-layer perceptron (§8.2).

## 5.2 Probability

A *probability* is often described as

*the extent to which an event is likely to occur, measured by the ratio of the favorable outcomes to the whole number of outcomes possible.*

We explain what this means by describing the basic terminology:

- An *experiment* is a procedure that yields an outcome, out of a set of possible outcomes. For example, tossing a coin is an experiment that yields one of two outcomes, heads or tails, which we also write as 1 or 0. Rolling a six-sided die yields outcomes 1, 2, 3, 4, 5, 6. Rolling two six-sided dice yields 36 outcomes  $(1, 1), (1, 2), \dots$ . Flipping a coin three times yields  $2^3 = 8$  outcomes

$$TTT, TTH, THT, THH, HTT, HTH, HHT, HHH,$$

or

$$000, 001, 010, 011, 100, 101, 110, 111.$$

- The *sample space* is the set  $S$  of all possible outcomes. If  $\#(S)$  is the number of outcomes in  $S$ , then for the four experiments above, we have  $\#(S)$  equals 2, 6, 36, and 8. The sample space  $S$  is also called the *population*.
- An *event* is a specific subset  $E$  of  $S$ . For example, when rolling two dice,  $E$  can be the outcomes where the sum of the dice equals 7. In this case, the outcomes in  $E$  are

$$(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1),$$

so here  $\#(S) = 36$  and  $\#(E) = 6$ . Another example is obtaining three heads when tossing a coin seven times. Here  $\#(S) = 2^7 = 128$  and

$\#(E) = 35$ , which is the number of ways you can choose three things out of seven things:

$$\#(E) = \text{7-choose-3} = \binom{7}{3} = \frac{7 \cdot 6 \cdot 5}{1 \cdot 2 \cdot 3} = 35.$$

- The *probability* of an outcome  $s$  is a number  $Prob(s)$  with the properties
  1.  $0 \leq Prob(s) \leq 1$ ,
  2. The sum of the probabilities of all outcomes equals one.
- The *probability*  $Prob(E)$  of an event  $E$  is the sum of the probabilities of the outcomes in  $E$ .
- Outcomes are *equally likely* when they have the same probability. When this is so, we must have

$$Prob(E) = \frac{\#(E)}{\#(S)}.$$

For example,

1. A coin is *fair* if the outcomes are equally likely. For one toss of a fair coin,  $Prob(\text{heads}) = 1/2$ .
2. More generally, tossing a coin results in outcomes

$$Prob(\text{head}) = p, \quad Prob(\text{tail}) = 1 - p,$$

with  $0 < p < 1$ .

3. A die is *fair* if the outcomes are equally likely. Roll a fair die and let  $E$  be the event that the outcome is less than 3. Then  $Prob(E) = 2/6 = 1/3$ .
4. The probability of obtaining a sum of 7 when rolling two fair dice is  $Prob(E) = 6/36 = 1/6$ .
5. The probability of obtaining three heads when tossing a fair coin seven times is  $Prob(E) = 35/128$ .
6. The probability of selecting an iris with petal length between 1 and 3 from the Iris dataset.



Now suppose we conduct an experiment by tossing a coin (always assumed fair unless otherwise mentioned) 10 times. Because the coin is fair, we expect to obtain heads around 5 times. Will we obtain heads exactly 5 times? Let's run the experiment with Python. In fact, we will run the experiment 20 times. If we count the number of heads after each run of the experiment, we obtain a digit between 0 and 10 inclusive.

To simulate this, we use `binomial(n,p,N)`. When  $N = 1$ , this returns the number of heads obtained after a single experiment, consisting of tossing a coin  $n$  times, where the probability of obtaining heads in each toss is  $p$ .

More generally, `binomial(n,p,N)` runs this experiment  $N$  times, returning a vector  $v$  with  $N$  components. For example, the code

```
from numpy.random import *
p = .5
n = 10
N = 20

v = binomial(n,p,N)
print(v)
```

returns

```
[9 6 7 4 4 4 3 3 7 5 6 4 6 9 4 5 4 7 6 7]
```

The sample space  $S$  corresponding to  $(p, n, N)$  consists of all vectors  $v = (v_1, v_2, \dots, v_N)$  with  $N$  components, with each component equal to 0, 1, ...,  $n$ . So here  $\#(S) = (n + 1)^N$ .

Now we conduct three experiments: tossing a coin 5 times, then 50 times, then 500 times. The code

```
p = .5
for n in [5,50,500]: print(binomial(n,p,1))
```

This returns the count of heads after 5 tosses, 50 tosses, and 500 tosses,

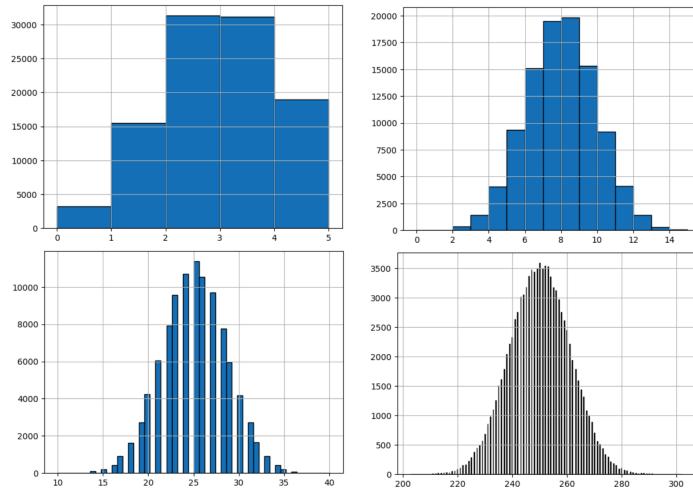


Figure 5.4: 100,000 sessions, with 5, 15, 50, and 500 tosses per session.

3, 28, 266

The *proportions* are the count divided by the total number of tosses in the experiment. For the above three experiments, the proportions after 5 tosses, 50 tosses, and 500 tosses, are

$$3/5=.600, \quad 28/50=.560, \quad 266/500=.532$$

Now we repeat each experiment 100,000 times and we plot the results in a histogram.

```
from matplotlib.pyplot import *
from numpy.random import *

N = 100000
p = .5

for n in [5,50,500]:
    data = binomial(n,p,N)
    hist(data,bins=n,edgecolor ='Black')
    grid()
```

```
show()
```

This results in Figure 5.4.



The takeaway from these graphs are the two fundamental results of probability:

1. **Law of Large Numbers.** The proportion in a repeated experiment is the *sample proportion*. The sample proportion tends to be near the underlying probability  $p$ . The underlying probability is the *population proportion*. The larger the sample size in the experiment, the closer the proportion is to  $p$ . Another way of saying this is: For large sample size, the sample mean is approximately equal to the population mean.
2. **Central Limit Theorem.** For large sample size, the shape of the graph of the proportions or counts is approximately normal. The normal distribution is studied in §5.4. Another way of saying this is: For large sample size, the shape of the sample mean histogram is approximately normal.

The law of large numbers is *qualitative* and the central limit theorem is *quantitative*. While the law of large numbers says one thing is close to another, it does not say how close. The central limit theorem provides a numerical measure of closeness, using the normal distribution.



Roll two six-sided dice. Let  $A$  be the event that at least one dice is an even number, and let  $B$  be the event that the sum is 6. Then

$$A = \{(2,*), (4,*), (6,*), (*,2), (*,4), (*,6)\}.$$

$$B = \{(1,5), (2,4), (3,3), (4,2), (5,1)\}.$$

The *intersection* of  $A$  and  $B$  is the event of outcomes in both events:

$$A \text{ and } B = \{(2,4), (4,2)\}.$$

The *union* of  $A$  and  $B$  is the event of outcomes in either event:

$$A \text{ or } B = \{(2, *), (4, *), (6, *), (*, 2), (*, 4), (*, 6), (1, 5), (3, 3), (5, 1)\}.$$

The *complement* of  $A$  is the events of outcomes not in  $A$ . So

$$\text{not } A = \{(1, 1), (1, 3), (1, 5), (3, 1), (3, 3), (3, 5), (5, 1), (5, 3), (5, 5)\}.$$

Since  $\#\text{(not } A) = 9$ , and  $\#(S) = 36$ ,

$$\#(A) = \#(S) - \#\text{(not } A) = 36 - 9 = 27.$$

Clearly  $\#(B) = 5$ .

The *difference* of  $A$  minus  $B$  is the event of outcomes in  $A$  but not in  $B$ :

$$A - B = A \text{ and not } B$$

$$= \{(2, * \text{ except } 4), (4, * \text{ except } 2), (6, *), (* \text{ except } 4, 2), (* \text{ except } 2, 4), (*, 6)\}.$$

Similarly,

$$B - A = \{(1, 5), (3, 3), (5, 1)\}.$$

Then  $A - B$  is part of  $A$  and  $B - A$  is part of  $B$ ,  $A \cap B$  is part of both, and all are part of  $A \cup B$ .

Hence

$$\text{Prob}(A) = \frac{27}{36} = \frac{3}{4}, \quad \text{Prob}(B) = \frac{5}{36}.$$

Events  $A$  and  $B$  are *independent* if

$$\text{Prob}(A \text{ and } B) = \text{Prob}(A) \times \text{Prob}(B).$$

The *conditional probability* of  $A$  given  $B$  is

$$\text{Prob}(A | B) = \frac{\text{Prob}(A \text{ and } B)}{\text{Prob}(B)}.$$

When  $A$  and  $B$  are independent,

$$\text{Prob}(A | B) = \frac{\text{Prob}(A \text{ and } B)}{\text{Prob}(B)} = \frac{\text{Prob}(A) \times \text{Prob}(B)}{\text{Prob}(B)} = \text{Prob}(A),$$

so the conditional probability equals the unconditional probability.

Are  $A$  and  $B$  above independent? Since

$$\text{Prob}(A | B) = \frac{\text{Prob}(A \cap B)}{\text{Prob}(B)} = \frac{2/36}{5/36} = \frac{2}{5}$$

which is not equal to  $\text{Prob}(A)$ , they are not independent.

### 5.3 Random Variables

Suppose a real number  $x$  is selected at random. Even if we don't know anything about  $x$ , we know  $x$  is a number, so our *confidence* that  $-\infty < x < \infty$  equals 100%, the *chance* that  $x$  satisfies  $-\infty < x < \infty$  equals 1, the *probability* that  $x$  satisfies  $-\infty < x < \infty$  equals 1.

When we say  $x$  is “selected at random”, we think of a machine  $X$  that is the source of the numbers  $x$  (Figure 5.5). Such a source of numbers is best called a *random number*. Unfortunately,<sup>1</sup> the standard 100-year-old terminology for such an  $X$  is *random variable*, and this is what we'll use.

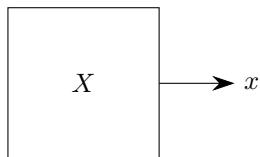


Figure 5.5: When we sample  $X$ , we get  $x$ .

For example, suppose we want to estimate the proportion of American college students who have a smart phone. Instead of asking every student, we take a sample and make an estimate based on the sample.

Let  $p$  be the actual proportion of students that in fact have a smartphone. If there are  $N$  students in total, and  $m$  of them have a smartphone, then  $p = m/N$ . For each student, let

$$X = \begin{cases} 1, & \text{if the student has a smartphone,} \\ 0, & \text{if not.} \end{cases}$$

Then  $X$  is a random variable:  $X$  is a machine that returns 0 or 1 depending on the chosen student.

A random variable taking on only two values is a *bernoulli* random variable. Since  $X$  takes on the two values 0 and 1,  $X$  is a bernoulli random variable.

Throughout we adopt the convention that random variables are written uppercase  $X$ , while the numbers they produce when sampled are written lowercase  $x$ . In other words, *when we sample  $X$ , we get  $x$* .

---

<sup>1</sup>The standard terminology is inaccurate, because the variability of the samples  $x$  is implied by the term *random*; the term *variable* is superfluous and may suggest something like double-randomness (whatever that is), which is not the case.

We will have occasion to meet many different random variables  $X$ ,  $Y$ ,  $Z$ , .... The letter  $Z$  is reserved for a standard random variable, one having mean zero and variance one. Samples from  $Z$  are written as  $z$ .



What is the chance, what is our confidence, what is the probability, of selecting  $x$  from an interval  $[a, b]$ ? If we write

$$\text{Prob}(a < X < b)$$

for the chance that  $X$  lies in the interval  $[a, b]$ , we are asking for  $\text{Prob}(a < X < b)$ . If we don't know anything about  $X$ , then we can't figure out the probability, and there is nothing we can say. Knowing something about  $X$  means knowing the *distribution* of  $X$ : Where  $X$  is more likely to be and where  $X$  is less likely to be. In effect, a random variable is a quantity  $X$  whose probabilities  $\text{Prob}(a < X < b)$  can be computed.

For example, take the Iris dataset and let  $X$  be the petal length of an iris (Figure 5.6) selected at random. Here the number of samples is  $N = 150$ .

```
from pandas import *

df = read_csv("iris.csv")
petal_length = df["Petal_length"].to_numpy()
```

```
3.7586666666666666
array([1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.5, 1.6, 1.1, 1.2, 1.5, 1.3, 1.4, 1.7, 1.5, 1.7, 1.5, 1.5, 1., 1.7, 1.7, 1.9, 1.6, 1.6, 1.5, 1.4, 1.6, 1.6, 1.5, 1.5, 1.4, 1.4, 1.5, 1.2, 1.3, 1.5, 1.3, 1.5, 1.3, 1.3, 1.3, 1.6, 1.9, 1.4, 1.6, 1.4, 1.5, 1.4, 4.7, 4.5, 4.9, 4., 4.6, 4.5, 4.7, 3.3, 4.6, 3.9, 3.5, 4.2, 4., 4.7, 3.6, 4.4, 4.5, 4.1, 4.5, 3.9, 4.8, 4., 4.9, 4.7, 4.3, 4.4, 4.4, 4.8, 5., 4.5, 3.5, 3.8, 3.7, 3.9, 5.1, 4.5, 4.5, 4.7, 4.4, 4.1, 4., 4.4, 4.6, 4., 3.3, 4.2, 4.2, 4.2, 4.3, 3., 4.1, 6., 5.1, 5.9, 5.6, 5.8, 6.6, 4.5, 6.3, 5.8, 6.1, 5.1, 5.3, 5.5, 5., 5.1, 5.3, 5.5, 6.7, 6.9, 5., 5.7, 4.9, 6.7, 4.9, 5.7, 6., 4.8, 4.9, 5.6, 5.8, 6.1, 6.4, 5.6, 5.1, 5.6, 6.1, 5.6, 5.5, 4.8, 5.4, 5.6, 5.1, 5.1, 5.9, 5.7, 5.2, 5., 5.2, 5.4, 5.1])
```

Figure 5.6:  $N = 150$  petal lengths and their mean.

The mean is the usual formula

$$m = E(X) = \frac{1}{N} \sum_{k=1}^N x_k.$$

Similarly, the second moment is

$$E(X^2) = \frac{1}{N} \sum_{k=1}^N x_k^2.$$

In general, given any function  $f(x)$ , we have the mean of  $f(x_1), f(x_2), \dots, f(x_N)$ ,

$$E(f(X)) = \frac{1}{N} \sum_{k=1}^N f(x_k). \quad (5.3.1)$$

If we let

$$f(x) = \begin{cases} 1, & 1 < x < 3, \\ 0, & \text{otherwise,} \end{cases}$$

then  $f(x_k)$  only counts when  $1 < x_k < 3$ , so

$$E(f(X)) = \frac{1}{N} \sum_{k=1}^N f(x_k) = \frac{\#\{\text{samples satisfying } 1 < x_k < 3\}}{N}.$$

But this is the probability that a randomly selected iris has petal length  $X$  between 1 and 3,

$$\text{Prob}(1 < X < 3) = E(f(X)),$$

when  $f(x)$  is as above.

This shows probabilities are special cases of means. Since we can compute means by (5.3.1), we can compute probabilities for  $X$ . By definition, this  $X$  is a random variable.



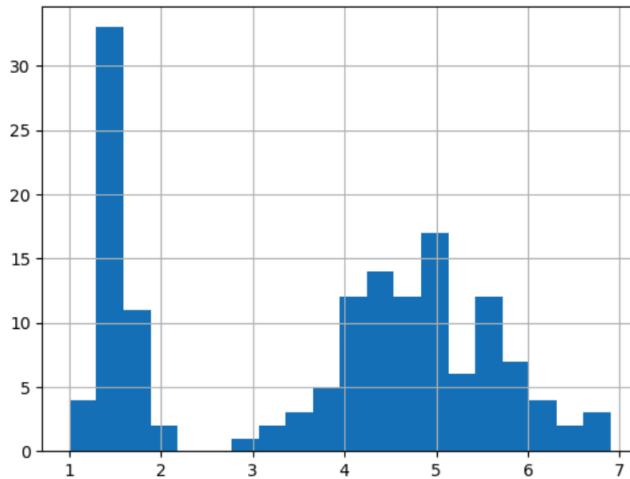


Figure 5.7: Histogram of  $N = 150$  petal lengths.

To see how the iris petal lengths are distributed, we plot a histogram,

```
from matplotlib.pyplot import *
grid()
hist(petal_length,bins=20)
show()
```

This results in Figure 5.7.

More generally, we take a random *batch* of samples of size  $n$  and take the mean of the samples in the batch. For example, the following code grabs a batch of  $n = 5$  petals lengths  $x_1, x_2, x_3, x_4, x_5$  at random and takes their mean,

$$\frac{x_1 + x_2 + x_3 + x_4 + x_5}{5}.$$

```
from numpy.random import *
rng = default_rng()
# n = batch_size
```

```

def random_batch_mean(n):
    rng.shuffle(petal_length)
    return mean(petal_length[:n])

random_batch_mean(5)

```

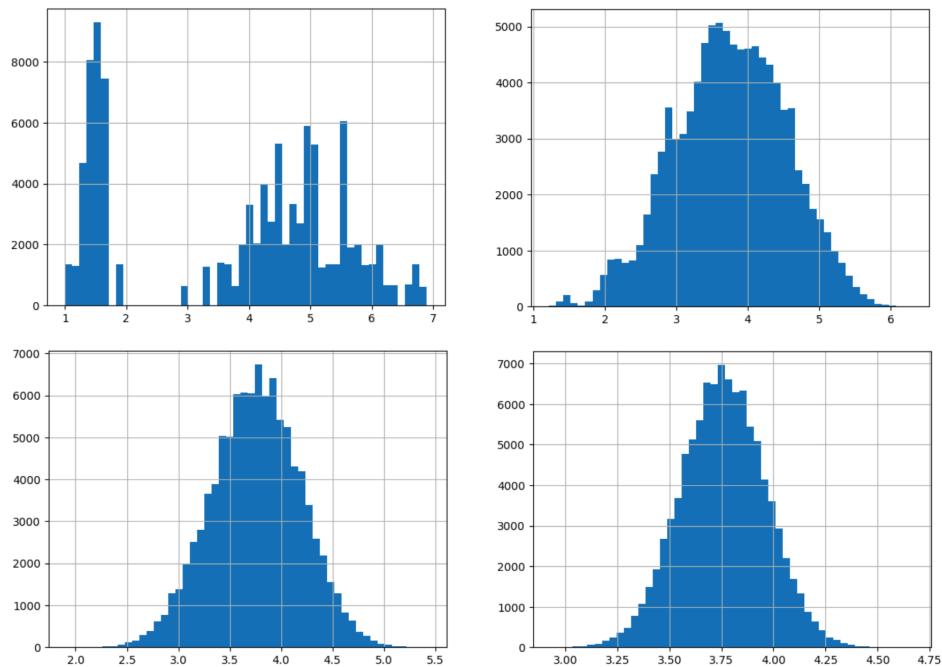


Figure 5.8: Means of 100,000 batches, of size  $n = 1, 5, 15, 50$ .

The five petal lengths are selected by first shuffling the petal lengths, then selecting the first five `petal_length[:5]`. Now repeat this computation 100,000 times, for batch sizes 1, 5, 15, 50. The resulting histograms are in Figure 5.8. Notice in the first subplot, the batch is size  $n = 1$ , so we recover the base histogram Figure 5.7. Figure 5.8 is of course another illustration of the central limit theorem.

`N = 100000`

```

for n in [1,5,15,50]:
    Xbar = [random_batch_mean(n) for _ in range(N)]
    hist(Xbar,bins=50)
    grid()
    show()

```



The simplest random variable is the bernoulli random variable  $X$  resulting from a coin toss, with  $X = 1$  corresponding to heads, and  $X = 0$  corresponding to tails. In this case,

$$\text{Prob}(X = 1) = p, \quad \text{Prob}(X = 0) = 1 - p,$$

which can be written

$$\text{Prob}(X = x) = p^x(1 - p)^{1-x}, \quad x = 0, 1.$$

This distribution is presented graphically in Figure 5.9.

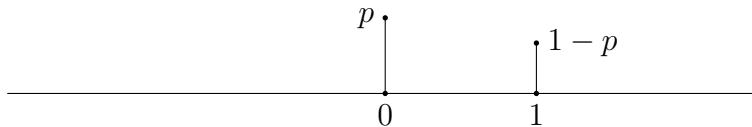


Figure 5.9: Distribution of a bernoulli random variable.

For example, if we let  $X$  be the number of heads obtained in  $n$  tosses of a coin with heads probability  $p$ , and sample  $X$  100,000 times, the resulting frequency histogram is in 5.4. Knowing something about  $X$  means knowing where  $X$  is more likely to be and where  $X$  is less likely to be. But this is exactly the information provided by the histogram. Therefore *the histogram is the approximate distribution of  $X$* . There is one caveat: Because we prefer our probabilities to sum to 1, with 1 corresponding to certainty, this last conclusion is correct only after rescaling the histogram to have total area 1.

Because of the central limit theorem, the normal distribution given on the left in Figure 5.10 plays a central role. This distribution is studied in §5.4. We say  $X$  follows a specific distribution, such as the curves in Figure

[5.10](#), when the probability  $\text{Prob}(a < X < b)$  is given by the green area in Figure [5.10](#). Thus

$$\text{chance} = \text{confidence} = \text{probability} = \text{area}.$$

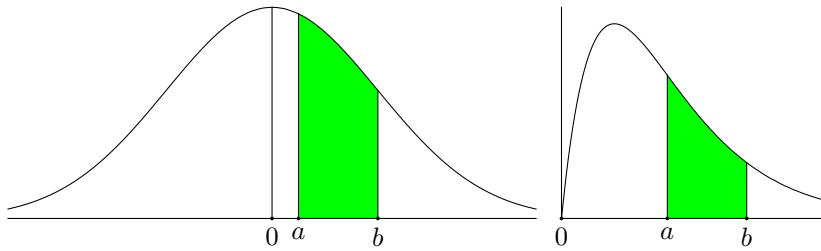


Figure 5.10: Confidence that  $X$  lies in interval  $[a, b]$ .

Given any distribution as in Figure [5.10](#), the *cumulative distribution function* at a point  $x$  is the area under the graph to the left of  $x$ ,

$$\text{cdf}_X(x) = \text{Prob}(X \leq x).$$

Then the green areas in Figure [5.10](#) is the difference between two areas, hence equal

$$\text{cdf}_X(b) - \text{cdf}_X(a).$$

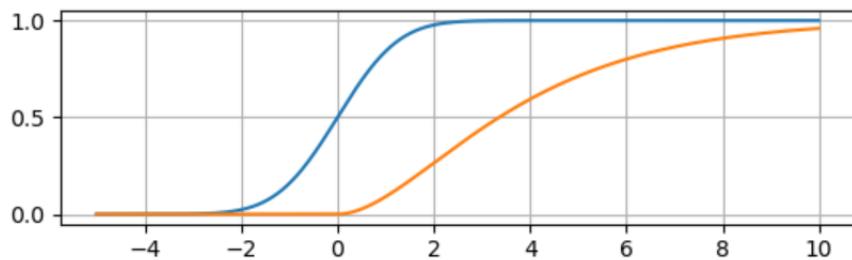


Figure 5.11: Cumulative distribution functions.

For the distributions in Figure [5.10](#), the cumulative distribution functions are in Figure [5.11](#).

For the bernoulli distribution in Figure 5.9, the cdf is in Figure 5.12. Because the bernoulli random variable takes on only the values  $x = 0, 1$ , these are the values where the cdf  $\text{Prob}(X \leq x)$  jumps.

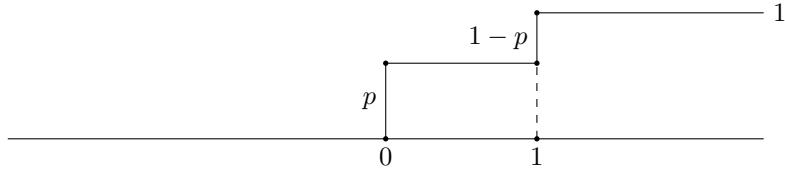


Figure 5.12: Cdf of a bernoulli distribution.

Let  $X$  be a random variable taking on values  $x_1, x_2, x_3, \dots$ , with probabilities  $p_1, p_2, p_3, \dots$  then the *mean* is

$$\mu = E(X) = p_1x_1 + p_2x_2 + p_3x_3 + \dots = \sum_{k=1}^N p_kx_k.$$

This is the *population mean*. It does not depend on a sampling of the population.

For example, suppose the population consists of 100 balls, of which 30 are red, 20 are green, and 50 are blue. The cost of each ball is

$$X(\text{ball}) = \begin{cases} \$1, & \text{red}, \\ \$2, & \text{green}, \\ \$3, & \text{blue}. \end{cases}$$

Then

$$\begin{aligned} p_{\text{red}} &= \text{Prob(red)} = \frac{\#\text{(red)}}{\#\text{(balls)}} = \frac{30}{100} = .3, \\ p_{\text{green}} &= \text{Prob(green)} = \frac{\#\text{(green)}}{\#\text{(balls)}} = \frac{20}{100} = .2, \\ p_{\text{blue}} &= \text{Prob(blue)} = \frac{\#\text{(blue)}}{\#\text{(balls)}} = \frac{50}{100} = .5. \end{aligned}$$

Then the average cost of a ball equals

$$\begin{aligned} E(X) &= p_{\text{red}} \cdot 1 + p_{\text{green}} \cdot 2 + p_{\text{blue}} \cdot 3 \\ &= \frac{30 \cdot 1 + 20 \cdot 2 + 50 \cdot 3}{100} = \frac{x_1 + x_2 + \dots + x_{100}}{100}. \end{aligned}$$

The *variance* is

$$\begin{aligned} \text{Var}(X) &= E((X - \mu)^2) = p_1(x_1 - \mu)^2 + p_2(x_2 - \mu)^2 + p_3(x_3 - \mu)^2 + \dots \\ &= \sum_{k=1}^N p_k(x_k - \mu)^2. \end{aligned}$$

This is the *variance* or *population variance*. A direct consequence of the formula is  $\text{Var}(X) = 0$  exactly when  $X$  is a non-random constant. The square root of the variance is the *standard deviation*. We also write  $\text{Var}(X) = \sigma^2$ , so  $\sigma$  is the standard deviation.

Going back to the smartphone random variable  $X$ , because  $p$  is the proportion of students having smartphones,  $\text{Prob}(X = 1) = p$ . This implies  $\text{Prob}(X = 0) = 1 - p$ . Since  $X$  takes on two values  $x_1 = 1$  and  $x_2 = 0$  with probabilities  $p_1 = p$  and  $p_2 = 1 - p$ , the mean is

$$\mu = E(X) = x_1 p_1 + x_2 p_2 = 1 \cdot p + 0 \cdot (1 - p) = p,$$

and the variance is

$$\sigma^2 = (x_1 - \mu)^2 p_1 + (x_2 - \mu)^2 p_2 = (1 - p)^2 \cdot p + (0 - p)^2 \cdot (1 - p) = p(1 - p).$$

Thus the standard deviation is  $\sqrt{p(1 - p)}$ .

More generally, if a random variable  $X$  takes on two values  $a$  and  $b$  with  $\text{Prob}(X = a) = p$ , then

$$E(X) = ap + b(1 - p).$$

In particular, if  $Y = \pm 1$  with probability  $1/2$ , then  $E(Y) = 0$ .

If  $\text{Prob}(X = a) = 1$ , then we say  $X$  is the constant  $a$ . In this case, the mean is

$$E(X) = a \cdot \text{Prob}(X = 1) = a,$$

and the variance is

$$E((X - a)^2) = (a - a)^2 \text{Prob}(X = 1) = 0.$$

Conversely, a random variable with variance zero is a constant.

Let  $X$  and  $Y$  be random variables and let  $a, b$  be constants. A basic property of the mean is *linearity*

$$E(aX + bY) = aE(X) + bE(Y).$$

Using this, we have

$$\begin{aligned} \text{Var}(X) &= \sigma^2 = E((X - \mu)^2) \\ &= E(X^2 - 2\mu X + \mu^2) = E(X^2) - 2\mu E(X) + \mu^2 = E(X^2) - E(X)^2. \end{aligned}$$

We conclude

$$E(X^2) = \mu^2 + \sigma^2 = (E(X))^2 + \text{Var}(X). \quad (5.3.2)$$

Let  $X$  have mean  $\mu$  and variance  $\sigma^2$ , and write

$$Z = \frac{X - \mu}{\sigma}.$$

Then

$$E(Z) = \frac{1}{\sigma} E(X - \mu) = \frac{E(X) - \mu}{\sigma} = \frac{\mu - \mu}{\sigma} = 0,$$

and

$$E(Z^2) = \frac{1}{\sigma^2} E((X - \mu)^2) = \frac{\sigma^2}{\sigma^2} = 1.$$

We conclude  $Z$  has mean zero and variance one.

A random variable is *standard* if its mean is zero and its variance is one. The variable  $Z$  is the *standardization* of  $X$ . For example, the standardization of the bernoulli random variable is

$$\frac{X - p}{\sqrt{p(1 - p)}}.$$

Since  $p(1 - p)$  is unchanged when  $p$  is replaced by  $1 - p$ , the graph of the bernoulli variance is a symmetric hump vanishing at  $p = 0$  and  $p = 1$  (Figure 5.13). Therefore the bernoulli variance is maximized at  $p = 1/2$ , where it equals  $p(1 - p) = 1/4$ .

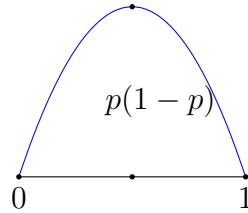


Figure 5.13: Binary variance.

If  $X$  is a random variable, so are  $X$ ,  $X^2$ ,  $X^3$ , .... These are *powers* of  $X$ . The *moments* of  $X$  are the power means

$$E(X), E(X^2), E(X^3), \dots$$

The moments of a random variable may be combined into a sum. To explain this, we bring in the *exponential series*

$$e^t = 1 + t + \frac{t^2}{2!} + \frac{t^3}{3!} + \dots$$

where  $t$  is any real number. The number  $e$ , Euler's constant (§4.4), is approximately 2.7, as can be seen from

$$e = e^1 = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \dots = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \dots$$

Since  $X$  has real values, so does  $tX$ , so  $e^{tX}$  is also a random variable. The *moment generating function* is the mean of  $e^{tX}$ ,

$$M(t) = M_X(t) = E(e^{tX}) = 1 + tE(X) + \frac{t^2}{2!}E(X^2) + \frac{t^3}{3!}E(X^3) + \dots$$

For example, for the smartphone random variable  $X = 0, 1$  with  $Prob(X = 1) = p$ ,  $X^2 = X$ ,  $X^3 = X$ , ..., so

$$M(t) = 1 + tE(X) + \frac{t^2}{2!}E(X^2) + \frac{t^3}{3!}E(X^3) + \dots = 1 + tp + \frac{t^2}{2!}p + \frac{t^3}{3!}p + \dots$$

which equals

$$M(t) = (1 - p) + pe^t.$$

In §5.2, we discussed independence of events. Now we do the same for random variables. Let  $X$  and  $Y$  be random variables. We say  $X$  and  $Y$  are *uncorrelated* if the expectations multiply,

$$E(XY) = E(X)E(Y).$$

Otherwise, we say  $X$  and  $Y$  are *correlated*.

By (5.3.2), a random variable  $X$  is always correlated to itself, unless it is a constant.

Suppose  $X$  and  $Y$  take on the values  $X = \pm 1$  and  $Y = 0, 1$  with the probabilities

$$(X, Y) = \begin{cases} (1, 1) & \text{with probability } a, \\ (1, 0) & \text{with probability } b, \\ (-1, 1) & \text{with probability } b, \\ (-1, 0) & \text{with probability } c. \end{cases} \quad (5.3.3)$$

We investigate when  $X$  and  $Y$  are uncorrelated. Here  $a > 0$ ,  $b > 0$ , and  $c > 0$ .

First, because the total probability equals 1,

$$a + 2b + c = 1. \quad (5.3.4)$$

Also we have

$$\text{Prob}(X = 1) = a + b = \text{Prob}(Y = 1), \quad \text{Prob}(X = -1) = b + c = \text{Prob}(Y = 0),$$

and

$$E(X) = a - c, \quad E(Y) = a + b.$$

Now  $X$  and  $Y$  are uncorrelated if

$$a - b = E(XY) = E(X)E(Y) = (a - c)(a + b). \quad (5.3.5)$$

Solving (5.3.4), (5.3.5) using Python,

```
from sympy import *

a,b,c = symbols('a,b,c')
eq1 = a + 2*b + c - 1
eq2 = a - b - (a-c)*(a+b)
solutions = solve([eq1,eq2],a,b)
print(solutions)
```

we see  $X$  and  $Y$  are uncorrelated if

$$b = \sqrt{c} - c, \quad a = c - 2\sqrt{c} + 1. \quad (5.3.6)$$

For example,  $X$  and  $Y$  are uncorrelated when  $c = 1/4$ , which leads to  $a = b = 1/4$ . Also,  $X$  and  $Y$  are uncorrelated if  $c = .01$ , which leads to  $a = .81$  and  $b = .09$ .

Let  $X$  and  $Y$  be random variables. We say  $X$  and  $Y$  are *independent* if all powers of  $X$  are uncorrelated with all powers of  $Y$ ,

$$E(X^n Y^m) = E(X^n) E(Y^m). \quad (5.3.7)$$

Clearly, if  $X$  and  $Y$  are independent, then, by taking  $n = 1$  and  $m = 1$ ,  $X$  and  $Y$  are uncorrelated.

Suppose  $X$  and  $Y$  satisfy (5.3.3) and (5.3.6). Since  $X = \pm 1$ ,  $X^n = 1$  for  $n$  even and  $X^n = X$  for  $X$  odd. Since  $Y = 0, 1$ ,  $Y^n = Y$  for all  $n$ . This is enough to show that, in this case,  $X$  and  $Y$  uncorrelated is equivalent to  $X$  and  $Y$  independent. However, this is certainly not true in general.

Here is an example of uncorrelated random variables that are dependent. Let  $X, Y$  be as above with  $a = b = c = 1/4$ . Then, as we have seen,  $X$  and  $Y$  are uncorrelated. Let  $U = XY$ . Then

$$E(U) = E(XY) = E(X)E(Y) = 0 \cdot \frac{1}{2} = 0.$$

Since  $UY = XYY = XY = U$ ,  $E(UY) = E(U) = 0$ . Hence  $Y$  and  $U$  are uncorrelated. But, since  $U^2 = Y$ ,  $Y$  and  $U^2$  are correlated, so  $Y$  and  $U$  are not independent.



Let  $X$  and  $Y$  be random variables. Expanding the exponentials into their series, and using (5.3.7), one can show

### Independence and Moment Generating Functions

Let  $X$  and  $Y$  be random variables. Then  $X$  and  $Y$  are independent if their moment generating functions multiply,

$$M_{X,Y}(a, b) \equiv E(e^{aX+bY}) = E(e^{aX}) E(e^{bY}) = M_X(a) M_Y(b).$$

The expectation on the left is the *joint* moment generating function  $M_{X,Y}(a, b)$  of the pair  $(X, Y)$ . One may have joint moment generating functions for triples  $(X, Y, Z)$ , by writing

$$M_{X,Y,Z}(a, b, c) = E(e^{aX+bY+cZ}).$$

Then we say  $X, Y, Z$  are independent if

$$M_{X,Y,Z}(a, b, c) = M_X(a) M_Y(b) M_Z(c).$$

Of course, this generalizes to any tuple of random variables.

As an illustration, consider an ordinary dice with  $X = 1, X = 2, \dots, X = 6$  equally probable. Then  $\text{Prob}(X = k) = 1/6$ ,  $k = 1, 2, \dots, 6$ . Evaluating the geometric sum,

$$M_X(t) = E(e^{tX}) = \sum_{k=1}^6 \frac{1}{6} e^{6k} = \frac{1}{6} \frac{e^{7t} - e^t}{e^t - 1}.$$

Now suppose we have a non-standard dice with unknown probabilities for  $Y = 0, Y = 1, \dots, Y = 6$ . if we are told the probabilities of the sum  $X + Y$  is uniform over  $1 \leq X + Y \leq 12$ , how should we choose the probabilities for  $Y$ ?

To answer this, use

$$M_{X+Y}(t) = E(e^{t(X+Y)}) = E(e^{tX}) E(e^{tY}).$$

Since

$$M_{X+Y}(t) = \frac{1}{12} \sum_{k=1}^{12} e^{tk} = \frac{1}{12} \frac{e^{13t} - e^t}{e^t - 1},$$

we obtain

$$\frac{1}{12} \frac{e^{13t} - e^t}{e^t - 1} = M_Y(t) \cdot \frac{1}{6} \frac{e^{7t} - e^t}{e^t - 1}.$$

Factoring

$$e^{13t} - e^t = e^t(e^{6t} - 1)(e^{6t} + 1), \quad e^{7t} - e^t = e^t(e^{6t} - 1),$$

we obtain

$$M_Y(t) = \frac{1}{2}(e^{6t} + 1).$$

This says

$$\text{Prob}(Y = 0) = \frac{1}{2}, \quad \text{Prob}(Y = 6) = \frac{1}{2},$$

and all other probabilities are zero.



Let  $X$  and  $Y$  be random variables. We say  $X$  and  $Y$  are *identically distributed* if their moments are equal,

$$E(X^n) = E(Y^n), \quad n \geq 1.$$

This is equivalent to  $X$  and  $Y$  having equal probabilities,

$$\text{Prob}(a < X < b) = \text{Prob}(a < Y < b).$$

For example, if  $X$  and  $Y$  satisfy (5.3.3), then  $X$  and  $2Y - 1$  are identically distributed. However,  $X$  and  $2Y - 1$  are independent if and only if  $X$  and  $Y$  are independent, which, as we saw above, happens only when (5.3.6) holds.

On the other hand, let  $X$  be any random variable, and let  $Y = X$ . Then  $X$  and  $Y$  are identically distributed, but are certainly correlated. So identical distributions does not imply independence, nor vice-versa.

Let  $X$  be a random variable. A *simple random sample* of size  $n$  is a sequence of random variables  $X_1, X_2, \dots, X_n$  that are independent and identically distributed. We also say the sequence  $X_1, X_2, \dots, X_n$  is an *i.i.d. sequence* (independent identically distributed).

For example, going back to the smartphone example, suppose we select  $n$  students at random, where we are allowed to select the same student twice. We obtain numbers  $x_1, x_2, \dots, x_n$ . So the result of a single selection experiment is a sequence of numbers  $x_1, x_2, \dots, x_n$ . To make statistical statements about the results, we repeat this experiment many times, and we obtain a sequence of numbers  $x_1, x_2, \dots, x_n$  each time.

This process can be thought of  $n$  machines producing  $x_1, x_2, \dots, x_n$  each time, or  $n$  random variables  $X_1, X_2, \dots, X_n$  (Figure 5.14). By making each of the  $n$  selections independently, we end up with an i.i.d. sequence, or a simple random sample.

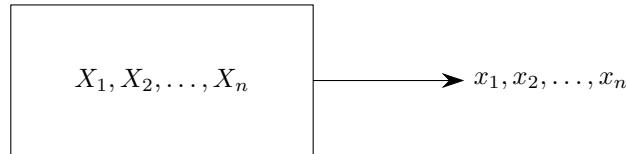


Figure 5.14: When we sample  $X_1, X_2, \dots, X_n$ , we get  $x_1, x_2, \dots, x_n$ .

Let  $X_1, X_2, \dots, X_n$  be a simple random sample. Then  $X_1, X_2, \dots, X_n$  are identically distributed. Let  $\mu$  be their common mean  $E(X)$ . The *sample*

mean is

$$\bar{X} = \frac{X_1 + X_2 + \cdots + X_n}{n} = \frac{1}{n} \sum_{k=1}^n X_k.$$

Then

$$E(\bar{X}) = \frac{1}{n} E(X_1 + X_2 + \cdots + X_n) = \frac{1}{n} (E(X_1) + E(X_2) + \cdots + E(X_n)) = \frac{1}{n} \cdot n\mu = \mu.$$

We conclude *the mean of the sample mean equals the population mean.*

Now let  $\sigma^2$  be the common variance of  $X_1, X_2, \dots, X_n$ . Since  $\sigma^2 = E(X^2) - E(X)^2$ , we have

$$E(X_k^2) = \mu^2 + \sigma^2.$$

When  $i \neq j$ , by independence,

$$E(X_i X_j) = E(X_i) E(X_j) = \mu \cdot \mu = \mu^2.$$

Putting this all together,

$$\begin{aligned} E(\bar{X}^2) &= \frac{1}{n^2} E \left( \left( \sum_{k=1}^n X_k \right)^2 \right) \\ &= \frac{1}{n^2} \sum_{i,j} E(X_i X_j) \\ &= \frac{1}{n^2} \left( \sum_{i \neq j} E(X_i X_j) + \sum_k E(X_k^2) \right) \\ &= \frac{1}{n^2} (n(n-1)\mu^2 + n(\mu^2 + \sigma^2)) = \mu^2 + \frac{1}{n}\sigma^2. \end{aligned}$$

Since the variance of  $\bar{X}$  equals

$$E(\bar{X}^2) - E(\bar{X})^2 = E(\bar{X}^2) - \mu^2,$$

we conclude *the variance of  $\bar{X}$  equals  $\sigma^2/n$ .* The standard deviation of  $\bar{X}$  is the *standard error*.

### Independence and Variances

If  $X_1, X_2, \dots, X_n$  is a simple random sample (i.i.d. sequence) with mean  $\mu$  and variance  $\sigma^2$ , then the mean and variance of  $\bar{X}$  are

$$E(\bar{X}) = \mu \quad \text{and} \quad \text{Var}(\bar{X}) = \frac{\sigma^2}{n}.$$

In particular, when  $X_1, X_2, \dots, X_n$  is a bernoulli simple random sample, the mean and variance of  $\bar{X}$  are  $p$  and  $p(1-p)/n$ .

More generally, by a similar calculation, we have

### Independence and Variances

Let  $X_1, X_2, \dots, X_n$  be a sequence of independent random variables with means  $\mu_1, \mu_2, \dots, \mu_n$ , and variances  $\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2$ , and let

$$S_n = X_1 + X_2 + \dots + X_n.$$

Then the mean and variance of  $S_n$  are

$$\mu_1 + \mu_2 + \dots + \mu_n \quad \text{and} \quad \sigma_1^2 + \sigma_2^2 + \dots + \sigma_n^2. \quad (5.3.8)$$



Now we restate the two fundamental results of probability in the language of this section. Let  $X_1, X_2, \dots$ , be independent identically distributed random variables, each with mean  $\mu$  and variance  $\sigma^2$ .

Let

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_n}{n}, \quad n \geq 1,$$

and let  $Z$  be a *normal* random variable with mean  $\mu$  and variance  $\sigma^2$ .

**1. Law of Large Numbers.** For every<sup>2</sup> sample  $x_1, x_2, \dots$ ,

$$\lim_{n \rightarrow \infty} \frac{x_1 + x_2 + \dots + x_n}{n} = \mu.$$

---

<sup>2</sup>This holds for almost every sample, in the sense that the exceptions form a negligible set of samples.

2. **Central Limit Theorem.** For every  $a < b$ ,

$$\lim_{n \rightarrow \infty} \text{Prob}(a < \bar{X} < b) = \text{Prob}(a < Z < b).$$

## 5.4 Normal Distribution

A random variable  $Z$  has a *standard normal distribution* or *Z distribution* if its moment generating functionfunction!moment generating!normal satisfies

$$M_Z(t) = E(e^{tZ}) = e^{t^2/2} = \exp(t^2/2).$$

In this case, we write  $Z \sim N(0, 1)$ .

This is equivalent to specifying the probability density that  $Z$  lies in a small interval  $[a, b]$  containing  $z$ . This is specified by the famous formula

$$\frac{\text{Prob}(a < Z < b)}{b - a} = \frac{1}{N} \cdot e^{-z^2/2}, \quad a < z < b. \quad (5.4.1)$$

Here  $N$  is a constant to make the total area under the graph equal to one (Figure 5.15). In other words, (5.4.1) is the pdf of the normal distribution.

When the interval  $[a, b]$  is not small, the correct formula is obtained by integration, which means dividing  $[a, b]$  into many small intervals and summing. We will not use this density formula directly.

Using Python, the normal probability density is plotted by

```
from scipy.stats import norm as Z
from numpy import *
from matplotlib.pyplot import *

mu, sdev = 0, 1

grid()
z = arange(mu-3*sdev, mu+3*sdev, .01)

# if mu, sdev not specified, then Z is standard
plot(z, Z(mu, sdev).pdf(z))
show()
```

Here pdf stands for *probability density function*.

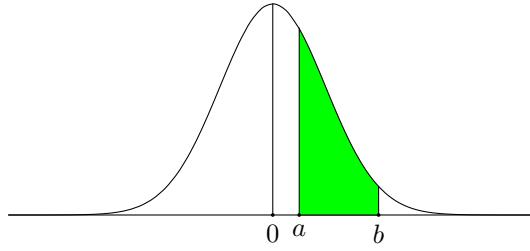


Figure 5.15: The standard normal distribution.



Expand both sides of the definition of  $M_Z(t)$  in exponential series. This results in

$$\begin{aligned} 1 + tE(Z) + \frac{t^2}{2!}E(Z^2) + \frac{t^3}{3!}E(Z^3) + \frac{t^4}{4!}E(Z^4) + \dots \\ = 1 + \frac{t^2}{2} + \frac{1}{2!} \left( \frac{t^2}{2} \right)^2 + \frac{1}{3!} \left( \frac{t^2}{2} \right)^3 + \dots \end{aligned}$$

From this, the odd moments of  $Z$  are zero, and the even moments are

$$E(Z^{2n}) = \frac{(2n)!}{2^n n!}, \quad n = 0, 1, 2, \dots$$

By separating the even and the odd factors, this simplifies to

$$\begin{aligned} E(Z^{2n}) &= \frac{(1 \cdot 3 \cdot 5 \cdots (2n-1))(2 \cdot 4 \cdots 2n)}{2^n n!} \\ &= \frac{(1 \cdot 3 \cdot 5 \cdots (2n-1))2^n n!}{2^n n!} \\ &= 1 \cdot 3 \cdot 5 \cdots (2n-1), \quad n \geq 1. \end{aligned} \tag{5.4.2}$$

For example,

$$E(Z) = 0, E(Z^2) = 1, E(Z^3) = 0, E(Z^4) = 3, E(Z^5) = 0, E(Z^6) = 15.$$

More generally, we say  $X$  has a *normal distribution* with mean  $\mu$  and variance  $\sigma^2$ , and we write  $X \sim N(\mu, \sigma)$ , if

$$M_X(t) = E(e^{tX}) = \exp(\mu t + \sigma^2 t^2 / 2). \tag{5.4.3}$$

Then

$$X \sim N(\mu, \sigma) \iff Z = \frac{X - \mu}{\sigma} \sim N(0, 1).$$

A normal distribution is a standard normal distribution when  $\mu = 0$  and  $\sigma = 1$ .



As mentioned in §5.3, the important result is

### Central Limit Theorem

Let  $X_1, X_2, \dots, X_n$  be independent identically distributed random variables. Then, for large  $n$ ,

$$S_n = X_1 + X_2 + \cdots + X_n$$

and  $\bar{X} = S_n/n$  have approximately normal distributions. More precisely, if  $\mu$  and  $\sigma$  are the mean and standard deviation of each  $X$ , and  $(a, b)$  is an interval,

$$\lim_{n \rightarrow \infty} \text{Prob}\left(\mu + a \cdot \frac{\sigma}{\sqrt{n}} < \bar{X} < \mu + b \cdot \frac{\sigma}{\sqrt{n}}\right) = \text{Prob}(a < Z < b).$$



The standard normal distribution is symmetric about zero, and has a specific width. Because of the symmetry, a random number  $Z$  following this distribution is equally likely to satisfy  $Z < 0$  and  $Z > 0$ , so  $\text{Prob}(Z < 0) = \text{Prob}(Z > 0)$ . Since the total area equals 1,

$$\text{Prob}(Z < 0) + \text{Prob}(Z > 0) = 1,$$

we expect the chance that  $Z < 0$  should equal  $1/2$ . In other words, because of the symmetry of the curve, we expect to be 50% confident that  $Z < 0$ , or 0 is at the 50-th percentile level. So

$$\text{chance} = \text{confidence} = \text{percentile} = \text{area}$$

To summarize, we expect  $\text{Prob}(Z < 0) = 1/2$ .

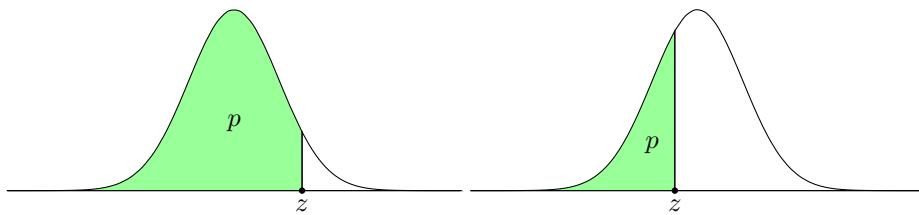


Figure 5.16:  $z = \text{norm.ppf}(p)$  and  $p = \text{norm.cdf}(z)$ .

When

$$\text{Prob}(Z < z) = p,$$

we say  $z$  is the *z-score*  $z$  corresponding to the *p-value*  $p$ . Equivalently, we say our confidence that  $Z < z$  is  $p$ , or the percentile of  $z$  equals  $100p$ . In Python, the relation between  $z$  and  $p$  (Figure 5.16) is specified by

```
from scipy.stats import norm as Z

p = Z.cdf(z)
z = Z.ppf(p)
```

`ppf` is the *percentile point function*, and `cdf` is the *cumulative distribution function*.

In Figure 5.17, the red areas are the *lower tail p-value*  $\text{Prob}(Z < z)$ , the *two-tail p-value*  $\text{Prob}(|Z| > z)$ , and the *upper tail p-value*  $\text{Prob}(Z > z)$ .

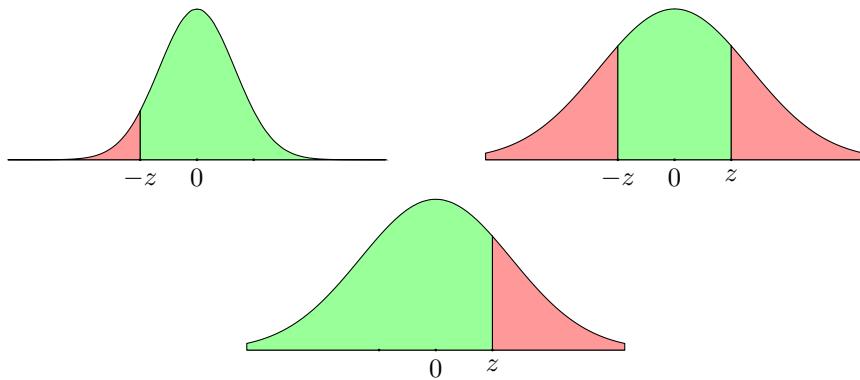


Figure 5.17: Confidence (green) or significance (red) (lower-tail, two-tail, upper-tail).

By symmetry of the graph, upper-tail and two-tail  $p$ -values can be computed from lower tail  $p$ -values.

$$\text{Prob}(a < Z < b) = \text{Prob}(Z < b) - \text{Prob}(Z < a),$$

and

$$\text{Prob}(|Z| < z) = \text{Prob}(-z < Z < z) = \text{Prob}(Z < z) - \text{Prob}(Z < -z),$$

and

$$\text{Prob}(Z > z) = 1 - \text{Prob}(Z < z).$$

To go backward, suppose we are given  $\text{Prob}(|Z| < z) = p$  and we want to compute the cutoff  $z$ . Then  $\text{Prob}(|Z| > z) = 1 - p$ , so  $\text{Prob}(Z > z) = (1 - p)/2$ . This implies

$$\text{Prob}(Z < z) = 1 - (1 - p)/2 = (1 + p)/2.$$

In Python,

```
from scipy.stats import norm as Z

# p = P(|Z| < z)

z = Z.ppf((1+p)/2)
```



Now let's zoom in closer to the graph and mark off 1, 2, 3 on the horizontal axis to obtain specific colored areas as in Figure 5.18. These areas are governed by the 68-95-99 rule (Table 5.19). Our confidence that  $|Z| < 1$  equals the blue area 0.685, our confidence that  $|Z| < 2$  equals the sum of the blue plus green areas 0.955, and our confidence that  $|Z| < 3$  equals the sum of the blue plus green plus red areas 0.997. This is summarized in Table 5.19.

The possibility  $|Z| > 1$  is called a 1-sigma event,  $|Z| > 2$  a 2-sigma event, and so on. So a 2-sigma event is 95.5% unlikely, or 4.5% likely. An event is considered *statistically significant* if it's a 2-sigma event or more. In other words, *something is significant if it's unlikely*. A six-sigma event  $|Z| > 6$  is 2 in a billion. You want a plane crash to be six-sigma.

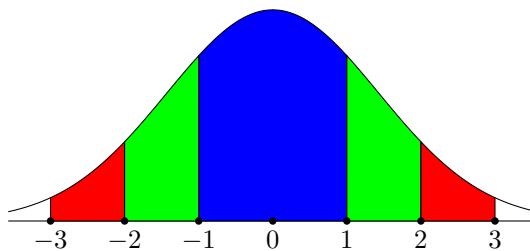


Figure 5.18: 68%, 95%, 99% confidence cutoffs for standard normal.

Figure 5.18 is not to scale, because a 1-sigma event should be where the curve inflects from convex to concave (in the figure this happens closer to 2.7). Moreover, according to Table 5.19, the left-over white area should be .03% (3 parts in 10,000), which is not what the figure suggests.

cutoff	abs confidence	two-tail $p$ -value
$z$	$1 - p$	$p$
1	.685	.315
2	.955	.045
3	.997	.003

Table 5.19: Cutoffs, confidence levels,  $p$ -values.



An event is *statistically significant* if its  $p$ -value is 5% or less (Table 5.20). For example,  $Z > z$  is statistically significant if  $\text{Prob}(Z > z)$  is .05 or less, which means  $z$  is greater than 1.64,  $Z < z$  is statistically significant if  $\text{Prob}(Z < z)$  is .05 or less, which means  $z$  is less than  $-1.64$ , and  $|Z| > z$  is statistically significant if  $\text{Prob}(|Z| > z)$  is .05 or less, which means  $|z|$  is greater than 1.96.

An event is *highly significant* if its  $p$ -value is 1% or less (Table 5.20). For example,  $Z > z$  is highly significant if  $\text{Prob}(Z > z)$  is .01 or less, which means  $z$  is greater than 2.33,  $Z < z$  is highly significant if  $\text{Prob}(Z < z)$  is .01 or less, which means  $z$  is less than  $-2.33$ , and  $|Z| > z$  is highly significant if  $\text{Prob}(|Z| > z)$  is .01 or less, which means  $|z|$  is greater than 2.56.

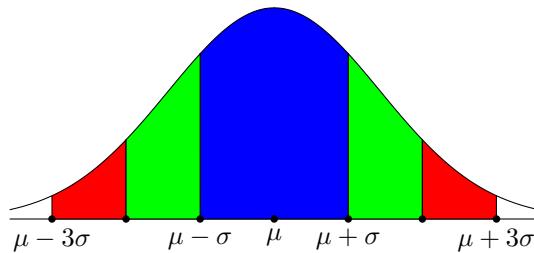


Figure 5.21: 68%, 95%, 99% cutoffs for non-standard normal.

event	type	<i>p</i> -value	<i>z</i> -score
$Z > z$	upper tail	.05	1.64
$Z < z$	lower tail	.05	-1.64
$ Z  > z$	two-tail	.05	1.96
$Z > z$	upper tail	.01	2.33
$Z < z$	lower tail	.01	-2.33
$ Z  > z$	two-tail	.01	2.56

Table 5.20: *p*-values at 5% and at 1%.

In general, the normal distribution is not centered at the origin, but elsewhere. We say  $X$  is normal with mean  $\mu$  and standard deviation (SD)  $\sigma$  if

$$Z = \frac{X - \mu}{\sigma}$$

is distributed according to a standard normal. We write  $N(\mu, \sigma)$  for the normal with mean  $\mu$  and SD  $\sigma$ . As its name suggests, it is easily checked that such a random variable  $X$  has mean  $\mu$  and SD  $\sigma$ . For the normal distribution with mean  $\mu$  and standard deviation  $\sigma$ , the cutoffs are as in Figure 5.21. In Python, `norm(m,s)` returns the normal with mean  $m$  and standard deviation  $s$ .

Here is a sample computation. Let  $X$  be a normal random variable with mean  $\mu$  and standard deviation  $\sigma$ , and suppose  $Prob(X < 7) = .15$ , and  $Prob(X < 19) = .9$ . Given this data, we find  $\mu$  and  $\sigma$  as follows.

With  $Z$  as above, we have

$$Prob(Z < (7 - \mu)/\sigma) = .15, \quad \text{and} \quad Prob(Z < (19 - \mu)/\sigma) = .9.$$

Also, since  $Z$  is standard, we compute

$$\begin{aligned} a &= Z.\text{ppf}(.15) \\ b &= Z.\text{ppf}(.9) \end{aligned}$$

By definition of  $\text{ppf}$  (see above), we then have

$$a = \frac{7 - \mu}{\sigma}, \quad b = \frac{19 - \mu}{\sigma}.$$

These are two equations in two unknowns. Multiplying both equations by  $\sigma$  then subtracting, we obtain  $\mu$  and  $\sigma$ ,

$$\sigma = \frac{19 - 7}{b - a}, \quad \mu = 7 - a\sigma.$$



Let  $\bar{X}$  be the sample mean

$$\bar{X} = \frac{X_1 + X_2 + \cdots + X_n}{n},$$

drawn from a normally distributed population with mean  $\mu$  and standard deviation  $\sigma$ . As we saw in §5.3, the standard deviation of  $\bar{X}$ , the standard error, is  $\sigma/\sqrt{n}$ .

To compute probabilities for  $\bar{X}$  when  $X$  has mean  $\mu$  and standard deviation  $\sigma$ , standardize  $\bar{X}$  by writing

$$Z = \sqrt{n} \cdot \frac{\bar{X} - \mu}{\sigma},$$

then compute standard normal probabilities.



Here are three examples. In the first example, suppose student grades are normally distributed with mean 80 and variance 16. This says the average of all grades is 80, and the SD is 4. If a grade is  $g$ , the standardized grade is

$$z = \frac{g - 80}{4}.$$

A student is picked and their grade was  $g = 90$ . Is this significant? Is it highly significant? In effect, we are asking, how unlikely is it to obtain such a grade? Remember,

$$\text{significant} = \text{unlikely}$$

Since the standard deviation is 4, the student's  $z$ -score is

$$z = \frac{g - 80}{4} = \frac{90 - 80}{4} = 2.5.$$

What's the upper-tail  $p$ -value corresponding to this  $z$ ? It's

$$\text{Prob}(Z > z) = \text{Prob}(Z > 2.5) = .0062,$$

or .62%. Since the upper-tail  $p$ -value is less than 1%, yes, this student's grade is both significant and highly significant.

For the second example, suppose a sample of  $n = 9$  students are selected and their sample average grade is  $\bar{g} = 84$ . Is this significant? Is it highly significant? This time we take

$$z = \sqrt{n} \cdot \frac{\bar{g} - 80}{4} = 3 \cdot \frac{84 - 80}{4} = 3.$$

What's the upper-tail  $p$ -value corresponding to this  $z$ ? It's

$$\text{Prob}(Z > z) = \text{Prob}(Z > 2.5) = 0.0013,$$

or .13%. Since the upper-tail  $p$ -value is less than 1%, yes, this sample average grade is both significant and highly significant.

For the third example, suppose a single selected student has the same grade as the sample average,  $g = 84$ . Is this significant? Here

$$z = \frac{g - 80}{4} = \frac{84 - 80}{4} = 1.$$

Since the upper-tail significance corresponding to  $z = 1$

$$\text{Prob}(Z > 1) = \frac{1}{2} \text{Prob}(|Z| > 1) = \frac{1}{2}(1 - \text{Prob}(|Z| < 1)) = \frac{1}{2}(1 - .68) = .16,$$

or 16%, this grade is not significant.



The constructor `numpy.random.normal` returns samples of normally distributed real numbers. For example.

```
from numpy.random import default_rng
from numpy import sqrt

rng = default_rng()

mean, sdev, n = 80, 4, 20
rng.normal(mean,sdev,n)
```

returns 20 normally distributed numbers, with specified mean and variance.

Suppose student grades are normally distributed with mean 80 and variance 16. How many students should be sampled so that the chance that at least one student's grade lies below 70 is at least 50%?

To solve this, if  $p$  is the chance that a single student has a grade below 70, then  $1 - p$  is the chance that the student has a grade above 70. If  $n$  is the sample size,  $(1 - p)^n$  is the chance that all sample students have grades above 70. Thus the requested chance is  $1 - (1 - p)^n$ . The following code shows the answer is  $n = 112$ .

```
from scipy.stats import norm as Z

x = 70
mean, sdev = 80, 4
p = Z(mean,sdev).cdf(x)

for n in range(2,200):
    q = 1 - (1-p)**n
    print(n, q)
```



Here is the code for computing tail probabilities for the sample mean  $\bar{X}$  drawn from a normally distributed population with mean  $\mu$  and standard deviation  $\sigma$ . When  $n = 1$ , this applies to a single normal random variable.

```
#####
# P-values
```

```
#####
from numpy import *
from scipy.stats import norm as Z

def pvalue(mean,sdev,n,xbar,type):
    Xbar = Z(mean,sdev/sqrt(n))
    if type == "lower-tail": p = Xbar.cdf(xbar)
    elif type == "upper-tail": p = 1 - Xbar.cdf(xbar)
    elif type == "two-tail": p = 2 *(1 - Xbar.cdf(abs(xbar)))
    else:
        print("What's the tail type?")
        return
    print("type: ",type)
    print("mean,sdev,n,xbar: ",mean,sdev,n,xbar)
    print("p-value: ",p)
    z = sqrt(n) * (xbar - mean) / sdev
    print("z-score: ",z)

type = "upper-tail"
mean = 80
sdev = 4
n = 1
xbar = 90

pvalue(mean,sdev,n,xbar,type)
```

## 5.5 Chi-squared Distribution

Let  $X$  and  $Y$  be independent standard normal random variables. Then  $(X, Y)$  is a random point in the plane. What is the probability that the point  $(X, Y)$  lies inside a square (Figure 5.22)? Specifically, assume the square is  $|X| \leq 1$  and  $|Y| \leq 1$ . Since  $X$  and  $Y$  independent, the probability  $(X, Y)$  lies in the square is

$$\begin{aligned} \text{Prob}(|X| < 1 \text{ and } |Y| < 1) &= \text{Prob}(|X| < 1) \text{Prob}(|Y| < 1) \\ &= \text{Prob}(|X| < 1)^2 = .685^2 = .469. \end{aligned}$$

What is the probability  $(X, Y)$  lies inside the unit circle,

$$\text{Prob}(X^2 + Y^2 < 1)?$$

Here the answer is not as straightforward, and leads us to introduce the chi-squared distribution.

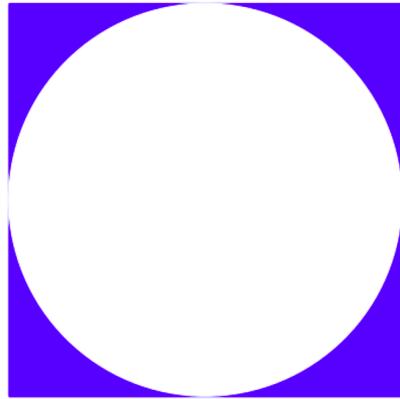


Figure 5.22:  $(X, Y)$  in the square and in the circle.



A random variable  $U$  has a *chi-squared distribution with degree 1* if

$$M_U(u) = E(e^{uU}) = \frac{1}{\sqrt{1 - 2u}}.$$

To compute the moments of  $U$ , we use the binomial theorem (7.1.22)

$$(1 + x)^p = \sum_{n=0}^{\infty} \binom{p}{n} x^n = 1 + px + \binom{p}{2} x^2 + \binom{p}{3} x^3 + \dots$$

to write out  $M_U(u)$ . Taking  $p = -1/2$  and  $x = -2u$ ,

$$\frac{1}{\sqrt{1 - 2u}} = (1 - 2u)^{-1/2} = \sum_{n=0}^{\infty} \binom{-1/2}{n} (-2u)^n.$$

Since

$$\frac{1}{\sqrt{1-2u}} = E(e^{uU}) = \sum_{n=0}^{\infty} \frac{u^n}{n!} E(U^n),$$

comparing coefficients of  $u^n/n!$  shows

$$E(U^n) = (-2)^n n! \binom{-1/2}{n}, \quad n = 0, 1, 2, \dots \quad (5.5.1)$$

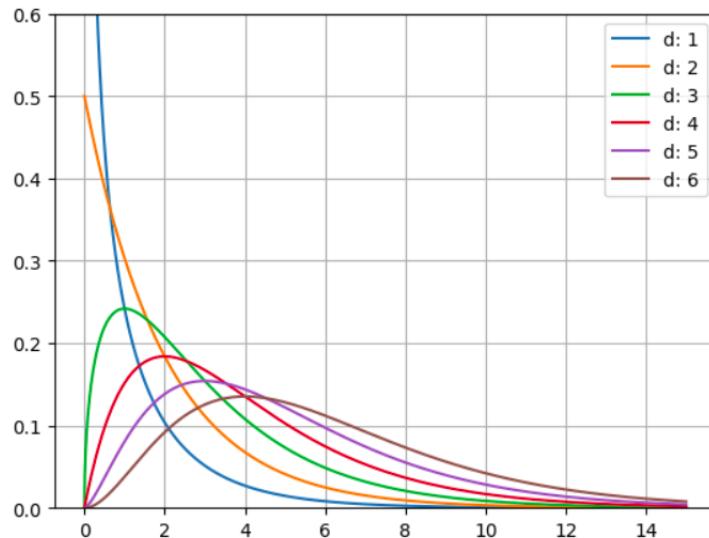


Figure 5.23: Chi-squared distribution with different degrees.

Using the definition

$$\binom{p}{n} = \frac{p \cdot (p-1) \cdots (p-n+1)}{n!},$$

$\binom{p}{n}$  makes sense for fractional  $p$  (see (4.3.12)). With this, we have

$$\begin{aligned} E(U^n) &= (-2)^n n! \frac{(-1/2) \cdot (-1/2 - 1) \cdots (-1/2 - n + 1)}{n!} \\ &= 1 \cdot 3 \cdot 5 \cdot 7 \cdots (2n-1). \end{aligned}$$

But this equals the right side of (5.4.2). Thus the left sides of (5.4.2) and (5.5.1) are equal. This shows

### Chi-squared is the Square of Normal

If  $Z$  is standard normal, then  $U = Z^2$  is chi-squared with degree 1.



More generally, we say  $U$  is *chi-squared with degree  $d$*  if

$$U = U_1 + U_2 + \cdots + U_d = Z_1^2 + Z_2^2 + \cdots + Z_d^2, \quad (5.5.2)$$

with independent standard normal  $Z_1, Z_2, \dots, Z_d$ .

By independence, the moment generating functions multiply (§5.3), so the moment generating function for chi-squared with degree  $d$  is

$$M_U(t) = E(e^{tU}) = \frac{1}{(1 - 2t)^{d/2}}.$$

Going back to the question posed at the beginning of the section, we have  $X$  and  $Y$  independent standard normal and we want

$$\text{Prob}(X^2 + Y^2 < 1).$$

If we set  $U = X^2 + Y^2$ , we want<sup>3</sup>  $\text{Prob}(U < 1)$ . Since  $U$  is chi-squared with degree  $d = 2$ , we use `chi2.cdf(u,d)`. Then the code

```
from scipy.stats import chi2 as U
d = 2
u = 1
U(d).cdf(u)
```

returns 0.39.




---

<sup>3</sup>Geometrically,  $\text{Prob}(U < 1)$  is the probability that a normally distributed point is inside the unit sphere in  $d$ -dimensional space.

Let us compute the mean and variance of a chi-squared  $U$  with degree  $d$ . By (5.5.2),

$$E(U) = \sum_{k=1}^d E(Z_k^2) = \sum_{k=1}^d 1 = d.$$

By (5.4.2) and independence,

$$\begin{aligned} E(U^2) &= \sum_{k,\ell=1}^d E(Z_k^2 Z_\ell^2) \\ &= \sum_{k \neq \ell} E(Z_k^2) E(Z_\ell^2) + \sum_{k=1}^d E(Z_k^4) \\ &= d(d-1) \cdot 1 + d \cdot 3 = d^2 + 2d. \end{aligned}$$

Since the variance equals  $E(U^2) - E(U)^2$ , we conclude

### Mean and Variance

The mean and variance of a chi-squared with degree  $d$  are

$$E(U) = d, \quad \text{and} \quad \text{Var}(U) = 2d.$$

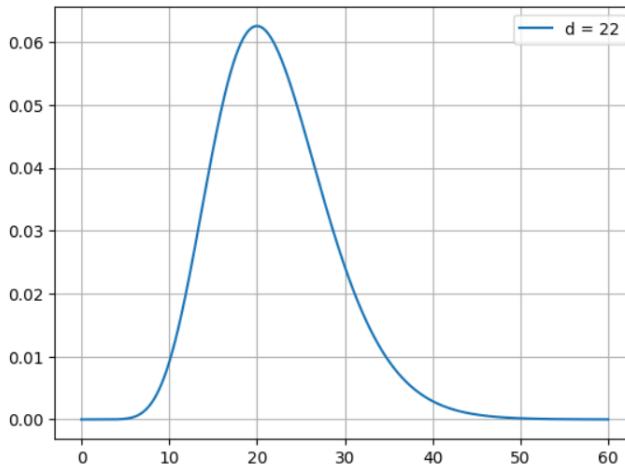


Figure 5.24: With degree  $d \geq 2$ , the chi-squared distribution peaks at  $d - 2$ .

Note that the peak (maximum likelihood point) in a chi-squared distribution with degree  $d$  is not at the mean  $d$ . It is at  $d - 2$  (Figure 5.24).



Because

$$\frac{1}{(1-2t)^{d/2}} \frac{1}{(1-2t)^{d'/2}} = \frac{1}{(1-2t)^{(d+d')/2}},$$

we obtain

### Independence and Chi-squared

If  $U$  and  $U'$  are independent chi-squared with degrees  $d$  and  $d'$ , then  $U + U'$  is chi-squared with degree  $d + d'$ .



To compute distributions for sample variances (below) and Pearson's Test (§6.7), we need to derive chi-squared for correlated normal samples. This is best approached using random vectors.

A *random vector* is a vector  $X = (X_1, X_2, \dots, X_d)$  in  $\mathbf{R}^d$  whose components are random variables. For example, a simple random sample  $X_1, X_2, \dots, X_n$  may be collected into the random vector

$$X = (X_1, X_2, \dots, X_n)$$

in  $\mathbf{R}^n$ .

If  $X$  is a random vector in  $\mathbf{R}^d$ , its *mean* is the vector

$$\mu = E(X) = (E(X_1), E(X_2), \dots, E(X_d)) = (\mu_1, \mu_2, \dots, \mu_d).$$

Assume the mean of  $X$  equals zero. The *variance* of  $X$  is the  $d \times d$  matrix  $Q$  whose  $(i, j)$ -th entry is

$$Q_{ij} = E(X_i X_j), \quad 1 \leq i, j \leq d.$$

In the notation of §2.2,

$$Q = E(X \otimes X).$$

Clearly,  $Q$  is a nonnegative matrix, since

$$v \cdot Qv = v \cdot E(X \otimes X)v = E(v \cdot (X \otimes X)v) = E((v \cdot X)^2)$$

is never negative.

A random vector  $X$  is *normal with mean  $\mu$  and variance  $Q$*  if for every vector  $w$ ,  $w \cdot X$  is normal with mean  $w \cdot \mu$  and variance  $w \cdot Qw$ .

Then  $\mu$  is the mean of  $X$ , and  $Q$  is the variance  $X$ . The random vector  $X$  is *standard normal* if  $\mu = 0$  and  $Q = I$ .

From §5.3, we see

### Normal Random Vectors

$Z_1, Z_2, \dots, Z_d$  is a simple random sample of standard normal random variables if and only if

$$Z = (Z_1, Z_2, \dots, Z_d)$$

is a standard normal random vector in  $\mathbf{R}^d$ .

If  $X$  is a normal random vector with mean zero and variance  $Q$ , then, by definition,  $w \cdot X$  is normal with mean 0 and variance  $w \cdot Qw$ . Using (5.4.3) with  $t = 1$ , the moment generating function of the random vector  $X$  is

$$M_X(w) = E(e^{w \cdot X}) = e^{w \cdot Qw/2}. \quad (5.5.3)$$

In §5.3 we studied correlation and independence. We saw how independence implies uncorrelatedness, but not conversely. Now we show that, for normal random vectors, they are in fact the same.

### Independence and Correlation

If  $(X, Y)$  is a normal random vector, then  $X$  and  $Y$  are uncorrelated iff  $X$  and  $Y$  are independent.

To see this, we write down

$$E(X \otimes X) = A, \quad E(X \otimes Y) = B, \quad E(Y \otimes Y) = C.$$

Then the variance of  $(X, Y)$  is

$$Q = \begin{pmatrix} E(X \otimes X) & E(X \otimes Y) \\ E(Y \otimes X) & E(Y \otimes Y) \end{pmatrix} = \begin{pmatrix} A & B \\ B^t & C \end{pmatrix}$$

. From this, we see  $X$  and  $Y$  are uncorrelated when  $B = 0$ .

With  $w = (u, v)$ , we write

$$w \cdot Qw = \begin{pmatrix} u \\ v \end{pmatrix} \cdot \begin{pmatrix} A & B \\ B^t & C \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = u \cdot Au + u \cdot Bv + v \cdot B^t u + v \cdot Cv.$$

Then

$$M_{X,Y}(w) = E(e^{w \cdot (X,Y)}) = e^{w \cdot Qw/2} = M_X(u) M_Y(v) e^{(u \cdot Bv + v \cdot B^t u)/2}.$$

From this,  $X$  and  $Y$  are independent when  $B = 0$ . Thus, for normal random vectors, independence and uncorrelatedness are the same.



If  $Z$  is a standard normal random vector in  $\mathbf{R}^d$ , then above we saw  $|Z|^2$  is chi-squared with degree  $d$ . Now we generalize this result to correlated normal random vectors.

Recall the pseudo-inverse  $Q^+$  of the matrix  $Q$  (§2.3).

### Correlated Normal Random Vector

Let  $X$  be a normal random vector with mean zero and variance  $Q$ .

Then

$$U = X \cdot Q^+ X$$

is chi-squared of degree  $r$ , where  $r$  is the rank of the matrix  $Q$ .

To derive this, we use the eigenvalue decomposition (§3.2) of  $Q$ ,

$$Q = USU^t.$$

Here  $S$  is a square diagonal matrix and  $U$  is an orthogonal matrix,  $U^t U = I$ . The diagonal entries of  $S$  are the eigenvalues of  $Q$ .

Since the rank of  $Q$  is  $r$ , and this equals the rank of  $S$ , only  $r$  of these eigenvalues are positive,  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_r > 0$ , and  $\lambda_{r+1} = \dots = \lambda_n = 0$ .

Moreover,

$$S^+ = \begin{pmatrix} 1/\lambda_1 & 0 & 0 & \dots & 0 \\ 0 & 1/\lambda_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 1/\lambda_r & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

and

$$Q^+ = US^+U^t.$$

If we set  $Y = U^t X = (Y_1, Y_2, \dots, Y_d)$ , then

$$E(Y \otimes Y) = E((U^t X) \otimes (U^t X)) = U^t E(X \otimes X)U = U^t Q U = S.$$

Writing this out, the random variables  $Y_1, Y_2, \dots, Y_d$  satisfy

$$E(Y_i Y_j) = \begin{cases} \lambda_i, & i = j \\ 0, & i \neq j. \end{cases}$$

Since  $\lambda_i = 0$  for  $i > r$ , we see  $Y_i = 0$  for  $i > r$ . If we set

$$Z_i = \frac{1}{\sqrt{\lambda_i}} Y_i, \quad i = 1, 2, \dots, r,$$

then  $Z$  is a standard normal  $r$ -vector. By (5.5.2),

$$|Z|^2 = Z_1^2 + Z_2^2 + \cdots + Z_r^2$$

is chi-squared with degree  $r$ . But

$$|Z|^2 = \sum_{k=1}^r Z_k^2 = \sum_{k=1}^r \frac{Y_k^2}{\lambda_k} = Y \cdot S^+ Y = (U^t X) \cdot S^+ (U^t X) = X \cdot (US^+U^t)X = X \cdot Q^+ X.$$

This completes the proof of the theorem.



Let  $v$  be a unit vector, and let  $Q = I - v \otimes v$ . Suppose  $X$  is a normal random vector with mean zero and variance  $Q$ . Then

$$E((X \cdot v)^2) = v \cdot Q v = v \cdot (I - v \otimes v)v = v - (v \cdot v)v = 0,$$

so  $X \cdot v = 0$ .

It is easy to check  $Q^3 = Q$  and  $Q^2$  is symmetric, so (§2.3)  $Q^+ = Q$ . Since  $X \cdot v = 0$ ,

$$X \cdot Q^+ X = X \cdot Q X = X \cdot (X - (v \cdot X)v) = |X|^2.$$

We conclude

### Singular Chi-squared

Let  $v$  be a unit vector, and let  $X$  be a normal random vector in  $\mathbf{R}^d$  with mean zero and variance  $I - v \otimes v$ . Then  $|X|^2$  is chi-squared with degree  $d - 1$ .



We use the above to derive the distribution of the sample variance. Let  $X_1, X_2, \dots, X_n$  be a random sample, and let  $\bar{X}$  be the sample mean,

$$\bar{X} = \frac{X_1 + X_2 + \cdots + X_n}{n}.$$

Let  $S^2$  be the *sample variance*,

$$S^2 = \frac{(X_1 - \bar{X})^2 + (X_2 - \bar{X})^2 + \cdots + (X_n - \bar{X})^2}{n - 1}. \quad (5.5.4)$$

Since  $(n - 1)S^2$  is a sum-of-squares similar to (5.5.2), we expect  $(n - 1)S^2$  to be chi-squared. In fact this is so, but the degree is  $n - 1$ , not  $n$ . We will show

### Independence of Sample Mean and Sample Variance

Let  $Z = (Z_1, Z_2, \dots, Z_n)$  be independent standard normal random variables, let  $\bar{Z}$  be the sample mean, and let  $S^2$  be the sample variance. Then  $(n - 1)S^2$  is chi-squared with degree  $n - 1$ , and  $\bar{Z}$  and  $S^2$  are independent.

To see this, let

$$v = \left( \frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}, \dots, \frac{1}{\sqrt{n}} \right).$$

Then  $v$  is a unit vector, and

$$Z \cdot v = \frac{1}{\sqrt{n}} \sum_{k=1}^n Z_k = \sqrt{n} \bar{Z}, \quad (Z \cdot v)v = (\bar{Z}, \dots, \bar{Z}).$$

Now let

$$X = Z - (Z \cdot v)v = (Z_1 - \bar{Z}, Z_2 - \bar{Z}, \dots, Z_n - \bar{Z}).$$

Then  $E(X) = 0$  and

$$E((Z \cdot v)^2) = E(v \cdot (Z \otimes Z)v) = v \cdot Iv = 1.$$

From this, it follows that

$$E(X \otimes X) = I - v \otimes v.$$

Hence

$$(n - 1)S^2 = |X|^2$$

is chi-squared with degree  $n - 1$ .

Now  $X$  and  $Z \cdot v$  are uncorrelated, since

$$E(X(Z \cdot v)) = E(Z(Z \cdot v)) - E((Z \cdot v)^2)v = v - v = 0.$$

Since they are normal,  $X$  and  $Z \cdot v$  are independent. Since  $(n - 1)S^2 = |X|^2$ , and  $\sqrt{n}\bar{Z} = Z \cdot v$ ,  $S^2$  and  $\bar{Z}$  are independent.



# Chapter 6

## Statistics

### 6.1 Estimation

In statistics, like any science, we start with a guess or an assumption or hypothesis, then we take a measurement, then we accept or modify our guess/assumption based on the result of the measurement. This is common sense, and applies to everything in life, not just statistics.

For example, suppose you see a sign on campus saying

*There is a lecture in room B120.*

How can you tell if this is true/correct or not? One approach is to go to room B120 and look. Either there is a lecture or there isn't. Problem solved.

But then someone might object, saying, wait, what if there is a lecture in room B120 tomorrow? To address this, you go every day to room B120 and check, for 100 days. You find out that in 85 of the 100 days, there is a lecture, and in 15 days, there is none. Based on this, you can say you are 85% confident there is a lecture there. Of course, you can never be sure, it depends on which day you checked, you can only provide a confidence level. Nevertheless, this kind of thinking allows us to quantify the probability that our hypothesis is correct.

In general, the measurement is significant if it is *unlikely*. When we obtain a significant measurement, then we are likely to reject our guess/assumption. So

$$\text{significance} = 1 - \text{confidence}.$$

In practice, our guess/assumption allows us to calculate a *p*-value, which is the probability that the measurement is *not* consistent with our assumption.

In the above scenario, the  $p$ -value is .15, determined by repeatedly sampling the room.

This is what statistics is about, summarized in Figure 6.1. The details may be more or less complicated depending on the problem situation or setup, but this is the central idea.

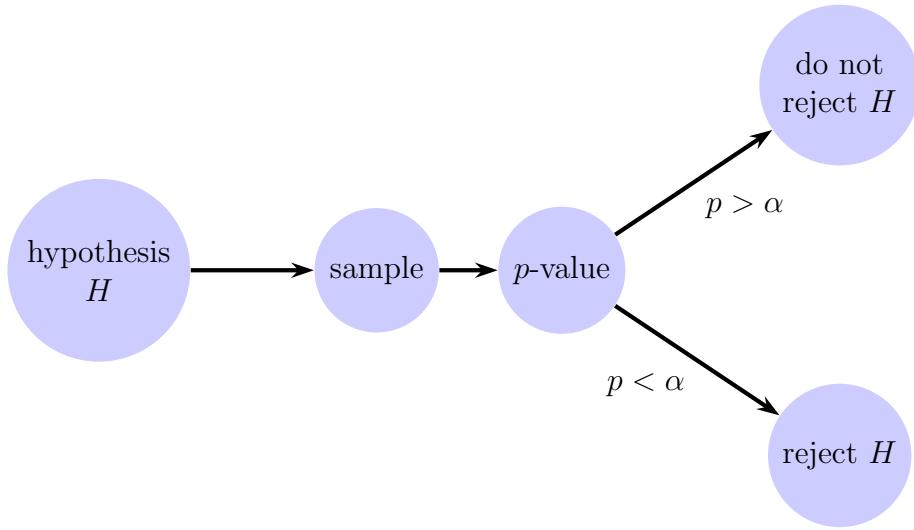


Figure 6.1: Statistics flowchart:  $p$ -value  $p$  and significance  $\alpha$ .



Here is a geometric example. The *null hypothesis* and the *alternate hypothesis* are

- $H_0$ : The angle between two randomly selected vectors in 784 dimensions is approximately  $90^\circ$
- $H_a$ : The angle between two randomly selected vectors in 784 dimensions is approximately  $60^\circ$ .

In §2.2, there is code (2.2) returning the angle `angle(u, v)` between two vectors. To test this hypothesis, we run the code

```
from numpy import *
from numpy.random import randn

# randn is standard normal

N = 784

for _ in range(20):
    u = randn(N)
    v = randn(N)
    print(angle(u,v))
```

to randomly select  $u, v$  twenty times. This code returns

```
86.27806537791886
87.91436653824776
93.00098725550777
92.73766421951748
90.005139015804
87.99643434444482
89.77813370637857
96.09801014394806
90.07032573539982
89.37679070400239
91.3405728939376
86.49851399221568
87.12755619082597
88.87980905998855
89.80377324818076
91.3006921339982
91.43977096117017
88.52516224405458
86.89606919838387
90.49100744167357
```

Here we see strong evidence supporting  $H_0$ . On the other hand, if we run the code

```
from numpy import *
from numpy.random import binomial

N = 784

n = 1 # one coin toss
#n = 3 # three coin tosses

for _ in range(20):
    u = binomial(n,.5,N)
    v = binomial(n,.5,N)
    print(angle(u,v))
```

to randomly select  $u, v$  twenty times, we obtain

```
59.43464627897324
59.14345748418916
60.31453922165891
60.38024365702492
59.24709660805488
59.27165957992343
61.21424657806321
60.55756381536082
61.59468919876665
61.33296028237481
60.03925473033243
60.25732069941224
61.77018692842784
60.672901794058326
59.628519516164666
59.41272458020638
58.43172340007064
59.863796136907744
59.45156367988921
59.95835532791699
```

Here we see strong evidence that  $H_0$  is false, as the angles are now close to  $60^\circ$ .



The difference between the two scenarios is the distribution. In the first scenario, we have `randn(n)`: the components are distributed according to a standard normal. In the second scenario, we have `binomial(1,.5,N)`: the components are distributed according to a fair coin toss. To see how the distribution affects things, we bring in the law of large numbers, which is discussed in §5.3.

Let  $X_1, X_2, \dots, X_n$  be a simple random sample from some population, and let  $\mu$  be the population mean. Recall this means  $X_1, X_2, \dots, X_n$  are i.i.d. random variables, with  $\mu = E(X)$ . The sample mean is

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_n}{n}.$$

Then we have the

### law of large numbers

For almost every simple random sample, for large sample size, the sample mean  $\bar{X}$  approximately equals the population mean  $\mu$ . More precisely, as the sample size  $n$  approaches infinity,  $\bar{X}$  approaches  $\mu$ .

We use the law of large numbers to explain the closeness of the vector angles to specific values.

Assume  $u = (x_1, x_2, \dots, x_n)$ , and  $v = (y_1, y_2, \dots, y_n)$  where all components are selected independently of each other, and each is selected according to the same distribution.

Let  $U = (X_1, X_2, \dots, X_n)$ ,  $V = (Y_1, Y_2, \dots, Y_n)$ , be the corresponding random variables. Then  $X_1, X_2, \dots, X_n$  and  $Y_1, Y_2, \dots, Y_n$  are independent and identically distributed (i.i.d.), with population mean  $E(X) = E(Y)$ .

From this,  $X_1Y_1, X_2Y_2, \dots, X_nY_n$  are i.i.d. random variables with population mean  $E(XY)$ . By the law of large numbers,<sup>1</sup>

$$\frac{X_1Y_1 + X_2Y_2 + \dots + X_nY_n}{n} \approx E(XY),$$

so

$$U \cdot V = X_1Y_1 + X_2Y_2 + \dots + X_nY_n \approx nE(XY).$$

---

<sup>1</sup>  $\approx$  means the ratio of the two sides approaches 1 as  $n$  grows without bound.

Similarly,  $U \cdot U \approx n E(X^2)$  and  $V \cdot V \approx n E(Y^2)$ . Hence (check that the  $n$ 's cancel)

$$\cos(U, V) = \frac{U \cdot V}{\sqrt{(U \cdot U)(V \cdot V)}} \approx \frac{E(XY)}{\sqrt{E(X^2)E(Y^2)}}.$$

Since  $X$  and  $Y$  are independent with mean  $\mu$  and variance  $\sigma^2$ ,

$$E(XY) = E(X)E(Y) = \mu^2, \quad E(X^2) = \mu^2 + \sigma^2, \quad E(Y^2) = \mu^2 + \sigma^2.$$

If  $\theta$  is the angle between  $U$  and  $V$ , we conclude

$$\cos(\theta) = \frac{U \cdot V}{\sqrt{(U \cdot U)(V \cdot V)}} \approx \frac{\mu^2}{\mu^2 + \sigma^2}.$$

When the distribution is  $N(0, 1)$ ,  $\mu = 0$ , so the angle is approximately  $90^\circ$ . When the distribution is bernoulli with parameter  $p$ ,

$$\frac{\mu^2}{\mu^2 + \sigma^2} = \frac{p^2}{p^2 + p(1-p)} = p.$$

For  $p = .5$ , this results in an angle of  $60^\circ$ .

The general result is

### Random Vectors in High Dimensions

Let  $U$  and  $V$  be two vectors selected randomly. Assume the components of  $U$  and  $V$  are independent and identically distributed with mean  $\mu$  and variance  $\sigma^2$ . Let  $\theta$  be the angle between them. When the vector dimension is high,

$$\cos(\theta) \quad \text{is approximately} \quad \frac{\mu^2}{\mu^2 + \sigma^2}.$$

## 6.2 Z-test

Suppose we want to estimate the proportion of American college students who have a smart phone. Instead of asking every student, we take a sample and make an estimate based on the sample.

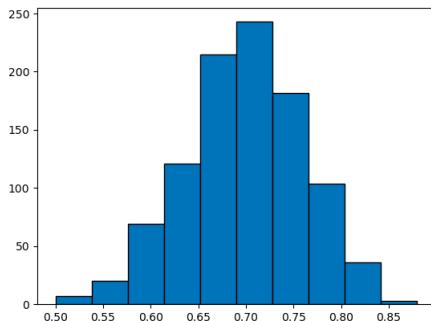


Figure 6.2: Histogram of sampling  $n = 25$  students, repeated  $N = 1000$  times.

The *population proportion*  $p$  is the actual proportion of students that in fact have a smart phone. Then  $0 < p < 1$ . Pick a student, and let

$$X = \begin{cases} 1, & \text{if the student has a smartphone,} \\ 0, & \text{if not.} \end{cases}$$

Then  $X$  is a bernoulli random variable with mean  $p$ .

For example, suppose the population proportion of students that have a smartphone is  $p = .7$ , and we sample  $n = 25$  students, obtaining a sample proportion  $\bar{X}$ . If we repeat the sampling  $N = 1000$  times, we will obtain 1000 values for  $X$ . Figure 6.2 displays the resulting histogram of  $\bar{X}$  values. Here is the code

```
from numpy import *
from matplotlib.pyplot import *
from numpy.random import binomial

p = .7
n = 25
N = 1000
v = binomial(n,p,N)/n

hist(v,edgecolor ='Black')
show()
```

Let  $X_1, X_2, \dots, X_n$  be a simple random sample of size  $n$ . This means  $n$  students were selected randomly and independently and whether or not they had smartphones was recorded in the variables  $X_1, X_2, \dots, X_n$ . Each of these variables equals one or zero with probability  $p$  or  $1 - p$ .

The sample mean (§5.3) is

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_n}{n} = \frac{1}{n} \sum_{k=1}^n X_k.$$

Because each  $X_k$  is 0 or 1, this is the *sample proportion* of the students in the sample that have smartphones. Like  $p$ ,  $\bar{X}$  is also between zero and one.

Because the samples vary, it is impossible to make absolute statements about the population. Instead, as we see below, the best we can do is make statements that come with a *confidence level*. Confidence levels are expressed as percentages, such as a 95% confidence level, or as a proportion, such as a .95 confidence level.

Often levels are expressed as *significance levels*. The *significance level* is the corresponding tail probability, so

$$\text{significance level} = 1 - \text{confidence level}.$$

A confidence level of zero indicates that we have *no faith at all* that selecting another sample will give similar results, while a confidence level of 1 indicates that we have *no doubt at all* that selecting another sample will give similar results.

When we say  $p$  is within  $\bar{X} \pm \epsilon$ , or

$$|p - \bar{X}| < \epsilon,$$

we call  $\epsilon$  the *margin of error*. The interval

$$(L, U) = (\bar{X} - \epsilon, \bar{X} + \epsilon)$$

is a *confidence interval*.

With the above setup, we have the population proportion  $p$ , and the four sample characteristics

- sample size  $n$
- sample proportion  $\bar{X}$ ,

- margin of error  $\epsilon$ ,
- confidence level  $\alpha$ .

Suppose we do not know  $p$ , but we know  $n$  and  $\bar{X}$ . We say the margin of error is  $\epsilon$ , at confidence level  $\alpha$ , if

$$\text{Prob}(|p - \bar{X}| < \epsilon) = \alpha.$$

Here are some natural questions:

1. Given a sample of size  $n = 20$  and sample proportion  $\bar{X} = .7$ , what can we say about the margin of error  $\epsilon$  with confidence  $\alpha = .95$ ?
2. Given a sample proportion  $\bar{X} = .7$ , what sample size  $n$  should we take to obtain a margin of error  $\epsilon = .15$  with confidence  $\alpha = .95$ ?
3. Given a sample proportion  $\bar{X} = .7$ , what sample size  $n$  should we take to obtain a margin of error  $\epsilon = .15$  with confidence  $\alpha = .99$ ?
4. Given a sample of size  $n = 20$  and sample proportion  $\bar{X} = .7$ , with what confidence level  $\alpha$  is the margin of error  $\epsilon = .1$ ?

The answers are at the end of the section.



Suppose each  $X_k$  in the sample  $X_1, X_2, \dots, X_n$  has mean  $\mu$  and standard deviation  $\sigma$ . From §5.3, we know the mean and standard deviation of  $\bar{X}$  are  $\mu$  and  $\sigma/\sqrt{n}$ . In particular, when  $X_1, X_2, \dots, X_n$  is a bernoulli sample, the mean and variance of the sample proportion  $\bar{X}$  are  $p$  and  $p(1-p)/n$ .

Therefore, the mean and variance of the standardized random variable

$$Z = \sqrt{n} \frac{\bar{X} - p}{\sqrt{p(1-p)}}$$

are zero and one.

Returning to our smartphone question, how close is the sample mean  $\bar{X}$  to the population mean  $E(X) = p$ ? Remember, both  $\bar{X}$  and  $p$  are between 0 and 1. More specifically, given a *margin of error*  $\epsilon$ , we want to compute the confidence level

$$\text{Prob}(|\bar{X} - p| < \epsilon).$$

This corresponds to the confidence interval

$$L, U = \bar{X} - \epsilon, \bar{X} + \epsilon.$$

The key result is the central limit theorem (§5.3):  $Z$  is approximately normal. How large should the sample size  $n$  be in order to apply the central limit theorem? When we have *success-failure condition*

$$np \geq 10, \quad n(1-p) \geq 10.$$

For example,  $p = .7$  and  $n = 50$  satisfies the success-failure condition.

Let  $\alpha$  be the two-tail significance level, say  $\alpha = .05$ . Assuming  $Z$  is exactly normal, let  $z^*$  be the  $z$ -score corresponding to significance  $\alpha$ ,

$$\text{Prob}(|Z| > z^*) = \alpha.$$

Let  $\sigma/\sqrt{n}$  be the standard error. By the central limit theorem,

$$\alpha \approx \text{Prob}\left(\frac{|\bar{X} - p|}{\sqrt{p(1-p)}} > \frac{z^*}{\sqrt{n}}\right).$$

To compute the confidence interval  $(L, U)$ , we solve

$$\frac{|\bar{X} - p|}{\sqrt{p(1-p)}} = \frac{z^*}{\sqrt{n}} \tag{6.2.1}$$

for  $p$ . But (6.2.1) may be rewritten as a quadratic equation in  $p$ , leading to the approximate solution

$$L, U = \bar{X} \pm \epsilon = \bar{X} \pm \frac{z^*}{\sqrt{n}} \cdot \sqrt{\bar{X}(1-\bar{X})}.$$

From here we obtain the margin of error

$$\epsilon = \frac{z^*}{\sqrt{n}} \cdot \sqrt{\bar{X}(1-\bar{X})}.$$



More generally, let  $z^*$  be the  $z$ -score corresponding to significance level  $\alpha$ , so

```

zstar = Z.ppf(alpha)      # lower-tail, zstar < 0
zstar = Z.ppf(1-alpha)    # upper-tail, zstar > 0
zstar = Z.ppf(1-alpha/2)  # two-tail,   zstar > 0

```

Given a population with known standard deviation  $\sigma$ , sample size  $n$ , and sample mean  $\bar{X}$ , the *margin of error* is

$$\epsilon = z^* \cdot \frac{\sigma}{\sqrt{n}},$$

and the intervals

$$(L, U) = \begin{cases} (\bar{X} - \epsilon, \bar{X}), & \text{lower-tail,} \\ (\bar{X}, \bar{X} + \epsilon), & \text{upper-tail,} \\ (\bar{X} - \epsilon, \bar{X} + \epsilon), & \text{two-tail,} \end{cases}$$

are the *confidence intervals* at significance level  $\alpha$ . When not specified, a confidence interval is usually taken to be two-tail.

In the Python code below, instead of working with the standardized statistic  $Z$ , we work directly with the  $\bar{X}$ . When  $\sigma$  is not known, we have to replace the normal distribution by the  $t$  distribution (§6.3).

```

#####
# Confidence Interval - Z
#####

from numpy import *
from scipy.stats import norm as Z

# significance level alpha

def confidence_interval(xbar,sdev,n,alpha,type):
    Xbar = Z(xbar,sdev/sqrt(n))
    if type == "two-tail":
        U = Xbar.ppf(1-alpha/2)
        L = Xbar.ppf(alpha/2)
    elif type == "upper-tail":
        U = Xbar.ppf(1-alpha)
        L = xbar

```

```

    elif type == "lower-tail":
        L = Xbar.ppf(alpha)
        U = xbar
    else: print("what's the test type?"); return
    return L, U

# when X is not bernoulli 0,1,
# Z-test assumes sdev is known!!!
# when X is bernoulli, sdev = sqrt(xbar*(1-xbar))

alpha = .02
sdev = 228
n = 35
xbar = 95

L, U = confidence_interval(xbar,sdev,n,alpha,type)

print("type: ", type)
print("significance, sdev, n, xbar: ", alpha,sdev,n,xbar)
print("lower, upper: ",L, U)

```

Now we can answer the questions posed at the start of the section. Here are the answers.

1. When  $n = 20$ ,  $\alpha = .95$ , and  $\bar{X} = .7$ , we have  $[L, U] = [.5, .9]$ , so  $\epsilon = .2$ .
2. When  $\bar{X} = .7$ ,  $\alpha = .95$ , and  $\epsilon = .15$ , we run `confidence_interval` for  $15 \leq n \leq 40$ , and select the least  $n$  for which  $\epsilon < .15$ . We obtain  $n = 36$ .
3. When  $\bar{X} = .7$ ,  $\alpha = .99$ , and  $\epsilon = .15$ , we run `confidence_interval` for  $1 \leq n \leq 100$ , and select the least  $n$  for which  $\epsilon < .15$ . We obtain  $n = 62$ .
4. When  $\bar{X} = .7$ ,  $n = 20$ , and  $\epsilon = .1$ , we have

$$z^* = \frac{\epsilon\sqrt{n}}{\sigma} = .976.$$

Since  $Prob(Z > z^*) = .165$ , the confidence level is  $1 - 2 * .165 = .68$  or 68%.



The speed limit on a highway is  $\mu_0 = 120$ . Ten automatic speed cameras are installed along a stretch of the highway to measure passing vehicles speeds. Because the cameras aren't perfect, the average speed  $\bar{X}$  measured by the cameras may not equal a vehicle's true speed  $\mu$ . As a consequence, some drivers who were driving at the speed limit may be fined. These drivers are *false positives*.

Suppose the distribution of a vehicle's measured speed is normal with standard deviation 2. What measured speed cutoff  $\mu^*$  should the authorities use to keep false positives below 1%? Here we are asked for the upper-tail confidence interval  $(L, U) = (\mu_0, \mu^*)$  at significance level .01. A driver will be fined if their average measured speed  $\bar{X}$  is higher than  $\mu^*$ .

Using the above code, the cutoff  $\mu^*$  equals 121.47.



One use of confidence intervals is *hypothesis testing*. Here we have two hypotheses, a null hypothesis and an alternate hypothesis. In the above setting where we are estimating a population parameter  $\mu$ , the *null hypothesis* is that  $\mu$  equals a certain value  $\mu_0$ , and the *alternate hypothesis* is that  $\mu$  is not equal to  $\mu_0$ . Hypothesis testing is of three types, depending on the alternate hypothesis:  $\mu \neq \mu_0$ ,  $\mu > \mu_0$ ,  $\mu < \mu_0$ . These are two-tail, lower-tail, and upper-tail hypotheses.

- $H_0: \mu = \mu_0$
- $H_a: \mu \neq \mu_0$  or  $\mu < \mu_0$  or  $\mu > \mu_0$ .

For example, going back to our smartphone setup, if we sample  $n = 20$  students, obtaining a mean  $\bar{x} = .7$ , then  $\sigma = \sqrt{\bar{x}(1 - \bar{x})} = .46$ , and the two-tail 5% confidence interval is then [.5, .9]. If  $\mu_0$  lies outside the confidence interval, we reject  $H_0$  and accept  $H_a$ , at the 5% level. Otherwise, if  $\mu_0$  lies within the interval, we do not reject  $H_0$ .

Suppose 35 people are randomly selected and the accuracy of their wrist-watches is checked, with positive errors representing watches that are ahead of the correct time and negative errors representing watches that are behind the correct time. The sample has a mean of 95 seconds and a population

standard deviation of 228 seconds. At the 2% significance, can we claim the population mean is  $\mu_0 = 0$ ?

Here

- $H_0: \mu = 0$
- $H_a: \mu \neq 0.$

Here the significance level is  $\alpha = .02$  and  $\mu_0 = 0$ . To decide whether to reject  $H_0$  or not, compute the *standardized test statistic*

$$z = \sqrt{n} \cdot \frac{\bar{x} - \mu_0}{\sigma} = 2.465.$$

Since  $z$  is a sample from an approximately normal distribution  $Z$ , the  $p$ -value

$$p = Prob(|Z| > z) = .0137.$$

On the other hand, the  $z$ -score corresponding to the requested significance level is  $z^* = 2.326$ , since

$$Prob(|Z| > 2.326) = .02.$$

Since  $p$  is less than  $\alpha$ , or equivalently, since  $|z| > z^*$ , we reject  $H_0$ . In other words, when the  $p$ -value is smaller than the significance level, it is *more* significant, and we reject  $H_0$ .

Equivalently, the 98% confidence interval is

$$(\bar{x} - \epsilon, \bar{x} + \epsilon) = (5.3, 184.6).$$

Since  $\mu_0 = 0$  is outside this interval, we reject  $H_0$ .



### Hypothesis Testing

There are three types of alternative hypotheses  $H_a$ :

$$\mu < \mu_0, \quad \mu > \mu_0, \quad \mu \neq \mu_0.$$

These are lower-tail, upper-tail, and two-tail tests. In every case, we

have a sample of size  $n$ , a statistic  $\bar{x}$ , a standard deviation  $\sigma$ , a standardized statistic

$$z = \sqrt{n} \cdot \frac{\bar{x} - \mu_0}{\sigma},$$

a significance level  $\alpha$ , the  $p$ -value

$$p = \text{Prob}(Z < z), \quad p = \text{Prob}(Z > z), \quad p = \text{Prob}(|Z| > z),$$

and the critical cutoff  $z^*$ ,

$$\text{Prob}(Z < z^*) = \alpha, \quad \text{Prob}(Z > z^*) = \alpha, \quad \text{Prob}(|Z| > z^*) = \alpha.$$

Then we reject  $H_0$  whenever  $z$  is more significant than  $z^*$ , which is the same as saying whenever the  $p$ -value  $p$  is less than the significance level  $\alpha$ .



In the Python code below, instead of working with the standardized statistic  $Z$ , we work directly with  $\bar{X}$ , which is normally distributed with mean  $\mu_0$  and standard deviation  $\sigma/\sqrt{n}$ .

```
#####
# Hypothesis Z-test
#####

from numpy import *
from scipy.stats import norm as Z

# significance level alpha

def ztest(mu0, sdev, n, xbar, type):
    Xbar = Z(mu0, sdev/sqrt(n))
    print("mu0, sdev, n, xbar: ", mu0, sdev, n, xbar)
    if type == "lower-tail": p = Xbar.cdf(xbar)
    elif type == "upper-tail": p = 1 - Xbar.cdf(xbar)
    elif type == "two-tail": p = 2 * (1 - Xbar.cdf(abs(xbar)))
    print("type: ", type)
```

```

print("pvalue: ",p)
if p < alpha: print("reject H0")
else: print("do not reject H0")

xbar = 122
n = 10
type = "upper-tail"
mu0 = 120
sdev = 2
alpha = .01

ztest(mu0, sdev, n, xbar,type)

```

Going back to the driving speed example, the hypothesis test is

- $H_0: \mu = \mu_0$
- $H_a: \mu > \mu_0$

If a driver's measured average speed is  $\bar{X} = 122$ , the above code rejects  $H_0$ . This is consistent with the confidence interval cutoff we found above.



There are two types of possible errors we can make. a *Type I error* is when  $H_0$  is true, but we reject it, and a *Type II error* is when  $H_0$  is not true but we fail to reject it.

	$H_0$ is true	$H_0$ is false
do not reject $H_0$	$1 - \alpha$	Type II error: $\beta$
reject $H_0$	Type I error: $\alpha$	Power: $1 - \beta$

Table 6.3: The error matrix.

We reject  $H_0$  when the  $p$ -value of  $Z$  is less than the significance level  $\alpha$ , which happens when  $z < z^*$  or  $z > z^*$  or  $|z| > z^*$ . In all cases, the chance of this happening is by definition  $\alpha$ . In other words,

$$\text{Prob}(\text{Type I error}) = \text{Prob}(p\text{-value} < \alpha \mid H_0) = \alpha.$$

Thus *the probability of a type I error is the significance level  $\alpha$ .*

We make a Type II error when we do not reject  $H_0$ , but  $H_0$  is false. To compute the probability of a Type II error, suppose the true value of  $\mu$  is  $\mu_1$ . Then we do not reject  $H_0$  if  $|z| < |z^*|$ , which is when  $\mu_0$  lies in the confidence interval  $\bar{x} \pm z^* \sigma / \sqrt{n}$ , or when  $\bar{x}$  lies in the interval

$$\mu_0 - \frac{z^* \sigma}{\sqrt{n}} < \bar{x} < \mu_0 + \frac{z^* \sigma}{\sqrt{n}}.$$

But when  $\mu = \mu_1$ ,  $\bar{X}$  is  $N(\mu_1, \sigma)$ , so the probability of this event can be computed.

Standardize  $\bar{X}$  by subtracting  $\mu_1$  and dividing by the standard error. Then we have a Type II error when

$$\sqrt{n} \frac{(\mu_0 - \mu_1)}{\sigma} - z^* < z < \sqrt{n} \frac{(\mu_0 - \mu_1)}{\sigma} + z^*.$$

If we set  $\delta$  to equal the standardized difference in the means,

$$\delta = \sqrt{n} \frac{(\mu_0 - \mu_1)}{\sigma},$$

then we have a Type II error when

$$\delta - z^* < Z < \delta + z^*,$$

or when  $|Z - \delta| < z^*$ . Hence

$$Prob(\text{Type II error}) = Prob(|Z - \delta| < z^*).$$

This calculation was for a two-tail test. When the test is upper-tail or lower-tail, a similar calculation leads to the code

```
#####
# Type1 and Type2 errors - Z
#####

from numpy import *
from scipy.stats import norm as Z
```

```

def type2_error(type,mu0,mu1,sdev,n,alpha):
    print("significance,mu0,mu1, sdev, n: ",
        ↪ alpha,mu0,mu1,sdev,n)
    print("prob of type1 error: ", alpha)
    delta = sqrt(n) * (mu0 - mu1) / sdev
    if type == "lower-tail":
        zstar = Z.ppf(alpha)
        type2 = 1 - Z.cdf(delta + zstar)
    elif type == "upper-tail":
        zstar = Z.ppf(1-alpha)
        type2 = Z.cdf(delta + zstar)
    elif type == "two-tail":
        zstar = Z.ppf(1 - alpha/2)
        type2 = Z.cdf(delta + zstar) - Z.cdf(delta - zstar)
    else: print("what's the test type?"); return
    print("test type: ",type)
    print("zstar: ", zstar)
    print("delta: ", delta)
    print("prob of type2 error: ", type2)
    print("power: ", 1 - type2)

mu0 = 120
mu1 = 122
sdev = 2
n = 10
alpha = .01
type = "upper-tail"

type2_error(type,mu0,mu1,sdev,n,alpha)

```



A type II error is when we do not reject the null hypothesis and yet it's false. The *power* of a test is the probability of rejecting the null hypothesis when it's false (Figure 6.3). If the probability of a type II error is  $\beta$ , then the power is  $1 - \beta$ .

Going back to the driving speed example, what is the chance that someone driving at  $\mu_1 = 122$  is not caught? This is a type II error; using the above

code, the probability is

$$\beta = \text{Prob}(\bar{X} = 120 \mid \mu = 122) = 20\%.$$

Therefore this test has power 80% to detect such a driver.

### 6.3 T-test

Let  $X_1, X_2, \dots, X_n$  be a simple random sample from a population. We repeat the previous section when we know neither the population mean  $\mu$ , nor the population variance  $\sigma^2$ . We only know the sample mean

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_n}{n}$$

and the sample variance

$$S^2 = \frac{1}{n-1} \sum_{k=1}^n (X_k - \bar{X})^2.$$

For example, assume  $X_1, X_2, \dots, X_n$  are bernoulli 0,1 random variables. Then as we've seen before,

$$(n-1)S^2 = \sum_{k=1}^n (X_k - \bar{X})^2 = n\bar{X}(1-\bar{X}).$$

When the sample  $Z_1, Z_2, \dots, Z_n$  is normal, we have

$$\bar{Z} = \frac{Z_1 + Z_2 + \dots + Z_n}{n}$$

and

$$S^2 = \frac{1}{n-1} \sum_{k=1}^n (Z_k - \bar{Z})^2.$$

In this case, from §5.5,

- $(n-1)S^2$  is chi-squared of degree  $n-1$ , and
- $\bar{X}$  and  $S^2$  are independent.



A random variable  $T$  has a *t-distribution with degree d* if the probability that  $T$  lies in a small interval  $[a, b]$  containing  $t$  is

$$\frac{Prob(a < T < b)}{b - a} = \frac{1}{N} \cdot \left(1 + \frac{t^2}{d}\right)^{-(d+1)/2}, \quad a < t < b. \quad (6.3.1)$$

Here  $N$  is a constant to make the total area under the graph equal to one (Figure 6.4). In other words, (6.3.1) is the **pdf** of the *t*-distribution.

When the interval  $[a, b]$  is not small, the correct formula is obtained by integration, which means dividing  $[a, b]$  into many small intervals and summing. We will not use this density formula directly.

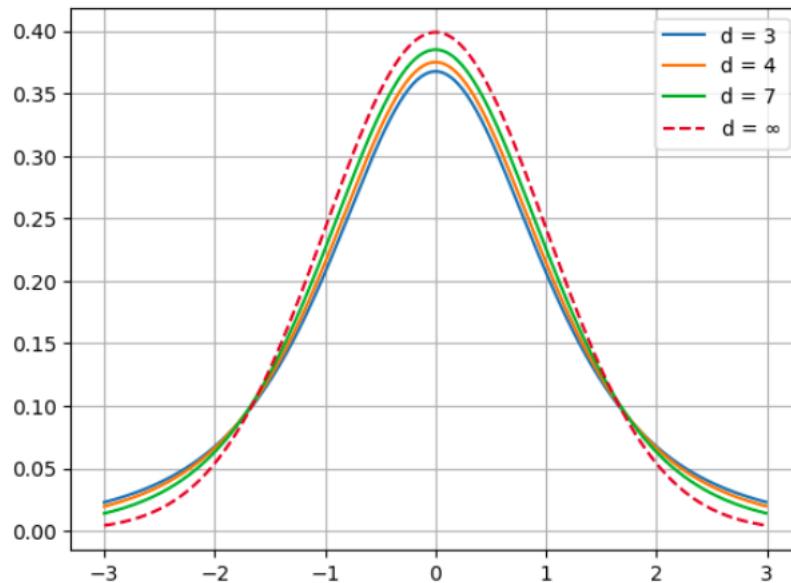


Figure 6.4: *t*-distribution, against normal (dashed).

By the compound-interest formula for the exponential (4.4.6), the *t*-distribution (6.3.1) approaches the standard normal distribution (5.4.1) as  $d \rightarrow \infty$ .

```

from numpy import *
from scipy.stats import t as T, norm as Z
from matplotlib.pyplot import *

for d in [3,4,7]:
    t = arange(-3,3,.01)
    plot(t,T(d).pdf(t),label="d = "+str(d))

plot(t,Z.pdf(t),"--",label=r"d = $\infty$")
grid()
legend()
show()

```

Using calculus, one can derive

### Relation Between $Z$ , $U$ , and $T$

Suppose  $Z$  and  $U$  are independent, where  $Z$  is standard normal, and  $U$  is chi-squared with  $d$  degrees of freedom. Then

$$T = \frac{Z}{\sqrt{U/d}}$$

is a  $t$ -distribution with degree  $d$ .



In the previous section, we normalized a sample mean by subtracting the mean  $\mu$  and dividing by the standard error  $\sigma/\sqrt{n}$ . Since now we don't know  $\sigma$ , it is reasonable to divide by the sample standard error, obtaining

$$\sqrt{n} \cdot \frac{\bar{X} - \mu}{S} = \sqrt{n} \cdot \frac{\bar{X} - \mu}{\sqrt{\frac{1}{n-1} \sum_{k=1}^n (X_k - \bar{X})^2}}.$$

If we standardize each variable by

$$X_k = \mu + \sigma Z_k,$$

then we can verify

$$\bar{X} = \mu + \sigma \bar{Z}, \quad \bar{Z} = \frac{Z_1 + Z_2 + \cdots + Z_n}{n},$$

and

$$S^2 = \sigma^2 \frac{1}{n-1} \sum_{k=1}^n (Z_k - \bar{Z})^2.$$

From this, we have

$$\sqrt{n} \cdot \frac{\bar{X} - \mu}{S} = \sqrt{n} \cdot \frac{\bar{Z}}{\sqrt{\frac{1}{n-1} \sum_{k=1}^n (Z_k - \bar{Z})^2}} = \sqrt{n} \cdot \frac{\bar{Z}}{\sqrt{U/(n-1)}}.$$

Using the last result with  $d = n - 1$ , we arrive at the main result in this section.

### Samples and $T$ Distributions

Let  $X_1, X_2, \dots, X_n$  be independent normal random variables with mean  $\mu$ . Let  $\bar{X}$  be the sample mean, let  $S^2$  be the sample variance, and let

$$T = \sqrt{n} \cdot \frac{\bar{X} - \mu}{S}.$$

Then  $T$  is distributed according to a  $t$ -distribution with degree  $(n - 1)$ .

The takeaway here is we do not need to know the standard deviations  $\sigma$  of  $X_1, X_2, \dots, X_n$  to compute  $T$ .



The  $t$ -score  $t^*$  corresponding<sup>2</sup> to significance  $\alpha$  is

```
tstar = T(d).ppf(alpha)      # lower-tail, tstar < 0
tstar = T(d).ppf(1-alpha)    # upper-tail, tstar > 0
```

---

<sup>2</sup>Geometrically,  $Prob(T > 1)$  is the probability that a normally distributed point is inside the light cone in  $(d + 1)$ -dimensional spacetime.

```
tstar = T(d).ppf(1-alpha/2) # two-tail, tstar > 0
```

Here  $d$  is the degree of  $T$ . Then we have

```
#####
# Confidence Interval - T
#####

from numpy import *
from scipy.stats import t as T

def confidence_interval(xbar,s,n,alpha,type):
    d = n-1
    if type == "two-tail":
        tstar = T(d).ppf(1-alpha/2)
        L = xbar - tstar * s / sqrt(n)
        U = xbar + tstar * s / sqrt(n)
    elif type == "upper-tail":
        tstar = T(d).ppf(1-alpha)
        L = xbar
        U = xbar + tstar* s / sqrt(n)
    elif type == "lower-tail":
        tstar = T(d).ppf(alpha)
        L = xbar + tstar* s / sqrt(n)
        U = xbar
    else: print("what's the test type?"); return
    print("type: ",type)
    return L, U

n = 10
xbar = 120
s = 2
alpha = .01
type = "upper-tail"
print("significance, s, n, xbar: ", alpha,s,n,xbar)

L,U = confidence_interval(xbar,s,n,alpha,type)
print("lower, upper: ", L,U)
```

Going back to the driving speed example from §6.2, instead of assuming the population standard deviation is  $\sigma = 2$ , we compute the sample standard deviation and find it's  $S = 2$ . Recomputing with  $T(9)$ , instead of  $Z$ , we see  $(L, U) = (120, 121.78)$ , so the cutoff now is  $\mu^* = 121.78$ , as opposed to  $\mu^* = 121.47$  there.



We turn now to *hypothesis testing*. As before, we have two hypotheses, a null hypothesis and an alternate hypothesis. In the above setting where we are estimating a population parameter, the *null hypothesis* is that  $\mu$  equals a certain value  $\mu_0$ , and the *alternate hypothesis* is that  $\mu$  is not equal to  $\mu_0$ .

- $H_0: \mu = \mu_0$
- $H_a: \mu \neq \mu_0$ .

Here is the code:

```
#####
# Hypothesis T-test
#####

from numpy import *
from scipy.stats import t as T

def ttest(mu0, s, n, xbar,type):
    d = n-1
    print("mu0, s, n, xbar: ", mu0,s,n,xbar)
    t = sqrt(n) * (xbar - mu0) / s
    print("t: ",t)
    if type == "lower-tail": p = T(d).cdf(t)
    elif type == "upper-tail": p = 1 - T(d).cdf(t)
    elif type == "two-tail": p = 2 * (1 - T(d).cdf(abs(t)))
    print("pvalue: ",p)
    if p < alpha: print("reject H0")
    else: print("do not reject H0")

xbar = 122
```

```

n = 10
type = "upper-tail"
mu0 = 120
s = 2
alpha = .01

ttest(mu0, s, n, xbar,type)

```

Going back to the driving speed example, the hypothesis test is

- $H_0: \mu = \mu_0$
- $H_a: \mu > \mu_0$

If a driver's measured average speed is  $\bar{X} = 122$ , the above code rejects  $H_0$ . This is consistent with the confidence interval cutoff we found above. However, the  $p$ -value obtained here is greater than the corresponding  $p$ -value in §6.2.



For Type I and Type II errors, the code is

```

#####
# Type1 and Type2 errors
#####

from numpy import *
from scipy.stats import t as T

def type2_error(type,mu0,mu1,n,alpha):
    d = n-1
    print("significance,mu0,mu1,n: ", alpha,mu0,mu1,n)
    print("prob of type1 error: ", alpha)
    delta = sqrt(n) * (mu0 - mu1) / sdev
    if type == "lower-tail":
        tstar = T(d).ppf(alpha)
        type2 = 1 - T(d).cdf(delta + tstar)
    elif type == "upper-tail":

```

```

tstar = T(d).ppf(1-alpha)
type2 = T(d).cdf(delta + tstar)
elif type == "two-tail":
    tstar = T(d).ppf(1 - alpha/2)
    type2 = T(d).cdf(delta + tstar) - T(d).cdf(delta -
→ tstar)
else: print("what's the test type?"); return

print("test type: ", type)
print("tstar: ", tstar)
print("delta: ", delta)

print("prob of type2 error: ", type2)
print("power: ", 1 - type2)

type2_error(type,mu0,mu1,n,alpha)

```

Going back to the driving speed example, if a driver's measured average speed is  $\bar{X} = 122$ , what is the chance they will not be fined? From the code, the probability of this Type II error is 37%, and the power to detect such a driver is 63%.

## 6.4 Two Means

Let  $X_1, X_2, \dots, X_n$  be a simple random sample from a population. Let  $Y_1, Y_2, \dots, Y_m$  be another simple random sample, and assume the two samples are independent. Assume also that each  $X_k$  is  $N(\mu_X, \sigma)$ , and each  $Y_k$  is  $N(\mu_Y, \sigma)$ . *The goal is to estimate  $\mu_X - \mu_Y$ .*

As before, let

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_n}{n}, \quad \bar{Y} = \frac{Y_1 + Y_2 + \dots + Y_m}{m},$$

and let

$$S_X^2 = \frac{1}{n-1} \sum_{k=1}^n (X_k - \bar{X})^2, \quad S_Y^2 = \frac{1}{m-1} \sum_{k=1}^m (Y_k - \bar{Y})^2.$$

Then  $S_X^2$  and  $S_Y^2$  are unbiased estimators of  $\sigma_X^2$  and  $\sigma_Y^2$ , which means

$$E(S_X^2) = \sigma_X^2, \quad E(S_Y^2) = \sigma_Y^2.$$

From before, we know

$$\frac{(n-1)S_X^2}{\sigma_X^2}$$

is chi-squared with degree  $n-1$ . Since the variance of chi-squared with degree  $r$  equals  $2r$ , the variance of  $(n-1)S_X^2/\sigma_X^2$  equals  $2(n-1)$ . Thus

$$Var(S_X^2) = \frac{\sigma_X^4}{(n-1)^2} Var\left(\frac{(n-1)S_X^2}{\sigma_X^2}\right) = \frac{\sigma_X^4}{(n-1)^2} \cdot 2(n-1) = \frac{2\sigma_X^4}{n-1}.$$

Similarly,

$$Var(S_Y^2) = \frac{2\sigma_Y^4}{m-1}.$$

Before, with a single mean, we used the result that

$$T = \frac{Z}{\sqrt{U/n}}$$

is a  $t$ -distribution with  $n$  degrees of freedom when

1.  $Z$  and  $U$  are independent
2.  $Z$  is  $N(0, 1)$
3.  $U$  is chi-squared with  $n$  degrees of freedom.

We apply this same result this time, but we proceed more carefully. To begin,  $\bar{X}$  and  $\bar{Y}$  are normal with means  $\mu_X$  and  $\mu_Y$  and variances  $\sigma^2/n$  and  $\sigma^2/m$  respectively. Hence

$$\frac{(\bar{X} - \bar{Y}) - (\mu_X - \mu_Y)}{\sqrt{\frac{\sigma^2}{n} + \frac{\sigma^2}{m}}} \sim N(0, 1).$$

Next,

$$(n-1)\frac{S_X^2}{\sigma^2} \quad \text{and} \quad (m-1)\frac{S_Y^2}{\sigma^2}$$

are chi-squared of degrees  $n-1$  and  $m-1$  respectively, so their sum

$$(n-1)\frac{S_X^2}{\sigma^2} + (m-1)\frac{S_Y^2}{\sigma^2}$$

is chi-squared with degree  $(n - 1) + (m - 1) = n + m - 2$ . If we let

$$S_p^2 = \frac{(n - 1)S_X^2 + (m - 1)S_Y^2}{n + m - 2}$$

be the *pooled sample variance*, then the above sum is  $(n + m - 2)S_p^2/\sigma^2$ . We conclude  $(n + m - 2)S_p^2/\sigma^2$  is chi-squared with degree  $n + m - 2$ .

By our main result above (the  $\sigma$ 's cancel),

$$T = \frac{\bar{X} - \bar{Y} - (\mu_X - \mu_Y)}{S_p \sqrt{\frac{1}{n} + \frac{1}{m}}}$$

is distributed according to a  $t$ -distribution with degree  $n + m - 2$ .

Based on this, let  $t_\alpha^*$  be the critical  $t$ -score of degree  $n + m - 2$  at significance  $\alpha$ . Then a confidence interval for  $\mu_X - \mu_Y$  at significance  $\alpha$  is

$$\bar{X} - \bar{Y} \pm S_p \cdot t_\alpha^* \cdot \sqrt{\frac{1}{n} + \frac{1}{m}}.$$

Here is code for computing confidence intervals for two means.

```
#####
# Confidence Interval - Two means
#####

import numpy as np
from scipy.stats import t

T = t

def confidence_interval(xbar,ybar,varx,vary,nx,ny,alpha):
    tstar = T.ppf(1-alpha/2, nx+ny-2)
    varp = (nx-1)*varx+(ny-1)*vary
    n = nx+ny-2
    varp = varp/n
    s_p = np.sqrt(varp)
    h = 1/nx + 1/ny
    L = xbar - ybar - tstar * s_p * np.sqrt(h)
    U = xbar - ybar + tstar * s_p * np.sqrt(h)
```

```
return L, U
```

Now we turn to the question of what to do when the variances  $\sigma_X^2$  and  $\sigma_Y^2$  are not equal. In this case, by independence, the population variance of  $\bar{X} - \bar{Y}$  is the sum of the population variances of  $\bar{X}$  and  $\bar{Y}$ , which is

$$\sigma_B^2 = \frac{\sigma_X^2}{n} + \frac{\sigma_Y^2}{m}. \quad (6.4.1)$$

Hence

$$\frac{(\bar{X} - \bar{Y}) - (\mu_X - \mu_Y)}{\sqrt{\frac{\sigma_X^2}{n} + \frac{\sigma_Y^2}{m}}} \sim N(0, 1).$$

We want to replace the population variance (6.4.1) by the sample variance

$$S_B^2 = \frac{S_X^2}{n} + \frac{S_Y^2}{m}.$$

Because  $S_B^2$  is not a straight sum, but is a more complicated linear combination of variances,  $S_B^2$  is not chi-squared.

Welch's approximation is to assume it is chi-squared with degree  $r$ , and to figure out the best  $r$  for this. More exactly, we seek the best choice of  $r$  so that

$$\frac{rS_B^2}{\sigma_B^2} = \frac{rS_B^2}{\frac{\sigma_X^2}{n} + \frac{\sigma_Y^2}{m}}$$

is close to chi-squared with degree  $r$ . By construction, we multiplied  $S_B^2$  by  $r/\sigma_B^2$  so that its mean equals  $r$ ,

$$E\left(\frac{rS_B^2}{\sigma_B^2}\right) = \frac{r}{\sigma_B^2} E(S_B^2) = r.$$

Since the variance of a chi-squared with degree  $r$  is  $2r$ , we compute the variance and set it equal to  $2r$ ,

$$2r = Var\left(\frac{rS_B^2}{\sigma_B^2}\right) = \frac{r^2}{(\sigma_B^2)^2} Var(S_B^2). \quad (6.4.2)$$

By independence,

$$Var(S_B^2) = Var\left(\frac{S_X^2}{n}\right) + Var\left(\frac{S_Y^2}{m}\right) = \frac{1}{n^2} Var(S_X^2) + \frac{1}{m^2} Var(S_Y^2).$$

But  $(n - 1)S_X^2/\sigma_X^2$  and  $(m - 1)S_Y^2/\sigma_Y^2$  are chi-squared, so

$$Var(S_B^2) = 2\frac{\sigma_X^4}{n^2(n-1)} + 2\frac{\sigma_Y^4}{m^2(m-1)}. \quad (6.4.3)$$

Combining (6.4.2) and (6.4.3), we arrive at Welch's approximation for the degrees of freedom,

$$r = \frac{\frac{\sigma_B^4}{\sigma_X^4}}{\frac{\sigma_X^4}{n^2(n-1)} + \frac{\sigma_Y^4}{m^2(m-1)}} = \frac{\left(\frac{\sigma_X^2}{n} + \frac{\sigma_Y^2}{m}\right)^2}{\frac{\sigma_X^4}{n^2(n-1)} + \frac{\sigma_Y^4}{m^2(m-1)}}.$$

In practice, this expression for  $r$  is never an integer, so one rounds it to the closest integer, and the population variances  $\sigma_X^2$  and  $\sigma_Y^2$  are replaced by the sample variances  $S_X^2$  and  $S_Y^2$ .

We summarize the results.

### Welch's T-statistic

If we have independent simple random samples, then the statistic

$$T = \frac{\bar{X} - \bar{Y} - (\mu_X - \mu_Y)}{\sqrt{\frac{S_X^2}{n} + \frac{S_Y^2}{m}}}$$

is approximately distributed according to a  $T$ -distribution with degrees of freedom

$$r = \frac{\left(\frac{S_X^2}{n} + \frac{S_Y^2}{m}\right)^2}{\frac{S_X^4}{n^2(n-1)} + \frac{S_Y^4}{m^2(m-1)}}.$$

## 6.5 Variances

Let  $X_1, X_2, \dots, X_n$  be a normally distributed simple random sample with mean 0 and variance 1.

Then we know

$$U = X_1^2 + X_2^2 + \cdots + X_n^2$$

is chi-squared with  $n$  degrees of freedom.

Throughout we let  $\chi_{\alpha,n}^2$  be the score corresponding to significance  $1 - \alpha$ ,

$$\text{Prob}(U \leq \chi_{\alpha,n}^2) = \alpha.$$

More generally, let  $X_1, X_2, \dots, X_n$  be a normally distributed simple random sample with mean  $\mu$  and variance  $\sigma^2$ , and let

$$S^2 = \frac{1}{n-1} \sum_{k=1}^n (X_k - \bar{X})^2$$

be the sample variance. Then

$$\frac{(n-1)S^2}{\sigma^2} \sim \chi_{n-1}^2.$$

Let

$$a = \chi_{\alpha/2,n-1}^2, \quad b = \chi_{1-\alpha/2,n-1}^2. \quad (6.5.1)$$

By definition of the score  $\chi_{\alpha,n}^2$ , we have

$$\text{Prob}\left(a \leq \frac{(n-1)S^2}{\sigma^2} \leq b\right) = 1 - \alpha.$$

From this, we get

$$\text{Prob}\left(\frac{(n-1)S^2}{b^2} \leq \sigma^2 \leq \frac{(n-1)S^2}{a^2}\right) = 1 - \alpha.$$

We conclude

### Confidence Interval

A  $(1 - \alpha)100\%$  confidence interval for the population variance  $\sigma^2$  is

$$\left(\frac{(n-1)S^2}{b^2} \leq \sigma^2 \leq \frac{(n-1)S^2}{a^2}\right)$$

where  $a$  and  $b$  are the  $\chi_{n-1}^2$  scores at significance  $1 - \alpha/2$  and  $\alpha/2$ .

```
#####
# Confidence Interval - Chi2
#####

from scipy.stats import chi2

def confidence_interval(s2,n,alpha):
    a = chi2.ppf(alpha/2,n-1)
    b = chi2.ppf(1-alpha/2,n-1)
    L = (n-1)*s2/b
    U = (n-1)*s2/a
    return L, U
```

Here is an example: A large candy manufacturer produces, packages and sells packs of candy targeted to weigh 52 grams. A quality control manager working for the company was concerned that the variation in the actual weights of the targeted 52-gram packs was larger than acceptable. That is, he was concerned that some packs weighed significantly less than 52-grams and some weighed significantly more than 52 grams. In an attempt to estimate  $\sigma^2$ , he took a random sample of  $n = 10$  packs off of the factory line. The random sample yielded a sample variance of 4.2 grams. Use the random sample to derive a 95% confidence interval for  $\sigma^2$ .

Here  $S^2 = 4.2$ ,  $n = 10$ , and  $\alpha = .05$ , resulting in

$$L, U = 1.99, 14.0$$

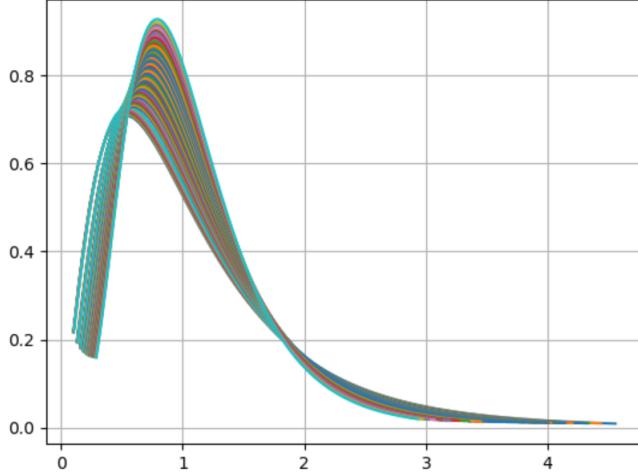
For hypothesis testing, given hypotheses

- $H_0: \sigma = \sigma_0$
- $H_a: \sigma \neq \sigma_0$

the standardized test statistic is

$$\frac{(n-1)S^2}{\sigma_0^2}.$$

and one compares the  $p$ -value of the standardized test statistic to the required significance score, whether two-tail, upper-tail, or lower-tail.

Figure 6.5: Fisher  $F$ -distribution.

Now we consider two populations with two variances. For this, we introduce the  $F$ -distribution. If  $U_1, U_2$  are independent chi-squared distributions with degrees  $n_1$  and  $n_2$ , then

$$F = \frac{U_1/n_1}{U_2/n_2}$$

is distributed according to an  $F$ -distribution with degrees  $(dfn, dfd) = (n_1, n_2)$ . The  $F$ -distribution for a range of degrees is shown in Figure 6.5. Here  $dfn$  and  $dfd$  stand for degrees of freedom for the numerator and denominator.

Let  $X_1, X_2, \dots, X_n$  be a simple random sample from a normally distributed population with mean  $\mu_X$  and variance  $\sigma_X^2$ . Similarly, let  $Y_1, Y_2, \dots, Y_m$  be a simple random sample from a normally distributed population with mean  $\mu_Y$  and variance  $\sigma_Y^2$ .

Then

$$\frac{(n-1)S_X^2}{\sigma_X^2}, \quad \text{and} \quad \frac{(m-1)S_Y^2}{\sigma_Y^2}$$

are independent chi-squared with degrees  $n$  and  $m$  respectively.

Hence

$$\frac{S_X^2}{S_Y^2} \cdot \frac{\sigma_Y^2}{\sigma_X^2}$$

is  $F$ -distributed with degrees  $(n, m)$ .

For example, suppose we sample two populations independently. Suppose the first sample size is 10, the second sample size is 5, and the first sample variance is  $\sigma_X^2 = 1.5$ , the second sample variance is  $\sigma_Y^2 = 2.3$ . What is a 95% confidence interval for  $\sigma_X/\sigma_Y$ ?

Let  $a_\alpha$  and  $b_\alpha$  be the critical  $f$ -scores corresponding to  $\alpha = .05$ , with degrees  $(dfn, dfd) = (10, 5)$ ,

```
from scipy.stats import f

alpha = .05
a = f.ppf(alpha/2, dfn, dfd)
b = f.ppf(1-alpha/2, dfn, dfd)
```

Then

$$\text{Prob} \left( a_\alpha < \frac{S_X^2}{S_Y^2} \frac{\sigma_Y^2}{\sigma_X^2} < b_\alpha \right) = 1 - \alpha,$$

which may be rewritten

$$\text{Prob} \left( \frac{1}{b_\alpha} \frac{S_X^2}{S_Y^2} < \frac{\sigma_X^2}{\sigma_Y^2} < \frac{1}{a_\alpha} \frac{S_X^2}{S_Y^2} \right) = 1 - \alpha.$$

Hence a  $100\%(1 - \alpha)$  confidence interval for  $\sigma_X/\sigma_Y$  is

$$L = \frac{1}{\sqrt{b_\alpha}} \frac{S_X}{S_Y}, \quad U = \frac{1}{\sqrt{a_\alpha}} \frac{S_X}{S_Y}.$$

Plugging in, we obtain,

$$L = 0.31389215230779993, \quad U = 1.6621265193149342$$

for the 95% confidence interval.

## 6.6 Maximum Likelihood Estimates

★ under construction ★

## 6.7 Chi-Squared Tests

Let  $X_1, X_2, \dots, X_n$  be i.i.d. random variables, where each  $X_k$  is *categorical*. This means each  $X_k$  is a discrete random variable taking values in one of  $d$  categories. For simplicity, assume the categories are

$$X_k = 0, 1, 2, \dots, d - 1.$$

When  $d = 2$ , this reduces to the bernoulli case  $X_k = 0, 1$ .

When  $d = 2$  and  $X_k = 0, 1$ , the sample mean  $\bar{X}$  is a proportion, the population mean is  $p = \text{Prob}(X_k = 1)$ , the population standard deviation is  $\sqrt{p(1-p)}$ , and the sample standard deviation is  $\sqrt{\bar{X}(1-\bar{X})}$ . By the central limit theorem, the test statistic

$$Z = \sqrt{n} \cdot \frac{\bar{X} - p}{\sqrt{p(1-p)}} \quad (6.7.1)$$

is approximately standard normal for large enough sample size, and consequently  $U = Z^2$  is approximately chi-squared with degree one. Pearson's test generalizes this from  $d = 2$  categories to  $d > 2$  categories.

Given a category  $j$ , let  $\#_j$  denote the number of times  $X_k = j$ ,  $1 \leq k \leq n$ . Then  $\#_j$  is the *count* that  $X_k = j$ , and  $\hat{p}_j = \#_j/n$  is the *observed frequency*, in  $n$  samples. Let  $p_j$  be the *expected frequency*,

$$p_j = \text{Prob}(X_k = j), \quad 0 \leq j < d$$

Since  $X_k$  are identically distributed, this does not depend on  $k$ .

By the central limit theorem,

$$\sqrt{n}(\hat{p}_j - p_j) = \sqrt{n} \left( \frac{\#_j}{n} - p_j \right), \quad 0 \leq j < d,$$

are approximately normal for large  $n$ . Based on this, Pearson [22] showed

### Goodness-Of-Fit Test

Let  $\hat{p} = (\hat{p}_1, \hat{p}_2, \dots, \hat{p}_d)$  be the observed frequencies and  $p =$

$(p_1, p_2, \dots, p_d)$  the expected frequencies. Then, for large  $n$ , the statistic

$$n \sum_{j=0}^{d-1} \frac{(\hat{p}_j - p_j)^2}{p_j} \quad (6.7.2)$$

is approximately chi-squared with degree  $d - 1$ .

By clearing denominators, (6.7.2) may be rewritten in terms of counts as follows,

$$\sum_{j=0}^{d-1} \frac{(\#_j - np_j)^2}{np_j} = \sum_{j=0}^{d-1} \frac{(observed - expected)^2}{expected}.$$

When  $d = 2$ , this statistic reduces to  $Z^2$ , where  $Z$  is given by (6.7.1). Here is the code.

```
from numpy import *
from scipy.stats import chi2 as U

def goodness_of_fit(observed, expected):
    # assume len(observed) == len(expected)
    d = len(observed)
    n = sum(observed)
    u = sum([(observed[i] - expected[i])**2/expected[i] for i
        ↪ in range(d)])
    deg = d-1
    pvalue = 1 - U(deg).cdf(u)
    return pvalue
```



Suppose a dice is rolled  $n = 120$  times, and the observed counts are

$$O_1 = 17, O_2 = 12, O_3 = 14, O_4 = 20, O_5 = 29, O_6 = 28.$$

Notice

$$O_1 + O_2 + O_3 + O_4 + O_5 + O_6 = 120.$$

If the dice is fair, the expected counts are

$$E_1 = 20, E_2 = 20, E_3 = 20, E_4 = 20, E_5 = 20, E_6 = 20.$$

Based on the observed counts, at 5% significance, what can we conclude about the dice?

Here there are  $d = 6$  categories, and  $\alpha = .05$ . The Pearson statistic (6.7.2) equals

$$u = 12.7$$

The dice is fair if  $u$  is not large and the dice is unfair if  $u$  is large. At significance level  $\alpha$ , the large/not-large cutoff  $u^*$  is

```
from scipy.stats import chi2 as U
d = 6
ustar = U(d-1).ppf(1-alpha)
```

Since this returns  $u^* = 11.07$  and  $u > u^*$ , we can conclude the dice is not fair.



We now derive the goodness-of-fit test. For each category  $0 \leq j < d$ , let

$$\tilde{X}_k^j = \begin{cases} \frac{1}{\sqrt{p_j}} & \text{if } X_k = j, \\ 0 & \text{if } X_k \neq j. \end{cases}$$

Then  $E(\tilde{X}_n^j) = \sqrt{p_j}$ , and

$$E(\tilde{X}_k^i \tilde{X}_k^j) = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

If

$$\mu = (\sqrt{p_1}, \sqrt{p_2}, \dots, \sqrt{p_d}) \quad \text{and} \quad \tilde{X}_k = (\tilde{X}_k^1, \tilde{X}_k^2, \dots, \tilde{X}_k^d),$$

then

$$E(\tilde{X}_k) = \mu, \quad E(\tilde{X}_k \otimes \tilde{X}_k) = I.$$

From this,

$$\text{Var}(\tilde{X}_k) = E(\tilde{X}_k \otimes \tilde{X}_k) - E(\tilde{X}_k) \otimes E(\tilde{X}_k) = I - \mu \otimes \mu.$$

From (5.3.8), we conclude the random vector

$$Z = \sqrt{n} \left( \frac{1}{n} \sum_{k=1}^n \tilde{X}_k - \mu \right)$$

has mean zero and variance  $I - \mu \otimes \mu$ . By the central limit theorem,  $Z$  is approximately normal for large  $n$ .

Since

$$|\mu|^2 = (\sqrt{p_0})^2 + (\sqrt{p_1})^2 + \cdots + (\sqrt{p_{d-1}})^2 = p_0 + p_1 + \cdots + p_{d-1} = 1,$$

$\mu$  is a unit vector. By the singular chi-squared result in §5.5,  $|Z|^2$  is approximately chi-squared with degree  $d - 1$ . Using

$$Z_j = \sqrt{n} \left( \frac{\hat{p}_j}{\sqrt{p_j}} - \sqrt{p_j} \right),$$

we write  $|Z|^2$  out,

$$|Z|^2 = \sum_{j=1}^d Z_j^2 = n \sum_{j=1}^d \left( \frac{\hat{p}_j}{\sqrt{p_j}} - \sqrt{p_j} \right)^2 = n \sum_{j=1}^d \frac{(\hat{p}_j - p_j)^2}{p_j},$$

obtaining (6.7.2).



Suppose  $X_1, X_2, \dots, X_n$  and  $Y_1, Y_2, \dots, Y_n$  are samples measuring two possibly related effects. Suppose the  $X$  variables take on  $d$  categories,  $X = 1, 2, \dots, d$ , and the  $Y$  variables take on  $e$  categories,  $Y = 1, 2, \dots, e$ . The goal is test whether the two effects are independent or not.

For example, suppose 300 people are polled and the results are collected in a *contingency table* (Figure 6.6).

	Democrat	Republican	Independent	Total
Women	68	56	32	156
Men	52	72	20	144
Total	120	128	52	300

Table 6.6: Contingency table [25].

Is a person's gender correlated with their party affiliation, or are the two variables independent? To answer this, we use the

### Chi-squared Independence Test

Let  $\hat{p} = (\hat{p}_1, \hat{p}_2, \dots, \hat{p}_d)$  be the observed frequencies corresponding to  $X_1, X_2, \dots, X_n$ , and let  $\hat{q} = (\hat{q}_1, \hat{q}_2, \dots, \hat{q}_e)$  be the observed frequencies corresponding to  $Y_1, Y_2, \dots, Y_n$ . Let  $\hat{r}_{ij}$  be the joint observed frequencies

$$\hat{r}_{ij} = \frac{\#\{k : X_k = i, Y_k = j\}}{n}, \quad i = 1, 2, \dots, d, j = 1, 2, \dots, e.$$

If  $X_1, X_2, \dots, X_n$  and  $Y_1, Y_2, \dots, Y_n$  are independent, then, for large  $n$ , the statistic

$$n \sum_{i,j=1}^{d,e} \frac{(\hat{r}_{ij} - \hat{p}_i \hat{q}_j)^2}{\hat{p}_i \hat{q}_j} \tag{6.7.3}$$

is approximately chi-squared with degree  $(d - 1)(e - 1)$ .

By clearing denominators, (6.7.3) may be rewritten in terms of counts as follows,

$$\begin{aligned} \sum_{i,j=1}^{d,e} \frac{(n \#_{ij}^{XY} - \#_i^X \#_j^Y)^2}{n \#_i^X \#_j^Y} &= -n + n \sum_{i,j=1}^{d,e} \frac{(\#_{ij}^{XY})^2}{\#_i^X \#_j^Y} \\ &= -n + n \sum_{i,j=1}^{d,e} \frac{(\text{observed})^2}{\text{expected}}. \end{aligned}$$



The code

```
def chi2_independence(table):
    observed = table
    n = sum(observed)
    d = len(observed)
    e = len(observed.T)
```

```
rowsum = array([ sum(observed[i,:]) for i in range(d) ])
colsum = array([ sum(observed[:,j]) for j in range(e) ])
expected = outer(rowsum,colsum)
u = -n + n*sum([[ observed[i,j]**2/expected[i,j] for j in
    ↪ range(e) ] for i in range(d) ])
deg = (d-1)*(e-1)
pvalue = 1 - U(deg).cdf(u)
return pvalue

table = array([[68,56,32],[52,72,20]])
chi2_independence(table)
```

returns a  $p$ -value of 0.0401, so, at the 5% significance level, the effects are not independent.



The independence test is Fisher's modification [7] of goodness-of-fit, and the derivation depends on maximum likelihood estimates.

# Chapter 7

## Calculus

### 7.1 Calculus

In this section, we focus on single-variable calculus, and in §7.3, we review multi-variable calculus. Recall the *slope* of a line  $y = mx + b$  equals  $m$ .

Let  $y = f(x)$  be a function as in Figure 7.1, and let  $a$  be a fixed point. The *derivative of  $f(x)$  at the point  $a$*  is the slope of the line tangent to the graph of  $f(x)$  at  $a$ . Then the derivative at a point  $a$  is a number  $f'(a)$  possibly depending on  $a$ .

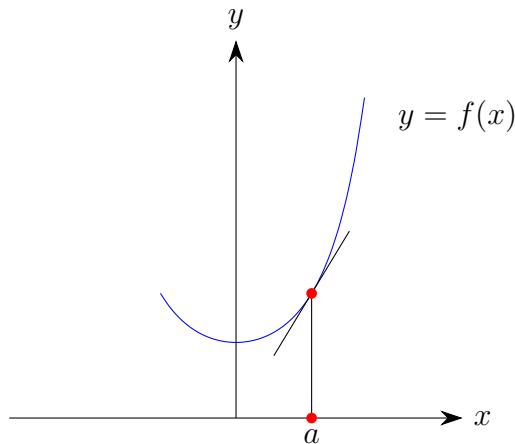


Figure 7.1:  $f'(a)$  is the slope of the tangent line at  $a$ .

Since the tangent line at  $a$  passes through the point  $(a, f(a))$ , and its

slope is  $f'(a)$ , the equation of the tangent line at  $a$  is

$$y = f(a) + f'(a)(x - a).$$

Based on the definition, natural properties of the derivative are

- A.** The derivative at  $x$  of  $f(x) - mx$  is  $f'(x) - m$ .
- B.** If  $f'(x) \geq 0$  on an interval  $[a, b]$ , then  $f(b) \geq f(a)$ .
- C.** If  $f'(x) \leq 0$  on an interval  $[a, b]$ , then  $f(b) \leq f(a)$ .

Using these properties, we determine the formula for  $f'(a)$ . Suppose the derivative is bounded between two extremes  $m$  and  $L$  at every point  $x$  in an interval  $[a, b]$ , say

$$m \leq f'(x) \leq L, \quad a \leq x \leq b.$$

Then by **A**, the derivative of  $h(x) = f(x) - mx$  at  $x$  equals  $h'(x) = f'(x) - m$ . By assumption,  $h'(x) \geq 0$  on  $[a, b]$ , so, by **B**,  $h(b) \geq h(a)$ . Since  $h(a) = f(a) - ma$  and  $h(b) = f(b) - mb$ , this leads to

$$\frac{f(b) - f(a)}{b - a} \geq m.$$

Repeating this same argument with  $f(x) - Lx$ , and using **C**, leads to

$$\frac{f(b) - f(a)}{b - a} \leq L.$$

We have shown

### First Derivative Bounds

If  $m \leq f'(x) \leq L$  for  $a \leq x \leq b$ , then

$$m \leq \frac{f(b) - f(a)}{b - a} \leq L. \tag{7.1.1}$$

When  $b$  is close to  $a$ , we expect both extremes  $m$  and  $L$  to be close to  $f'(a)$ . From (7.1.1), we arrive at the formula for the derivative,

### Derivative Definition

$$f'(a) = \lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a}. \quad (7.1.2)$$

From (7.1.2), the derivative of a line  $f(x) = mx + b$  equals  $f'(a) = m$ : If the graph of a function is a line, then the tangent line to the graph is that line.

Below we also write

$$y' = f'(x) = \frac{dy}{dx}$$

or

$$f'(a) = \left. \frac{dy}{dx} \right|_{x=a}$$

When the particular point  $a$  is understood from the context, we write  $y'$ .



From (7.1.2), the basic properties of the derivative are

- *Sum rule.*  $h = f + g$  implies  $h' = f' + g'$ ,
- *Product rule.*  $h = fg$  implies  $h' = f'g + fg'$ ,
- *Quotient rule.*  $h = f/g$  implies  $h' = (f'g - fg')/g^2$ .
- *Chain rule.*  $u = f(x)$  and  $y = g(u)$  implies

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$

To visualize the chain rule, suppose

$$\begin{aligned} u &= f(x) = \sin x, \\ y &= g(u) = u^2. \end{aligned}$$

These are two functions  $f, g$  in composition, as in Figure 7.2.

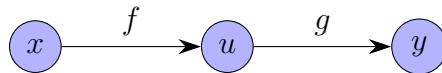


Figure 7.2: Composition of two functions.

Suppose  $x = \pi/4$ . Then  $u = \sin(\pi/4) = 1/\sqrt{2}$ , and  $y = u^2 = 1/2$ . Since

$$\frac{dy}{du} = 2u = \frac{2}{\sqrt{2}}, \quad \frac{du}{dx} = \cos x = \frac{1}{\sqrt{2}},$$

by the chain rule,

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = \frac{2}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} = 1.$$

Since the chain rule is important for machine learning, it is discussed in detail in §7.4.

Since a constant function  $f(x) = c$  is a line with slope zero, *the derivative of a constant is zero*. Since  $f(x) = x$  is a line with slope 1,  $(x)' = 1$ .

By the product rule,

$$(x^2)' = x'x + xx' = 1x + x1 = 2x.$$

Similarly one obtains the *power rule*

$$(x^n)' = nx^{n-1}. \tag{7.1.3}$$

Using the chain rule, the power rule can be derived for any rational number  $n$ , positive or negative. For example, since  $(\sqrt{x})^2 = x$ , we can write  $x = f(g(x))$  with  $f(x) = x^2$  and  $g(x) = \sqrt{x}$ . By the chain rule,

$$1 = (x)' = f'(g(x))g'(x) = 2g(x)g'(x) = 2\sqrt{x}(\sqrt{x})'.$$

Solving for  $(\sqrt{x})'$  yields

$$(\sqrt{x})' = \frac{1}{2\sqrt{x}},$$

which is (7.1.3) with  $n = 1/2$ . In this generality, the variable  $x$  is restricted to positive values only.



The *second derivative*  $f''(x)$  of  $f(x)$  is the derivative of the derivative,

$$f''(x) = (f'(x))'.$$

For example,

$$(x^n)'' = (nx^{n-1})' = n(n-1)x^{n-2} = \frac{n!}{(n-2)!}x^{n-2} = P(n, 2)x^{n-2}$$

(for  $n!$  and  $P(n, k)$  see §4.1).

More generally, the  $k$ -th derivative  $f^{(k)}(x)$  is the derivatives taken  $k$  times, so

$$(x^n)^{(k)} = n(n-1)(n-2)\dots(n-k+1)x^{n-k} = \frac{n!}{(n-k)!}x^{n-k} = P(n, k)x^{n-k}.$$

When  $k = 0$ ,  $f^{(0)}(x) = f(x)$ , and, when  $k = 1$ ,  $f^{(1)}(x) = f'(x)$ .



We use the above to derive the Taylor series. Suppose  $f(x)$  is given by a finite or infinite sum

$$f(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + \dots \quad (7.1.4)$$

Then  $f(0) = c_0$ . Taking derivatives, by the sum, product, and power rules,

$$\begin{aligned} f'(x) &= c_1 + 2c_2x + 3c_3x^2 + 4c_4x^3 + \dots \\ f''(x) &= 2c_2 + 3 \cdot 2c_3x + 4 \cdot 3c_4x^2 + \dots \\ f'''(x) &= 3 \cdot 2c_3 + 4 \cdot 3 \cdot 2c_4x + \dots \\ f^{(4)}(x) &= 4 \cdot 3 \cdot 2c_4 + \dots \end{aligned} \quad (7.1.5)$$

Inserting  $x = 0$ , we obtain  $f'(0) = c_1$ ,  $f''(0) = 2c_2$ ,  $f'''(0) = 3 \cdot 2c_3$ ,  $f^{(4)}(0) = 4 \cdot 3 \cdot 2c_4$ . This can be encapsulated by  $f^{(n)}(0) = n!c_n$ ,  $n = 0, 1, 2, 3, 4, \dots$ , which is best written

$$\frac{f^{(n)}(0)}{n!} = c_n, \quad n \geq 0.$$

Going back to (7.1.4), we derived

### Taylor Series About 0

For almost every function  $f(x)$ ,

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n = f(0) + f'(0)x + f''(0)\frac{x^2}{2} + f'''(0)\frac{x^3}{6} + f^{(4)}(0)\frac{x^4}{24} + \dots \quad (7.1.6)$$

More generally, let  $a$  be a fixed point. Then any function  $f(x)$  can be expanded in powers  $(x - a)^n$ , and we have

### Taylor Series About $a$

For almost every function  $f(x)$ ,

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n = f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2} + \dots \quad (7.1.7)$$



We review the derivative of sine and cosine. Recall the angle  $\theta$  in radians is the length of the subtended arc (in red) in Figure 7.3. Following the figure, with  $P = (x, y)$ , we have  $x = \cos \theta$ ,  $y = \sin \theta$ . By the figure, the arclength  $\theta$  is greater than the diagonal, which in turn is greater than  $y$ . Moreover  $\theta$  is less than  $1 - x + y$ , so

$$y < \theta < 1 - x + y.$$

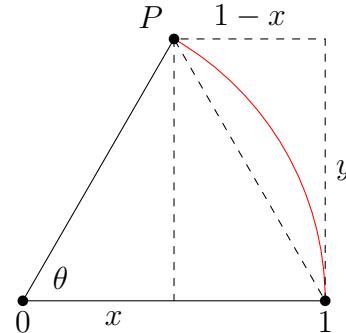


Figure 7.3: Angle  $\theta$  in the plane,  $P = (x, y)$ .

From this we have

$$\sin \theta < \theta < 1 - \cos \theta + \sin \theta,$$

which implies

$$1 - \frac{1 - \cos \theta}{\theta} < \frac{\sin \theta}{\theta} < 1. \quad (7.1.8)$$

From the Figure,  $0 < \sin \theta < \theta$ , and  $\sin^2 \theta + \cos^2 \theta = 1$ , so

$$0 \leq \frac{1 - \cos \theta}{\theta} = \frac{1 - \cos^2 \theta}{\theta(1 + \cos \theta)} = \frac{\sin \theta}{\theta} \cdot \frac{\sin \theta}{1 + \cos \theta} \leq \sin \theta \leq \theta.$$

This implies

$$\lim_{\theta \rightarrow 0} \frac{1 - \cos \theta}{\theta} = 0.$$

Taking the limit  $\theta \rightarrow 0$  in (7.1.8), this implies

$$\lim_{\theta \rightarrow 0} \frac{\sin \theta}{\theta} = 1.$$

From (1.5.5),

$$\sin(\theta + \phi) = \sin \theta \cos \phi + \cos \theta \sin \phi,$$

so

$$\lim_{\phi \rightarrow 0} \frac{\sin(\theta + \phi) - \sin \theta}{\phi} = \lim_{\phi \rightarrow 0} \sin \theta \cdot \frac{\cos \phi - 1}{\phi} + \cos \theta \cdot \frac{\sin \phi}{\phi} = \cos \theta.$$

Thus the derivative of sine is cosine,

$$(\sin \theta)' = \cos \theta.$$

Similarly,

$$(\cos \theta)' = -\sin \theta.$$

Using the chain rule, we compute the derivative of the inverse arcsin  $x$  of  $\sin \theta$ . Since

$$\theta = \arcsin x \quad \iff \quad x = \sin \theta,$$

we have

$$1 = x' = (\sin \theta)' = \theta' \cdot \cos \theta = \theta' \cdot \sqrt{1 - x^2},$$

or

$$(\arcsin x)' = \theta' = \frac{1}{\sqrt{1 - x^2}}.$$

We use this to compute the derivative of the arcsine law (3.2.13). With  $x = \sqrt{\lambda}/2$ , by the chain rule,

$$\begin{aligned} \left( \frac{2}{\pi} \arcsin \left( \frac{1}{2} \sqrt{\lambda} \right) \right)' &= \frac{2}{\pi} \frac{1}{\sqrt{1 - x^2}} \cdot x' \\ &= \frac{2}{\pi} \frac{1}{\sqrt{1 - \lambda/4}} \cdot \frac{1}{4\sqrt{\lambda}} = \frac{1}{\pi \sqrt{\lambda(4 - \lambda)}}. \end{aligned} \tag{7.1.9}$$

This shows the derivative of the arcsine law is the density in Figure 3.11.



For the parabola in Figure 7.4,  $y = x^2$  so, by the power rule,  $y' = 2x$ . Since  $y' > 0$  when  $x > 0$  and  $y' < 0$  when  $x < 0$ , this agrees with the increase/decrease of the graph. In particular, the minimum of the parabola occurs when  $y' = 0$ .

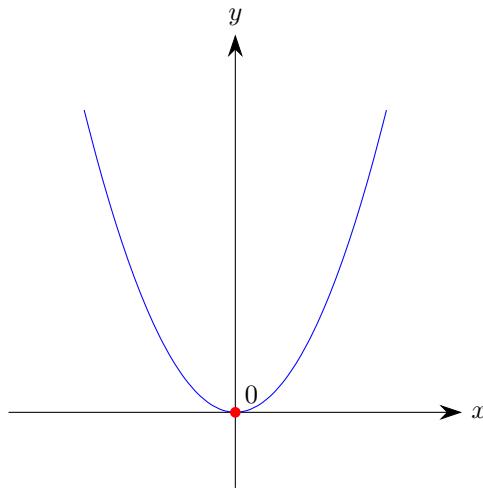


Figure 7.4: Increasing or decreasing?



For the curve  $y = x^4 - 2x^2$  in Figure 7.5,

$$y' = 4x^3 - 4x = 4x(x^2 - 1) = 4x(x - 1)(x + 1),$$

so  $y'$  is a product of the three factors  $4x$ ,  $x - 1$ ,  $x + 1$ . Since the zeros of these factors are 0, 1, and  $-1$ , and  $y' > 0$  when all factors are positive, or two of them are negative, this agrees with the increase/decrease in the figure.

Here  $y' = 0$  occurs at the two minima  $x = \pm 1$  and at the local maximum 0. Notice 0 is not a global maximum as there is no highest value for  $y$ .

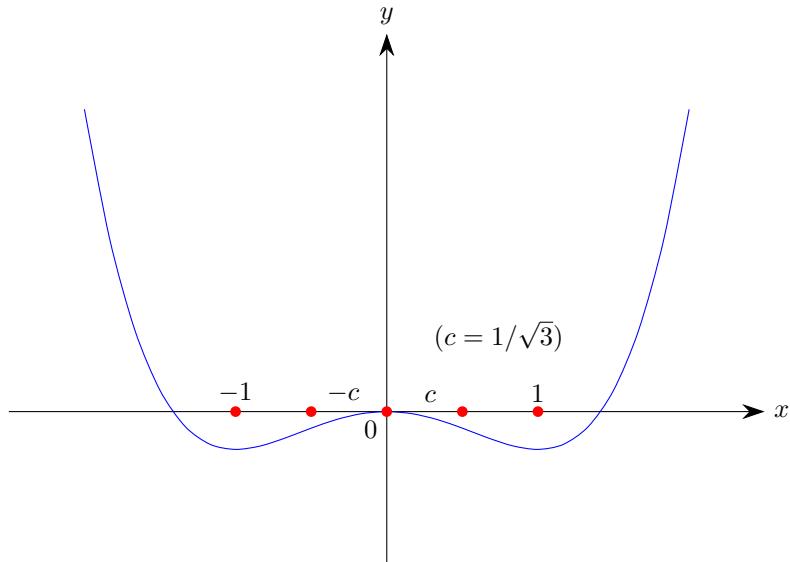


Figure 7.5: Increasing or decreasing?



Let  $y = f(x)$  be a function. A *critical point* is a point  $x^*$  where the derivative equals zero,  $f'(x^*) = 0$ . Above we saw local or global maximizers or minimizers are critical points. In general, however, this need not be so. A critical point may be neither. For example, for  $f(x) = x^3$ ,  $x^* = 0$  is a critical point, but is neither a maximizer nor a minimizer. Here, for  $y = x^3$ ,  $x^* = 0$  is a *saddle point*.



Now we look at the increase/decrease in  $y'$ , rather than in  $y$ . Applying the above logic to  $y'$  instead to  $y$ , we see  $y'$  is increasing when  $y'' \geq 0$ , and  $y'$  is decreasing when  $y'' \leq 0$ . In the first case, we say  $f(x)$  is *convex*, while in the second case, we say  $f(x)$  is *concave*.

If we look at Figure 7.4, the slope at  $x$  equals  $y' = 2x$ . Thus as  $x$  increases,  $y'$  increases. Even though the parabola height  $y$  decreases when  $x < 0$  and increases when  $x > 0$ , its slope  $y'$  is always increasing: When  $x < 0$ , as  $x$  increases,  $y' = 2x$  is less and less negative, while, when  $x > 0$ , as  $x$  increases,  $y'$  is more and more positive.

Since  $y'$  increases when its derivative is positive, the parabola's behavior is encapsulated in

$$y'' = (y')' = (2x)' = 2 > 0.$$

In general,

### Second Derivative Test for Convexity

$y = f(x)$  is convex iff  $y'' \geq 0$ , and concave if  $y'' \leq 0$ .

A point where  $y'' = 0$  is an *inflection point*. For example, the parabola in Figure 7.4 is convex everywhere. Analytically, for the parabola,  $y'' = 2 > 0$  everywhere,

For the graph in Figure 7.5 it is clear the graph is convex away from 0, and concave near 0. Analytically,

$$y'' = (x^4 - 2x^2)'' = (4x^3 - 4x)' = 12x^2 - 4 = 4(3x^2 - 1),$$

so the inflection points are  $x = \pm 1/\sqrt{3}$ . Hence the graph is convex when  $|x| > 1/\sqrt{3}$ , and the graph is concave when  $|x| < 1/\sqrt{3}$ . Since  $1/\sqrt{3} < 1$ , the graph is convex near  $x = \pm 1$ .



A function  $f(x)$  is *strictly convex* if  $y'' > 0$ . Geometrically,  $f(x)$  is strictly convex if each chord joining any two points on the graph lies strictly above the graph. Similarly, one defines *strictly concave* to mean  $y'' < 0$ .

### Second Derivative Test for Strict Convexity

Suppose  $y = f(x)$  has a second derivative  $y''$ . Then  $y$  is strictly convex if  $y'' > 0$ , and strictly concave if  $y'' < 0$ .

For example,  $x^2$  and  $e^x$  are strictly convex everywhere, and  $x^4 - 2x^2$  is strictly convex for  $|x| > 1/\sqrt{3}$ .

This was also derived in (4.4.12). Since

$$(e^x)^{(n)} = e^x, \quad n \geq 0,$$

writing the Taylor series centered at zero for the exponential function yields the exponential series (4.4.10).



Suppose  $y = f(x)$  is convex, so  $y'$  is increasing. Then  $a \leq t \leq x \leq b$  implies  $f'(a) \leq f'(t) \leq f'(x) \leq f'(b)$ . Taking  $m = f'(a)$  and  $L = f'(x)$  in (7.1.1),

$$f'(a) \leq \frac{f(x) - f(a)}{x - a} \leq f'(x), \quad a \leq x \leq b.$$

Since the tangent line at  $a$  is  $y = f'(a)(x - a) + f(a)$ , rearranging this last inequality, we obtain

### Convex Function Graph Lies Above the Tangent Line

If  $f(x)$  is convex on  $[a,b]$ , then

$$f(x) \geq f(a) + f'(a)(x - a), \quad a \leq x \leq b.$$

For example, the function in Figure 7.6 is convex near  $x = a$ , and the graph lies above its tangent line at  $a$ .



Let  $p_m(x)$  be the parabola

$$p_m(x) = f(a) + f'(a)(x - a) + \frac{m}{2}(x - a)^2.$$

Then  $p_m''(x) = m$ . Moreover the graph of  $p_m(x)$  is tangent to the graph of  $f(x)$  at  $x = a$ , in the sense  $f(a) = p_m(a)$  and  $f'(a) = p'_m(a)$ . Because of this, we call  $p_m(x)$  a *tangent parabola*.

When  $y$  is convex, we saw above the graph of  $y$  lies above its tangent line. When  $m \leq y'' \leq L$ , we can specify the size of the difference between the graph and the tangent line. In fact, the graph is constrained to lie above or below the lower or upper tangent parabolas.

### Second Derivative Bounds

If  $m \leq f''(x) \leq L$  on  $[a, b]$ , the graph lies between  $p_m(x)$  and  $p_L(x)$ ,

$$\frac{m}{2}(x - a)^2 \leq f(x) - f(a) - f'(a)(x - a) \leq \frac{L}{2}(x - a)^2. \quad (7.1.10)$$

$$a \leq x \leq b.$$

To see this, suppose  $f''(x) \geq m$ . then  $g(x) = f(x) - p_m(x)$  satisfies

$$g''(x) = f''(x) - p_m''(x) = f''(x) - m \geq 0,$$

so  $g(x)$  is convex, so  $g(x)$  lies above its tangent line at  $x = a$ . Since  $g(a) = 0$  and  $g'(a) = 0$ , the tangent line is 0, and we conclude  $g(x) \geq 0$ , which is the left half of (7.1.10). Similarly, if  $f''(x) \leq L$ , then  $p_L(x) - f(x)$  is convex, leading to the right half of (7.1.10).

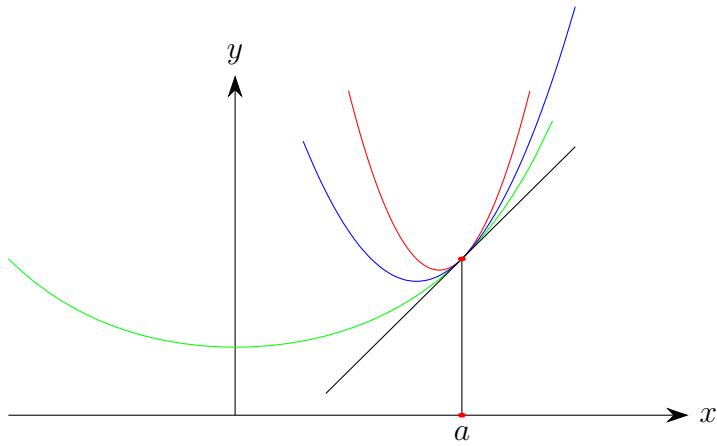


Figure 7.6: Tangent parabolas  $p_m(x)$  (green),  $p_L(x)$  (red),  $L > m > 0$ .



Now suppose  $f(x)$  is strongly convex in the sense  $L \geq f''(x) \geq m$  on an interval  $[a, b]$ , for some positive constants  $m$  and  $L$ . By (7.1.1),

$$t = \frac{f'(b) - f'(a)}{b - a} \implies L \geq t \geq m,$$

which implies

$$t^2 - (m + L)t + mL = (t - m)(t - L) \leq 0, \quad a \leq t \leq b.$$

This yields

### Coercivity for Strongly Convex Functions

If  $m \leq f''(x) \leq L$  for  $a \leq x \leq b$ , then

$$\frac{f'(b) - f'(a)}{b - a} \geq \frac{mL}{m + L} + \frac{1}{m + L} \left| \frac{f'(b) - f'(a)}{b - a} \right|^2. \quad (7.1.11)$$



We now compute the derivatives of the exponential function (§4.4). By (4.4.10),

$$\frac{e^x - 1}{x} = 1 + \frac{x}{2} + \frac{x^2}{6} + \dots,$$

so

$$(e^x)'|_{x=0} = \lim_{x \rightarrow 0} \frac{e^x - 1}{x} = 1.$$

By the law of exponents and  $t = x - a$ ,

$$\lim_{x \rightarrow a} \frac{e^x - e^a}{x - a} = e^a \cdot \lim_{x \rightarrow a} \frac{e^{x-a} - 1}{x - a} = e^a \cdot \lim_{t \rightarrow 0} \frac{e^t - 1}{t} = e^a \cdot 1 = e^a.$$

This derives

### Derivative of the Exponential Function

The exponential function satisfies

$$(e^x)' = e^x, \quad (e^x)'' = e^x.$$

In particular, since  $e^x > 0$ , the exponential function is convex.

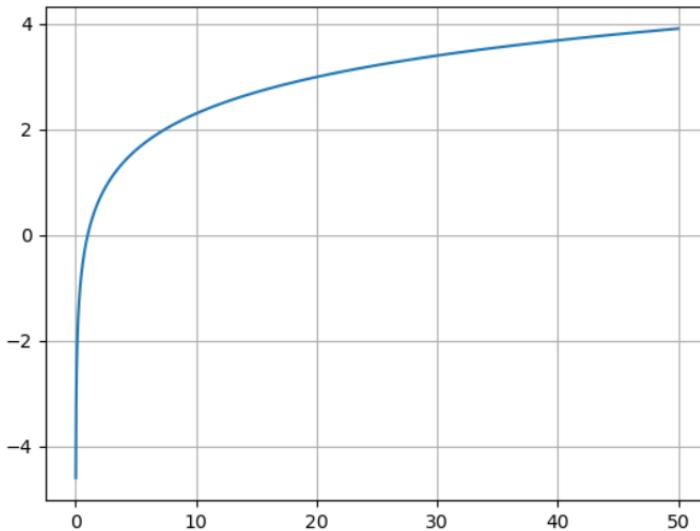


The *logarithm* function is the inverse of the exponential function,

$$y = \log x \iff x = e^y.$$

This is the same as saying

$$\log(e^y) = y, \quad e^{\log x} = x.$$

Figure 7.7: The logarithm function  $\log x$ .

From here, we see the logarithm is defined only for  $x > 0$  and is strictly increasing (Figure 7.7).

Since  $e^0 = 1$ ,

$$\log 1 = 0.$$

Since  $e^\infty = \infty$  (Figure 4.10),

$$\log \infty = \infty.$$

Since  $e^{-\infty} = 1/e^\infty = 1/\infty = 0$ ,

$$\log 0 = -\infty.$$

We also see  $\log x$  is negative when  $0 < x < 1$ , and positive when  $x > 1$ .

Moreover, by the law of exponents,

$$\log(ab) = \log a + \log b.$$

For  $a > 0$  and  $b$  real, define

$$a^b = e^{b \log a}.$$

Then, by definition,

$$\log(a^b) = b \log a,$$

and

$$(a^b)^c = (e^{b \log a})^c = e^{bc \log a} = a^{bc}.$$



By definition of the logarithm,  $y = \log x$  is shorthand for  $x = e^y$ . Use the chain rule to find  $y'$ :

$$x = e^y \quad \Rightarrow \quad 1 = x' = (e^y)' = e^y y' = xy',$$

so

$$y = \log x \quad \Rightarrow \quad y' = \frac{1}{x}.$$

### Derivative of the Logarithm

$$y = \log x \quad \Rightarrow \quad y' = \frac{1}{x}. \quad (7.1.12)$$



For gradient descent, we need the relation between a convex function and its dual. If  $f(x)$  is convex, its *convex dual* is

$$g(p) = \max_x(px - f(x)). \quad (7.1.13)$$

Below we see  $g(p)$  is also convex. This may not always exist, but we will work with cases where no problems arise.

Let  $q > 0$ . The simplest example is

$$f(x) = \frac{q}{2}x^2 \quad \Rightarrow \quad g(p) = \frac{1}{2q}p^2.$$

For each  $p$ , the point  $x$  where  $px - f(x)$  equals the maximum  $g(p)$  — the *maximizer* — depends on  $p$ . If we denote the maximizer by  $x = x(p)$ , then

$$g(p) = px(p) - f(x(p)).$$

Since the maximum occurs when the derivative is zero, we have

$$0 = (px - f(x))' = p - f'(x) \iff x = x(p).$$

Hence

$$g(p) = px - f(x) \iff p = f'(x).$$

Also, by the chain rule, differentiating with respect to  $p$ ,

$$g'(p) = (px - f(x))' = x + px' - f'(x)x' = x.$$

From this, we conclude

$$p = f'(x) \iff x = g'(p). \quad (7.1.14)$$

Thus  $f'(x)$  is the inverse function of  $g'(p)$ . Since  $g(p) = px - f(x)$  is the same as  $f(x) = px - g(p)$ , we have

### Dual of the Dual

If  $g(p)$  is the convex dual of a convex  $f(x)$ , then  $f(x)$  is the convex dual of  $g(p)$ .

Since  $f'(x)$  is the inverse function of  $g'(p)$ , we have

$$f'(g'(p)) = p.$$

Differentiating with respect to  $p$  again yields

$$f''(g'(p))g''(p) = 1.$$

We derived

### Second Derivatives of Dual Functions

Let  $f(x)$  be a strictly convex function, and let  $g(p)$  be the convex dual of  $f(x)$ . Then  $g(p)$  is strictly convex and

$$g''(p) = \frac{1}{f''(x)}, \quad (7.1.15)$$

where  $x = g'(p)$ ,  $p = f'(x)$ .

Since  $f''(x) > 0$ , also  $g''(p) > 0$ , so  $g(p)$  is strictly convex.



The *logistic function*

$$q = \sigma(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}, \quad -\infty < z < \infty, \quad (7.1.16)$$

was defined in §5.1. By the quotient and chain rules, its derivative is

$$q' = -\frac{-e^{-z}}{(1 + e^{-z})^2} = \sigma(z)(1 - \sigma(z)) = q(1 - q). \quad (7.1.17)$$

The logistic function is also called the *expit function* and the *sigmoid function*.

The inverse of the logistic function is the *logit function*. The logit function is found by solving  $q = \sigma(z)$  for  $z$ , obtaining

$$z = \sigma^{-1}(q) = \log\left(\frac{q}{1 - q}\right). \quad (7.1.18)$$

The logit function is also called the *log-odds function*. Its derivative is

$$z' = \frac{1 - q}{q} \cdot \left(\frac{q}{1 - q}\right)' = \frac{1 - q}{q} \cdot \frac{1}{(1 - q)^2} = \frac{1}{q(1 - q)}.$$

Notice the derivatives of  $\sigma$  and its inverse  $\sigma^{-1}$  are reciprocals. This result holds in general, and is called the inverse function theorem.

The *partition function* is

$$Z(z) = \log(1 + e^z). \quad (7.1.19)$$

Then  $Z'(z) = \sigma(z)$  and  $Z''(z) = \sigma'(z) = \sigma(1 - \sigma) > 0$ . This shows  $Z(z)$  is strictly convex.

The maximum

$$\max_z(pz - Z(z))$$

is attained when  $(pz - Z(z))' = 0$ , which happens when  $p = Z'(z) = \sigma(z)$ . Inserting the log-odds function  $z = \sigma^{-1}(p)$ , we obtain

$$\max_z(pz - Z(z)) = p \log\left(\frac{p}{1 - p}\right) - Z\left(\log\left(\frac{p}{1 - p}\right)\right), \quad (7.1.20)$$

which simplifies to  $I(p)$ . Thus *the convex dual of the partition function is the information*. The information is studied further in §7.2, and the multinomial extension is in §7.6.



For the chi-squared distribution (§5.5), we will need Newton's generalization of the binomial theorem to general exponents.

### Newton's Binomial Theorem

Let  $n$  be any real number. For  $a > 0$  and  $-a < x < a$ ,

$$(a + x)^n = a^n + na^{n-1}x + \binom{n}{2}a^{n-2}x^2 + \binom{n}{3}a^{n-3}x^3 + \dots$$

This makes sense because the binomial coefficient  $\binom{n}{k}$  is defined for any real number  $n$  (4.3.12), (4.3.13).

In summation notation,

$$(a + x)^n = \sum_{k=0}^{\infty} \binom{n}{k} a^{n-k} x^k. \quad (7.1.21)$$

The only difference between (4.3.7) and (7.1.21) is the upper limit of the summation, which is set to infinity. When  $n$  is a whole number, by (4.3.10), we have

$$\binom{n}{k} = 0, \quad \text{for } k > n,$$

so (7.1.21) is a sum of  $n+1$  terms, and equals (4.3.7) exactly. When  $n$  is not a whole number, the sum (7.1.21) is an infinite sum.

Actually, in §5.5, we will need the special case  $a = 1$ , which we write in slightly different notation,

$$(1 + x)^p = \sum_{n=0}^{\infty} \binom{p}{n} x^n. \quad (7.1.22)$$

Newton's binomial theorem (7.1.21) is a special case of the Taylor series centered at zero (7.1.6). To see this, set

$$f(x) = (a + x)^n.$$

Then, by the power rule,

$$f^{(k)}(x) = n(n-1)(n-2)\dots(n-k+1)(a+x)^{n-k},$$

so

$$\frac{f^{(k)}(0)}{k!} = \frac{n(n-1)(n-2)\dots(n-k+1)}{k!} a^{n-k} = \binom{n}{k} a^{n-k}.$$

Writing out the Taylor series,

$$(a+x)^n = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} = \sum_{k=0}^{\infty} \binom{n}{k} a^{n-k} x^k,$$

which is Newton's binomial theorem.

## 7.2 Entropy and Information

A function  $f(x)$  is *concave* if  $-f(x)$  is convex. We use convexity of  $e^x$  to show concavity of  $\log x$ .

Let  $a = e^x$  and  $b = e^y$ . Then  $x = \log a$  and  $y = \log b$ . Taking the log of both sides of (4.4.12), since  $\log x$  is increasing, we have

$$(1-t)x + ty \leq \log((1-t)a + tb),$$

or

$$(1-t)\log a + t\log b \leq \log((1-t)a + tb).$$

Since the inequality sign is reversed, this shows

### Concavity of the Logarithm Function

The logarithm function is strictly concave,

$$\log((1-t)a + tb) \geq (1-t)\log a + t\log b, \quad a > 0, b > 0, \quad (7.2.1)$$

for  $0 \leq t \leq 1$ .

We use calculus to derive the strict concavity. By the power rule,

$$y'' = (\log x)'' = \left(\frac{1}{x}\right)' = (x^{-1})' = -1x^{-2}.$$

Since  $x > 0$ ,  $y'' < 0$ , which shows  $\log x$  is in fact strictly concave everywhere it is defined.

Since  $\log x$  is strictly concave,

$$\log\left(\frac{1}{x}\right) = -\log x$$

is strictly convex.



Let  $p$  be a probability, i.e. a number between 0 and 1. The *entropy* of  $p$  is

$$H(p) = -p \log p - (1-p) \log(1-p), \quad 0 < p < 1. \quad (7.2.2)$$

This is sometimes called *absolute entropy* to contrast with relative entropy which we see below.

To graph  $H(p)$ , we compute its first and second derivatives. Here the independent variable is  $p$ . By the product rule,

$$H'(p) = (-p \log p - (1-p) \log(1-p))' = -\log p + \log(1-p) = \log\left(\frac{1-p}{p}\right).$$

Thus  $H'(p) = 0$  when  $p = 1/2$ ,  $H'(p) > 0$  on  $p < 1/2$ , and  $H'(p) < 0$  on  $p > 1/2$ . Since this implies  $H(p)$  is increasing on  $p < 1/2$ , and decreasing on  $p > 1/2$ ,  $p = 1/2$  is a global maximum of the graph.

Notice as  $p$  increases,  $1-p$  decreases, so  $(1-p)/p$  decreases. Since  $\log$  is increasing, as  $p$  increases,  $H'(p)$  decreases. Thus  $H(p)$  is concave.

Taking the second derivative, by the chain rule and the quotient rule,

$$H''(p) = \left( \log\left(\frac{1-p}{p}\right) \right)' = -\frac{1}{p(1-p)},$$

which is negative, leading to the strict concavity of  $H(p)$ .

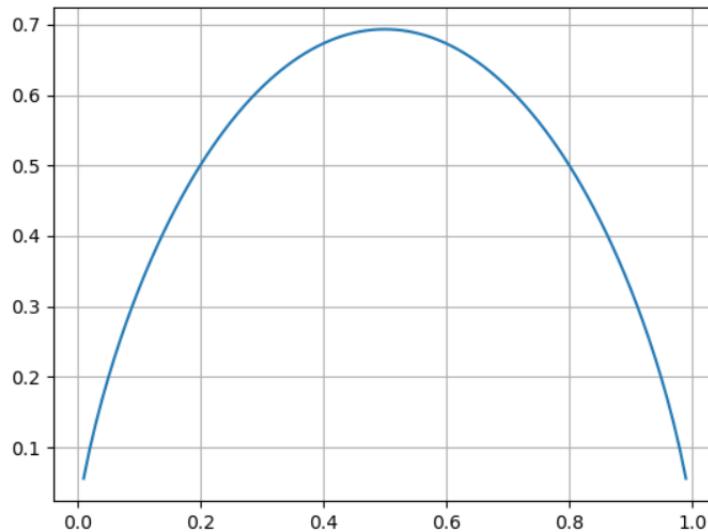


Figure 7.8: The absolute entropy function  $H(p)$ .

A crucial aspect of Figure 7.8 is its limiting values at the edges  $p = 0$  and  $p = 1$ ,

$$H(0) = \lim_{p \rightarrow 0} H(p) \quad \text{and} \quad H(1) = \lim_{p \rightarrow 1} H(p).$$

Figure 7.8 suggests  $H(0) = 0$  and  $H(1) = 0$ .

To check this, for the first limit, since  $H(p)$  is increasing near  $p = 0$ , it is clear there is a definite value  $H(0)$ . The entropy is the sum of two terms,  $-p \log p$ , and  $-(1-p) \log(1-p)$ . When  $p \rightarrow 0$ , the second term approaches  $-\log 1 = 0$ , so  $H(0)$  is the limit of the first term,

$$H(0) = -\lim_{p \rightarrow 0} p \log p.$$

When  $p \rightarrow 0$ , also  $2p \rightarrow 0$ . Replacing  $p$  by  $2p$ ,

$$\begin{aligned} H(0) &= -\lim_{p \rightarrow 0} p \log p \\ &= -\lim_{p \rightarrow 0} 2p \log(2p) \\ &= \lim_{p \rightarrow 0} -2p \log 2 + 2H(0) = 2H(0). \end{aligned}$$

Thus  $H(0) = 0$ . Since  $H(p)$  is symmetric,  $H(1-p) = H(p)$ , we also have  $H(1) = 0$ . This completes the discussion of Figure 7.8.

We can now explain the meaning of the entropy function. Suppose an event has probability  $p$ . If  $p$  is near 1, then we have confidence that the event is likely, and, if  $p$  is near 0, we have confidence the event is unlikely. If  $p = 1/2$ , then we have no information either way. Thus we can view the entropy as the negative of our information about the event: High entropy equals low information, or

### Entropy and Information

Entropy equals negative information.

Because of this, we call

$$I(p) = -H(p) = p \log p + (1-p) \log(1-p) \quad (7.2.3)$$

the *information* in  $p$ . In (7.1.20), we saw the information is the convex dual of the partition function. The derivative of  $I(p)$  is

$$I'(p) = \log\left(\frac{p}{1-p}\right). \quad (7.2.4)$$

Then  $I'(p)$  is the inverse of the derivative  $\sigma(x)$  (7.1.16) of the dual  $Z(x)$  (7.1.19) of  $I(p)$ , as it should be (7.1.14).



The clearest explanation of  $H(p)$  is in terms of counting coin tosses. Recall the binomial coefficient  $\binom{n}{k}$  is the number of ways of selecting  $k$  objects from  $n$  objects (4.3.10).

Toss a coin  $n$  times, and let  $\#_n = \#_n(p)$  be the number of outcomes where the proportion  $k/n$  of heads is  $p$ . Then the number of heads is  $k = np$ , so,

$$\#_n(p) = \binom{n}{np}.$$

When  $p$  is an irrational,  $np$  is replaced by the floor  $\lfloor np \rfloor$ , but we ignore this point. Using (4.1.1), a straightforward calculation results in

### Entropy and Coin-Tossing Counting

Toss a coin  $n$  times, and let  $\#_n(p)$  be the number of outcomes where the proportion of heads is  $p$ . Then we have the approximation

$$\#_n(p) \sim e^{nH(p)}, \quad \text{for } n \text{ large.}$$

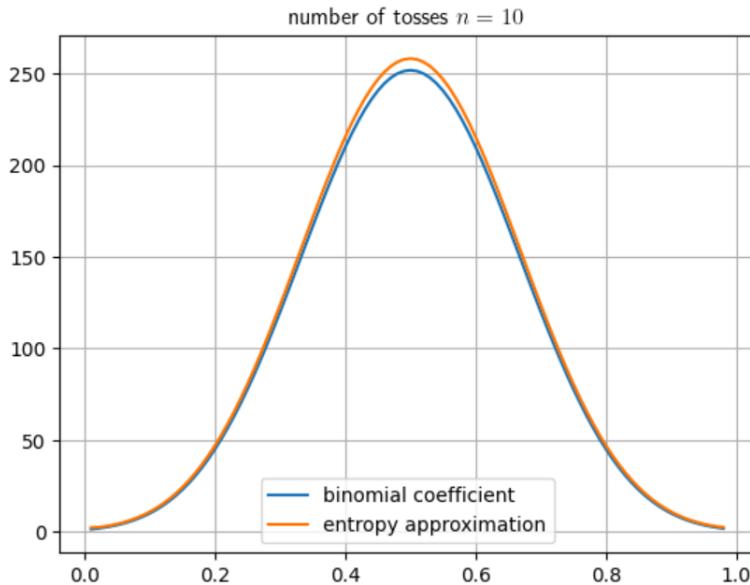


Figure 7.9: Asymptotics of binomial coefficients.

In more detail, using (4.1.6), one can derive the asymptotic equality

$$\#_n(p) \approx \frac{1}{\sqrt{2\pi n}} \cdot \frac{1}{\sqrt{p(1-p)}} \cdot e^{nH(p)}, \quad \text{for } n \text{ large.} \quad (7.2.5)$$

Figure 7.9 is returned by the code below, which compares both sides of the asymptotic equality (7.2.5) for  $n = 10$ .

```
from numpy import *
from scipy.special import comb
from matplotlib.pyplot import *
```

```

n = 10
def H(p): return - p*log(p) - (1-p)*log(1-p)
p = arange(.01,.99,.01)

grid()
plot(p, comb(n, n*p), label="binomial coefficient")
plot(p, exp(n*H(p))/sqrt(2*n*pi*p*(1-p)), label="entropy
    ↪ approximation")
title("number of tosses " + "$n=" + str(n) + "$", usetex=True)
legend()
show()

```

The `usetex=True` option assumes  $\text{\TeX}$  is installed on your system.



Let  $p$  and  $q$  be two probabilities,

$$0 < p < 1, \quad \text{and} \quad 0 < q < 1.$$

When do we consider  $p$  and  $q$  close to each other? If  $p$  and  $q$  were just numbers,  $p$  and  $q$  are considered close if the distance  $|p - q|$  is small or the distance squared  $|p - q|^2$  is small. But here  $p$  and  $q$  are probabilities, so it makes sense to consider them close if their information content is close.

To this end, we define the *relative information*  $I(p, q)$  by

$$I(p, q) = p \log\left(\frac{p}{q}\right) + (1-p) \log\left(\frac{1-p}{1-q}\right).$$

Then

$$I(q, q) = 0,$$

which agrees with our understanding that  $I(p, q)$  measures the difference in information between  $p$  and  $q$ . Because  $I(p, q)$  is not symmetric in  $p, q$ , we think of  $q$  as a base or reference probability, against which we compare  $p$ .

Equivalently, instead of measuring relative information, we can measure the *relative entropy*,

$$H(p, q) = -I(p, q).$$

Since  $-\log(x)$  is strictly convex,

$$\begin{aligned} I(p, q) &= -p \log\left(\frac{q}{p}\right) - (1-p) \log\left(\frac{1-q}{1-p}\right) \\ &> -\log\left(p \cdot \frac{q}{p} + (1-p) \cdot \frac{1-q}{1-p}\right) \\ &= -\log 1 = 0. \end{aligned}$$

This shows  $I(p, q)$  is positive and  $H(p, q)$  is negative, when  $p \neq q$ .

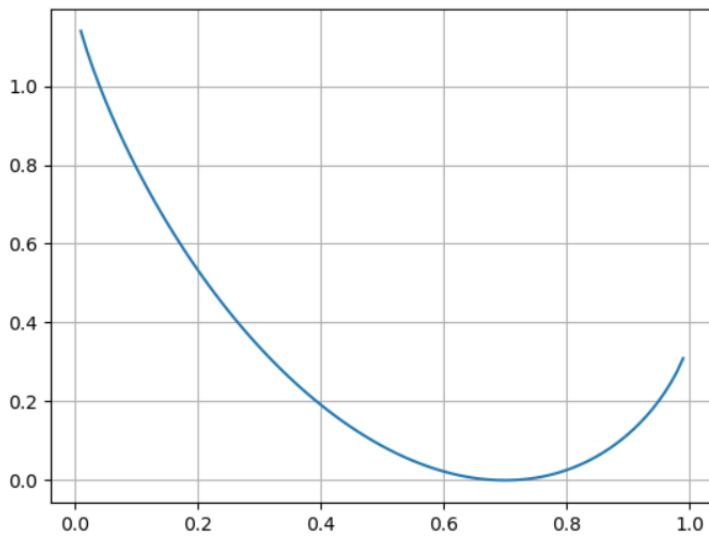


Figure 7.10: The relative information  $I(p, q)$  with  $q = .7$ .

Since

$$I(p, q) = -H(p) - p \log(q) - (1-p) \log(1-q)$$

and  $H(0) = 0 = H(1)$ ,  $I(p, q)$  is well-defined for  $p = 0$ , and  $p = 1$ ,

$$I(1, q) = -\log q, \quad I(0, q) = -\log(1-q).$$

Taking derivatives (with independent variable  $p$ ),

$$\frac{d^2}{dp^2} I(p, q) = -H''(p) = \frac{1}{p(1-p)},$$

hence  $I$  is strictly convex in  $p$ . Thus  $q$  is a global minimum of the graph of  $I(p, q)$  (Figure 7.10). Also

$$\frac{d^2}{dq^2} I(p, q) = \frac{p}{q^2} + \frac{1-p}{(1-q)^2},$$

so  $I(p, q)$  is strictly convex in  $q$  as well.



Assume the probability of heads in a single toss of a coin is  $q$ . Then we expect the long-term *proportion* of heads in  $n$  tosses to equal roughly  $q$ . Now let  $p$  be another probability,  $0 \leq p \leq 1$ .

Toss a coin  $n$  times, and let  $P_n(p, q)$  be the probability of obtaining outcomes where the proportion of heads equals  $p$ , given that the base heads-probability is  $q$ .

If  $p$  is not equal to  $q$ , we expect this outcome to be unlikely. In other words, we expect  $P_n(p, q)$  to be small for large  $n$ . In fact, as  $n \rightarrow \infty$ , we expect  $P_n(p, q) \rightarrow 0$ .

We derive a formula for the speed of this decay. With  $k = np$  in the binomial distribution (5.1.5),

$$P_n(p, q) = \binom{n}{np} q^{np} (1-q)^{n-np}.$$

Using (4.1.1), a straightforward calculation results in

### Relative Entropy and Coin-Tossing Probabilities

Assume the heads-probability of a coin is  $q$ . Toss the coin  $n$  times, and let  $P_n(p, q)$  be the probability of obtaining outcomes where the proportion of heads is  $p$ . Then we have the approximation

$$P_n(p, q) \sim e^{nH(p, q)}, \quad \text{for } n \text{ large.} \quad (7.2.6)$$

In more detail, using (4.1.6), one can derive the asymptotic equality

$$P_n(p, q) \approx \frac{1}{\sqrt{2\pi n}} \cdot \frac{1}{\sqrt{p(1-p)}} \cdot e^{nH(p,q)}, \quad \text{for } n \text{ large.} \quad (7.2.7)$$

The *law of large numbers* (§6.1) states that the proportion of heads equals approximately  $q$  for large  $n$ . Therefore, when  $p \neq q$ , we expect the probability that the proportion of heads equal  $p$  should become successively smaller as  $n$  get larger, and in fact vanish when  $n = \infty$ . Since  $H(p, q) < 0$  when  $p \neq q$ , (7.2.7) implies this is so. Thus (7.2.7) may be viewed as a quantitative strengthening of the law of large numbers, in the setting of coin-tossing.

## 7.3 Multi-variable Calculus

Let

$$f(x) = f(x_1, x_2, \dots, x_d)$$

be a scalar function of a point  $x = (x_1, x_2, \dots, x_d)$  in  $\mathbf{R}^d$ , and suppose  $v$  is a unit vector in  $\mathbf{R}^d$ . Then, along the line  $x(t) = x + tv$ ,  $g(t) = f(x + tv)$  is a function of the single variable  $t$ . Hence its derivative  $g'(0)$  at  $t = 0$  is well-defined. Since  $g'(0)$  depends on the point  $x$  and on the direction  $v$ , this rate of change is the directional derivative of  $f(x)$  at  $x$  in the direction  $v$ .

More explicitly, the *directional derivative* of  $f(x)$  at  $x$  in the direction  $v$  is

$$D_v f(x) = \left. \frac{d}{dt} \right|_{t=0} f(x + tv). \quad (7.3.1)$$

In multiple dimensions, there are many directions  $v$  emanating from a point  $x$ , we may ask: How does the direction  $v$  affect the rate of change of temperature  $f$ ? More specifically, in which direction  $v$  does the temperature  $f$  increase? In which direction  $v$  does the temperature decrease? In which direction does the temperature have the greatest increase? In which direction does the temperature have the greatest decrease? In one dimension, there are only two directions, so the directional derivative is either  $f'(x)$  or  $-f'(x)$ .



When we select specific directions, the directional derivatives have specific names. Let  $e_1, e_2, \dots, e_d$  be the standard basis in  $\mathbf{R}^d$ . The *partial derivative*

in the  $k$ -th direction,  $k = 1, \dots, d$ , is

$$\frac{\partial f}{\partial x_k}(x) = \left. \frac{d}{ds} \right|_{t=0} f(x + te_k).$$

The partial derivative in the  $k$ -th direction is just the one-dimensional derivative considering  $x_k$  as the independent variable, with all other  $x_j$ 's constants.



Below we exhibit the multi-variable chain rule in two ways. The first interpretation is geometric, and involves motion in time and directional derivatives. This interpretation is relevant to gradient descent, §8.3.

The second interpretation is combinatorial, and involves repeated compositions of functions. This interpretation is relevant to computing gradients in networks, specifically backpropagation §7.4, §8.2.

These two interpretations work together when training neural networks, §8.4.



For the first interpretation of the chain rule, suppose the components  $x_1, x_2, \dots, x_d$  are functions of a single variable  $t$  (usually time), so we have

$$x_1 = x_1(t), \quad x_2 = x_2(t), \quad \dots, \quad x_d = x_d(t).$$

Inserting these into  $f(x_1, x_2, \dots, x_d)$ , we obtain a function

$$f(t) = f(x_1(t), x_2(t), \dots, x_d(t))$$

of a single variable  $t$ . Then we have

### Multi-Variable Chain Rule

With  $f(t) = f(x_1(t), x_2(t), \dots, x_d(t))$ ,

$$\frac{df}{dt} = \frac{\partial f}{\partial x_1} \cdot \frac{dx_1}{dt} + \frac{\partial f}{\partial x_2} \cdot \frac{dx_2}{dt} + \dots + \frac{\partial f}{\partial x_d} \cdot \frac{dx_d}{dt}.$$

The *gradient* of  $f(x)$  is the vector

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_d} \right). \quad (7.3.2)$$

The  $\mathbf{R}^d$ -valued function  $x(t) = (x_1(t), x_2(t), \dots, x_d(t))$  represents a curve or path in  $\mathbf{R}^d$ , and the vector

$$x'(t) = (x'_1(t), x'_2(t), \dots, x'_d(t))$$

represents its *velocity* at time  $t$ .

With this notation, the chain rule may be written

$$\frac{df}{dt} = \nabla f(x(t)) \cdot x'(t).$$

Let  $v = (v_1, v_2, \dots, v_d)$ . The simplest application of the multi-variable chain rule is to select  $x(t) = x + tv$ . Then the chain rule becomes

### Directional Derivative Formula

The directional derivative of  $f(x)$  in the direction  $v$  is the dot product of the gradient  $\nabla f(x)$  and  $v$ ,

$$\left. \frac{d}{dt} \right|_{t=0} f(x + tv) = \nabla f(x) \cdot v. \quad (7.3.3)$$



In §8.6, we will need to compute the gradient of a function  $f(W)$  of matrices  $W$ . Towards this, recall the collection of matrices with a fixed shape may be added and scaled. It follows if  $W$  and  $V$  are matrices with the same shape, then  $W + sV$  also has the same shape, for any scalar  $s$ .

If  $G$  and  $V$  are two matrices with the same shape, we think of  $\text{trace}(V^t G)$  as a dot product between  $G$  and  $V$ . This is consistent with the definition of norm squared (2.2.12). By analogy with (7.3.3), we say

### Directional Derivative Matrix Formula

A matrix  $G$  is the gradient of  $f(W)$  at  $W$  if

$$\left. \frac{d}{ds} \right|_{s=0} f(W + sV) = \text{trace}(V^t G). \quad \text{for all } V. \quad (7.3.4)$$

Then the gradient  $G$  has the same shape as  $W$ .



Here is an example of the second interpretation of the chain rule. Suppose

$$\begin{aligned} r &= f(x) = \sin x, \\ s &= g(x) = \frac{1}{1 + e^{-x}}, \\ t &= h(x) = x^2, \\ u &= r + s + t, \\ y &= k(u) = \cos u. \end{aligned}$$

These are multiple functions in composition, as in Figure 7.11.

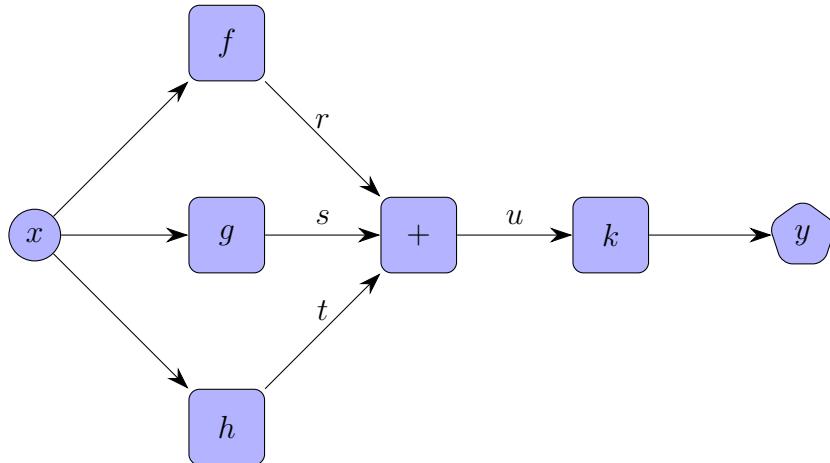


Figure 7.11: Composition of multiple functions.

The input variable is  $x$  and the output variable is  $y$ . The intermediate variables are  $r, s, t, u$ . Suppose  $x = \pi/4$ . Then

$$x, r, s, t, u, y = 0.79, 0.71, 0.69, 0.62, 2.01, -0.43.$$

To compute derivatives, start with

$$\frac{dy}{du} = k'(u) = -\sin u = -0.90.$$

Next, to compute  $dy/dr$ , the chain rule says

$$\frac{dy}{dr} = \frac{dy}{du} \frac{du}{dr} = -0.90 * 1 = -0.90,$$

and similarly,

$$\frac{dy}{ds} = \frac{dy}{dt} = -0.90.$$

By the chain rule,

$$\frac{dy}{dx} = \frac{dy}{dr} \cdot \frac{dr}{dx} + \frac{dy}{ds} \cdot \frac{ds}{dx} + \frac{dy}{dt} \cdot \frac{dt}{dx}.$$

By (7.1.17),  $s' = s(1 - s) = 0.22$ , so

$$\frac{dr}{dx} = \cos x = 0.71, \quad \frac{ds}{dx} = s(1 - s) = 0.22, \quad \frac{dt}{dx} = 2x = 1.57.$$

We obtain

$$\frac{dy}{dx} = -0.90 * 0.71 - 0.90 * 0.22 - 0.90 * 1.57 = -2.25.$$

The chain rule is discussed in further detail in §7.4.



Let  $y = f(x)$  be a function. A *critical point* is a point  $x^*$  satisfying

$$\nabla f(x^*) = 0.$$

Let  $x^*$  be a local or global minimizer of  $y = f(x)$ . Then for any vector  $v$  and scalar  $t$  near zero,  $f(x^*) \leq f(x^* + tv)$ . Hence

$$\nabla f(x) \cdot v = \left. \frac{d}{dt} \right|_{t=0} f(x^* + tv) = \lim_{t \rightarrow 0} \frac{f(x^* + tv) - f(x^*)}{t} \geq 0.$$

This is so for any direction  $v$ . Replacing  $v$  by  $-v$ , we conclude  $\nabla f(x^*) \cdot v = 0$ . Since  $v$  is any direction,  $\nabla f(x^*) = 0$ ,  $x^*$  is a critical point. Thus *a minimizer is a critical point*. Similarly, *a maximizer is a critical point*.

As in the single-variable case, a critical point may be neither a minimizer nor a maximizer, for example  $x^* = (0, 0)$  and  $y = x_1^2 - x_2^2$ . Such a point is a *saddle point*.

If  $x^*$  is a critical point and  $D^2 f(x^*) > 0$ , then  $x^*$  is a local or global minimizer. This is the same as saying all eigenvalues of the symmetric matrix  $D^2 f(x^*)$  are positive. When  $D^2 f(x^*) < 0$ ,  $x^*$  is a local or global maximum.

If  $D^2f(x^*)$  has both positive and negative eigenvalues,  $x^*$  is a saddle point.



Let  $Q$  be a  $d \times d$  symmetric matrix, let  $b$  be a vector, and let

$$f(x) = \frac{1}{2}x \cdot Qx - b \cdot x = \frac{1}{2} \sum_{i,j=1}^d q_{ij}x_i x_j - \sum_{j=1}^d b_j x_j. \quad (7.3.5)$$

When  $Q$  is a covariance matrix and  $b = 0$ ,  $f(x)$  is the variance corresponding to covariance matrix  $Q$ .

In this case,

$$\frac{\partial f}{\partial x_i} = \frac{1}{2} \sum_{j=1}^d q_{ij}x_j + \frac{1}{2} \sum_{j=1}^d q_{ji}x_j - b_i = (Qx - b)_i.$$

Here we used  $Q = Q^t$ . Thus  $\nabla f(x) = Qx - b$ , and

$$D_v f(x) = v \cdot (Qx - b).$$

A multi-variable function  $f(x)$  is *convex* if its restriction to any line is convex. Explicitly,  $f(x)$  is convex if the single-variable function  $g(t) = f(x_0 + tv)$  is convex for every point  $x_0$  and every direction  $v$ .

For example, when  $f(x)$  is given by (7.3.5),

$$\begin{aligned} g(t) &= f(x_0 + tv) \\ &= \frac{1}{2}(x_0 + tv) \cdot Q(x_0 + tv) - b \cdot (x_0 + tv) \\ &= \frac{1}{2}x_0 \cdot Qx_0 - b \cdot x_0 + tv \cdot (Qx_0 - b) + \frac{1}{2}t^2v \cdot Qv \\ &= f(x_0) + tv \cdot (Qx_0 - b) + \frac{1}{2}t^2v \cdot Qv. \end{aligned} \quad (7.3.6)$$

From this follows

$$g'(t) = v \cdot (Qx_0 - b) + tv \cdot Qv, \quad g''(t) = f(v) = \frac{1}{2}v \cdot Qv.$$

This shows

### Quadratic Convexity

Let  $Q$  be a symmetric matrix and  $b$  a vector. The quadratic function

$$f(x) = \frac{1}{2}x \cdot Qx - b \cdot x$$

has gradient

$$\nabla f(x) = Qx - b. \quad (7.3.7)$$

Moreover  $f(x)$  is convex everywhere exactly when  $Q$  is a covariance matrix,  $Q \geq 0$ .



By (2.2.2),

$$D_v f(x) = \nabla f(x) \cdot v = |\nabla f(x)| |v| \cos \theta,$$

where  $\theta$  is the angle between the vector  $v$  and the gradient vector  $\nabla f(x)$ .

Since  $-1 \leq \cos \theta \leq 1$ , we conclude

### Gradient is Direction of Greatest Increase

Let  $v$  be a unit vector and let  $x_0$  be a point in  $\mathbf{R}^d$ . As the direction  $v$  varies, the directional derivative varies between the two extremes

$$-|\nabla f(x_0)| \leq D_v f(x_0) \leq |\nabla f(x_0)|.$$

The directional derivative achieves its greatest value when  $v$  points in the direction of  $\nabla f(x_0)$ , and achieves its least value in the opposite direction, when  $v$  points in the direction of  $-\nabla f(x_0)$ .



## 7.4 Back Propagation

In this section, we compute outputs and derivatives on a graph. We consider two cases, when the graph is a chain, or the graph is a network of neurons.

The derivatives are taken with respect to the outputs at each node of the graph. In §8.2, we consider a third case, and compute outputs and derivatives on a neural network.

To compute node outputs, we do forward propagation. To compute derivatives, we do back propagation. Corresponding to the three cases, we will code three versions of forward and back propagation. In all cases, back propagation depends on the chain rule.

The chain rule (§7.1) states

$$r = f(x), y = g(r) \quad \Rightarrow \quad \frac{dy}{dx} = \frac{dy}{dr} \cdot \frac{dr}{dx}.$$

In this section, we work out the implications of the chain rule on repeated compositions of functions.

Suppose

$$\begin{aligned} r &= f(x) = \sin x, \\ s &= g(r) = \frac{1}{1 + e^{-r}}, \\ y &= h(s) = s^2. \end{aligned}$$

These are three functions  $f, g, h$  composed in a *chain* (Figure 7.12).

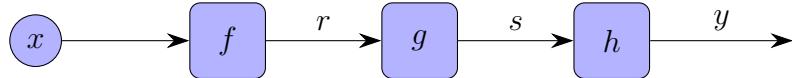


Figure 7.12: Composition of three functions in a chain.

The chain in Figure 7.12 has four nodes and four edges. The *outputs* at the nodes are  $x, r, s, y$ . Start with output  $x = \pi/4$ . Evaluating the functions in order,

$$x = 0.785, \quad r = 0.707, \quad s = 0.670, \quad y = 0.448.$$

Notice these values are evaluated in the forward direction:  $x$  then  $r$  then  $s$  then  $y$ . This is *forward propagation*.

Now we evaluate the derivatives of the output  $y$  with respect to  $x, r, s$ ,

$$\frac{dy}{dx}, \quad \frac{dy}{dr}, \quad \frac{dy}{ds}.$$

With the above values for  $x, r, s$ , we have

$$\frac{dy}{ds} = 2s = 2 * 0.670 = 1.340.$$

Since  $g$  is the logistic function, by (7.1.17),

$$g'(r) = g(r)(1 - g(r)) = s(1 - s) = 0.670 * (1 - 0.670) = 0.221.$$

From this,

$$\frac{dy}{dr} = \frac{dy}{ds} \cdot \frac{ds}{dr} = 1.340 * g'(r) = 1.340 * 0.221 = 0.296.$$

Repeating one more time,

$$\frac{dy}{dx} = \frac{dy}{dr} \cdot \frac{dr}{dx} = 0.296 * \cos x = 0.296 * 0.707 = 0.209.$$

Thus the derivatives are

$$\frac{dy}{dx} = 0.209, \quad \frac{dy}{dr} = 0.296, \quad \frac{dy}{ds} = 1.340.$$

Notice the derivatives are evaluated in the backward direction: First  $dy/dy = 1$ , then  $dy/ds$ , then  $dy/dr$ , then  $dy/dx$ . This is *back propagation*.



Here is another example. Let

$$\begin{aligned} r &= x^2, \\ s &= r^2 = x^4, \\ y &= s^2 = x^8. \end{aligned}$$

This is the same function  $h(x) = x^2$  composed with itself three times. With  $x = 5$ , we have

$$x = 5, \quad r = 25, \quad s = 625, \quad y = 390625.$$

Applying the chain rule as above, check that

$$\frac{dy}{dx} = 625000, \quad \frac{dy}{dr} = 62500, \quad \frac{dy}{ds} = 1250.$$



To evaluate  $x, r, s, y$  in Figure 7.12, first we built the list of functions and the list of derivatives

```

from numpy import *

def f(x): return sin(x)
def g(r): return 1/(1+ exp(-r))
def h(s): return s**2
# this for next example
def k(t): return cos(t)

func_chain = [f,g,h]

def df(x): return cos(x)
def dg(r): return g(r)*(1-g(r))
def dh(s): return 2*s
# this for next example
def dk(t): return -sin(t)

der_chain = [df,dg,dh]

```

Then we evaluate the output vector  $x = (x, r, s, y)$ , leading to the first version of forward propagation,

```

# first version: chains

def forward_prop(x_in,func_chain):
    x = [x_in]
    while func_chain:
        f = func_chain.pop(0) # first func
        x_out = f(x_in)
        x.append(x_out) # insert at end
        x_in = x_out
    return x

from numpy import *
x_in = pi/4
x = forward_prop(x_in,func_chain)

```

Now we evaluate the gradient vector  $\delta = (dy/dx, dy/dr, dy/ds, dy/dy)$ . Since  $dy/dy = 1$ , we set

```
# dy/dy = 1
delta_out = 1
```

The code for the first version of back propagation is

```
# first version: chains

def backward_prop(delta_out,x,der_chain):
    delta = [delta_out]
    while der_chain:
        # discard last output
        x.pop(-1)
        df = der_chain.pop(-1) # last der
        der = df(x[-1])
        # chain rule -- multiply by previous der
        der = der * delta[0]
        delta.insert(0,der) # insert at start
    return delta

delta = backward_prop(delta_out,x,der_chain)
```

Note forward propagation must be run prior to back propagation.



To apply this code to the second example, use

```
d = 3
func_chain, der_chain = [h]*d, [dh]*d
x_in, delta_out = 5, 1

x = forward_prop(x_in,func_chain)
delta = backward_prop(delta_out,x,der_chain)
```



Now we work with the network in Figure 7.13, using the multi-variable chain rule (§7.3). The functions are

$$\begin{aligned} a &= f(x, y) = x + y, \\ b &= g(y, z) = \max(y, z), \\ J &= h(a, b) = ab. \end{aligned}$$

The composite function is

$$J = (x + y) \max(y, z),$$

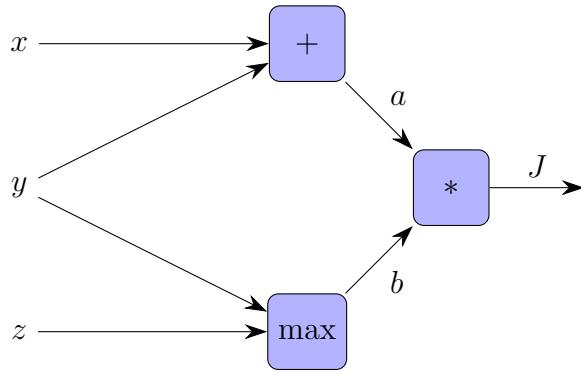


Figure 7.13: A network composition [27].

Here there are three input nodes  $x, y, z$ , and three hidden nodes  $+$ ,  $\max$ ,  $*$ . Starting with inputs  $(x, y, z) = (1, 2, 0)$ , and plugging in, we obtain node outputs

$$(x, y, z, a, b, J) = (1, 2, 0, 3, 2, 6)$$

(Figure 7.15). This is forward propagation.



Now we compute the derivatives

$$\frac{\partial J}{\partial x}, \quad \frac{\partial J}{\partial y}, \quad \frac{\partial J}{\partial z}, \quad \frac{\partial J}{\partial a}, \quad \frac{\partial J}{\partial b}.$$

This we do in reverse order. First we compute

$$\frac{\partial J}{\partial a} = b = 2, \quad \frac{\partial J}{\partial b} = a = 3.$$

Then

$$\frac{\partial a}{\partial x} = \mathbf{1}, \quad \frac{\partial a}{\partial y} = \mathbf{1}.$$

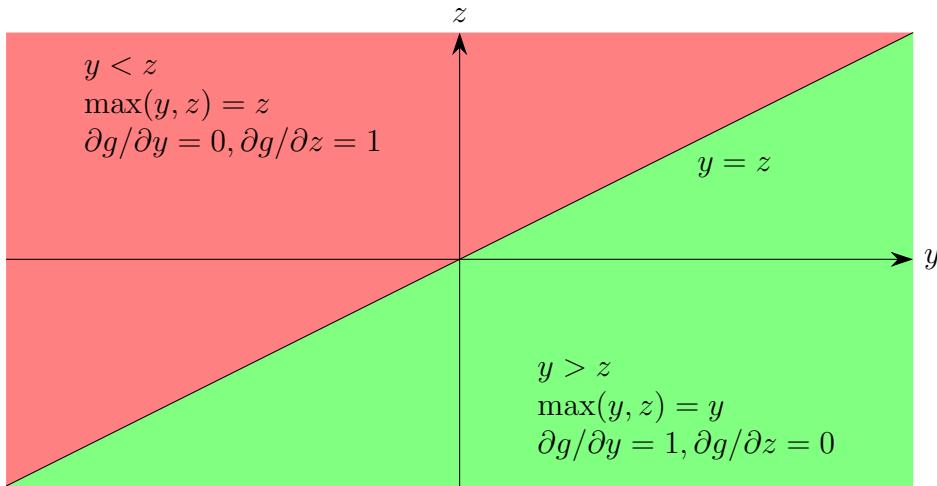


Figure 7.14: The function  $g = \max(y, z)$ .

Let

$$\mathbf{1}(y > z) = \begin{cases} 1, & y > z, \\ 0, & y < z. \end{cases}$$

By Figure 7.14, since  $y = 2$  and  $z = 0$ ,

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = \mathbf{1}, \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = \mathbf{0}.$$

By the chain rule,

$$\begin{aligned} \frac{\partial J}{\partial x} &= \frac{\partial J}{\partial a} \frac{\partial a}{\partial x} = \mathbf{2 * 1} = 2, \\ \frac{\partial J}{\partial y} &= \frac{\partial J}{\partial a} \frac{\partial a}{\partial y} + \frac{\partial J}{\partial b} \frac{\partial b}{\partial y} = \mathbf{2 * 1 + 3 * 1} = 5, \\ \frac{\partial J}{\partial z} &= \frac{\partial J}{\partial b} \frac{\partial b}{\partial z} = \mathbf{3 * 0} = 0. \end{aligned}$$

Hence we have

$$\left( \frac{\partial J}{\partial x}, \frac{\partial J}{\partial y}, \frac{\partial J}{\partial z}, \frac{\partial J}{\partial a}, \frac{\partial J}{\partial b}, \frac{\partial J}{\partial J} \right) = (2, 5, 0, 2, 3, 1).$$

The outputs (blue) and the derivatives (red) are displayed in Figure 7.15.

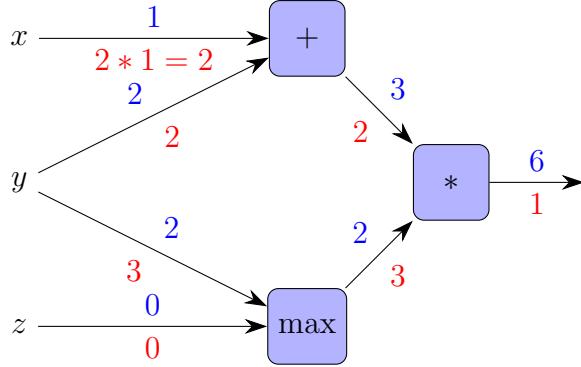


Figure 7.15: Forward and backward propagation [27].

Summarizing, by the chain rule,

- derivatives are computed backward,
- derivatives along successive edges are multiplied,
- derivatives along several outgoing edges are added.



To do this in general, recall a directed graph (§4.2) as in Figure 7.13 has an adjacency matrix  $W = (w_{ij})$  with  $w_{ij}$  equal to one or zero depending on whether  $(i, j)$  is an edge or not.

Suppose a directed graph has  $d$  nodes, and, for each node  $i$ , let  $x_i$  be the outgoing signal. Then  $x = (x_1, x_2, \dots, x_d)$  is the *outgoing vector*. In the case of Figure 7.13,  $d = 6$  and

$$x = (x_1, x_2, x_3, x_4, x_5, x_6) = (x, y, z, a, b, J).$$

With this order, the adjacency matrix is

$$W = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

This we code as a list of lists,

```
d = 6
w = [ [None]*d for _ in range(d) ]
w[0][3] = w[1][3] = w[1][4] = w[2][4] = w[3][5] = w[4][5] = 1
```

More generally, in a weighed directed graph, the weights  $w_{ij}$  are numeric scalars. In this case, for each node  $j$ , let

$$x_j^- = (w_{1j}x_1, w_{2j}x_2, \dots, w_{dj}x_d). \quad (7.4.1)$$

Then  $x_j^-$  is the *list* of node signals, each weighed accordingly. If  $(i, j)$  is not an edge, then  $w_{ij} = 0$ , so  $x_i$  does not appear in  $x_j^-$ : In other words,  $x_j^-$  is the weighed *list of incoming signals* at node  $j$ .

An *activation function* at node  $j$  is a function  $f_j$  of the incoming signals  $x_j^-$ . Then the outgoing signal at node  $j$  is

$$x_j = f_j(x_j^-) = f_j(w_{1j}x_1, w_{2j}x_2, \dots, w_{dj}x_d). \quad (7.4.2)$$

By the chain rule,

$$\frac{\partial x_j}{\partial x_i} = \begin{cases} \frac{\partial f_j}{\partial x_i} \cdot w_{ij}, & \text{if } (i, j) \text{ is an edge,} \\ 0, & \text{if } (i, j) \text{ is not an edge.} \end{cases} \quad (7.4.3)$$

For example, if  $(1, 5), (7, 5), (2, 5)$  are the edges pointing to node 5 and we ignore zeros in (7.4.1), then  $x_5^- = (w_{15}x_1, w_{75}x_7, w_{25}x_2)$ , so

$$x_5 = f_5(x_5^-) = f_5(w_{15}x_1, w_{75}x_7, w_{25}x_2).$$

The *incoming vector* is

$$x^- = (x_1^-, x_2^-, \dots, x_d^-).$$

Then  $x^-$  is a list of lists. In the case of Figure 7.13, if we ignore zeros,

$$x^- = (x_1^-, x_2^-, x_3^-, x_4^-, x_5^-, x_6^-) = (((), (), (), (x, y), (y, z), (a, b)),$$

and

$$f_4(x, y) = x + y, \quad f_5(y, z) = \max(y, z), \quad J(a, b) = ab.$$

Note there is nothing incoming at the input nodes, so there is no point defining  $f_1, f_2, f_3$ .

```
activate = [None]*d

activate[3] = lambda x,y: x+y
activate[4] = lambda y,z: max(y,z)
activate[5] = lambda a,b: a*b
```

Assume `activate[j]` is the function at node  $j$ . To compute the outgoing signal  $x_j$  at node  $j$ , we collect the incoming signals  $x_j^-$  following (7.4.1)

```
def incoming(x,w,j):
    return [outgoing(x,w,i) * w[i][j] if w[i][j] else 0 for i
            → in range(d)]
```

then plug them into the activation function,

```
def outgoing(x,w,j):
    if x[j] != None: return x[j]
    else: return activate[j](*incoming(x,w,j))
```

Here `*` is the unpacking operator.

Summarizing, at each node  $j$ , we have the outgoing signal  $x_j$ , and a list  $x_j^-$  of incoming signals.



A node with an attached activation function is a *neuron*. A *network* is a directed weighed graph where the nodes are neurons. The code in this section works for any network without cycles. In §8.2, we specialize to neural networks. Neural networks are networks with a restricted class of activation functions.



Let  $x_{\text{in}}$  be the outgoing vector over the input nodes. If there are  $m$  input nodes, and  $d$  nodes in total, then the length of  $x_{\text{in}}$  is  $m$ , and the length of  $x$  is  $d$ . In the example above,  $x_{\text{in}} = (x, y, z)$ .

We assume the nodes are ordered so that the initial portion of  $x$  equals  $x_{\text{in}}$ ,

```
m = len(x_in)
x[:m] = x_in
```

Here is the second version of forward propagation.

```
# second version: networks

def forward_prop(x_in,w):
    d = len(w)
    x = [None]*d
    m = len(x_in)
    x[:m] = x_in
    for j in range(m,d): x[j] = outgoing(x,w,j)
    return x
```

For this code to work, we assume there are no cycles in the graph: All backward paths end at inputs.



Let  $x_{\text{out}}$  be the output nodes. For Figure 7.13, this means  $x_{\text{out}} = (J)$ . Then by forward propagation,  $J$  is also a function of all node outputs. For Figure 7.13, this means  $J$  is a function of  $x, y, z, a, b$ .

Therefore, at each node  $i$ , we have the derivatives

$$\delta_i = \frac{\partial J}{\partial x_i}(x_i), \quad i = 1, 2, \dots, d.$$

Then  $\delta = (\delta_1, \delta_2, \dots, \delta_d)$  is the *gradient vector*. We first compute the derivatives of  $J$  with respect to the output nodes  $x_{\text{out}}$ , and we assume these derivatives are assembled into a vector  $\delta_{\text{out}}$ .

In Figure 7.13, there is one output node  $J$ , and

$$\delta_J = \frac{\partial J}{\partial J} = 1.$$

Hence  $\delta_{\text{out}} = (1)$ .

We assume the nodes are ordered so that the terminal portion of  $x$  equals  $x_{\text{out}}$  and the terminal portion of  $\delta$  equals  $\delta_{\text{out}}$ ,

```

d = len(x)
m = len(x_out)
x[d-m:] = x_out
delta[d-m:] = delta_out

```

For each  $i, j$ , let

$$g_{ij} = \frac{\partial f_j}{\partial x_i}.$$

Then we have a  $d \times d$  gradient matrix  $g = (g_{ij})$ . When  $(i, j)$  is not an edge,  $g_{ij} = 0$ .

These are the *local* derivatives, not the derivatives obtained by the chain rule. For example, even though we saw above  $\partial J/\partial y = 1$ , here the local derivative is zero, since  $J$  does not depend directly on  $y$ .

For the example above, with  $(x_1, x_2, x_3, x_4, x_5, x_6) = (x, y, z, a, b, J)$ ,

```

g = [ [None]*d for _ in range(d) ]

# note g[i][i] remains undefined

g[0][3] = lambda x,y: 1
g[1][3] = lambda x,y: 1
g[1][4] = lambda y,z: 1 if y>z else 0
g[2][4] = lambda y,z: 1 if z>y else 0
g[3][5] = lambda a,b: b
g[4][5] = lambda a,b: a

```

By the chain rule and (7.4.3),

$$\frac{\partial J}{\partial x_i} = \sum_{i \rightarrow j} \frac{\partial J}{\partial x_j} \cdot \frac{\partial x_j}{\partial x_i} = \sum_{i \rightarrow j} \frac{\partial J}{\partial x_j} \cdot \frac{\partial f_j}{\partial x_i} \cdot w_{ij},$$

so

$$\delta_i = \sum_{i \rightarrow j} \delta_j \cdot g_{ij} \cdot w_{ij}.$$

The code is

```

def derivative(x,delta,g,i):
    if delta[i] != None: return delta[i]
    else:
        return sum([ derivative(x,delta,g,j) *
→ g[i][j](*incoming(x,g,j)) * w[i][j] if g[i][j] != None
→ else 0 for j in range(d) ])

```

This leads to our second version of back propagation,

```

# second version: networks

def backward_prop(x,delta_out,g):
    d = len(g)
    delta = [None]*d
    m = len(delta_out)
    delta[d-m:] = delta_out
    for i in range(d-m): delta[i] = derivative(x,delta,g,i)
    return delta

```

## 7.5 Convex Functions

Let  $f(x)$  be a scalar function of points  $x = (x_1, \dots, x_d)$  in  $\mathbf{R}^d$ . For example, in two dimensions,

$$f(x) = f(x_1, x_2) = \max(|x_1|, |x_2|), \quad f(x) = f(x_1, x_2) = \frac{x_1^2}{4} + x_2^2$$

are scalar functions of points in  $\mathbf{R}^2$ . More generally, if  $Q$  is a  $d \times d$  matrix,  $f(x) = x \cdot Qx$  is such a function. Here, to obtain  $x \cdot Qx$ , we think of the point  $x$  as a vector, then use row-times-column multiplication to obtain  $Qx$ , then take the dot product with  $x$ . We begin with functions in general.

A *level set* of  $f(x)$  is the set

$$E : \quad f(x) = 1.$$

Here we write the level set of level 1. One can have level sets corresponding to any level  $\ell$ ,  $f(x) = \ell$ . In two dimensions, level sets are also called *contour lines*.

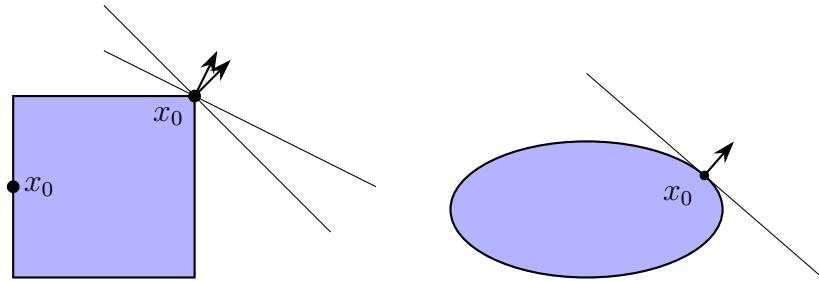


Figure 7.16: Level sets and sublevel sets in two dimensions.

For example, the covariance ellipsoids  $x \cdot Qx = 1$  are level sets. In two dimensions, the square and ellipse in Figure 7.16 are level sets

$$\max(|x_1|, |x_2|) = 1, \quad \frac{x_1^2}{4} + x_2^2 = 1.$$

The contour lines of

$$f(x) = f(x_1, x_2) = \frac{x_1^2}{16} + \frac{x_2^2}{4}$$

are in Figure 7.17.

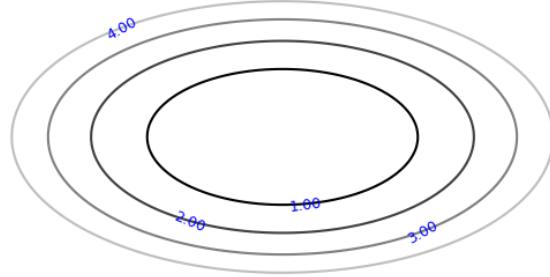
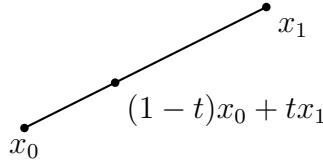


Figure 7.17: Contour lines in two dimensions.



A *sublevel set* of  $f(x)$  is the set

$$E : \quad f(x) \leq 1.$$

Figure 7.18: Line segment  $[x_0, x_1]$ .

Here we write the sublevel set of level 1. One can have sublevel sets corresponding to any level  $c$ ,  $f(x) \leq c$ . For example, in Figure 7.16, the (blue) interior of the square, together with the square itself, is a sublevel set. Similarly, the interior of the ellipse, together with the ellipse itself, is a sublevel set. The interiors of the ellipsoids, together with the ellipsoids themselves, in Figure 7.22 are sublevel sets. Note we always consider the level set to be part of the sublevel set.

The level set  $f(x) = 1$  is the *boundary* of the sublevel set  $f(x) \leq 1$ . Thus the square and the ellipse in Figure 7.16 are boundaries of their respective sublevel sets, and the covariance ellipsoid  $x \cdot Qx = 1$  is the boundary of the sublevel set  $x \cdot Qx \leq 1$ .



Given points  $x_0$  and  $x_1$  in  $\mathbf{R}^d$ , the *line segment* joining them is

$$[x_0, x_1] = \{(1-t)x_0 + tx_1 : 0 \leq t \leq 1\}.$$

A scalar function  $f(x)$  is *convex* if<sup>1</sup> for any two points  $x_0$  and  $x_1$  in  $\mathbf{R}^d$ ,

$$f((1-t)x_0 + tx_1) \leq (1-t)f(x_0) + tf(x_1), \quad \text{for } 0 \leq t \leq 1. \quad (7.5.1)$$

This says the line segment joining any two points  $(x_0, f(x_0))$  and  $(x_1, f(x_1))$  on the graph of  $f(x)$  lies above the graph of  $f(x)$ . For example, in two dimensions, the function  $f(x) = f(x_1, x_2) = x_1^2 + x_2^2/4$  is convex because its graph is the paraboloid in Figure 7.19.

More generally, given points  $x_1, x_2, \dots, x_N$ , a linear combination

$$t_1x_1 + t_2x_2 + \cdots + t_Nx_N$$

---

<sup>1</sup>We only consider convex functions that are continuous.

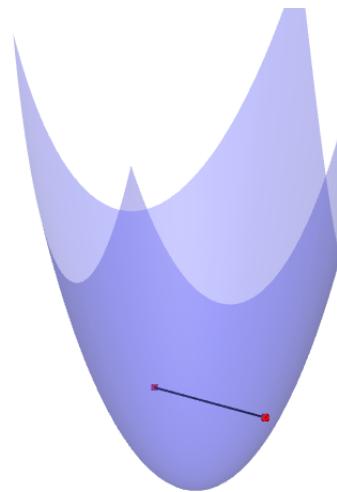


Figure 7.19: Convex: The line segment lies above the graph.

is a *convex combination* if  $t_1, t_2, \dots, t_N$  are nonnegative, and

$$t_1 + t_2 + \cdots + t_N = 1.$$

For example, if  $0 \leq t \leq 1$ ,  $(1 - t)x_0 + tx_1$  is a convex combination of  $x_0$  and  $x_1$ . Then a convex function also satisfies

$$f(t_1x_1 + \cdots + t_Nx_N) \leq t_1f(x_1) + \cdots + t_Nf(x_N), \quad (7.5.2)$$

for any convex combination.



Recall (§2.2) a *nonnegative* matrix is a symmetric matrix  $Q$  satisfying  $x \cdot Qx \geq 0$  for all  $x$ , and every such matrix is the covariance matrix of some dataset. This is equivalent to the nonnegativity of the eigenvalues of  $Q$ . When the eigenvalues of  $Q$  are positive,  $Q$  is invertible.

### Quadratic is Convex

If  $Q$  is a nonnegative matrix and  $b$  is a vector, then

$$f(x) = \frac{1}{2}x \cdot Qx - b \cdot x$$

is a convex function. When  $Q$  is invertible,  $f(x)$  is strictly convex.

This was derived in the previous section, but here we present a more geometric proof.

To derive this result, let  $x_0$  and  $x_1$  be any points, and let  $v = x_1 - x_0$ . Then  $x_0 + tv = (1-t)x_0 + tx_1$  and  $x_1 = x_0 + v$ . Let  $g_0 = Qx_0 - b$ . By (7.3.6),

$$f(x_0 + tv) = f(x_0) + tv \cdot (Qx_0 - b) + \frac{1}{2}t^2v \cdot Qv = f(x_0) + tv \cdot g_0 + \frac{1}{2}t^2v \cdot Qv. \quad (7.5.3)$$

Inserting  $t = 1$  in (7.5.3), we have  $f(x_1) = f(x_0) + v \cdot g_0 + v \cdot Qv/2$ . Since  $t^2 \leq t$  for  $0 \leq t \leq 1$  and  $v \cdot Qv \geq 0$ , by (7.5.3),

$$\begin{aligned} f((1-t)x_0 + tx_1) &= f(x_0 + tv) \\ &\leq f(x_0) + tv \cdot g_0 + \frac{1}{2}tv \cdot Qv \\ &= (1-t)f(x_0) + tf(x_0) + tv \cdot g_0 + \frac{1}{2}tv \cdot Qv \\ &= (1-t)f(x_0) + tf(x_1). \end{aligned}$$

When  $Q$  is invertible, then  $v \cdot Qv > 0$ , and we have strict convexity.



A *convex set* is a subset  $E$  in  $\mathbf{R}^d$  that contains the line segment joining any two points in it: *If  $x_0$  and  $x_1$  are in  $E$ , then the line segment  $[x_0, x_1]$  is in  $E$ .* To be consistent with sublevel sets, we only consider convex sets that contain their boundaries.

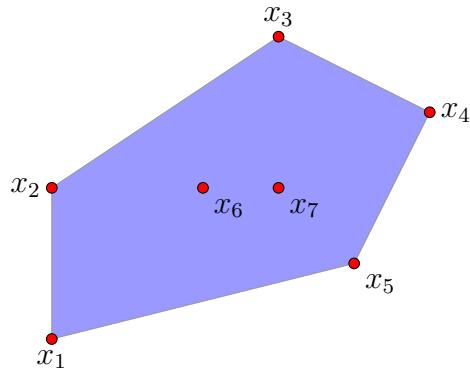
More generally, given points  $x_1, x_2, \dots, x_N$  in  $E$ , the convex combination

$$x = t_1x_1 + t_2x_2 + \cdots + t_Nx_N$$

is also in  $E$ . The set of all convex combinations of  $x_1, x_2, \dots, x_N$  is the *convex hull* of  $x_1, x_2, \dots, x_N$  (Figure 7.20). If  $E$  is convex and contains  $x_1, x_2, \dots, x_N$ , then  $E$  contains their convex hull.

The interiors of the square and the ellipse in Figure 7.16, together with their boundaries, are convex sets. The interior of the ellipsoid in Figure 7.22, together with the ellipsoid, is a convex set.

The following code generates convex hulls,

Figure 7.20: Convex hull of  $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ .

```
from scipy.spatial import ConvexHull
from numpy import *
from numpy.random import *

rng = default_rng()

# 30 random points in 2-D
points = rng.random((30, 2))

hull = ConvexHull(points)
```

and this plots the facets of the convex hull

```
from matplotlib.pyplot import *

plot(points[:,0], points[:,1], 'o')
for facet in hull.simplices:
    plot(points[facet,0], points[facet,1], 'k-')

facet = hull.simplices[0]
plot(points[facet, 0], points[facet, 1], 'r--')
grid()
show()
```

resulting in Figure 7.21.

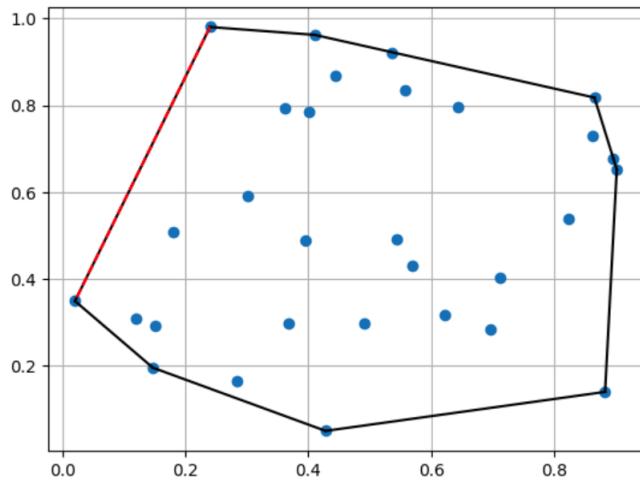


Figure 7.21: A convex hull with one facet highlighted.



If  $f(x)$  is a function, its graph is the set of points  $(x, y)$  in  $\mathbf{R}^{d+1}$  satisfying  $y = f(x)$ , and its *epigraph* is the set of points  $(x, y)$  satisfying  $y \geq f(x)$ . If  $f(x)$  is defined on  $\mathbf{R}^d$ , its sublevel sets are in  $\mathbf{R}^d$ , and its epigraph is in  $\mathbf{R}^{d+1}$ . Then  $f(x)$  is a convex function exactly when its epigraph is a convex set (Figure 7.19). From convex functions, there are other ways to get convex sets:

### Sublevel of Convex is Convex

If  $f(x)$  is a convex function, then the sublevel set

$$E : \quad f(x) \leq 1$$

is a convex set.

This is an immediate consequence of the definition:  $f(x_0) \leq 1$  and  $f(x_1) \leq 1$  implies

$$f((1-t)x_0 + tx_1) \leq (1-t)f(x_0) + tf(x_1) \leq (1-t) + t = 1.$$

From these results, we have

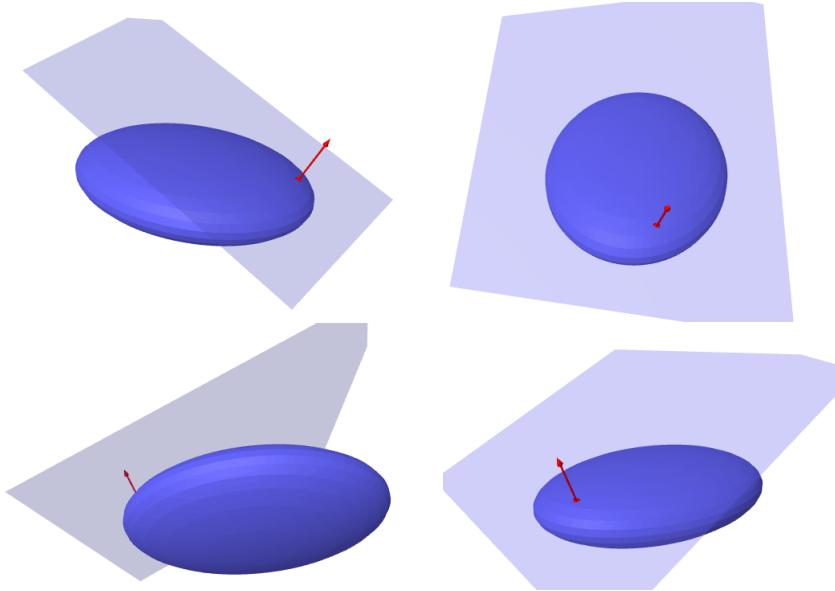


Figure 7.22: Convex set in three dimensions with supporting hyperplane.

### Ellipsoids are Boundaries of Convex Sets

If  $Q$  is a covariance matrix, then  $x \cdot Qx \leq 1$  is a convex set.



Let  $n$  be a nonzero vector in  $\mathbf{R}^d$ . In two dimensions, the vectors orthogonal to  $n$  form a line (Figure 7.23). In three dimensions, the vectors orthogonal to  $n$  form a plane (Figure 7.23). In  $d$  dimensions, these vectors form the orthogonal complement  $n^\perp$  (2.7.5), which is a  $(d - 1)$ -dimensional subspace. This subspace is a hyperplane passing through the origin.

In general, given a point  $x_0$  and a nonzero vector  $n$ , the *hyperplane through  $x_0$  with normal  $n$*  consists of all solutions  $x$  of

$$H : \quad n \cdot (x - x_0) = 0. \quad (7.5.4)$$

A hyperplane *separates* the whole space  $\mathbf{R}^d$  into two half-spaces,

$$n \cdot (x - x_0) < 0 \quad n \cdot (x - x_0) = 0 \quad n \cdot (x - x_0) > 0.$$

The vector  $n$  is the *normal vector* to the hyperplane. Note replacing  $n$  by any nonzero multiple of  $n$  leaves the hyperplane unchanged.

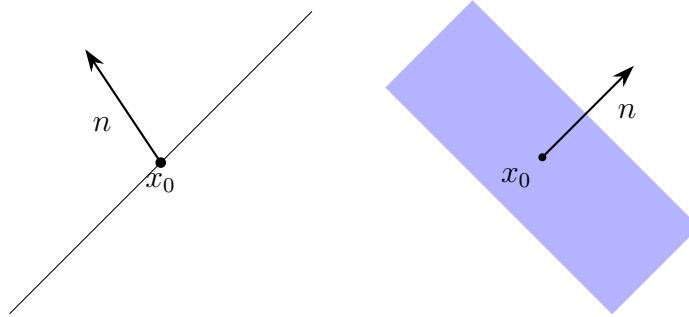


Figure 7.23: Hyperplanes in two and three dimensions.

### Separating Hyperplane

Let  $E$  be a convex set and let  $x^*$  be a point not in  $E$ . Then there is a hyperplane separating  $x^*$  and  $E$ : For some  $x_0$  in  $E$  and nonzero  $n$ ,

$$n \cdot (x - x_0) \leq 0 \quad \text{and} \quad n \cdot (x^* - x_0) > 0. \quad (7.5.5)$$

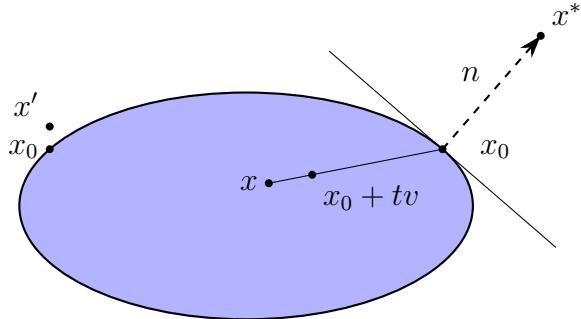


Figure 7.24: Separating hyperplane theorem.

A diagram of the proof is Figure 7.24. Let  $x_0$  be the point in  $E$  closest to  $x^*$ . This means  $x_0$  minimizes  $|x - x^*|^2$  over  $x$  in  $E$ . If  $x$  is in  $E$ , then by convexity, the line segment  $[x_0, x]$  is in  $E$ , hence  $x_0 + tv$ ,  $v = x - x_0$ , is in  $E$  for  $0 \leq t \leq 1$ . Since  $x_0$  is the point of  $E$  closest to  $x^*$ ,

$$|x_0 - x^*|^2 \leq |x_0 + tv - x^*|^2 \quad \text{for} \quad 0 \leq t \leq 1.$$

Expanding, we have

$$|x_0 - x^*|^2 \leq |x_0 - x^*|^2 + 2t(x_0 - x^*) \cdot v + t^2|v|^2, \quad 0 \leq t \leq 1.$$

Cancelling  $|x_0 - x^*|^2$  then, for  $t > 0$ , canceling  $t$ , we obtain

$$0 \leq 2(x_0 - x^*) \cdot v + t|v|^2, \quad 0 \leq t \leq 1.$$

Since this is true for small positive  $t$ , sending  $t \rightarrow 0$ , results in  $v \cdot (x_0 - x^*) \geq 0$ . Setting  $n = x^* - x_0$ , we obtain

$$n \cdot (x - x_0) \leq 0 \quad \text{and} \quad n \cdot (x^* - x_0) > 0.$$



Now suppose  $x_0$  is a point in the boundary of a convex set  $E$ . Since  $x_0$  is in  $E$ , we cannot find a separating hyperplane for  $x^* = x_0$ . In this case, the best we can hope for is a hyperplane passing through  $x_0$ , with  $E$  to one side of the hyperplane:

$$x \text{ in } E \implies (x - x_0) \cdot n \leq 0. \quad (7.5.6)$$

Such a hyperplane is a *supporting hyperplane for  $E$  at  $x_0$* . Figures 7.16 and 7.22 display examples of supporting hyperplanes. Here is the basic result relating convex sets and supporting hyperplanes.

### Supporting Hyperplane for Convex Set

Let  $E$  be a convex set and let  $x_0$  be a point on the boundary of  $E$ . Then there is a supporting hyperplane at  $x_0$ .

If  $x_0$  is in the boundary of  $E$ , there are points  $x'$  not in  $E$  approximating  $x_0$  (Figure 7.24). Applying the separating hyperplane theorem to  $x'$ , and taking the limit  $x' \rightarrow x_0$ , results in a supporting hyperplane at  $x_0$ . We skip the details here.  $\square$

Supporting hyperplanes characterize convex sets in the following sense: If through every point  $x_0$  in the boundary of  $E$ , there is a supporting hyperplane, then  $E$  is convex.



In Figure 7.16, there are multiple supporting hyperplanes. However, at every other point  $a$  on the boundary of the square, there is a unique (up to scalar multiple) supporting hyperplane. For the ellipse or ellipsoid, at every point of the boundary, there is a unique supporting hyperplane.

Now we derive the analogous concepts for convex functions.

Let  $f(x)$  be a function and let  $a$  be a point at which there is a gradient  $\nabla f(a)$ . The *tangent hyperplane* for  $f(x)$  at  $a$  is

$$y = f(a) + \nabla f(a) \cdot (x - a). \quad (7.5.7)$$

### Convex Function Graph Lies Above the Tangent Hyperplane

If  $f(x)$  is convex and has a gradient  $\nabla f(a)$ , then

$$f(x) \geq f(a) + \nabla f(a) \cdot (x - a). \quad (7.5.8)$$

This vector result is obtained by applying the corresponding scalar result in §7.1 to the function  $f(a + tv)$ , where  $v = x - a$ . As in the scalar case, there is a similar result (7.5.16) with tangent paraboloids.



We now address the existence of a global minimizer of a convex function. A (*global*) *minimizer* for  $f(x)$  is a vector  $x^*$  satisfying

$$f(x^*) = \min_x f(x),$$

where the minimum is taken over all vectors  $x$ . A minimizer is the location of the bottom of the graph of the function. For example, the parabola (Figure 7.4) and the relative information (Figure 7.10) both have global minimizers.

We say a function  $f(x)$  is *strictly convex* if  $g(t) = f(a + tv)$  is strictly convex for every point  $a$  and direction  $v$ . This is the same as saying the inequality (7.5.1) is strict for  $0 < t < 1$ .

We say a function  $f(x)$  is proper if the sublevel set  $f(x) \leq c$  is bounded for every level  $c$ . Before we state this precisely, we contrast a level versus a bound.

Let  $f(x)$  be a function. A *level* is a scalar  $c$  determining a sublevel set  $f(x) \leq c$ . A *bound* is a scalar  $C$  determining a bounded set  $|x|^2 \leq C$ .

We say  $f(x)$  is *proper* if for every level  $c$ , there is a bound  $C$  so that

$$f(x) \leq c \implies |x|^2 \leq C. \quad (7.5.9)$$

This is same as saying  $f(x)$  rises to  $+\infty$  as  $|x| \rightarrow \infty$ . The exact formula for the bound  $C$ , which depends on the level  $c$  and the function  $f(x)$ , is not important for our purposes. What matters is the existence of *some* bound  $C$  for *each* level  $c$ .

More vividly, suppose  $x$  is scalar, and think of the graph of  $y = f(x)$  as the cross-section of a river. Then  $f(x)$  is proper if the river never floods its banks, no matter how much it rains. So  $y = \sin x$  is not proper, but  $y = x^2 + \sin x$  is proper.

What does it mean for  $f(x)$  to not be proper? Unpacking the definition,  $f(x)$  is not proper if there is *some* level  $c$  with no corresponding bound  $C$ . This means there is some level  $c$  and a sequence  $x_1, x_2, \dots$  with  $f(x_n) \leq c$  and  $|x_n| \rightarrow \infty$ .

For example, the functions in Figure 7.4 are proper and strictly convex, while the function in Figure 7.5 is proper but neither convex nor strictly convex.

Intuitively, if  $f(x)$  goes up to  $+\infty$  when  $x$  is far away, then its graph must have a minimizer at some point  $x^*$ . The precise statement is below.



The following result describes when the residual (2.6.1) is proper.

### Properness of Residual on Row Space

Let  $A$  be a matrix, and  $b$  a vector with dimensions so that the residual

$$f(x) = |Ax - b|^2$$

is defined. Then  $f(x)$  is proper on the row space of  $A$ .

To see this, suppose  $f(x)$  is not proper. In this case, by (7.5.9), there would be a level  $c$  and a sequence  $x_1, x_2, \dots$  in the row space of  $A$  satisfying  $|x_n| \rightarrow \infty$  and  $f(x_n) \leq c$  for  $n \geq 1$ .

Let  $x'_n = x_n/|x_n|$ . Then  $x'_n$  are unit vectors in the row space of  $A$ , hence  $x'_n$  is a bounded sequence. From §A.2, this implies  $x'_n$  subconverges to some  $x^*$ , necessarily a unit vector in the row space of  $A$ .

By the triangle inequality (2.2.4),

$$|Ax'_n| = \frac{1}{|x_n|} |Ax_n| \leq \frac{1}{|x_n|} (|Ax_n - b| + |b|) \leq \frac{1}{|x_n|} (\sqrt{c} + |b|).$$

Moreover  $Ax'_n$  subconverges to  $Ax^*$ . Since  $|x_n| \rightarrow \infty$ , taking the limit  $n \rightarrow \infty$ ,

$$|Ax^*| = \lim_{n \rightarrow \infty} |Ax'_n| \leq \frac{1}{\infty} (\sqrt{c} + |b|) = 0.$$

Thus  $x^*$  is both in the row space of  $A$  and in the null space of  $A$ . Since the row space and the null space are orthogonal, this implies  $x^* = 0$ . But we can't have  $1 = |x^*| = |0| = 0$ . This contradiction shows there is no such sequence  $x_n$ , and we conclude  $f(x)$  is proper.

When the row space is the source space,

### Properness of Residual

When the  $N \times d$  matrix  $A$  has rank  $d$ ,

$$f(x) = |Ax - b|^2 \tag{7.5.10}$$

is proper on  $\mathbf{R}^d$ .



Now we turn to minimizers.

### Existence of Global Minimizer

Suppose  $f(x)$  is a continuous proper function defined on all of  $\mathbf{R}^d$ . Then  $f(x)$  has a global minimizer  $x^*$ ,

$$f(x^*) \leq f(x). \tag{7.5.11}$$

To see this, pick any point  $a$ . Then, by properness, the sublevel set  $S$  given by  $f(x) \leq f(a)$  is bounded. By continuity of  $f(x)$ , there is a minimizer  $x^*$  (see §A.2). Since for all  $x$  outside the sublevel set, we have  $f(x) > f(a)$ ,  $x^*$  is a global minimizer.

### Existence and Uniqueness of Global Minimizer

Suppose  $f(x)$  is a continuous strictly convex proper function defined on all of  $\mathbf{R}^d$ . Then  $f(x)$  has a unique global minimizer  $x^*$ .

Let  $x_1$  be another global minimizer. Then  $f(x_1) = f(x^*)$ . Let  $x_2 = (x^* + x_1)/2$  be their midpoint. By strict convexity,

$$f(x_2) < \frac{1}{2}(f(x^*) + f(x_1)) = f(x^*),$$

contradicting the fact that  $x^*$  is a global minimizer. Thus there cannot be another global minimizer.



As a consequence,

### Existence of Residual Minimizer

Let  $A$  be a matrix and  $b$  a vector so that the residual

$$f(x) = |Ax - b|^2 \quad (7.5.12)$$

is well-defined. Then there is a residual minimizer  $x^*$  in the row space of  $A$ ,

$$f(x^*) \leq f(x). \quad (7.5.13)$$



The global minimizer  $x^*$  is located by the first derivative test.

### First Derivative Test for Global Minimizer

Let  $f(x)$  be a strictly convex proper function having a gradient  $\nabla f(x)$  at every point. Then the global minimizer  $x^*$  is the unique point satisfying

$$\nabla f(x^*) = 0. \quad (7.5.14)$$

Let  $a$  be any point, and  $v$  any direction, and let  $g(t) = f(a + tv)$ . Then

$$g'(0) = \nabla f(a) \cdot v.$$

If  $a$  is a minimizer, then  $t = 0$  is a minimum of  $g(t)$ , so  $g'(0) = 0$ . Since  $v$  is any direction, this shows  $\nabla f(a) = 0$ .

If there were another point  $b$  satisfying  $\nabla f(b) = 0$ , let  $v = b - a$ . Then  $b = a + v$  and  $g(t)$  is strictly convex in  $t$ , and also  $g'(1) = \nabla f(b) \cdot v = 0$ . By convexity,  $g'(t)$  is increasing in  $t$ . If  $g'(0) = 0$  and  $g'(1) = 0$ , then  $g'(t) = 0$  for  $0 < t < 1$ . This implies  $g(t)$  is a linear on  $0 < t < 1$ , contradicting strict convexity. This establishes the first derivative test.



Suppose the second partials

$$\frac{\partial^2 f}{\partial x_i \partial x_j}, \quad 1 \leq i, j \leq d,$$

exist. Then the *second derivative* of  $f(x)$  is the symmetric matrix

$$D^2 f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \dots \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \dots \\ \dots & \dots & \dots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \dots \end{pmatrix}$$

Replacing  $x$  by  $x + tv$  in (7.3.3), we have

$$\frac{d}{dt} f(x + tv) = \nabla f(x + tv) \cdot v.$$

Differentiating and using the chain rule again,

### Second Directional Derivative

The second derivative  $Q = D^2 f(x)$  satisfies

$$\frac{d^2}{dt^2} \Big|_{t=0} f(x + tv) = v \cdot Qv. \quad (7.5.15)$$

In particular,  $f(x)$  is convex if the second directional derivative is non-negative for all  $x$  and  $v$ .

This implies

### Second Derivative Test for Multi-variable Strict Convexity

Suppose  $f(x)$  has a second derivative  $D^2 f(x)$ , and assume  $D^2 f(x)$  is positive definite, at every  $x$ . Then  $f(x)$  is strictly convex.

An important example of a strictly convex proper function is  $f(x) = x \cdot Qx/2 - b \cdot x$  when  $Q > 0$  (§7.3). Also (§8.5) loss functions in linear regression and logistic regression are strictly convex and proper under the right assumptions.



Recall  $m \leq Q \leq L$  means the eigenvalues of the symmetric matrix  $Q$  are between  $L$  and  $m$ . The following is the multi-variable version of (7.1.10). The proof is the same as in the scalar case.

### Second Derivative Bounds

If  $m \leq D^2 f(x) \leq L$ , then

$$\frac{m}{2}|x - a|^2 \leq f(x) - f(a) - \nabla f(a) \cdot (x - a) \leq \frac{L}{2}|x - a|^2. \quad (7.5.16)$$

If we choose  $a = x^*$ , where  $x^*$  is the global minimizer, then by (7.5.14), we see the graph of  $f(x)$  lies between two quadratics globally.

### Upper and Lower Paraboloids

If  $m \leq D^2 f(x) \leq L$  and  $x^*$  is the global minimum, then

$$\frac{m}{2}|x - x^*|^2 \leq f(x) - f(x^*) \leq \frac{L}{2}|x - x^*|^2. \quad (7.5.17)$$



We describe the convex dual in the multi-variable setting (the single-variable case was done in (7.1.13)). If  $f(x)$  is a scalar convex function of  $x$ , and  $x = (x_1, x_2, \dots, x_d)$  has  $d$  features, the *convex dual* is

$$g(p) = \max_x (p \cdot x - f(x)). \quad (7.5.18)$$

Here the maximum is over all vectors  $x$ , and  $p = (p_1, p_2, \dots, p_d)$ , the *dual variable*, also has  $d$  features. We will work in situations where a maximizer exists in (7.5.18).

Let  $Q > 0$  be a positive matrix. The simplest example is

$$f(x) = \frac{1}{2}x \cdot Qx \implies g(p) = \frac{1}{2}p \cdot Q^{-1}p.$$

This is established by the identity

$$\frac{1}{2}(p - Qx) \cdot Q^{-1}(p - Qx) = \frac{1}{2}p \cdot Q^{-1}p - p \cdot x + \frac{1}{2}x \cdot Qx. \quad (7.5.19)$$

To see this, since the left side of (7.5.19) is greater or equal to zero, we have

$$\frac{1}{2}p \cdot Q^{-1}p - p \cdot x + \frac{1}{2}x \cdot Qx \geq 0.$$

Since (7.5.19) equals zero iff  $p = Qx$ , we are led to (7.5.18).

Moreover, switching  $p \cdot Q^{-1}p$  with  $x \cdot Qx$ , we also have

$$f(x) = \max_p (p \cdot x - g(p)). \quad (7.5.20)$$

Thus the convex dual of the convex dual of  $f(x)$  is  $f(x)$ . In §7.6, we compute the convex dual of the partition function.

If  $x$  is a maximizer in (7.5.18), then the derivative is zero,

$$0 = \nabla_x(p \cdot x - f(x)) \implies p = \nabla_x f(x).$$

Here  $\nabla$  is with respect to  $x$ . The maximizer  $x = x(p)$  depends on  $p$ , so by the chain rule

$$\begin{aligned} \nabla_p g(p) &= \nabla(p \cdot x(p) - f(x(p))) \\ &= x + p \nabla x(p) - \nabla f(x) \nabla x(p) = x + (p - \nabla f(x)) \nabla x(p) = x. \end{aligned}$$

Here  $\nabla$  is with respect to  $p$  and, since  $x = x(p)$  is vector-valued,  $\nabla x(p)$  is a  $d \times d$  matrix. We conclude

$$p = \nabla_x f(x) \iff x = \nabla_p g(p).$$

Thus the vector-valued function  $\nabla f(x)$  is the inverse of the vector-valued function  $\nabla g(p)$ , or

$$\nabla g(\nabla f(x)) = x.$$

Differentiating, we obtain

$$D^2 g(\nabla f(x)) D^2 f(x) = I.$$

This yields

### Second Derivatives of Dual Functions

Let  $f(x)$  be a strictly convex function with second derivative  $D^2 f(x)$ , and let  $g(p)$  be its convex dual. Then

$$D^2 g(p) = (D^2 f(x))^{-1}, \quad p = \nabla f(x).$$

Moreover, if  $m \leq D^2 f(x) \leq L$ , then

$$\frac{1}{L} \leq D^2 g(p) \leq \frac{1}{m}.$$

Using this, and writing out (7.5.16) for  $g(p)$  instead of  $f(x)$  (we skip the details) yields

### Dual Second Derivative Bounds

Let  $p = \nabla f(x)$  and  $q = \nabla f(a)$ . If  $f(x)$  is convex and  $m \leq D^2 f(x) \leq L$ , then

$$\frac{1}{2m}|p - q|^2 \geq f(x) - f(a) - q \cdot (x - a) \geq \frac{1}{2L}|p - q|^2. \quad (7.5.21)$$

This is used in gradient descent.



Now let  $f(x)$  be strongly convex in the sense  $m \leq D^2 f(x) \leq L$ . Then we have the vector version of (7.1.11).

### Coercivity of the Gradient

Let  $p = \nabla f(x)$  and  $q = \nabla f(a)$ . If  $m \leq D^2 f(x) \leq L$ , then

$$(p - q) \cdot (x - a) \geq \frac{mL}{m + L}|x - a|^2 + \frac{1}{m + L}|p - q|^2. \quad (7.5.22)$$

This is derived by using (7.5.21), the details are in [3]. This result is used in gradient descent.

## 7.6 Multinomial Probability

In multinomial probability, there are  $d$  classes or categories, and  $p$  is a vector of probabilities,  $p = (p_1, p_2, \dots, p_d)$ . Here  $p_k$  is the probability we are in class  $k$ . Then each  $p_k$  is nonnegative,  $p_k \geq 0$ , and the sum is one,

$$p_1 + p_2 + \dots + p_d = 1.$$

The *partition function* is

$$Z(z) = \log(e^{z_1} + e^{z_2} + \dots + e^{z_d}). \quad (7.6.1)$$

Then  $Z$  is a function of  $d$  scalar variables  $z = (z_1, z_2, \dots, z_d)$ . If we insert  $z = 0$ , we obtain  $Z(0) = \log d$ .

Let

$$\mathbf{1} = (1, 1, \dots, 1).$$

Then

$$p \cdot \mathbf{1} = \sum_{k=1}^d p_k = 1.$$

Because

$$Z(z + a\mathbf{1}) = Z(z_1 + a, z_2 + a, \dots, z_d + a) = Z(z) + a,$$

$Z$  is not bounded below and does not have a minimum.



The *softmax* function is the vector-valued function  $q = \sigma(z)$  with components

$$q_k = \sigma_k(z) = \frac{e^{z_k}}{e^{z_1} + e^{z_2} + \dots + e^{z_d}} = \frac{e^{z_k}}{e^Z}, \quad k = 1, 2, \dots, d.$$

By the chain rule, the gradient of the partition function is the softmax function,

$$\nabla Z(z) = \sigma(z). \tag{7.6.2}$$

When  $d = 2$ , the vector softmax function reduces to the scalar logistic function (5.1.12), since

$$\begin{aligned} q_1 &= \frac{e^{z_1}}{e^{z_1} + e^{z_2}} = \frac{1}{1 + e^{-(z_1 - z_2)}} = \sigma(z_1 - z_2), \\ q_2 &= \frac{e^{z_2}}{e^{z_1} + e^{z_2}} = \frac{1}{1 + e^{-(z_2 - z_1)}} = \sigma(z_2 - z_1). \end{aligned}$$

Because of this, the softmax function is the multinomial analog of the logistic function, and we use the same symbol  $\sigma$  to denote both functions.

```
from scipy.special import softmax

z = array([z1, z2, z3])
q = softmax(z)
```



In the previous section, we studied convex functions and the existence and uniqueness of the global minimum. As we saw above,  $Z$  does not have a global minimum over unrestricted  $z$ .

Since  $\sigma(z) = \nabla Z(z)$ , a critical point  $z^*$  of  $Z$  must satisfy  $\sigma(z^*) = 0$ . For  $Z$ , a critical point cannot be unique, because

$$\sigma(z_1, z_2, \dots, z_d) = \sigma(z_1 + a, z_2 + a, \dots, z_d + a),$$

or

$$\sigma(z) = \sigma(z + a\mathbf{1}).$$

To guarantee uniqueness of a global minimum of  $Z$ , we have to restrict attention to the subspace of vectors  $z = (z_1, z_2, \dots, z_d)$  orthogonal to  $\mathbf{1}$ , the vectors satisfying

$$z_1 + z_2 + \dots + z_d = 0.$$

Now suppose  $z$  is orthogonal to  $\mathbf{1}$ . Since the exponential function is convex,

$$\frac{e^Z}{d} = \frac{1}{d} \sum_{k=1}^d e^{z_k} \geq \exp\left(\frac{1}{d} \sum_{k=1}^d z_k\right) = e^0 = 1.$$

This establishes

### Restricted Global Minimum of the Partition Function

If  $z$  satisfies  $z_1 + z_2 + \dots + z_d = 0$ , then  $Z(z) \geq Z(0) = \log d$ .



The inverse of the softmax function is obtained by solving  $p = \sigma(z)$  for  $z$ , obtaining

$$z_k = Z + \log p_k, \quad k = 1, 2, \dots, d. \quad (7.6.3)$$

Define

$$\log p = (\log p_1, \log p_2, \dots, \log p_d).$$

Then the inverse of  $p = \sigma(z)$  is

$$z = Z\mathbf{1} + \log p. \quad (7.6.4)$$



The function

$$I(p) = p \cdot \log p = \sum_{k=1}^d p_k \log p_k \quad (7.6.5)$$

is the (*absolute*) *information*. Since  $0 \leq p \leq 1$ ,  $\log p \leq 0$ , hence  $I(p) \leq 0$ .

Since  $\log$  is concave,

$$\sum_{k=1}^d p_k \log(e^{z_k}) \leq \log \left( \sum_{k=1}^d p_k e^{z_k} \right).$$

This implies

$$\begin{aligned} p \cdot z &= \sum_{k=1}^d p_k z_k = \sum_{k=1}^d p_k \log(e^{z_k}) \\ &\leq \log \left( \sum_{k=1}^d p_k e^{z_k} \right) = \log \left( \sum_{k=1}^d e^{z_k + \log p_k} \right) = Z(z + \log p). \end{aligned}$$

Replacing  $z$  by  $z - \log p$ , this establishes

$$I(p) \geq p \cdot z - Z(z). \quad (7.6.6)$$

By (7.6.4), (7.6.6) is an equality when  $p = \sigma(z)$ . We conclude

### Information and Partition are Convex Duals

For all  $p$ ,

$$I(p) = \max_z (p \cdot z - Z(z)).$$

For all  $z$ ,

$$Z(z) = \max_p (p \cdot z - I(p)).$$

The second equality follows by switching  $Z$  and  $I$  in (7.6.6), and repeating the same logic used to derive the first equality.



Inserting  $z = 0$  in (7.6.6), we have

### Absolute Information is Bounded

For all  $p = (p_1, p_2, \dots, p_d)$ ,

$$0 \geq I(p) \geq -\log(d). \quad (7.6.7)$$



The (*absolute*) *entropy*, the analog of (7.2.2), is then

$$H(p) = -I(p) = -\sum_{k=1}^d p_k \log(p_k). \quad (7.6.8)$$

Since

$$D^2 I(p) = \text{diag}\left(\frac{1}{p_1}, \frac{1}{p_2}, \dots, \frac{1}{p_d}\right),$$

we see  $I(p)$  is strictly convex, and  $H(p)$  is strictly concave.

In Python, the entropy is

```
from scipy.stats import entropy

p = array([p_1,p_2,p_3])
entropy(p)
```



Now

$$\frac{\partial^2 Z}{\partial z_j \partial z_k} = \frac{\partial \sigma_j}{\partial z_k} = \begin{cases} \sigma_j - \sigma_j \sigma_k, & \text{if } j = k, \\ -\sigma_j \sigma_k, & \text{if } j \neq k. \end{cases}$$

Hence we have

$$D^2 Z(z) = \nabla \sigma(z) = \text{diag}(q) - q \otimes q, \quad q = \sigma(z). \quad (7.6.9)$$

Let  $\bar{v} = v \cdot q = \sum q_k v_k$ . Since  $Q = D^2 Z(z)$  satisfies

$$v \cdot Qv = \sum_{k=1}^d q_k v_k^2 - (v \cdot q)^2 = \sum_{k=1}^d q_k (v_k - \bar{v})^2,$$

which is nonnegative,  $Q$  is a covariance matrix, and  $Z$  is convex.

In fact  $Z$  is strictly convex in directions  $v$  orthogonal to  $\mathbf{1} = (1, 1, \dots, 1)$ . If  $v \cdot Qv = 0$ , then, since  $q_k > 0$  for all  $k$ ,  $v = \bar{v}\mathbf{1}$ . If  $v$  is orthogonal to  $\mathbf{1}$ , this forces  $\bar{v} = 0$ , which, by using  $v \cdot Qv = 0$  again, implies  $v = 0$ . This shows  $Z$  is strictly convex in directions  $v$  orthogonal to  $\mathbf{1}$ .

Moreover,  $Z$  is proper (7.5.9) in directions orthogonal to  $\mathbf{1}$ . To see this, suppose  $z \cdot \mathbf{1} = 0$  and  $Z(z) \leq c$ . Since  $z_j \leq Z(z)$ , this implies

$$z_j \leq c, \quad j = 1, 2, \dots, d.$$

Given  $1 \leq j \leq d$ , add the inequalities  $z_k \leq c$  over all indices  $k \neq j$ . Since  $z \cdot \mathbf{1} = 0$ ,  $-z_j = \sum_{k \neq j} z_k$ . Hence

$$-z_j \leq (d-1)c, \quad j = 1, 2, \dots, d.$$

Combining the last two inequalities,

$$|z_j| = \max(z_j, -z_j) \leq (d-1)c, \quad j = 1, 2, \dots, d,$$

which implies

$$|z|^2 = \sum_{k=1}^d z_k^2 \leq d(d-1)^2 c^2.$$

Setting  $C = \sqrt{d}(d-1)c$ , we conclude

$$Z(z) \leq c \quad \text{and} \quad z \cdot \mathbf{1} = 0 \quad \implies \quad |z| \leq C. \quad (7.6.10)$$

By (7.5.9), we have shown

### The Partition Function is Proper and Strictly Convex

On the subspace  $z \cdot \mathbf{1} = 0$ ,  $Z(z)$  is proper and strictly convex.



The *relative information* is

$$I(p, q) = \sum_{k=1}^d p_k \log(p_k/q_k). \quad (7.6.11)$$

Here  $p = (p_1, p_2, \dots, p_d)$  and  $q = (q_1, q_2, \dots, q_d)$  are probability distributions.

Let

$$\log q = (\log q_1, \log q_2, \dots, \log q_d).$$

Then

$$p \cdot \log q = \sum_{k=1}^d p_k \log q_k,$$

and

$$I(p, q) = I(p) - p \cdot \log q. \quad (7.6.12)$$

Similarly, the *relative entropy* is

$$H(p, q) = -I(p, q). \quad (7.6.13)$$

In Python, the code

```
from scipy.stats import entropy

p = array([p1,p2,p3])
q = array([q1,q2,q3])
entropy(p,q)
```

returns the relative information, not the relative entropy. See below for more on this terminology confusion.



The *relative partition function* is

$$Z(z, q) = \log \left( \sum_{k=1}^d e^{z_k} q_k \right),$$

If we insert  $q_k = \exp(\log(q_k))$  in the definition of  $Z(z, q)$ , one obtains

$$Z(z, q) = Z(z + \log q).$$

From this, using the change of variable  $z' = z + \log q$ ,

$$\begin{aligned} \max_z (p \cdot z - Z(z, q)) &= \max_z (p \cdot z - Z(z + \log q)) \\ &= \max_{z'} (p \cdot (z' - \log q) - Z(z')) \\ &= \max_z (p \cdot z - Z(z)) - p \cdot \log q \\ &= I(p) - p \cdot \log q \\ &= I(p, q). \end{aligned}$$

As before, this shows

### Relative Information and Relative Partition are Convex Duals

For all  $p$  and  $q$ ,

$$I(p, q) = \max_z (p \cdot z - Z(z, q)).$$

For all  $z$  and  $q$ ,

$$Z(z, q) = \max_p (p \cdot z - I(p, q)).$$



In logistic regression (§8.5), the *output* is  $z$ , the *computed target* is  $q = \sigma(z)$ , the *desired target* is  $p$ , and the *information error function* is  $I(p, q)$ . To compute the information error, by (7.6.4),

$$q = \sigma(z) \implies \log q = z - Z(z)\mathbf{1}.$$

By (7.6.12), this yields

### Fundamental Information Error Identity

For all  $p$  and all  $z$ ,

$$I(p, \sigma(z)) = I(p) - p \cdot z + Z(z). \quad (7.6.14)$$

This identity is the direct analog of (7.5.19). The identity (7.5.19) is relevant to linear regression. Similarly, (7.6.14) relevant to logistic regression.



The *cross-information* is

$$I_{\text{cross}}(p, q) = - \sum_{k=1}^d p_k \log q_k,$$

and the *cross-entropy* is

$$H_{\text{cross}}(p, q) = -I_{\text{cross}}(p, q) = \sum_{k=1}^d p_k \log q_k.$$

The cross-information is usually erroneously called cross-entropy, see the discussion at the end of the section.

Cross-information and relative information are related by

$$I(p, q) = I(p) + I_{\text{cross}}(p, q).$$

A probability vector  $p = (p_1, p_2, \dots, p_d)$  is *one-hot encoded at slot  $j$*  if  $p_j = 1$ . When  $p$  is one-hot encoded at slot  $j$ , then  $p_k = 0$  for  $k \neq j$ .

When  $p$  is one-hot encoded, then  $I(p) = 0$ , so

$$I(p, q) = I_{\text{cross}}(p, q). \quad (7.6.15)$$

In general, from (7.6.14),

$$I_{\text{cross}}(p, \sigma(z)) = -p \cdot z + Z(z).$$



From (7.6.2) and (7.6.14),

$$\nabla_z I(p, \sigma(z)) = q - p, \quad q = \sigma(z). \quad (7.6.16)$$

Since  $I(p, \sigma(z))$  and  $I_{\text{cross}}(p, \sigma(z))$  differ by the constant  $I(p)$ , we also have

$$\nabla_z I_{\text{cross}}(p, \sigma(z)) = q - p, \quad q = \sigma(z),$$

so it doesn't matter whether  $I$  or  $I_{\text{cross}}$  is used in gradient descent (§8.3). Nevertheless, we stick with  $I$ , because the calculations are cleaner with  $I$  (§8.5).



The *relative softmax function* is

$$\sigma_k(z, q) = \frac{e^{z_k} q_k}{e^{z_1} q_1 + e^{z_2} q_2 + \cdots + e^{z_d} q_d}, \quad k = 1, 2, \dots, d.$$

Then the relative version of (7.6.14) is

$$I(p, \sigma(z, q)) = I(p, q) - p \cdot z + Z(z, q).$$



Here is the multinomial analog of (7.2.6). Suppose a dice has  $d$  faces, and suppose the probability of rolling the  $k$ -th face in a single roll is  $q_k$ . Then  $q = (q_1, q_2, \dots, q_d)$  is a probability vector. Let  $p = (p_1, p_2, \dots, p_d)$  be another probability vector. Roll the dice  $n$  times, and let  $P_n(p, q)$  be the probability that the proportion of times the  $k$ -th face is rolled equals  $p_k$ ,  $k = 1, 2, \dots, d$ . Then we have the approximation

$$P_n(p, q) \sim e^{nH(p, q)}, \quad \text{for } n \text{ large.}$$



In the literature, in the industry, in Wikipedia, and in Python, the terminology<sup>2</sup> is confused: The relative information  $I(p, q)$  is almost always called relative entropy.

Since the entropy  $H$  is the negative of the information  $I$ , this is looking at things upside-down. In other settings,  $I(p, q)$  is called the *Kullback–Leibler divergence*, which is not exactly intuitive terminology.

Also, in machine learning,  $I_{\text{cross}}(p, q)$  is called the cross-entropy, not cross-information, continuing the confusion.

Rubbing salt into the wound, in Python, `entropy(p)` is  $H(p)$ , which is correct, but `entropy(p, q)` is  $I(p, q)$ , which is incorrect, or at the very least, inconsistent, even within Python.

---

<sup>2</sup>Of course the correct quantities are used, it's the naming that is incorrect.

How does one keep things straight? By remembering that it's convex functions that we like to minimize, not concave functions. In machine learning, loss functions are built to be minimized, and information, in any form, is convex, while entropy, in any form, is concave. Table 7.25 summarizes the situation.

$H = -I$	Information	Entropy
Absolute	$I(p)$	$H(p)$
Cross	$I_{\text{cross}}(p, q)$	$H_{\text{cross}}(p, q)$
Relative	$I(p, q)$	$H(p, q)$
Curvature	Convex	Concave
Error	$I(p, q)$ with $q = \sigma(z)$	

Table 7.25: The third row is the sum of the first and second rows, and the  $H$  column is the negative of the  $I$  column.



# Chapter 8

## Machine Learning

### 8.1 Overview

This first section is an overview of the chapter. Here is a summary of the structure of neural networks.

- A graph consists of nodes and edges (§4.2).
- If each edge has a direction, the graph is directed.
- If each edge has a weight, the graph is weighed.
- In a directed graph, there are input nodes, output nodes, and hidden nodes.
- A node with an activation function is a neuron (§7.4).
- Each neuron has incoming signals and an outgoing signal.
- The outgoing signal is the activation function applied to the incoming signals.
- A network is a weighed directed graph (§4.2) where the nodes are neurons (§7.4).
- A neural network is a network where each activation function is a function of the *sum* of the incoming signals (§8.2).

The goal is to train a neural network. To *train* a neural network means to find weights  $W$  so that the input-output behavior of the network is as close as possible to a given dataset of sample pairs  $(x_k, y_k)$ ,  $k = 1, 2, \dots, N$ . Here is a summary of how neural networks are trained (§8.4).

1. Start with a sample pair  $(x_k, y_k)$  and a weight matrix  $W$ .
2. Using  $x_k$  as incoming signals at the input nodes, compute the network's outgoing signals at all nodes (forward propagation).
3. Compute the error  $J = J(x_k, y_k, W)$  between the outgoing signals at the output nodes and  $y_k$ .
4. Compute the derivatives  $\delta_{\text{out}}$  of  $J$  at the output nodes.
5. Compute the derivatives  $\delta$  of  $J$  at all nodes (back propagation).
6. Then the weight gradient is given by  $\nabla_W J = x \otimes \delta$ .
7. Update  $W$  using gradient descent (§8.3),  $W^+ = W - t \nabla_W J$  (§8.4).
8. Repeat steps 1-7 over all sample pairs  $(x_k, y_k)$ ,  $k = 1, 2, \dots, N$  (§8.9).
9. Repeat step 8 until convergence.

Steps 1-7 is an *iteration*, and step 8 is an *epoch*. An iteration uses a single sample (more generally a batch of samples), and an epoch uses the entire dataset. The *mean error function* over the dataset is

$$J(W) = \sum_{k=1}^N J(x_k, y_k, W).$$

Sometimes  $J(W)$  is normalized by dividing by  $N$ , but this does not change the results. With the dataset given, the mean error is a function of the weights.

A weight matrix  $W^*$  is *optimal* if it is a minimizer of the mean error,

$$J(W^*) \leq J(W), \quad \text{for all } W.$$

*Convergence* means  $W$  is close to  $W^*$ . Now we turn to the details.

## 8.2 Neural Networks

In §7.4, we saw two versions of forward and back propagation. In this section we see a third version. We begin by reviewing the definition of graph and network as given in §4.2 and §7.4.

A *graph* consists of *nodes* and *edges*. Nodes are also called *vertices*, and an edge is an ordered pair  $(i, j)$  of nodes. Because the ordered pair  $(i, j)$  is not the same as the ordered pair  $(j, i)$ , our graphs are *directed*.

The edge  $(i, j)$  is *incoming* at node  $j$  and *outgoing* at node  $i$ . If a node  $j$  has no outgoing edges, then  $j$  is an *output node*. If a node  $i$  has no incoming edges, then  $i$  is an *input node*. If a node is neither an input nor an output, it is a *hidden node*.

We assume our graphs have no cycles: every path terminates at an output node in a finite number of steps.

A graph is *weighed* if a scalar weight  $w_{ij}$  is attached to each edge  $(i, j)$ . If  $(i, j)$  is not an edge, we set  $w_{ij} = 0$ . If a network has  $d$  nodes, the edges are completely specified by the  $d \times d$  *weight matrix*  $W = (w_{ij})$ .

A node with an attached activation function (7.4.2) is a *neuron*. A *network* is a directed weighed graph where the nodes are neurons. In the next paragraph, we define a special kind of network, a neural network.



In a network, in §7.4, the activation function  $f_j$  at node  $j$  was allowed to be any function of the *incoming list* (7.4.1) at node  $j$

$$(w_{1j}x_1, w_{2j}x_2, \dots, w_{dj}x_d).$$

Because  $w_{ij} = 0$  if  $(i, j)$  is not an edge, the nonzero entries in the incoming list at node  $j$  correspond to the edges incoming at node  $j$ .

A *neural network* is a network where every activation function is restricted to be a function of the *sum* of the entries of the incoming list.

For example, all the networks in this section are neural networks, but the network in Figure 7.13 is not a neural network.

Let

$$x_j^- = \sum_{i \rightarrow j} w_{ij}x_i \tag{8.2.1}$$

be the sum of the incoming list at node  $j$ . Then, in a neural network, the outgoing signal at node  $j$  is

$$x_j = f_j(x_j^-) = f_j \left( \sum_{i \rightarrow j} w_{ij} x_i \right). \quad (8.2.2)$$

If the network has  $d$  nodes, the *outgoing vector* is

$$x = (x_1, x_2, \dots, x_d),$$

and the *incoming vector* is

$$x^- = (x_1^-, x_2^-, \dots, x_d^-).$$

In a network, in §7.4,  $x_j^-$  was a list or vector; in a neural network,  $x_j^-$  is a scalar.

Let  $W$  be the weight matrix. If the network has  $d$  nodes, the *activation vector* is

$$f = (f_1, f_2, \dots, f_d).$$

Then a neural network may be written in vector-matrix form

$$x = f(W^t x).$$

However, this representation is more useful when the network has structure, for example in a dense shallow layer (8.2.12).



A *perceptron* is a network of the form

$$y = f(w_1 x_1 + w_2 x_2 + \dots + w_d x_d) = f(w \cdot x)$$

(Figure 8.1). This is the simplest neural network.

Thus a perceptron is a linear function followed by an activation function. By our definition of neural network,

### Neural Network

Every neural network is a combination of perceptrons.

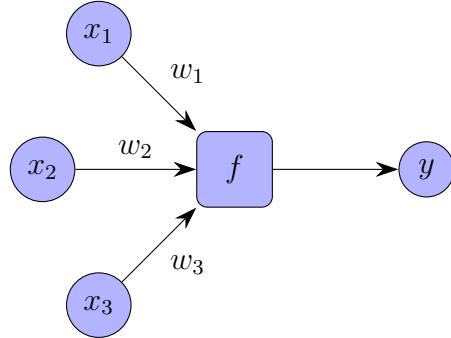


Figure 8.1: A perceptron with activation function  $f$ .



When an input  $x_0$  is fixed to equal 1,  $x_0 = 1$ , the corresponding weight  $w_0$  is called a *bias*,

$$y = f(w_1x_1 + w_2x_2 + \cdots + w_dx_d + w_0) = f(w \cdot x + w_0).$$

There is no computational advantage in separating out the bias.

Nevertheless, the bias is important, as it shifts the threshold in the activation function.

In §5.1, Bayes theorem is used to express a conditional probability in terms of a perceptron,

$$\text{Prob}(H \mid x) = \sigma(w \cdot x + w_0).$$

This is a basic example of how a perceptron computes probabilities.



Perceptrons gained wide exposure after Minsky and Papert's famous 1969 book [18], from which Figure 8.2 is taken.

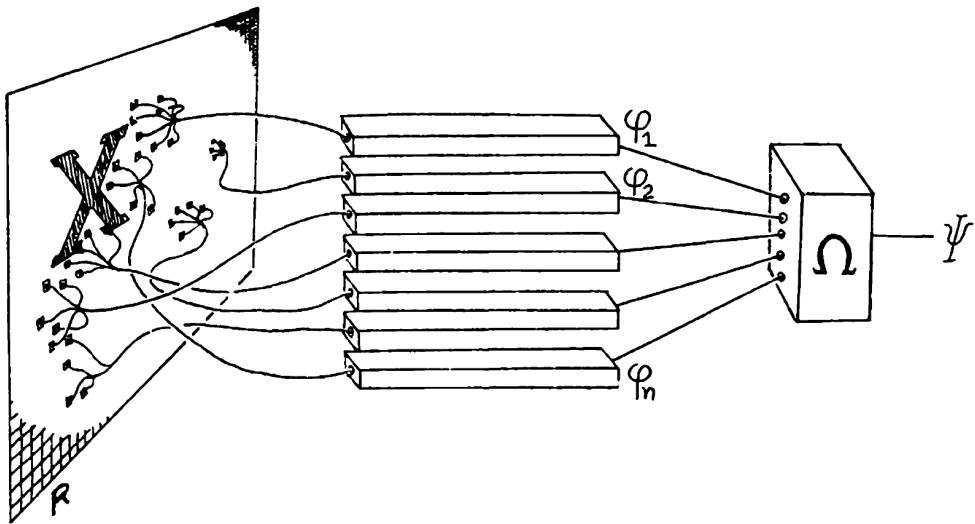


Figure 8.2: Perceptrons in parallel [18].



Here is a listing of common activation functions.

- The identity function,

$$\text{id}(z) = z$$

and its derivative  $\text{id}' = 1$ .

- The binary output,

$$\text{bin}(z) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z < 0, \end{cases}$$

and its derivative  $\text{bin}' = 0, z \neq 0$ , and  $\text{bin}'(0)$  undefined.

- The logistic or sigmoid function (Figure 5.2)

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

and its derivative  $\sigma' = \sigma(1 - \sigma)$ .

- The hyperbolic tangent function

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

and its derivative  $\tanh' = 1 - \tanh^2$ .

- The rectified linear unit relu,

$$\text{relu}(z) = \begin{cases} z & \text{if } z > 0, \\ 0 & \text{if } z < 0, \end{cases}$$

and its derivative  $\text{relu}' = \text{bin}$ .

Here is the code

```
# activation functions

def relu(z): return 0 if z < 0 else z
def bin(z): return 0 if z < 0 else 1
def sigmoid(z): return 1/(1+exp(-z))
def id(z): return z
# tanh already part of numpy
def one(z): return 1
def zero(z): return 0

# derivative of relu is bin
# derivative of bin is zero
# derivative of s=sigmoid is s*(1-s)
# derivative of id is one
# derivative of tanh is 1-tanh**2

def D_relu(z): return bin(z)
def D_bin(z): return 0
def D_sigmoid(z): return sigmoid(z)*(1-sigmoid(z))
def D_id(z): return 1
def D_relu(z): return bin(z)
def D_tanh(z): return 1 - tanh(z)**2

der_dict = { relu:D_relu, id:D_id, bin:D_bin,
```

→ sigmoid:D\_sigmoid, tanh: D\_tanh}



The neural network in Figure 8.3 has weight matrix

$$W = \begin{pmatrix} 0 & 0 & w_{13} & w_{14} & 0 & 0 \\ 0 & 0 & w_{23} & w_{24} & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{35} & w_{36} \\ 0 & 0 & 0 & 0 & w_{45} & w_{46} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (8.2.3)$$

and activation functions  $f_3, f_4, f_5, f_6$ . Here 1 and 2 are plain nodes, and 3, 4, 5, 6 are neurons.

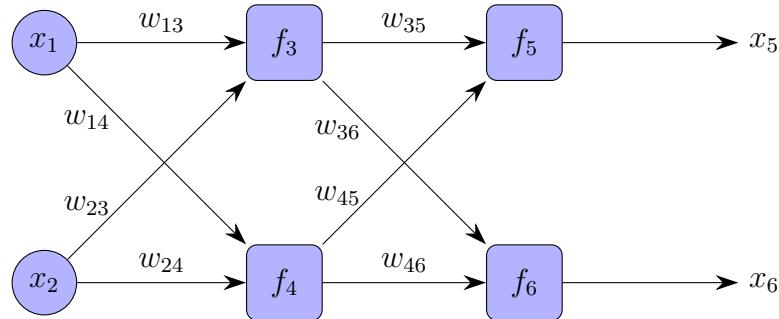


Figure 8.3: Network of neurons.

Let  $x_{\text{in}}$  and  $x_{\text{out}}$  be the outgoing vectors corresponding to the input and output nodes. Then the network in Figure 8.3 has outgoing vectors

$$x = (x_1, x_2, x_3, x_4, x_5, x_6), \quad x_{\text{in}} = (x_1, x_2), \quad x_{\text{out}} = (x_5, x_6).$$

Here are the incoming and outgoing signals at each of the four neurons  $f_3, f_4, f_5, f_6$ .

Neuron	Incoming	Outgoing
$f_3$	$x_3^- = w_{13}x_1 + w_{23}x_2$	$x_3 = f_3(w_{13}x_1 + w_{23}x_2)$
$f_4$	$x_4^- = w_{14}x_1 + w_{24}x_2$	$x_4 = f_4(w_{14}x_1 + w_{24}x_2)$
$f_5$	$x_5^- = w_{35}x_3 + w_{45}x_4$	$x_5 = f_5(w_{35}x_3 + w_{45}x_4)$
$f_6$	$x_6^- = w_{36}x_3 + w_{46}x_4$	$x_6 = f_6(w_{36}x_3 + w_{46}x_4)$

Table 8.4: Incoming and Outgoing signals.



Now we specialize the forward propagation code in §7.4 to neural networks. The key diagram is Figure 8.5.

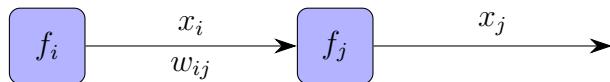


Figure 8.5: Forward and back propagation between two neurons.

Assume the activation function at node  $j$  is `activate[j]`. By (8.2.1) and (8.2.2), the code is

```
def incoming(x,w,j):
    return sum([ outgoing(x,w,i)*w[i][j] if w[i][j] != None
    → else 0 for i in range(d) ])

def outgoing(x,w,j):
    if x[j] != None: return x[j]
    else: return activate[j](incoming(x,w,j))
```

We assume the nodes are ordered so that the initial portion of  $x$  equals  $x_{\text{in}}$ ,

```
m = len(x_in)
x[:m] = x_in
```

Here is the third version of forward propagation.

```
# third version: neural networks

def forward_prop(x_in,w):
    d = len(w)
    x = [None]*d
    m = len(x_in)
    x[:m] = x_in
    for j in range(m,d): x[j] = outgoing(x,w,j)
    return x
```



For Figure 8.3, we define a weight matrix as follows,

$$W = \begin{pmatrix} 0 & 0 & 0.1 & -2.0 & 0 & 0 \\ 0 & 0 & 0.1 & -2.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.3 & -0.3 \\ 0 & 0 & 0 & 0 & 0.22 & 0.22 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (8.2.4)$$

and activation functions

```
activate = [None]*d

activate[2] = relu
activate[3] = id
activate[4] = sigmoid
activate[5] = tanh
```

The code for  $W$  is

```
w = [ [None]*d for _ in range(d) ]

# remember in Python, index starts from 0
```

```
w[0][2] = w[1][2] = 0.1
w[0][3] = w[1][3] = -2.0
w[2][4] = w[2][5] = -0.3
w[3][4] = w[3][5] = 0.22
```

Then the code

```
x_in = [1.5, 2.5]
x = forward_prop(x_in, w)
```

returns the outgoing vector

$$x = (1.5, 2.5, 0.4, -8.0, 0.132, -0.954). \quad (8.2.5)$$

From this, the incoming vector is

$$x^- = (0, 0, 0.4, -8.0, -1.88, -1.88).$$

and the outputs are

$$x_{\text{out}} = (0.132, -0.954).$$



Let

$$y_1 = 0.427, \quad y_2 = -0.288, \quad y = (y_1, y_2)$$

be targets, and let  $J(x_{\text{out}}, y)$  be a function of the outputs  $x_{\text{out}}$  of the output nodes, and the targets  $y$ . For example, for Figure 8.3,  $x_{\text{out}} = (x_5, x_6)$  and we may take  $J$  to be the *mean square error function* or *mean square loss*

$$J(x_{\text{out}}, y) = \frac{1}{2}(x_5 - y_1)^2 + \frac{1}{2}(x_6 - y_2)^2, \quad (8.2.6)$$

The code for this  $J$  is

```
def J(x_out,y):
    m = len(y)
    return sum([(x_out[i] - y[i])**2/2 for i in range(m)])
```

and the code

```

y0 = [0.132,-0.954]
y = [0.427, -0.288]

J(x_out,y0), J(x_out,y)

```

returns 0 and 0.266.



By forward propagation,  $J$  is also a function of all nodes. Then, at each node  $j$ , we have the derivatives

$$\frac{\partial J}{\partial x_j^-}, \quad f'_j(x_j^-), \quad \frac{\partial J}{\partial x_j}. \quad (8.2.7)$$

These are the *downstream derivative*, *local derivative*, and *upstream derivative* at node  $j$ . (The terminology reflects the fact that derivatives are computed backward.)

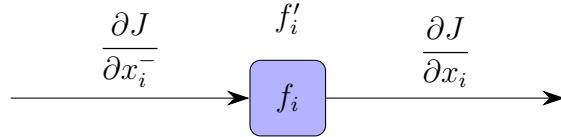


Figure 8.6: Downstream, local, and upstream derivatives at node  $i$ .

From (8.2.2),

$$\frac{\partial x_j}{\partial x_j^-} = f'_j(x_j^-). \quad (8.2.8)$$

By the chain rule and (8.2.8), the key relation between these derivatives is

$$\frac{\partial J}{\partial x_i^-} = \frac{\partial J}{\partial x_i} \cdot f'_i(x_i^-), \quad (8.2.9)$$

or

$$\text{downstream} = \text{upstream} \times \text{local}.$$

```
def local(x,w,i):
    return der_dict[activate[i]](incoming(x,w,i))
```



Let

$$\delta_i = \frac{\partial J}{\partial x_i^-}, \quad i = 1, 2, \dots, d.$$

Then we have the outgoing vector  $x = (x_1, x_2, \dots, x_d)$  and the *downstream gradient vector*  $\delta = (\delta_1, \delta_2, \dots, \delta_d)$ . Strictly speaking, we should write  $\delta_i^-$  for the downstream derivatives. However, in §8.4, we don't need upstream derivatives. Because of this, we will write  $\delta_i$ .

Let  $x_{\text{out}}$  be the output nodes, and let  $\delta_{\text{out}}$  be the downstream derivatives of  $J$  corresponding to  $x_{\text{out}}$ . Then  $\delta_{\text{out}}$  is a function of  $x_{\text{out}}$ ,  $y$ ,  $w$ . We assume the nodes are ordered so that the terminal portions of  $x$  and  $\delta$  equal  $x_{\text{out}}$  and  $\delta_{\text{out}}$  respectively,

```
d = len(x)
m = len(x_out)
x[d-m:] = x_out
delta[d-m:] = delta_out
```



Once we have the incoming vector  $x^-$  and outgoing vector  $x$ , we can differentiate  $J$  and compute the downstream derivatives  $\delta_{\text{out}}$  with respect to each node in  $x_{\text{out}}$ . For example, in Figure 8.3, there are two output nodes  $x_5, x_6$ , and we compute

$$\delta_5 = \frac{\partial J}{\partial x_5^-}, \quad \delta_6 = \frac{\partial J}{\partial x_6^-}, \quad \delta_{\text{out}} = (\delta_5, \delta_6)$$

as follows. Using (8.2.5) and (8.2.6), the upstream derivative is

$$\frac{\partial J}{\partial x_5} = (x_5 - y_1) = -0.294.$$

At node 5, the activation function is  $f_5 = \sigma$ . Since  $\sigma' = \sigma(1 - \sigma)$ , the local derivative at node 5 is

$$\sigma'(x_5^-) = \sigma(x_5^-)(1 - \sigma(x_5^-)) = x_5(1 - x_5) = 0.114.$$

Hence the downstream derivative at node 5 is

$$\delta_5 = \text{upstream} \times \text{local} = -0.294 * 0.114 = -0.0337.$$

Similarly,

$$\delta_6 = -0.059.$$

We conclude

$$\delta_{\text{out}} = (-0.0337, -0.059).$$

The code for this is

```
# delta_out for mean square error

def delta_out(x_out,y,w):
    d = len(w)
    m = len(y)
    return [ (x_out[i] - y[i]) * local(x,w,d-m+i) for i in
            ↪ range(m) ]

delta_out(x_out,y_star,w)
```



We compute  $\delta$  recursively via back propagation as in §7.4. From Figure 8.5 and (8.2.1) and (8.2.8),

$$\begin{aligned}\frac{\partial J}{\partial x_i^-} &= \sum_{i \rightarrow j} \frac{\partial J}{\partial x_j^-} \cdot \frac{\partial x_j^-}{\partial x_i} \cdot \frac{\partial x_i}{\partial x_i^-} \\ &= \left( \sum_{i \rightarrow j} \frac{\partial J}{\partial x_j^-} \cdot w_{ij} \right) \cdot f'_i(x_i^-).\end{aligned}$$

This yields the downstream derivative at node  $i$ ,

$$\delta_i = \left( \sum_{i \rightarrow j} \delta_j \cdot w_{ij} \right) \cdot f'_i(x_i^-). \quad (8.2.10)$$

The code is

```
def downstream(x,delta,w,i):
    if delta[i] != None: return delta[i]
    else:
        upstream = sum([ downstream(x,delta,w,j) * w[i][j] if
            → w[i][j] != None else 0 for j in range(d) ])
        return upstream * local(x,w,i)
```

Using this, we have the third version of back propagation,

```
# third version: neural networks

def backward_prop(x,y,w):
    d = len(w)
    delta = [None]*d
    m = len(y)
    x_out = x[d-m:]
    delta[d-m:] = delta_out(x_out,y_star,w)
    for i in range(d-m): delta[i] = downstream(x,delta,w,i)
    return delta
```

With  $W$ ,  $x$ , and targets  $y$  as above, the code

```
delta = backward_prop(x,y,w)
```

returns

$$\delta = (0.0437, 0.0437, 0.0279, -0.0204, -0.0337, -0.059).$$



Above we computed the upstream, downstream, and local derivatives of  $J$  at a given node (8.2.7). Since the incoming signals  $x_j^-$  depend also on the weights  $w_{ij}$ ,  $J$  also depends on  $w_{ij}$ . By (8.2.1),

$$\frac{\partial x_j^-}{\partial w_{ij}} = x_i,$$

see also Table 8.4. From this,

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial x_j^-} \cdot \frac{\partial x_j^-}{\partial w_{ij}} = \delta_j \cdot x_i.$$

We have shown

### Weight Gradient of Output

If  $(i, j)$  is an edge, then

$$\frac{\partial J}{\partial w_{ij}} = x_i \cdot \delta_j. \quad (8.2.11)$$

This result is key for neural network training (§8.4).



Perceptrons can be assembled in parallel (Figure 8.2). If a network has no hidden nodes, the network is *shallow*. In a shallow network, all nodes are either input nodes or output nodes (Figure 8.7).

A shallow network is *dense* if all input nodes point to all output nodes:  $w_{ij}$  is defined for all  $i, j$ . A shallow network can always be assumed dense by inserting zero weights at missing edges.

Neural networks can also be assembled in series, with each component a *layer* (Figure 8.8). Usually each layer is a dense shallow network. For example, Figure 8.3 consists of two dense shallow networks in layers. We say a network is *deep* if there are multiple layers.

The weight matrix  $W$  (8.2.3) is  $6 \times 6$ , while the weight matrices  $W_1, W_2$  of each of the two dense shallow network layers in Figure 8.3 are  $2 \times 2$ .

In a single shallow layer with  $n$  input nodes and  $m$  output nodes (Figure 8.7), let  $x$  and  $z$  be the layer's input node vector and output node vector. Then  $x$  and  $z$  are  $n$  and  $m$  dimensional respectively, and  $W$  is  $m \times n$ .

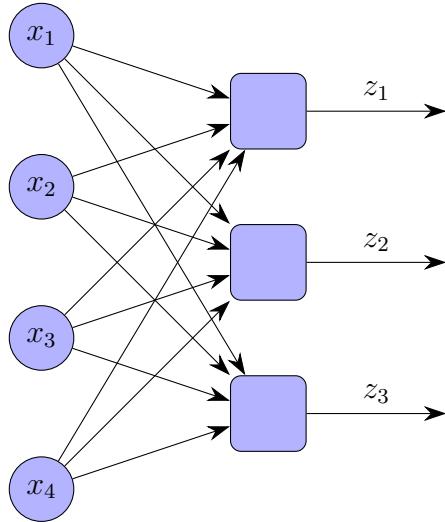


Figure 8.7: A shallow dense layer.

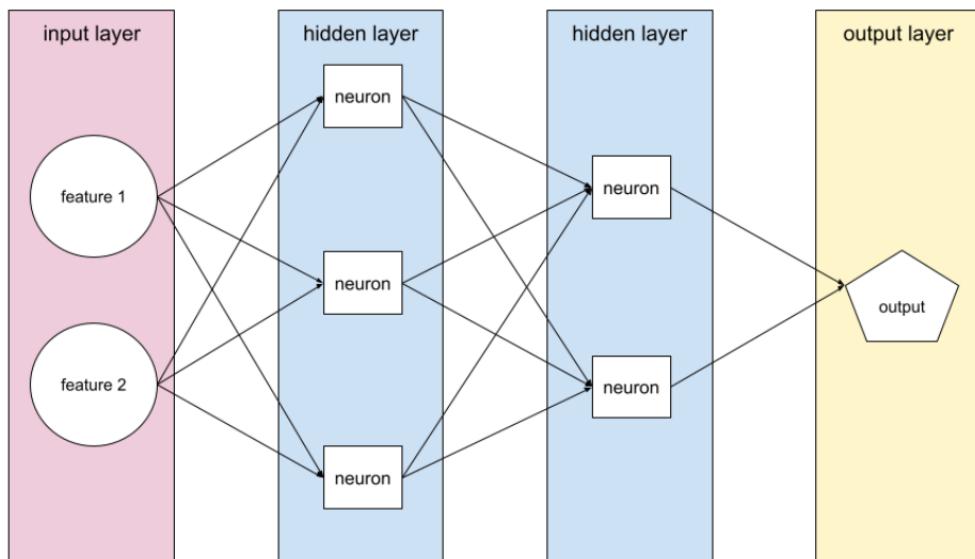


Figure 8.8: Layered neural network [9].

If we have the same activation function  $f$  at every output node, then we

may apply it componentwise,

$$f(z^-) = f(z_1^-, z_2^-, \dots, z_m^-) = (f(z_1^-), f(z_2^-), \dots, f(z_m^-)).$$

Our convention is to let  $w_{ij}$  denote the weight on the edge  $(i, j)$ . With this convention, the formulas (8.2.1), (8.2.2) reduce to the matrix multiplication formulas

$$z^- = W^t x, \quad z = f(W^t x). \quad (8.2.12)$$

Thus a dense shallow network can be thought of as a vector-valued perceptron. This allows for vectorized forward and back propagation.

### 8.3 Gradient Descent

Let  $f(w)$  be a scalar function of a vector  $w = (w_1, w_2, \dots, w_d)$  in  $\mathbf{R}^d$ . A basic problem is to minimize  $f(w)$ , that is, to find or compute a minimizer  $w^*$ ,

$$f(w) \geq f(w^*), \quad \text{for every } w.$$

This goal is so general, that anything concrete one insight one provides towards this goal is widely useful in many settings. The setting we have in mind is  $f = J$ , where  $J$  is the mean error from §8.1.

Usually  $f(w)$  is a measure of cost or lack of compatibility. Because of this,  $f(w)$  is called the *loss function* or *cost function*.

A neural network is a black box with inputs  $x$  and outputs  $y$ , depending on unknown *weights*  $w$ . To *train* the network is to select weights  $w$  in response to training data  $(x, y)$ . The optimal weights  $w^*$  are selected as minimizers of a loss function  $f(w)$  measuring the error between predicted outputs and actual outputs, corresponding to given training inputs.

From §7.5, if the loss function  $f(w)$  is continuous and proper, there is a global minimizer  $w^*$ . If  $f(w)$  is in addition strictly convex,  $w^*$  is unique. Moreover, if the gradient of the loss function is  $g = \nabla f(w)$ , then  $w^*$  is a critical point,  $g^* = \nabla f(w^*) = 0$ .



Let  $g(w)$  be any function of a scalar variable  $w$ . From the definition of derivative (7.1.2), if  $b$  is close to  $a$ , we have the approximation

$$g(b) - g(a) \approx g'(a)(b - a).$$

Inserting  $a = w$  and  $b = w^+$ ,

$$g(w^+) \approx g(w) + g'(w)(w^+ - w).$$

Assume  $w^*$  is a root of  $g(w) = 0$ , so  $g(w^*) = 0$ . If  $w^+$  is close to  $w^*$ , then  $g(w^+)$  is close to zero, so

$$0 \approx g(w) + g'(w)(w^+ - w).$$

Solving for  $w^+$ ,

$$w^+ \approx w - \frac{g(w)}{g'(w)}.$$

Since the global minimizer  $w^*$  satisfies  $f'(w^*) = 0$ , we insert  $g(w) = f'(w)$  in the above approximation,

$$w^+ \approx w - \frac{f'(w)}{f''(w)}.$$

This leads to *Newton's method* of computing approximations  $w_0, w_1, w_2, \dots$  of  $w^*$  using the recursion

$$w_{n+1} = w_n - \frac{f'(w_n)}{f''(w_n)}, \quad n = 1, 2, \dots$$

Because calculating  $f''(w)$  is computationally expensive, *first-order descent methods* replace the second derivative terms  $f''(w_n)$  by constants, known as learning rates.

In the multi-variable case, Newton's method becomes

$$w_{n+1} = w_n - D^2 f(w_n)^{-1} \nabla f(w_n), \quad n = 1, 2, \dots,$$

and the second-derivative term is even more expensive to compute.

These first-order methods, collectively known as *gradient descent*, are the subject of this chapter. In presenting §8.3 and §8.8, we follow [3], [19], [28], [30].



Here is code for Newton's method.

```

from numpy import *

def newton(loss,grad,curv,w,num_iter):
    g = grad(w)
    c = curv(w)
    trajectory = array([[w],[loss(w)]])
    for _ in range(num_iter):
        w -= g/c
        trajectory = column_stack([trajectory,[w,loss(w)]] )
        g = grad(w)
        c = curv(w)
        if allclose(g,0): break
    return trajectory

```

When applied to the function

$$f(w) = w^4 - 6w^2 + 2w,$$

the code returns `trajectory`

```

def loss(w): return w**4 - 6*w**2 + 2*w      # f(w)
def grad(w): return 4*w**3 - 12*w + 2         # f'(w)
def curv(w): return 12*w**2 - 12               # f''(w)

u0 = -2.72204813
w0 = 2.45269774
num_iter = 20
trajectory = newton(loss,grad,curv,w0,num_iter)

```

which can be plotted

```

from matplotlib.pyplot import *

def plot_descent(a,b,loss,curv,delta,trajectory):
    w = arange(a,b,delta)
    plot(w,loss(w),color='red',linewidth=1)
    plot(w,curv(w),"--",color='blue',linewidth=1)
    plot(*trajectory,color='green',linewidth=1)
    scatter(*trajectory,s=10)

```

```
title("num_iter= " + str(len(trajectory.T)))
grid()
show()
```

with the code

```
ylim(-15,10)
delta = .01
plot_descent(u0,w0,loss,curv,delta,trajectory)
```

returning Figure 8.9.

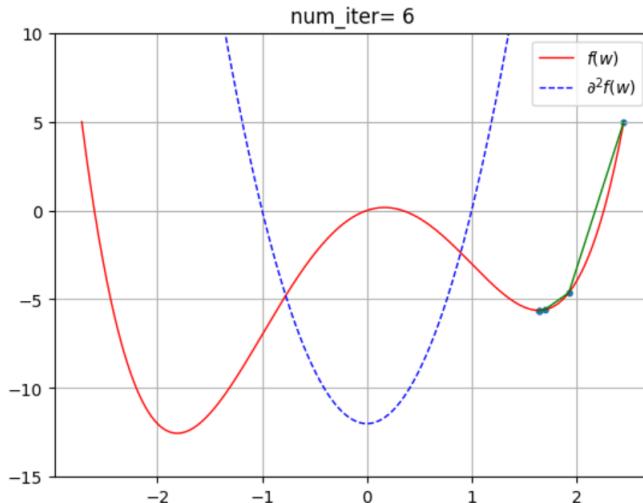


Figure 8.9: Double well newton descent.



A *descent sequence* is a sequence  $w_0, w_1, w_2, \dots$  where the loss function decreases

$$f(w_0) \geq f(w_1) \geq f(w_2) \geq \dots$$

In a descent sequence, the point after the current point  $w = w_n$  is the successive point  $w^+ = w_{n+1}$ , and the point before the current point is the previous point  $w^- = w_{n-1}$ . Then  $(w^-)^+ = w = (w^+)^-$ .

Recall (§7.3) the gradient  $\nabla f(w)$  at a given point  $w$  is the direction of greatest increase of the function, starting from  $w$ . Because of this, it is natural to construct a descent sequence by moving, at any given  $w$ , in the direction  $-\nabla f(w)$  opposite to the gradient.

A *gradient descent* is a descent sequence  $w_0, w_1, w_2, \dots$  where each successive point  $w^+$  is obtained from the previous point  $w$  by moving in the direction opposite to the gradient  $g = \nabla f(w)$  at  $w$ ,

### Basic Gradient Descent Step

$$w^+ = w - t\nabla f(w). \quad (8.3.1)$$

The step-size  $t$ , which determines how far to go in the direction opposite to  $g$ , is the *learning rate*.



Let us unpack (8.3.1), so we understand how it applies to weights in networks (§7.4). In a neural network, weights  $w_1, w_2, \dots$  are attached to edges, and the final outputs are combined into a loss function. As a result, the loss function is a function of the weights,

$$f(w) = f(w_1, w_2, \dots).$$

In (8.3.1),  $w = (w_1, w_2, \dots)$  is the weight vector, consisting all of weights combined into a single vector. By the gradient formula (7.3.2), (8.3.1) is equivalent to

$$\begin{aligned} w_1^+ &= w_1 - t \frac{\partial f}{\partial w_1}, \\ w_2^+ &= w_2 - t \frac{\partial f}{\partial w_2}, \\ &\dots = \dots \end{aligned}$$

In other words,

### Each Weight is Computed Separately

To update a weight in a specific edge using gradient descent, one needs only the derivative of the loss function relative to that specific weight.

Of course, the derivative relative to a specific weight may depend on other derivatives and other weights, when one applies backpropagation (§7.4). This principle also holds for modified gradient descent (§8.8).



In practice, the learning rate is selected by trial and error. Which learning rate does the theory recommend?

Given an initial point  $w_0$ , the *sublevel set at  $w_0$*  (see §7.5) consists of all points  $w$  where  $f(w) \leq f(w_0)$ . Only the part of the sublevel set that is connected to  $w_0$  counts.

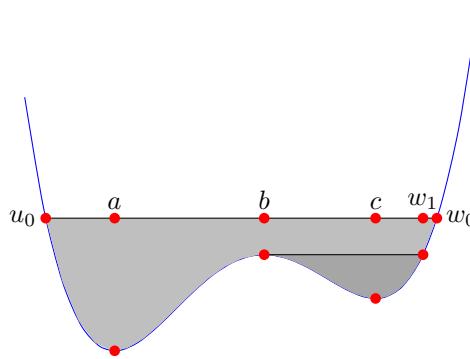


Figure 8.10: Double well cost function and sublevel sets at  $w_0$  and at  $w_1$ .

In Figure 8.10, the sublevel set at  $w_0$  is the interval  $[u_0, w_0]$ . The sublevel set at  $w_1$  is the interval  $[b, w_1]$ . Notice we do not include any points to the left of  $b$  in the sublevel set at  $w_1$ , because points to the left of  $b$  are separated from  $w_1$  by the gap at the point  $b$ .

Suppose the second derivative  $D^2 f(w)$  is never greater than a constant  $L$  on the sublevel set. This means

$$D^2 f(w) \leq L, \quad \text{on } f(w) \leq f(w_0), \tag{8.3.2}$$

in the sense the eigenvalues of  $D^2 f(w)$  are never greater than  $L$ .

Because the second derivative is the derivative of the first derivative,  $D^2f(w)$  measures how fast the gradient  $\nabla f(w)$  changes from point to point. From this point of view,  $D^2f(w)$  is a measure of the curvature of the function  $f(w)$ , and (8.3.2) says the rate of change of the gradient is never greater than  $L$ .

Given such a bound  $L$  on the curvature, If the learning rate  $t$  is no larger than  $1/L$ , we say we are doing *short step* gradient descent. Then we have

### Short Step Gradient Descent

Let  $L$  be as above and  $w^+$  as in (8.3.1). If  $t \leq 1/L$ , then

$$f(w^+) \leq f(w) - \frac{t}{2} |\nabla f(w)|^2. \quad (8.3.3)$$

To see this, fix  $w$  and let  $S$  be the sublevel set  $\{w' : f(w') \leq f(w)\}$ . Since the gradient pushes  $f$  down, for  $t > 0$  small,  $w^+$  stays in  $S$ . Insert  $x = w^+$  and  $a = w$  into the right half of (7.5.16) and simplify. This leads to

$$f(w^+) \leq f(w) - t |\nabla f(w)|^2 + \frac{t^2 L}{2} |\nabla f(w)|^2.$$

Since  $tL \leq 1$  when  $0 \leq t \leq 1/L$ , we have  $t^2 L \leq t$ . This derives (8.3.3).

The curvature of the loss function and the learning rate are inversely proportional. Where the curvature of the graph of  $f(w)$  is large, the learning rate  $1/L$  is small, and gradient descent proceeds in small time steps.



When the sublevel set is bounded, there is a bound  $L$  satisfying (8.3.2). From §7.5, the sublevel set is bounded when  $f(w)$  is proper: Large  $|w|$  implies high cost  $f(w)$ . The graphs in Figures 7.4, 7.5, 8.10, are proper.

In practice, when the loss function is not proper, it is modified by an extra term that forces properness. This is called *regularization*. If the extra term is proportional to  $|w|^2$ , it is *ridge* regularization, and if the extra term is proportional to  $|w|$ , it is *LASSO* regularization.



Now let  $w_0, w_1, w_2, \dots$  be a short-step gradient descent sequence,  $t \leq 1/L$ . By (8.3.3),  $w_n$  remains in the sublevel set  $f(w) \leq f(w_0)$ . If this sublevel set is

bounded,  $w_n$  subconverges to a limit  $w^*$  (Appendix A.2). Inserting  $w = w_n$ ,  $w^+ = w_{n+1}$  in (8.3.3),

$$f(w_{n+1}) \leq f(w_n) - \frac{1}{2L} |\nabla f(w_n)|^2.$$

Since  $f(w_n)$  and  $f(w_{n+1})$  both converge to  $f(w^*)$ , and  $\nabla f(w_n)$  converges to  $\nabla f(w^*)$ , we conclude

$$f(w^*) \leq f(w^*) - \frac{1}{2L} |\nabla f(w^*)|^2.$$

Since this implies  $\nabla f(w^*) = 0$ , we have derived the following.

### Gradient Descent Converges to a Critical Point

Fix an initial weight  $w_0$  and let  $L$  be as above. If the short-step gradient descent sequence starting from  $w_0$  converges to some point  $w^*$ , then  $w^*$  is a critical point.

For example, let  $f(w) = w^4 - 6w^2 + 2w$  (Figures 8.9, 8.10, 8.11). Then

$$f'(w) = 4w^3 - 12w + 2, \quad f''(w) = 12w^2 - 12.$$

Thus the inflection points (where  $f''(w) = 0$ ) are  $\pm 1$  and, in Figure 8.10, the critical points are  $a$ ,  $b$ ,  $c$ .

Let  $u_0$  and  $w_0$  be the points satisfying  $f(w) = 5$  as in Figure 8.11. Then  $u_0 = -2.72204813$  and  $w_0 = 2.45269774$ , so  $f''(u_0) = 76.914552$  and  $f''(w_0) = 60.188$ . Thus we may choose  $L = 76.914552$ . With this  $L$ , the short-step gradient descent starting at  $w_0$  is guaranteed to converge to one of the three critical points. In fact, the sequence converges to the right-most critical point  $c$  (Figure 8.10).

This exposes a flaw in basic gradient descent. Gradient descent may converge to a local minimizer, and miss the global minimizer. In §8.8, modified gradient descent will address some of these shortcomings.

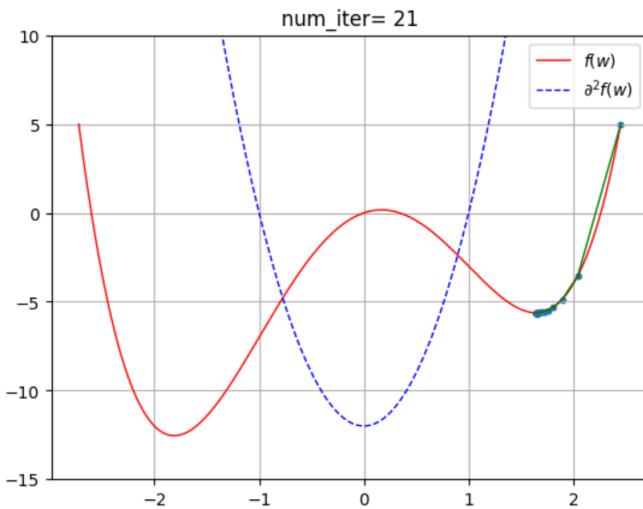


Figure 8.11: Double well gradient descent.

The code for gradient descent is

```
from numpy import *
from matplotlib.pyplot import *

def gd(loss,grad,w,learning_rate,num_iter):
    g = grad(w)
    trajectory = array([[w],[loss(w)]])
    for _ in range(num_iter):
        w -= learning_rate * g
        trajectory = column_stack([trajectory,[w,loss(w)]])
        g = grad(w)
        if allclose(g,0): break
    return trajectory
```

When applied to the double well function  $f(w)$ ,

```
u0 = -2.72204813
w0 = 2.45269774
L = 76.914552
learning_rate = 1/L
```

```

num_iter = 100
trajectory = gd(loss,grad,w0,learning_rate,num_iter)

ylim(-15,10)
delta = .01
plot_descent(u0,w0,loss,curv,delta,trajectory)

```

the code returns Figure 8.11.

## 8.4 Network Training

A neural network with weight matrix  $W$  defines an *input-output map*

$$x_{\text{in}} \rightarrow x_{\text{out}}.$$

Given inputs  $x_{\text{in}}$  and target outputs  $y$ , we seek to modify the weight matrix  $W$  so that the input-output map is

$$x_{\text{in}} \rightarrow y.$$

This is *training*.

Let (§8.2)

$$x^- = (x_1^-, x_2^-, \dots, x_d^-), \quad x = (x_1, x_2, \dots, x_d)$$

be the network's incoming vector and outgoing vector, and let

$$\delta = (\delta_1, \delta_2, \dots, \delta_d)$$

be the downstream gradient vector, relative to some mean error function  $J$ .

From (8.2.1),

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial x_j^-} \cdot \frac{\partial x_j^-}{\partial w_{ij}} = \frac{\partial J}{\partial x_j^-} \cdot x_i = x_i \delta_j. \quad (8.4.1)$$

This we derived as (8.2.11), but here it is again:

### The Weight Gradient of $J$ is a Tensor Product

Let  $w_{ij}$  be the weight along an edge  $(i, j)$ , let  $x_i$  be the outgoing signal from the  $i$ -th node, and let  $\delta_j$  be the downstream derivative of the output  $J$  with respect to the  $j$ -th node. Then the derivative  $\partial J / \partial w_{ij}$  equals  $x_i \delta_j$ . In this partial sense,

$$\nabla_W J = x \otimes \delta. \quad (8.4.2)$$

When  $W$  is the weight matrix between successive layers in a layered neural network (Figure 8.8), (8.4.2) is not partial, it is exactly correct.

Using (8.4.1), we update the weight  $w_{ij}$  using gradient descent

```
def update_weights(x,delta,w,learning_rate):
    d = len(w)
    for i in range(d):
        for j in range(d):
            if w[i][j]:
                w[i][j] = w[i][j] - learning_rate*x[i]*delta[j]
```

The learning rate is discussed in §8.3. The triple

forward propagation  $\rightarrow$  backward propagation  $\rightarrow$  update weights  
is an *iteration*. Starting with a given  $W_0$ , we repeat this iteration until we obtain the target outputs  $y$ . Here is the code.

```
def train_nn(x_in,y,w0,learning_rate,n_iter):
    trajectory = []
    cost = 1
    # build a local copy
    w = [ row[:] for row in w0 ]
    d = len(w0)
    for _ in range(n_iter):
        x = forward_prop(x_in,w)
        delta = backward_prop(x,y,w)
        update_weights(x,delta,w,learning_rate)
        m = len(y)
        x_out = x[d-m:]
```

```

cost = J(x_out,y)
trajectory.append(cost)
if allclose(0,cost): break
return w, trajectory

```

Here `n_iter` is the maximum number of iterations allowed, and the iterations stop if the cost  $J$  is close to zero.

The cost or error function  $J$  enters the code only through the function `delta_out`, which is part of the function `backward_prop`.

Let  $W_0$  be the weight matrix (8.2.4). Then

```

x_in = [1.5,2.5]
learning_rate = .01
y0 = 0.4265356063
y1 = -0.2876478137
y = [y0,y1]
n_iter = 10000

w, trajectory = train_nn(x_in,y,w0,learning_rate,n_iter)

```

returns the cost `trajectory`, which can be plotted using the code

```

from matplotlib.pyplot import *

for lr in [.01,.02,.03,.035]:
    w, trajectory = train_nn(x_in,y,w0,lr,n_iter)
    n = len(trajectory)
    label = str(n) + ", " + str(lr)
    plot(range(n),trajectory,label=label)

grid()
legend()
show()

```

resulting in Figure 8.12.

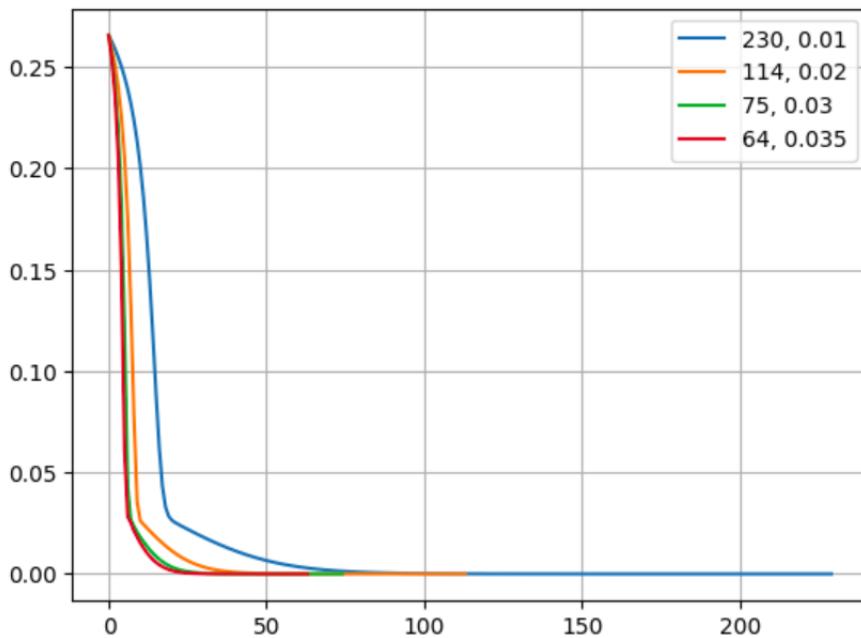


Figure 8.12: Cost trajectory and number of iterations as learning rate varies.

The convergence here is surprisingly easy to attain. However, the convergence here is a mirage. It is a reflection of *overfitting*, in the sense that we trained the weights to obtain the input-output map corresponding to a single sample: There is no reason the trained weights produce the input-output map for other samples.

Only after we train the weights repeatedly against all samples in a training dataset, can we hope to achieve training with some predictive power. This is the subject of §8.9.

## 8.5 Shallow Learning

Let  $x_1, x_2, \dots, x_N$  be a dataset, with corresponding labels or targets  $y_1, y_2, \dots, y_N$ . As in §8.1, the loss function is

$$J(W) = \sum_{k=1}^N J(x_k, y_k, W). \quad (8.5.1)$$

In this section, we focus on a single-layer perceptron (Figure 8.7),

$$J(x, y, W) = J(W^t x, y) = J(z, y).$$

Here  $x$  is the input,  $W$  is the weight matrix,  $z = W^t x$  is the network computed output, and  $y$  is the desired output or target.

A basic attribute of a neural network is its *trainability*. Can a given network be trained to achieve desired input-output behavior? As stated, this question is imprecise and not clearly defined. In fact, for deep networks, it is not at all clear how to turn this vague idea into an actionable definition.

In the case of a single-layer perceptron, the situation is straightforward enough to be able to both make the question precise, and to provide actionable criteria that guarantee trainability. This we do in the two cases

- linear regression, and
- logistic regression.

These cases correspond to the loss functions

- mean square error  $J(z, y) = |z - y|^2/2$ , and
- information error  $J(z, p) = I(p, \sigma(z))$ .

With any loss function  $J$ , the goal is to minimize  $J$ . With this in mind, from §7.5, we recall

### Ideal Loss Function

If a loss function  $J(W)$  is strictly convex and proper, then  $J$  has a unique global minimizer,

$$J(W^*) \leq J(W),$$

characterized as the unique critical point  $W^*$ .

Often, in machine learning,  $J$  is neither convex nor proper. Nevertheless, this result is an important benchmark to start with. Lack of properness is often addressed by *regularization*, which is the modification of  $J$  by a proper forcing term. Lack of convexity is addressed by using some type of accelerated gradient descent.

We say the loss function (8.5.1) of a single-layer perceptron is *trainable* if it is strictly convex and proper (§7.5). In this section, we determine conditions on the dataset that guarantee trainability in the above two cases.



The first loss function is (8.5.1) with

$$J(x, y, W) = \frac{1}{2}|W^t x - y|^2. \quad (8.5.2)$$

Then (8.5.1) is the *mean square error* or *mean square loss*, and the problem of minimizing (8.5.1) is *linear regression* (Figure 8.13).

We use (7.3.4) to compute the gradient of  $J(x, y, W)$ . Let  $V$  be a weight matrix, and let  $v = V^t x$ ,  $z = W^t x$ . Then  $(W + sV)^t x = z + sv$ , and the directional derivative is

$$\begin{aligned} \frac{d}{ds} \Big|_{s=0} J(x, y, W + sV) &= \frac{d}{ds} \Big|_{s=0} \frac{1}{2}|z + sv - y|^2 \\ &= v \cdot (z - y) = (V^t x) \cdot (z - y) \\ &= \text{trace}((V^t x) \otimes (z - y)) = \text{trace}(V^t(x \otimes (z - y))). \end{aligned}$$

By (7.3.4), this implies the weight gradient for mean square loss is

$$G = \nabla_W J(x, y, W) = x \otimes (z - y), \quad z = W^t x. \quad (8.5.3)$$

Note this result is a special case of (8.4.2).

Now we use (7.5.15) to check convexity of  $J(x, y, W)$ . With  $V$  and  $v$  as above,

$$\frac{d^2}{ds^2} \Big|_{s=0} J(x, y, W + sV) = |v|^2 = |V^t x|^2. \quad (8.5.4)$$

Since this is nonnegative,  $J(x, y, W)$  is a convex function of  $W$ .

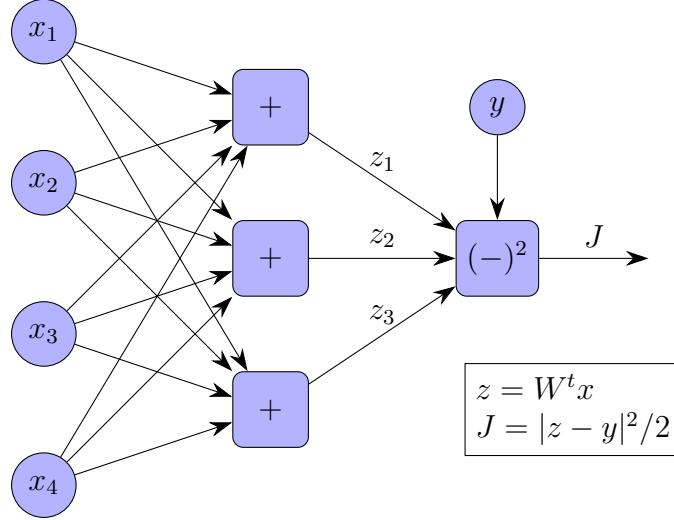


Figure 8.13: Linear regression neural network.

Since  $J(W)$  is the sum of  $J(x, y, W)$  over all samples,  $J(W)$  is convex. To check strict convexity of  $J(W)$ , suppose

$$\frac{d^2}{ds^2} \Big|_{s=0} J(W + sV) = 0.$$

Then (8.5.4) vanishes for all samples  $x = x_k$ ,  $y = y_k$ , which implies

$$V^t x_k = 0, \quad k = 1, 2, \dots, N. \quad (8.5.5)$$

Recall the feature space is the vector space of all inputs  $x$ , and (§2.9) a dataset is *full-rank* if the span of the dataset is the entire feature space. When this happens, (8.5.5) implies  $V = 0$ , hence  $J(W)$  is strictly convex.

To check properness of  $J(W)$ , by definition (7.5.9), we have to show

$$J(W) \leq c \quad \implies \quad \|W\|^2 \leq C. \quad (8.5.6)$$

Here  $\|W\|$  is the norm of the matrix  $W$  (2.2.11). The exact formula for the bound  $C$ , which is not important for our purposes, depends on the level  $c$  and the dataset.

If  $J(W) \leq c$ , by (8.5.1), (8.5.2), and the triangle inequality,

$$|W^t x_k| \leq \sqrt{2c} + |y_k|, \quad k = 1, 2, \dots, N.$$

If  $x$  is in the span of the dataset, then  $x$  is a linear combination of samples  $x_k$ . Hence there is a bound  $C_1$ , depending on  $x$  but not on  $W$ , such that

$$|W^t x| \leq C_1. \quad (8.5.7)$$

Let  $e_1, e_2, \dots$  be the standard basis in feature space, and assume the dataset is full-rank. Then  $e_1, e_2, \dots$  are in the span of the dataset. By (2.2.11) and (8.5.7), there is a bound  $C$ , not depending on  $W$ , with

$$\|W\|^2 = \sum_j |W^t e_j|^2 \leq C.$$

Since this establishes (8.5.6), we have shown

### Trainability: Linear Regression

Suppose the dataset  $x_1, x_2, \dots, x_N$  is full-rank. Then the mean square loss  $J(W)$  is trainable.

This is a simple, clear geometric criterion for convergence of gradient descent to the global minimum of  $J$ , valid for linear regression.



Let  $x_1, x_2, \dots, x_N$  be a dataset, with corresponding labels or targets  $p_1, p_2, \dots, p_N$ . In logistic regression, we assume the targets  $p$  reflect finitely many (say  $d$ ) classes or categories. Hence each target  $p$  is a probability vector  $p = (p_1, p_2, \dots, p_d)$ . Because of this, we use  $p$  instead of  $y$  to denote the targets.

In logistic regression, there are two main sub-cases where things work out best: Strict probabilities and one-hot encoded probabilities.

A probability  $p = (p_1, p_2, \dots, p_d)$  is *strict* if  $p_1, p_2, \dots, p_d$  are all positive (none are zero).

A target  $p = (p_1, p_2, \dots, p_d)$  is *one-hot encoded at slot  $j$*  if  $p_j = 1$ . When  $p$  is one-hot encoded at slot  $j$ , then  $p_k = 0$  for  $k \neq j$ .

For example, in classification problems, each sample  $x$  lies in one of  $d$  classes, and the target  $p$  is one-hot encoded at the slot corresponding to the class: If there are three classes 0, 1, 2, then the one-hot encoded target  $p$  is

$$(1, 0, 0), \quad (0, 1, 0), \quad (0, 0, 1),$$

depending on which class  $x$  lies in.

The second loss function is (8.5.1), with

$$J(x, p, W) = I(p, q), \quad q = \sigma(z), \quad z = W^t x.$$

Here  $I(p, q)$  is the relative information, measuring the error between the desired target  $p$  and the computed target  $q$ , and  $q = \sigma(z)$  is the softmax function, squashing the network's output  $z = W^t x$  into the probability  $q$ .

When  $p$  is one-hot encoded, by (7.6.15),

$$J(x, p, W) = I_{\text{cross}}(p, \sigma(W^t x)).$$

Because of this, in the literature, in the one-hot encoded case, (8.5.1) is called the *cross-entropy loss*.

In either case, strict or one-hot encoded,  $J(W)$  is *logistic loss* or *logistic error*, and the problem of minimizing (8.5.1) is *logistic regression* (Figure 8.14).

Since we will be considering both strict and one-hot encoded probabilities, we work with  $I(p, q)$  rather than  $I_{\text{cross}}(p, q)$ . Table 7.25 is a useful summary of the various information and entropy concepts.

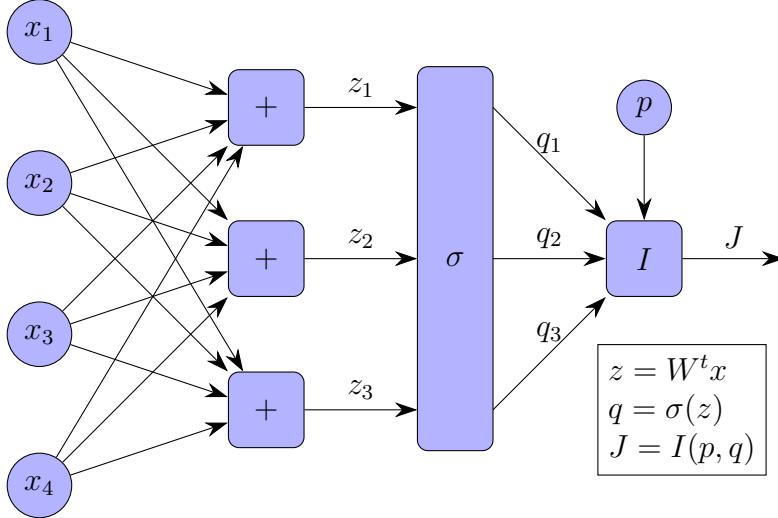


Figure 8.14: Logistic regression neural network.



We compute the gradient  $\nabla_W J(x, p, W)$ . By (7.6.2) and (7.6.14),

$$\nabla_z I(p, \sigma(z)) = \nabla_z Z(z) - p = q - p, \quad q = \sigma(z), \quad (8.5.8)$$

and, by (7.6.9),

$$D_z^2 I(p, \sigma(z)) = D_z^2 Z(z) = \text{diag}(q) - q \otimes q, \quad q = \sigma(z). \quad (8.5.9)$$

Let  $V$  be a weight matrix, and let  $v = V^t x$ ,  $z = W^t x$ . Then  $(W + sV)^t x = z + sv$ , and, by (8.5.8), the directional derivative is

$$\begin{aligned} \frac{d}{ds} \Big|_{s=0} J(x, y, W + sV) &= \frac{d}{ds} \Big|_{s=0} I(p, \sigma(z + sv)) \\ &= v \cdot (q - p) = (V^t x) \cdot (q - p) \\ &= \text{trace}((V^t x) \otimes (q - p)) \\ &= \text{trace}(V^t (x \otimes (q - p))). \end{aligned}$$

By (7.3.4), this shows the gradient for log loss is

$$G = \nabla_W J(x, p, W) = x \otimes (q - p), \quad q = \sigma(W^t x). \quad (8.5.10)$$

As before, this result is a special case of (8.4.2). Since  $q$  and  $p$  are probabilities,  $p \cdot \mathbf{1} = q \cdot \mathbf{1}$ , hence the gradient satisfies  $G\mathbf{1} = 0$ .

Recall (§7.6) we have strict convexity of  $Z(z)$  in directions orthogonal to  $\mathbf{1}$ , when  $z \cdot \mathbf{1} = 0$ . Since  $z = W^t x$ ,  $z \cdot \mathbf{1} = x \cdot W\mathbf{1}$ . Hence, to force  $z \cdot \mathbf{1} = 0$ , it is natural to impose the constraint

$$W\mathbf{1} = 0 \quad (8.5.11)$$

on the weight matrix, or

$$\sum_{j=1}^d w_{ij} = 0, \quad i = 1, 2, \dots, d.$$

If we initiate gradient descent with a weight matrix  $W$  satisfying  $W\mathbf{1} = 0$ , since the gradient  $G$  satisfies  $G\mathbf{1} = 0$ , all successive  $W$ 's will also satisfy  $W\mathbf{1} = 0$ .



Turning to convexity, we will establish

### Strict Convexity: Logistic Regression

Suppose the dataset  $x_1, x_2, \dots, x_N$  is full-rank. Then the logistic loss  $J(W)$ , restricted to the subspace  $W\mathbf{1} = 0$ , is strictly convex.

To see this, given a vector  $v$  and probability  $q$ , set  $\bar{v} = \sum_{j=1}^d v_j q_j$ . Then

$$\sum_{j=1}^d v_j^2 q_j - \left( \sum_{j=1}^d v_j q_j \right)^2 = \sum_{j=1}^d (v_j - \bar{v})^2 q_j.$$

If either side is zero, and  $q$  is strict, then  $v = \bar{v}\mathbf{1}$ , so  $v$  is a multiple of  $\mathbf{1}$ .

From this identity, and by (7.5.15) and (8.5.9), the second derivative of  $I(p, \sigma(z))$  in the direction of a vector  $v$  is

$$\frac{d^2}{ds^2} \Big|_{s=0} I(p, \sigma(z + sv)) = \sum_{j=1}^d (v_j - \bar{v})^2 q_j, \quad q = \sigma(z).$$

Let  $V$  be a weight matrix satisfying  $V\mathbf{1} = 0$  and let  $v = V^t x$ . Then  $v \cdot \mathbf{1} = x \cdot V\mathbf{1} = 0$ , so  $v$  is orthogonal to  $\mathbf{1}$ , and

$$(W + sV)^t x = z + sv.$$

If  $z = W^t x$ , it follows the second derivative of  $J(x, p, W)$  in the direction of  $V$  is

$$\frac{d^2}{ds^2} \Big|_{t=0} J(x, p, W + sV) = \sum_{j=1}^d (v_j - \bar{v})^2 q_j, \quad v = V^t x. \quad (8.5.12)$$

This shows the second derivative of  $J(x, p, W)$  is nonnegative, establishing the convexity of  $J(x, p, W)$ . Since  $J(W)$  is the sum of  $J(x, p, W)$  over all samples, we conclude  $J(W)$  is convex.

Moreover, if (8.5.12) vanishes, then, by the previous paragraph, since  $q = \sigma(z)$  is strict,  $v$  is a multiple of  $\mathbf{1}$ . Since  $v$  is orthogonal to  $\mathbf{1}$ ,  $v = 0$ . Since  $v = V^t x$ , the vanishing of (8.5.12) implies  $V^t x = 0$ .

If

$$\frac{d^2}{ds^2} \Big|_{s=0} J(W + sV) = \sum_{k=1}^N \frac{d^2}{ds^2} \Big|_{s=0} J(x_k, p_k, W + sV)$$

vanishes, then, since the summands are nonnegative, (8.5.12) vanishes, for every sample  $x = x_k$ ,  $p = p_k$ , hence

$$V^t x_k = 0, \quad k = 1, 2, \dots, N.$$

When the dataset is full-rank, this implies  $V = 0$ . This establishes strict convexity of  $J(W)$  in the subspace  $W\mathbf{1} = 0$ .



Now we turn to properness of  $J(W)$ .

### Properness: Logistic Regression

Let  $x_1, x_2, \dots, x_N$  be a dataset with corresponding targets  $p_1, p_2, \dots, p_N$ . For each class  $i$ , let  $K_i$  be the *convex hull* of the samples  $x$  whose corresponding targets  $p = (p_1, p_2, \dots, p_d)$  satisfy  $p_i > 0$ . If the span of the intersection  $K_i \cap K_j$  is full-rank for every class  $i$  and class  $j$ , then the logistic loss  $J(W)$  is proper on the subspace  $W\mathbf{1} = 0$ .

The convex hull is discussed in §7.5, see Figures 7.20 and 7.21. If  $K_i$  were just the samples  $x$  whose corresponding targets  $p$  satisfy  $p_i > 0$  (with no convex hull), then the intersection  $K_i \cap K_j$  may be empty.

For example, if  $p$  were one-hot encoded, then  $x$  belongs to at most one  $K_i$ . Thus taking the convex hull in the definition of  $K_i$  is crucial. This is clearly seen in Figure 8.26: The samples never intersect, but the convex hulls may do so.

To establish properness of  $J(W)$ , by definition (7.5.9), we have to show

$$W\mathbf{1} = 0 \quad \text{and} \quad J(W) \leq c \quad \implies \quad \|W\|^2 \leq C. \quad (8.5.13)$$

The exact formula for the bound  $C$ , which is not important for our purposes, depends on the level  $c$  and the dataset.

Suppose  $J(W) \leq c$ , with  $W\mathbf{1} = 0$ . Then  $I(p, \sigma(W^t x)) = J(x, p, W) \leq c$  for every sample  $x$  and corresponding target  $p$ .

Let  $x$  be a sample, let  $z = W^t x$ , and suppose the corresponding target  $p$  satisfies  $p_i \geq \epsilon$ , for some class  $i$ , and some  $\epsilon > 0$ . If  $j \neq i$ , then

$$\epsilon(z_j - z_i) \leq p_i(Z(z) - z_i) \leq \sum_{k=1}^d p_k(Z(z) - z_k) = Z(z) - p \cdot z.$$

Here we used  $z_j < Z(z)$ , and  $Z(z) - z_k > 0$  for all  $k$ . By (7.6.14),

$$Z(z) - p \cdot z = I(p, \sigma(z)) - I(p) \leq c + \log d.$$

Combining the last two inequalities,

$$\epsilon(z_j - z_i) \leq c + \log d.$$

By definition of  $K_i$ ,  $p_i > 0$  for all targets  $p$  corresponding to samples  $x$  in  $K_i$ . Therefore there is a positive  $\epsilon_i$  such that  $p_i \geq \epsilon_i$  for all targets  $p$  corresponding to samples  $x$  in  $K_i$ . Let  $\epsilon$  be the least of  $\epsilon_1, \epsilon_2, \dots, \epsilon_d$ . Then

$$\epsilon(z_j - z_i) \leq c + \log d, \quad j \neq i, \text{ for samples } x \text{ in } K_i.$$

By taking convex combinations of samples  $x$  in  $K_i$ , the last inequality remains valid for all  $x$  in  $K_i$ , so

$$\epsilon(z_j - z_i) \leq c + \log d, \quad j \neq i, \text{ for all } x \text{ in } K_i.$$

Repeating the same argument for  $x$  in  $K_j$ ,

$$\epsilon(z_i - z_j) \leq c + \log d, \quad j \neq i, \text{ for all } x \text{ in } K_j.$$

Let  $C_1 = (c + \log d)/\epsilon$ . Combining the last two inequalities,

$$|z_i - z_j| \leq C_1, \quad j \neq i, \text{ for all } x \text{ in } K_i \cap K_j. \quad (8.5.14)$$

Let  $x$  be any vector in feature space, and let  $z = W^t x$ . Since  $\text{span}(K_i \cap K_j)$  is full-rank for every  $i$  and  $j$ ,  $x$  is a linear combination of vectors in  $K_i \cap K_j$ . This implies, by (8.5.14), there is a bound  $C_2$ , depending on  $x$  but not on  $W$ , such that

$$|z_i - z_j| \leq C_2, \quad \text{for every } i \text{ and } j, \quad (8.5.15)$$

Since  $z \cdot \mathbf{1} = 0$ ,  $z_i = -\sum_{j \neq i} z_j$ . Summing (8.5.15) over  $j \neq i$ ,

$$d|z_i| = |(d-1)z_i + z_i| = \left| \sum_{j \neq i} (z_i - z_j) \right| \leq (d-1)C_2.$$

This implies there is a bound  $C_3$ , depending on  $x$  but not on  $W$ , such that

$$|W^t x|^2 = |z|^2 = \sum_{i=1}^d z_i^2 \leq C_3.$$

Let  $e_1, e_2, \dots$  be the standard basis in feature space. By (2.2.11),

$$\|W\|^2 = \sum_k |W^t e_k|^2.$$

Inserting  $x = e_1, x = e_2, \dots$  into the last inequality, we conclude there is a bound  $C$ , depending only on level  $c, d$ , and the dataset, satisfying (8.5.13).



If the span of  $K_i \cap K_j$  is full-rank, then the span of the dataset itself is full-rank. Putting the last two results together, we conclude

### Trainability: Logistic Regression

Let  $x_1, x_2, \dots, x_N$  be a dataset with corresponding targets  $p_1, p_2, \dots, p_N$ . For each class  $i$ , let  $K_i$  be the *convex hull* of the samples  $x$  whose corresponding targets  $p = (p_1, p_2, \dots, p_d)$  satisfy  $p_i > 0$ . If the span of the intersection  $K_i \cap K_j$  is full-rank for every class  $i$  and class  $j$ , then the logistic loss  $J(W)$  is trainable on the subspace  $W\mathbf{1} = 0$ .

By the definition of  $K_i$  here, the union of  $K_i$  over classes  $i = 1, 2, \dots, d$  contains the whole dataset. This is not necessarily the case in the results below.

As a special case, let  $K$  be the samples whose corresponding targets are strict. Then  $K \subset K_i$  for all classes  $i$ . If the span of  $K$  is full-rank, then  $\text{span}(K_i \cap K_j)$  is full-rank. This derives the first consequence,

### Trainability: Strict Logistic Regression

Let  $x_1, x_2, \dots, x_N$  be a dataset, with corresponding targets  $p_1, p_2, \dots, p_N$ . Let  $K$  be the samples whose corresponding targets are strict. If the span of  $K$  is full-rank, then the logistic loss  $J(W)$  is trainable on the subspace  $W\mathbf{1} = 0$ .

If a target  $p$  is one-hot encoded at slot  $i$ , then  $p_i = 1 > 0$ . This derives the second consequence,

### Trainability: One-hot Encoded Logistic Regression

Let  $x_1, x_2, \dots, x_N$  be a dataset with corresponding targets  $p_1, p_2, \dots, p_N$ . For each class  $i$ , let  $K_i$  be the *convex hull* of the samples whose corresponding targets are one-hot encoded at slot  $i$ . If the span of the intersection  $K_i \cap K_j$  is full-rank for every  $i$  and  $j$ , then the logistic loss  $J(W)$  is trainable on the subspace  $W\mathbf{1} = 0$ .

In this case, each sample  $x$  belongs in at most one  $K_i$ , so taking convex hulls is crucial, see the examples in the next section. Here not all samples need be one-hot encoded: The requirement is that there is sufficient overlap between the targets that are one-hot encoded.



We end the section by comparing the three regressions: linear, strict logistic, and one-hot encoded logistic.

In classification problems, it is one-hot encoded logistic regression that is relevant. Because of this, in the literature, logistic regression often defaults to the one-hot encoded case.

In linear regression, not only does  $J(W)$  have a minimum, but so does  $J(z, y)$ . Properness ultimately depends on properness of a quadratic  $|z|^2$ .

In strict logistic regression, by (8.5.8), the critical point equation

$$\nabla_z J(z, p) = 0$$

can always be solved, so there is at least one minimum for each  $J(z, p)$ . Here properness ultimately depends on properness of the partition function  $Z(z)$ .

In one-hot encoded regression,  $J(z, p) = I(p, \sigma(z))$  and  $\nabla_z J(z, p) = 0$  can never be solved, because  $q = \sigma(z)$  is always strict and  $p$  is one-hot encoded, see (8.5.10). Nevertheless, trainability of  $J(W)$  is achievable if there is sufficient overlap between the sample categories.

In linear regression, the minimizer is expressible in terms of the regression equation, and thus can be solved in principle using the pseudo-inverse. In practice, when the dimensions are high, gradient descent may be the only option for linear regression.

In logistic regression, the minimizer cannot be found in closed form, so we have no choice but to apply gradient descent, even for low dimensions.

## 8.6 Regression Examples

Let  $(x_k, y_k)$ ,  $k = 1, 2, \dots, N$ , be a dataset in the plane. The simplest regression problem is to determine the line  $y = mx + b$  minimizing the residual

$$J(m, b) = \sum_{k=1}^N (y_k - mx_k - b)^2. \quad (8.6.1)$$

Then the line is the *regression line*.

More generally, given a dataset  $x_1, x_2, \dots, x_N$  in  $\mathbf{R}^d$ , and scalar targets  $y_1, y_2, \dots, y_N$ , we want to minimize

$$J(w) = \sum_{k=1}^N (y_k - w \cdot x_k)^2$$

over all weight vectors  $w$  in  $\mathbf{R}^d$ . Here we are fitting a *regression hyperplane*

$$y = w \cdot x = w_1 x_1 + w_2 x_2 + \dots + w_d x_d.$$

This corresponds to (8.5.2), where  $W$  is the  $d \times 1$  matrix  $W = w$ .

For example, Figure 8.16 is a dataset and Figure 8.15 is a plot of population versus employed, with the mean and the regression line shown.

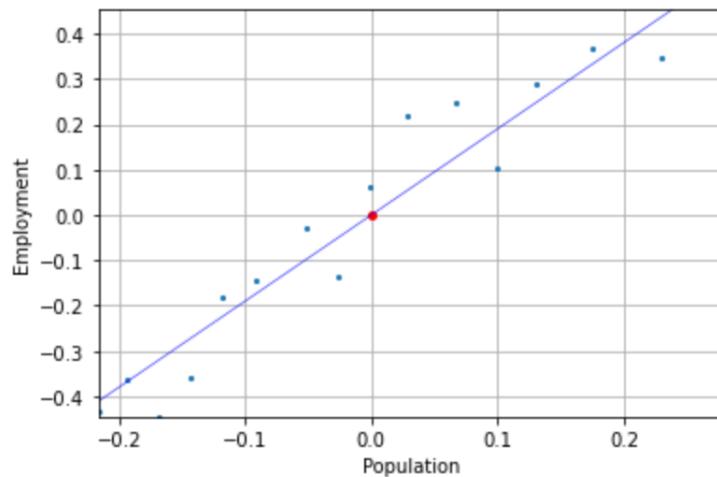


Figure 8.15: Population versus employed: Linear Regression.

GNP.deflator	GNP	Unemployed	Armed Forces	Population	Year	Employed
83	234.289	235.6	159	107.608	1947	60.323
88.5	259.426	232.5	145.6	108.632	1948	61.122
88.2	258.054	368.2	161.6	109.773	1949	60.171
89.5	284.599	335.1	165	110.929	1950	61.187
96.2	328.975	209.9	309.9	112.075	1951	63.221
98.1	346.999	193.2	359.4	113.27	1952	63.639
99	365.385	187	354.7	115.094	1953	64.989
100	363.112	357.8	335	116.219	1954	63.761
101.2	397.469	290.4	304.8	117.388	1955	66.019
104.6	419.18	282.2	285.7	118.734	1956	67.857
108.4	442.769	293.6	279.8	120.445	1957	68.169
110.8	444.546	468.1	263.7	121.95	1958	66.513
112.6	482.704	381.3	255.2	123.366	1959	68.655
114.2	502.601	393.1	251.4	125.368	1960	69.564
115.7	518.173	480.6	257.2	127.852	1961	69.331
116.9	554.894	400.7	282.7	130.081	1962	70.551

Table 8.16: Longley Economic Data [15].



Let  $X$  be the  $N \times d$  matrix with rows  $x_1, x_2, \dots, x_N$ , and let  $Y$  be the vector  $(y_1, y_2, \dots, y_N)$ . Then we can rewrite the residual as

$$J(w) = |Xw - Y|^2. \quad (8.6.2)$$

From §2.3, any weight  $w^*$  minimizing (8.6.2) is a solution the regression equation

$$X^t X w^* = X^t Y. \quad (8.6.3)$$

Since the pseudo-inverse provides a solution of the regression equation, we have

### Linear Regression

The weight  $w^* = X^+ Y$  minimizes the residual (8.6.2) and solves the regression equation (8.6.3).

We work out the regression equation in the plane, when both features  $x$  and  $y$  are scalar. In this case,  $w = (m, b)$  and

$$X = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \dots & \dots \\ x_N & 1 \end{pmatrix}, \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{pmatrix}.$$

In the scalar case, the regression equation (8.6.3) is  $2 \times 2$ . To simplify the computation of  $X^t X$ , let

$$\bar{x} = \frac{1}{N} \sum_{k=1}^N x_k, \quad \bar{y} = \frac{1}{N} \sum_{k=1}^N y_k.$$

Then  $(\bar{x}, \bar{y})$  is the mean of the dataset. Also, let  $x$  and  $y$  denote the vectors  $(x_1, x_2, \dots, x_N)$  and  $(y_1, y_2, \dots, y_N)$ , and let, as in §1.6,

$$\text{cov}(x, y) = \frac{1}{N} \sum_{k=1}^N (x_k - \bar{x})(y_k - \bar{y}) = \frac{1}{N} x \cdot y - \bar{x}\bar{y}.$$

Then  $\text{cov}(x, y)$  is the covariance between  $x$  and  $y$ ,

$$X^t X = N \begin{pmatrix} x \cdot x & \bar{x} \\ \bar{x} & 1 \end{pmatrix}, \quad X^t Y = N \begin{pmatrix} x \cdot y \\ \bar{y} \end{pmatrix}.$$

With  $w = (m, b)$ , the regression equation reduces to

$$\begin{aligned} (x \cdot x)m + \bar{x}b &= x \cdot y, \\ m\bar{x} + b &= \bar{y}. \end{aligned}$$

The second equation says the regression line passes through the mean  $(\bar{x}, \bar{y})$ . Multiplying the second equation by  $\bar{x}$  and subtracting the result from the first equation cancels the  $b$  and leads to

$$\text{cov}(x, x)m = (x \cdot x - \bar{x}^2)m = (x \cdot y - \bar{x}\bar{y}) = \text{cov}(x, y).$$

This derives

### Linear Regression in the Plane

The regression line in two dimensions passes through the mean  $(\bar{x}, \bar{y})$  and has slope

$$m = \frac{\text{cov}(x, y)}{\text{cov}(x, x)}.$$



Now we use linear regression to do *polynomial regression*. Return to the dataset  $(x_k, y_k)$  in  $\mathbf{R}^2$  (Figure 8.15). We can expand or “lift” the dataset from  $\mathbf{R}^2$  to  $\mathbf{R}^6$  by working with the vectors  $(1, x_k, x_k^2, x_k^3, x_k^4, y_k)$  instead of  $(x_k, y_k)$ .

Assuming the data is given by Figure 8.16, we build the code for Figures 8.15 and 8.17. We begin by assuming the data is given as arrays,

```
from numpy import *
from pandas import read_csv

df = read_csv("longley.csv")

X = df["Population"].to_numpy()
Y = df["Employed"].to_numpy()
```

Then we standardize the data

```
X = X - mean(X)
Y = Y - mean(Y)

varx = sum(X**2)/len(X)
vary = sum(Y**2)/len(Y)

X = X/sqrt(varx)
Y = Y/sqrt(vary)
```

After this, we compute the optimal weight  $w^*$  and construct the polynomial. The regression equation is solved using the pseudo-inverse (§2.3).

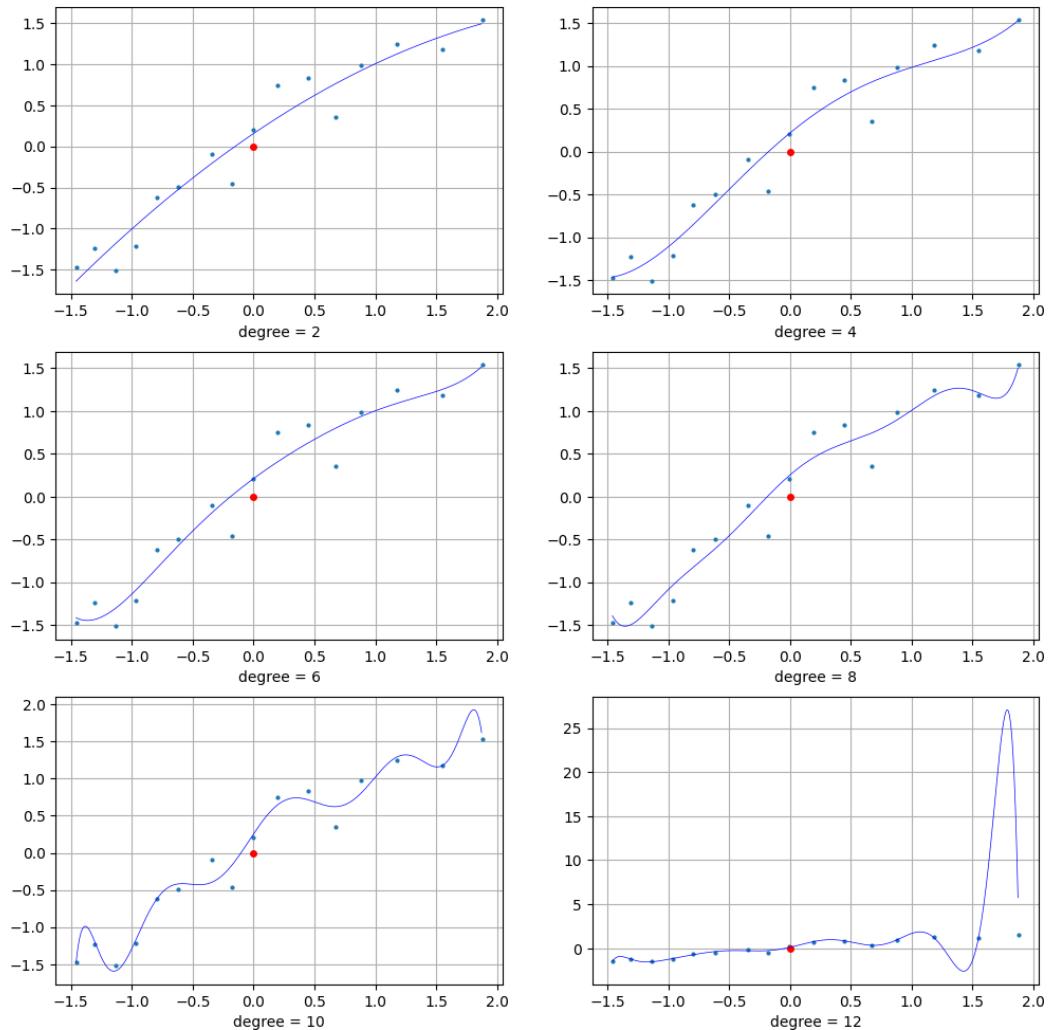


Figure 8.17: Polynomial regression: Degrees 2, 4, 6, 8, 10, 12.

```
from numpy.linalg import pinv

# polynomial function - degree d-1
def poly(x,d):
    A = column_stack([ X**i for i in range(d) ]) # Nxd
    Aplus = pinv(A)
    b = Y # Nx1
```

```
wstar = dot(Aplus,b)
return sum([x**i*wstar[i] for i in range(d)],axis=0)
```

Then we plot the data and the polynomial in six subplots.

```
from matplotlib.pyplot import *

xmin,ymin = amin(X), amin(Y)
xmax, ymax = amax(X), amax(Y)

figure(figsize=(12,12))
# six subplots
rows, cols = 3,2

# x interval
x = arange(xmin,xmax,.01)

for i in range(6):
    d = 3 + 2*i # degree = d-1
    subplot(rows, cols,i+1)
    plot(X,Y,"o",markersize=2)
    plot([0],[0],marker="o",color="red",markersize=4)
    plot(x,poly(x,d),color="blue",linewidth=.5)
    xlabel("degree = %s" % str(d-1))
    grid()

show()
```

Running this code with degree 1 returns Figure 8.15. Taking too high a power can lead to overfitting, for example for degree 12.



Here is an example of a simple logistic regression problem. A group of students takes an exam. For each student, we know the amount of time  $x$  they studied, and the outcome  $p$ , whether or not they passed the exam.

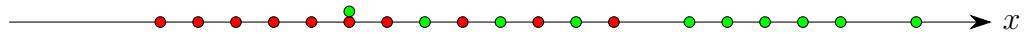
$x$	$p$	$x$	$p$	$x$	$p$	$x$	$p$	$x$	$p$
0.5	0	.75	0	1.0	0	1.25	0	1.5	0
1.75	0	1.75	1	2.0	0	2.25	1	2.5	0
2.75	1	3.0	0	3.25	1	3.5	0	4.0	1
4.25	1	4.5	1	4.75	1	5.0	1	5.5	1

Table 8.18: Hours studied and outcomes.

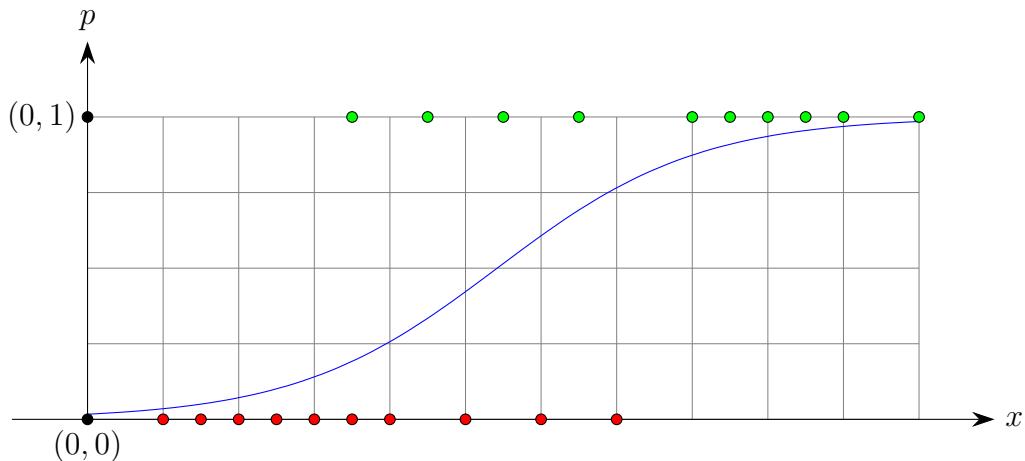
More generally, we may only know the amount of study time  $x$ , and the *probability*  $p$  that the student passed, where now  $0 \leq p \leq 1$ .

For example, the data may be as in Figure 8.18, where  $p_k$  equals 1 or 0 according to whether they passed or not.

As stated, the samples of this dataset are scalars, and the dataset is one-dimensional (Figure 8.19).

Figure 8.19: Exam dataset:  $x$ .

Plotting the dataset on the  $(x, p)$  plane, the goal is to fit a curve as in Figure 8.20.

Figure 8.20: Exam dataset:  $(x, p)$  [29].

To apply the results from the previous section, we incorporate the bias

and rewrite the dataset as

$$(x_1, 1), (x_2, 1), \dots, (x_N, 1), \quad N = 20,$$

resulting in Figure 8.21. Since these vectors are not parallel, the dataset is full-rank in  $\mathbf{R}^2$ , hence  $J(m, b)$  is strictly convex. In Figure 8.21, the shaded area is bounded by the vectors corresponding to the overlap between passing and failing students' hours.

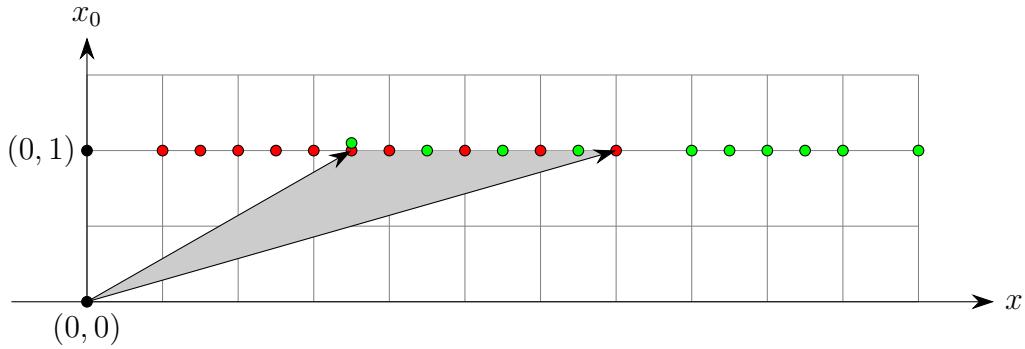


Figure 8.21: Exam dataset:  $(x, x_0)$ .

Let  $\sigma(z)$  be the sigmoid function (5.1.12). Then, as in the previous section, the goal is to minimize the loss function

$$J(m, b) = \sum_{k=1}^N I(p_k, q_k), \quad q_k = \sigma(mx_k + b), \quad (8.6.4)$$

Once we have the minimizer  $(m^*, b^*)$ , we have the best-fit curve

$$q = \sigma(m^*x + b^*)$$

(Figure 8.20).

If the targets  $p$  are one-hot encoded, the dataset is as follows.

$x$	$p$	$x$	$p$	$x$	$p$	$x$	$p$	$x$	$p$
0.5	(1,0)	.75	(1,0)	1.0	(1,0)	1.25	(1,0)	1.5	(1,0)
1.75	(1,0)	1.75	(0,1)	2.0	(1,0)	2.25	(0,1)	2.5	(1,0)
2.75	(0,1)	3.0	(1,0)	3.25	(0,1)	3.5	(1,0)	4.0	(0,1)
4.25	(0,1)	4.5	(0,1)	4.75	(0,1)	5.0	(0,1)	5.5	(0,1)

Table 8.22: Hours studied and one-hot encoded outcomes.

Each sample  $(x, 1)$  in the dataset is in  $\mathbf{R}^2$ , and each target is one-hot encoded as  $(p, 1 - p)$ . Since the weight matrix must satisfy (8.5.11)  $W\mathbf{1} = 0$ , we have

$$W = \begin{pmatrix} b & -b \\ m & -m \end{pmatrix}.$$

Since  $z = W^t x$ , the outputs must satisfy  $z_1 = z$  and  $z_2 = -z$ . This leads to a neural network with two inputs and two outputs (Figure 8.23).

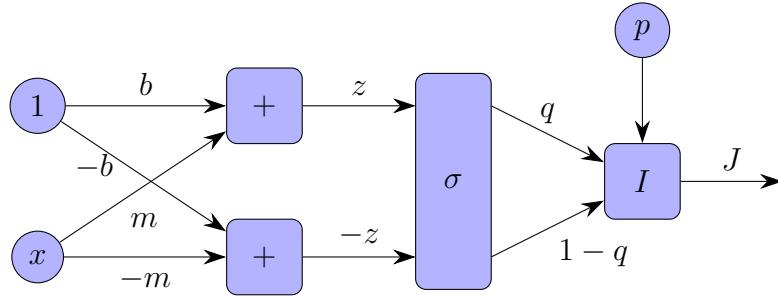


Figure 8.23: Neural network for student exam outcomes.

Since here  $d = 2$ , the networks in Figures 8.23 and 8.24 are equivalent. In Figure 8.23,  $\sigma$  is the softmax function,  $I$  is given by (7.6.5), and  $p, q$  are probability vectors. In Figure 8.24,  $\sigma$  is the sigmoid function,  $I$  is given by (7.2.3), and  $p, q$  are probability scalars.

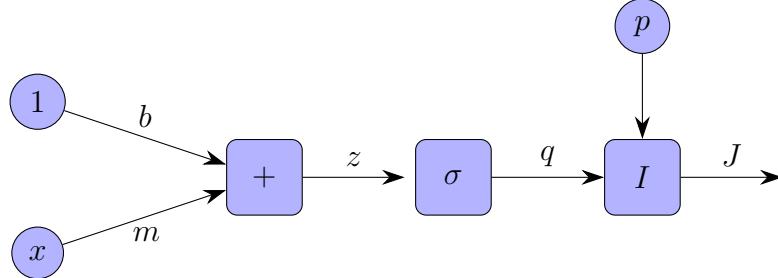
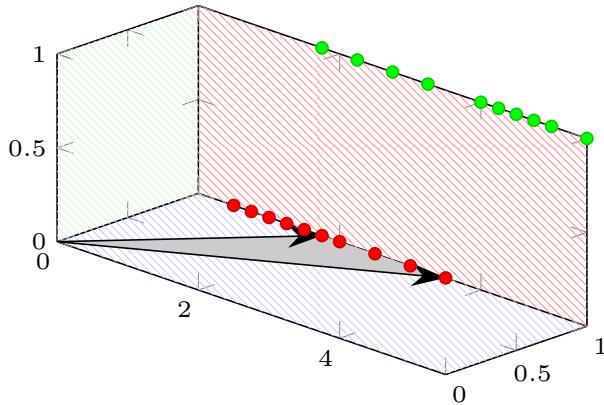


Figure 8.24: Equivalent neural network for student exam outcomes.

Figure 8.20 is a plot of  $x$  against  $p$ . However, the dataset, with the bias input included, has two inputs  $x, 1$  and one output  $p$ , and should be plotted in three dimensions  $(x, 1, p)$ . Then (Figure 8.25) samples lie on the line  $(x, 1)$  in the horizontal plane, and  $p$  is on the vertical axis.

Figure 8.25: Exam dataset:  $(x, x_0, p)$ .

The horizontal plane in Figure 8.25, which is the plane in Figure 8.21, is feature space. The convex hulls  $K_0$  and  $K_1$  are in feature space, so the convex hull  $K_0$  of the samples corresponding to  $p = 0$  is the line segment joining  $(.5, 1, 0)$  and  $(3.5, 1, 0)$ , and the convex hull  $K_1$  of the samples corresponding to  $p = 1$  is the line segment joining  $(1.75, 1, 0)$  and  $(5.5, 1, 0)$ . In Figure 8.25,  $K_0$  is the line segment joining the blue points, and  $K_1$  is the projection onto feature space of the line segment joining the red points. Since  $K_0 \cap K_1$  is the line segment joining  $(1.75, 1, 0)$  and  $(3.5, 1, 0)$ , the span of  $K_0 \cap K_1$  is all of feature space. By the results of the previous section,  $J(w)$  is proper.

Here is the descent code.

```
from numpy import *
from scipy.special import expit

X = [0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 1.75, 2.0, 2.25, 2.5,
      → 2.75, 3.0, 3.25, 3.5, 4.0, 4.25, 4.5, 4.75, 5.0, 5.5]
P = [0,0,0,0,0,0,1,0,1,0,1,0,1,0,1,1,1,1,1,1]

def gradient(m,b):
    return sum([ (expit(m*x+b) - p) * array([x,1]) for x,p in
      → zip(X,P) ],axis=0)

# gradient descent
w = array([0,0]) # starting m,b
```

```

g = gradient(*w)
t = .01 # learning rate

while not allclose(g,0):
    wplus = w - t * g
    if allclose(w,wplus): break
    else: w = wplus
    g = gradient(*w)

print("descent result: ",w)
print("gradient: ",gradient(*w))

```

This code returns

$$m^* = 1.49991537, \quad b^* = -4.06373862.$$

These values are used to graph the sigmoid in Figure 8.20.

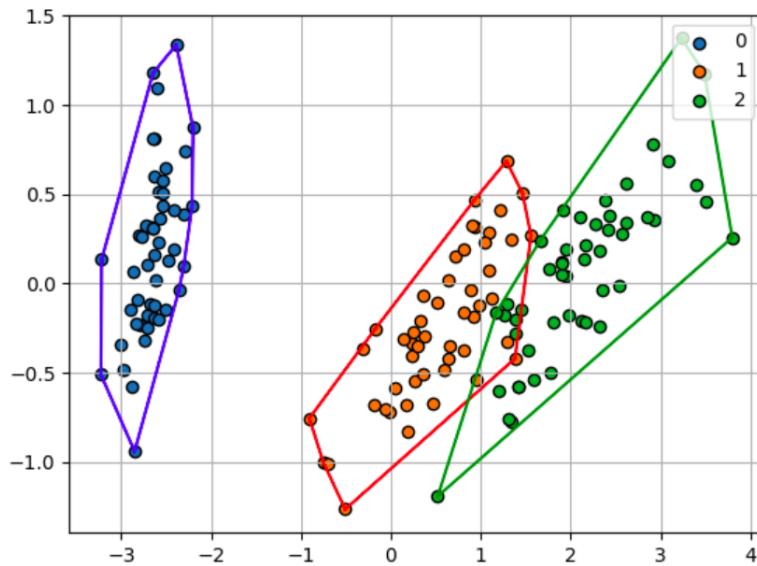


Figure 8.26: Convex hulls of Iris classes in  $\mathbf{R}^2$ .

The Iris dataset consists of 150 samples divided into three groups, leading to three convex hulls  $K_0, K_1, K_2$  in  $\mathbf{R}^4$ . If the dataset is projected onto the top two principal components, then the projections of these three hulls do not pair-intersect (Figure 8.26). It follows we have no guarantee the logistic loss is proper.

On the other hand, the MNIST dataset consists of 60,000 samples divided into ten groups. If the MNIST dataset is projected onto the top two principal components, the projections of the ten convex hulls  $K_0, K_1, \dots, K_9$  onto  $\mathbf{R}^2$ , do intersect (Figure 8.27).

This does not guarantee that the ten convex hulls  $K_0, K_1, \dots, K_9$  in  $\mathbf{R}^{784}$  intersect, but at least this is so for the 2d projection of the MNIST dataset. Therefore the logistic loss of the 2d projection of the MNIST dataset is proper.

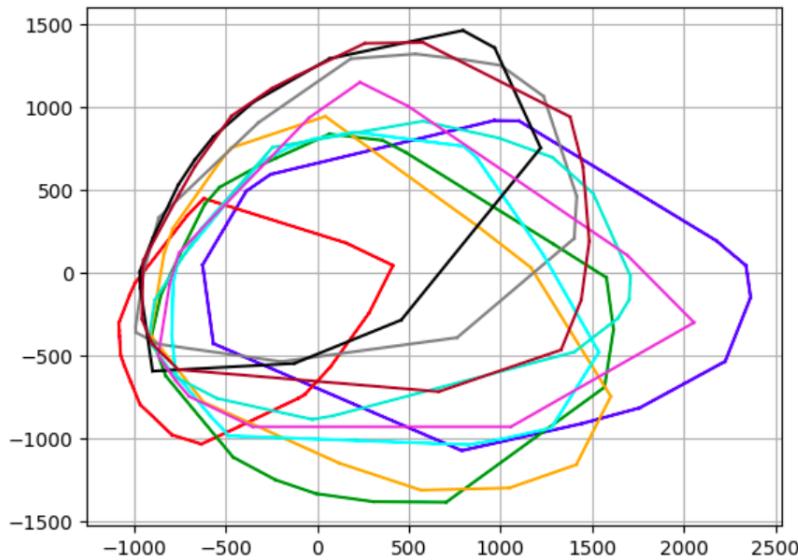


Figure 8.27: Convex hulls of MNIST classes in  $\mathbf{R}^2$ .



We say two sets  $A, B$  in  $\mathbf{R}^d$  are *linearly separable* if there is a hyperplane

$$z = w \cdot x + w_0$$

with

$$\begin{aligned} z \leq 0, & \quad \text{for } x \text{ in } A, \\ z \geq 0, & \quad \text{for } x \text{ in } B. \end{aligned}$$

In the case of two classes, the results in §7.5 and §8.5 lead to the following result [12].

### Dataset Binary Classifier

Suppose a dataset  $x_1, x_2, \dots, x_N$  is divided into two classes, and suppose neither class lies in a hyperplane. There are two possibilities.

- The two classes are linearly separable.
- The two classes are not linearly separable: When the means of the classes are distinct, the log loss  $J(w, w_0)$  is trainable (strictly convex and proper), and there is a unique minimizer  $(w^*, w_0^*)$  with  $w^* \neq 0$ , hence an optimal single-layer perceptron

$$q = \sigma(w^* \cdot x + w_0^*).$$

## 8.7 Strict Convexity

In this section, we work with loss functions that are smooth and strictly convex. While this is not always the case, this assumption is a base case against which we can test different optimization or training models.

By smooth and strictly convex, we mean there are positive constants  $m$  and  $L$  satisfying

$$m \leq D^2 f(w) \leq L, \quad \text{for every } w. \quad (8.7.1)$$

Recall this means the eigenvalues of the symmetric matrix  $D^2 f(w)$  are between  $L$  and  $m$ . In this situation, the *condition number*<sup>1</sup>  $r = m/L$  is between zero and one:  $0 < r \leq 1$ .

In the previous section, we saw that basic gradient descent converged to a critical point. If  $f(x)$  is strictly convex, there is exactly one critical point, the global minimum. From this we have

---

<sup>1</sup>In the literature, the condition number is often defined as  $L/m$ .

### Gradient Descent on a Strictly Convex Function

If the short-step gradient descent sequence starting from  $w_0$  converges to  $w^*$ , then  $w^*$  is the global minimum.

The simplest example of a convex loss function is the *quadratic case*

$$f(w) = \frac{1}{2}w \cdot Qw - b \cdot w, \quad (8.7.2)$$

where  $Q$  is a covariance matrix. Then  $D^2 f(w) = Q$ . If the eigenvalues of  $Q$  are between positive constants  $m$  and  $L$ , then  $f(w)$  is smooth and strictly convex.

By (7.3.7), the gradient for this example is  $g = Qw - b$ . Hence the minimizer is the unique solution  $w^* = Q^{-1}b$  of the linear system  $Qw = b$ . Thus gradient descent is a natural tool for solving linear systems and computing inverses, at least for covariance matrices  $Q$ .

By (7.5.17),  $f(w)$  lies between two quadratics,

$$\frac{m}{2}|w - w^*|^2 \leq f(w) - f(w^*) \leq \frac{L}{2}|w - w^*|^2. \quad (8.7.3)$$

How far we are from our goal  $w^*$  can be measured by the error  $E(w) = |w - w^*|^2$ . Another measure of error is  $E(w) = f(w) - f(w^*)$ . The goal is to drive the error between  $w$  and  $w^*$  to zero.

When  $f(w)$  is smooth and strictly convex in the sense of (8.7.1), the estimate (8.7.3) shows these two error measures are equivalent. We use both measures below.



Let  $t = 1/L$ . Inserting  $x = w$  and  $a = w^*$  in the left half of (7.5.21) and using  $\nabla f(w^*) = 0$  implies

$$f(w) \leq f(w^*) + \frac{1}{2m}|\nabla f(w)|^2.$$

Let  $E(w) = f(w) - f(w^*)$ . Combining this inequality with (8.3.3), and recalling  $r = m/L = mt$ , we arrive at

$$E(w^+) \leq (1 - r)E(w). \quad (8.7.4)$$

Iterating this implies

$$E(w_2) \leq (1-r)E(w_1) \leq (1-r)(1-r)E(w_0) = (1-r)^2 E(w_0).$$

In general, this leads to

### Gradient Descent I

Let  $r = m/L$  and set  $E(w) = f(w) - f(w^*)$ . Then the descent sequence  $w_0, w_1, w_2, \dots$  given by (8.3.1) with learning rate

$$t = \frac{1}{L}$$

converges to  $w^*$  at the rate

$$E(w_n) \leq (1-r)^n E(w_0), \quad n = 1, 2, \dots \quad (8.7.5)$$

This is the basic gradient descent result GD-I.



Using coercivity of the gradient (7.5.22), we can obtain an improved result GD-II.

Let  $E(w) = |w - w^*|^2$  and set the learning rate at  $t = 2/(m+L)$ . Inserting  $x = w$  and  $a = w^*$  in (7.5.22) and using  $\nabla f(w^*) = 0$  implies

$$g \cdot (w - w^*) \geq \frac{mL}{m+L} |w - w^*|^2 + \frac{1}{m+L} |g|^2.$$

Using this and (8.3.1) and  $t = 2/(m+L)$ ,

$$\begin{aligned} E(w^+) &= E(w) - 2tg \cdot (w - w^*) + t^2|g|^2 \\ &\leq \left(1 - 2t \frac{mL}{m+L}\right) E(w) + \left(t^2 - \frac{2t}{m+L}\right) |g|^2 \\ &= \left(\frac{L-m}{L+m}\right)^2 E(w). \end{aligned}$$

This implies

### Gradient Descent II

Let  $r = m/L$  and set  $E(w) = |w - w^*|^2$ . Then the descent sequence  $w_0, w_1, w_2, \dots$  given by (8.3.1) with learning rate

$$t = \frac{2}{m + L}$$

converges to  $w^*$  at the rate

$$E(w_n) \leq \left( \frac{1-r}{1+r} \right)^{2n} E(w_0), \quad n = 1, 2, \dots \quad (8.7.6)$$

GD-II improves GD-I in two ways: The learning rate is larger,

$$\frac{2}{m+L} > \frac{1}{L},$$

and the convergence rate is smaller,

$$\left( \frac{1-r}{1+r} \right)^2 < (1-r),$$

implying faster convergence.

For example, if  $L = 6$  and  $m = 2$ , then  $r = 1/3$ , the learning rates are  $1/6$  versus  $1/4$ , and the convergence rates are  $2/3$  versus  $1/4$ . Even though GD-II improves GD-I, the improvement is not substantial. In the next section, we use momentum to derive better convergence rates.



Let  $g$  be the gradient of the loss function at a point  $w$ . Then the line passing through  $w$  in the direction of  $g$  is  $w - tg$ . When the loss function is quadratic (7.3.7),  $f(w - tg)$  is a quadratic function of the scalar variable  $t$ . In this case, *the minimizer  $t$  along the line  $w - tg$  is explicitly computable* as

$$t = \frac{g \cdot g}{g \cdot Qg}.$$

This leads to gradient descent with varying time steps  $t_0, t_1, t_2, \dots$ . As a consequence, one can show the error is lowered as follows,

$$E(w^+) = \left( 1 - \frac{1}{(u \cdot Qu)(u \cdot Q^{-1}u)} \right) E(w), \quad u = \frac{g}{|g|}.$$

Using a well-known inequality, *Kantorovich's inequality*, one can show that here the convergence rate is also (8.7.6). Thus, after all this work, there is no advantage here, it simpler to stick with GD-II!

Nevertheless, the idea here, the *line-search for a minimizer*, is a sound one, and is useful in some situations.

## 8.8 Accelerated Gradient Descent

In this section, we modify the gradient descent method by adding a term incorporating previous gradients, leading to *gradient descent with momentum*. After this, we consider other variations, leading to the most frequently used descent methods.

Recall in a descent sequence, the current point is  $w$ , the next point is  $w^+$ , and the previous point is  $w^-$ .

In gradient descent with momentum, we add a *momentum term* to the current point  $w$ , obtaining the *lookahead point*

$$w^\circ = w + s(w - w^-). \quad (8.8.1)$$

Here  $s$  is the *decay rate*. The momentum term reflects the direction induced by the previous step. Because this mimics the behavior of a ball rolling downhill, gradient descent with momentum is also called *heavy ball descent*.

Then the descent sequence  $w_0, w_1, w_2, \dots$  is generated by

### Momentum Gradient Descent Step

$$w^+ = w - t\nabla f(w) + s(w - w^-). \quad (8.8.2)$$

Here we have two hyperparameters, the learning rate and the decay rate.



We study convergence for the simplest case of a quadratic (8.7.2). In this case,  $\nabla f(w) = Qw - b$ , and the sequence satisfies the recursion

$$w_{n+1} = w_n - t(Qw_n - b) + s(w_n - w_{n-1}), \quad n = 0, 1, 2, \dots \quad (8.8.3)$$

To initialize the recursion, we set  $w_{-1} = w_0^- = w_0$ . This implies  $w_1 = w_0 - t(Qw_0 - b)$ .

We measure the convergence using the error  $E(w) = |w - w^*|^2$ , and we assume  $m < Q < L$  strictly, in the sense every eigenvalue  $\lambda$  satisfies

$$m < \lambda < L. \quad (8.8.4)$$

As before, we set  $r = m/L$ .

Let  $v$  be an eigenvector of  $Q$  with eigenvalue  $\lambda$ . To solve (8.8.3), we assume a solution of the form

$$w_n = w^* + \rho^n v, \quad Qv = \lambda v. \quad (8.8.5)$$

Inserting this into (8.8.3) and using  $Qw^* = b$  leads to the quadratic equation

$$\rho^2 = (1 - t\lambda + s)\rho - s.$$

By the quadratic formula,

$$\rho = \rho_{\pm} = \frac{(1 - \lambda t + s) \pm \sqrt{(1 - \lambda t + s)^2 - 4s}}{2}.$$

Assume the discriminant  $(1 - \lambda t + s)^2 - 4s$  is negative. This happens exactly when

$$\frac{(1 - \sqrt{s})^2}{\lambda} < t < \frac{(1 + \sqrt{s})^2}{\lambda}. \quad (8.8.6)$$

If we assume

$$\frac{(1 - \sqrt{s})^2}{m} \leq t \leq \frac{(1 + \sqrt{s})^2}{L}, \quad (8.8.7)$$

then (8.8.6) holds for every eigenvalue  $\lambda$  of  $Q$ .

Multiplying (8.8.7) by  $\lambda$  and factoring the discriminant as a difference of two squares leads to

$$4s - (1 - \lambda t + s)^2 \geq (1 - s)^2 \frac{(L - \lambda)(\lambda - m)}{mL}. \quad (8.8.8)$$

When (8.8.6) holds, the roots are conjugate complex numbers  $\rho, \bar{\rho}$ , where

$$\rho = x + iy = \frac{(1 - \lambda t + s) + i\sqrt{-(1 - \lambda t + s)^2 + 4s}}{2}. \quad (8.8.9)$$

It follows the absolute value of  $\rho$  equals

$$|\rho| = \sqrt{x^2 + y^2} = \sqrt{s}.$$

To obtain the fastest convergence, we choose  $s$  and  $t$  to minimize  $|\rho| = \sqrt{s}$ , while still satisfying (8.8.7). This forces (8.8.7) to be an equality,

$$\frac{(1 - \sqrt{s})^2}{m} = t = \frac{(1 + \sqrt{s})^2}{L}.$$

These are two equations in two unknowns  $s, t$ . Solving, we obtain

$$\sqrt{s} = \frac{1 - \sqrt{r}}{1 + \sqrt{r}}, \quad t = \frac{1}{L} \cdot \frac{4}{(1 + \sqrt{r})^2}.$$

Let  $\tilde{w}_n = w_n - w^*$ . Since  $Qw_n - b = Q\tilde{w}_n$ , (8.8.3) is a 2-step *linear* recursion in the variables  $\tilde{w}_n$ . Therefore the general solution depends on two constants  $A, B$ .

Let  $\lambda_1, \lambda_2, \dots, \lambda_d$  be the eigenvalues of  $Q$  and let  $v_1, v_2, \dots, v_d$  be the corresponding orthonormal basis of eigenvectors.

Since (8.8.3) is a 2-step *vector* linear recursion,  $A$  and  $B$  are vectors, and the general solution depends on  $2d$  constants  $A_k, B_k, k = 1, 2, \dots, d$ .

If  $\rho_k, k = 1, 2, \dots, d$ , are the corresponding roots (8.8.9), then (8.8.5) is a solution of (8.8.3) for each of  $2d$  roots  $\rho = \rho_k, \rho = \bar{\rho}_k, k = 1, 2, \dots, d$ . Therefore the linear combination

$$w_n = w^* + \sum_{k=1}^d (A_k \rho_k^n + B_k \bar{\rho}_k^n) v_k, \quad n = 0, 1, 2, \dots \quad (8.8.10)$$

is the general solution of (8.8.3). Inserting  $n = 0$  and  $n = 1$  into (8.8.10), then taking the dot product of the result with  $v_k$ , we obtain two linear equations for two unknowns  $A_k, B_k$ ,

$$\begin{aligned} A_k + B_k &= (w_0 - w^*) \cdot v_k, \\ A_k \rho_k + B_k \bar{\rho}_k &= (w_1 - w^*) \cdot v_k = (1 - t\lambda_k)(w_0 - w^*) \cdot v_k, \end{aligned}$$

for each  $k = 1, 2, \dots, d$ . Solving for  $A_k, B_k$  yields

$$A_k = \left( \frac{1 - t\lambda_k - \bar{\rho}_k}{\rho_k - \bar{\rho}_k} \right) (w_0 - w^*) \cdot v_k, \quad B_k = \bar{A}_k.$$

Let

$$C = \max_{\lambda} \frac{(L - m)(L - m)}{(L - \lambda)(\lambda - m)}. \quad (8.8.11)$$

Using (8.8.8), one verifies the estimate

$$|A_k|^2 = |B_k|^2 \leq C |(w_0 - w^*) \cdot v_k|^2.$$

Now use (2.9.4) twice, first with  $v = w_n - w^*$ , then with  $v = w_0 - w^*$ . By (8.8.10) and the triangle inequality,

$$\begin{aligned} |w_n - w^*|^2 &= \sum_{k=1}^d |(w_n - w^*) \cdot v_k|^2 \\ &= \sum_{k=1}^d |A_k \rho_k^n + B_k \bar{\rho}_k^n|^2 \\ &\leq \sum_{k=1}^d (|A_k| + |B_k|)^2 |\rho_k|^{2n} \\ &\leq 4Cs^n \sum_{k=1}^d |(w_0 - w^*) \cdot v_k|^2 \\ &= 4Cs^n |w_0 - w^*|^2. \end{aligned}$$

This derives the following result.

### Momentum Gradient Descent - Heavy Ball

Suppose the loss function  $f(w)$  is quadratic (8.7.2), let  $r = m/L$ , and set  $E(w) = |w - w^*|^2$ . Let  $C$  be given by (8.8.11). Then the descent sequence  $w_0, w_1, w_2, \dots$  given by (8.8.2) with learning rate and decay rate

$$t = \frac{1}{L} \cdot \frac{4}{(1 + \sqrt{r})^2}, \quad s = \left( \frac{1 - \sqrt{r}}{1 + \sqrt{r}} \right)^2,$$

converges to  $w^*$  at the rate

$$E(w_n) \leq 4C \left( \frac{1 - \sqrt{r}}{1 + \sqrt{r}} \right)^{2n} E(w_0), \quad n = 1, 2, \dots \quad (8.8.12)$$

This heavy ball descent, due to Polyak [21], is an improvement over GD-II (8.7.6), because  $\sqrt{r}$  is substantially larger than  $r$  when  $r$  is small. The downside of this momentum method is that the convergence (8.8.12) is only

guaranteed for  $f(w)$  quadratic (8.7.2). In fact, there are examples of non-quadratic  $f(w)$  where heavy ball descent does not converge to  $w^*$ . Nevertheless, this method is widely used.



The momentum method can be modified by evaluating the gradient at the lookahead point  $w^\circ$  (8.8.1),

### Momentum Descent Step With Lookahead Gradient

$$\begin{aligned} w^\circ &= w + s(w - w^-), \\ w^+ &= w^\circ - t\nabla f(w^\circ). \end{aligned} \tag{8.8.13}$$

This leads to *accelerated gradient descent*, or *momentum descent with lookahead gradient*. This result, due to Nesterov [19], is valid for any convex function satisfying (8.7.1), not just quadratics.

The iteration (8.8.13) is in two steps, a momentum step followed by a basic gradient descent step. The momentum step takes us from the current point  $w$  to the lookahead point  $w^\circ$ , and the gradient descent step takes us from  $w^\circ$  to the successive point  $w^+$ .

Starting from  $w_0$ , and setting  $w_{-1} = w_0$ , here it turns out the loss sequence  $f(w_0), f(w_1), f(w_2), \dots$  is not always decreasing. Because of this, we seek another function  $V(w)$  where the corresponding sequence  $V(w_0), V(w_1), V(w_2), \dots$  is decreasing.

To explain this, it's best to assume  $w^* = 0$  and  $f(w^*) = 0$ . This can always be arranged by translating the coordinate system. Then it turns out

$$V(w) = f(w) + \frac{L}{2}|w - \rho w^-|^2, \tag{8.8.14}$$

with a suitable choice of  $\rho$ , does the job. With the choices

$$t = \frac{1}{L}, \quad s = \frac{1 - \sqrt{r}}{1 + \sqrt{r}}, \quad \rho = 1 - \sqrt{r},$$

we will show

$$V(w^+) \leq \rho V(w). \tag{8.8.15}$$

In fact, we see below (8.8.22), (8.8.23) that  $V$  is reduced by an additional quantity proportional to the momentum term.

The choice  $t = 1/L$  is a natural choice from basic gradient descent (8.3.3). The derivation of (8.8.15) below forces the choices for  $s$  and  $\rho$ .

Given a point  $w$ , while  $w^+$  is well-defined by (8.8.13), it is not clear what  $w^-$  means. There are two ways to insert meaning here. Either evaluate  $V(w)$  along a sequence  $w_0, w_1, w_2, \dots$  and set, as before,  $w_n^- = w_{n-1}$ , or work with the function  $W(w) = V(w^+)$  instead of  $V(w)$ . If we assume  $(w^+)^- = w$ , then  $W(w)$  is well-defined. With this understood, we nevertheless stick with  $V(w)$  as in (8.8.14) to simplify the calculations.

We first show how (8.8.15) implies the result. Using  $(w_0)^- = w_0$  and (8.7.3),

$$V(w_0) = f(w_0) + \frac{L}{2}|w_0 - \rho w_0|^2 = f(w_0) + \frac{m}{2}|w_0|^2 \leq 2f(w_0).$$

Moreover  $f(w) \leq V(w)$ . Iterating (8.8.15), we obtain

$$f(w_n) \leq V(w_n) \leq \rho^n V(w_0) \leq 2\rho^n f(w_0).$$

This derives

### Momentum Descent - Lookahead Gradient

Let  $r = m/L$  and set  $E(w) = f(w) - f(w^*)$ . Then the sequence  $w_0, w_1, w_2, \dots$  given by (8.8.13) with learning rate and decay rate

$$t = \frac{1}{L}, \quad s = \frac{1 - \sqrt{r}}{1 + \sqrt{r}}$$

converges to  $w^*$  at the rate

$$E(w_n) \leq 2(1 - \sqrt{r})^n E(w_0), \quad n = 1, 2, \dots \quad (8.8.16)$$

While the convergence rate for accelerated descent is slightly worse than heavy ball descent, the value of accelerated descent is its validity for all convex functions satisfying (8.7.1), and the fact, also due to Nesterov [19], that this convergence rate is best-possible among all such functions.

Now we derive (8.8.15). Assume  $(w^+)^- = w$  and  $w^* = 0$ ,  $f(w^*) = 0$ . We know  $w^\circ = (1 + s)w - sw^-$  and  $w^+ = w^\circ - tg^\circ$ , where  $g^\circ = \nabla f(w^\circ)$ .

By the basic descent step (8.3.1) with  $w^\circ$  replacing  $w$ , (8.3.3) implies

$$f(w^+) \leq f(w^\circ) - \frac{t}{2}|g^\circ|^2. \quad (8.8.17)$$

Here we used  $t = 1/L$ .

By (7.5.16) with  $x = w$  and  $a = w^\circ$ ,

$$f(w^\circ) \leq f(w) - g^\circ \cdot (w - w^\circ) - \frac{m}{2}|w - w^\circ|^2. \quad (8.8.18)$$

By (7.5.16) with  $x = w^* = 0$  and  $a = w^\circ$ ,

$$f(w^\circ) \leq g^\circ \cdot w^\circ - \frac{m}{2}|w^\circ|^2. \quad (8.8.19)$$

Multiply (8.8.18) by  $\rho$  and (8.8.19) by  $1 - \rho$  and add, then insert the sum into (8.8.17). After some simplification, this yields

$$f(w^+) \leq \rho f(w) + g^\circ \cdot (w^\circ - \rho w) - \frac{r}{2t} (\rho|w - w^\circ|^2 + (1 - \rho)|w^\circ|^2) - \frac{t}{2}|g^\circ|^2. \quad (8.8.20)$$

Since

$$(w^\circ - \rho w) - tg^\circ = w^+ - \rho w,$$

we have

$$\frac{1}{2t}|w^+ - \rho w|^2 = \frac{1}{2t}|w^\circ - \rho w|^2 - g^\circ \cdot (w^\circ - \rho w) + \frac{t}{2}|g^\circ|^2.$$

Adding this to (8.8.20) leads to

$$V(w^+) \leq \rho f(w) - \frac{r}{2t} (\rho|w - w^\circ|^2 + (1 - \rho)|w^\circ|^2) + \frac{1}{2t}|w^\circ - \rho w|^2. \quad (8.8.21)$$

Let

$$R(a, b) = r (\rho s^2 |b|^2 + (1 - \rho)|a + sb|^2) - |(1 - \rho)a + sb|^2 + \rho|(1 - \rho)a + pb|^2.$$

Solving for  $f(w)$  in (8.8.14) and inserting into (8.8.21) leads to

$$V(w^+) \leq \rho V(w) - \frac{1}{2t} R(w, w - w^-). \quad (8.8.22)$$

If we can choose  $s$  and  $\rho$  so that  $R(a, b)$  is a *positive* scalar multiple of  $|b|^2$ , then, by (8.8.22), (8.8.15) follows, completing the proof.

Based on this, we choose  $s, \rho$  to make  $R(a, b)$  independent of  $a$ , which is equivalent to  $\nabla_a R = 0$ . But

$$\nabla_a R = 2(1 - \rho) \left( (r - (1 - \rho)^2) a + (\rho^2 - s(1 - r)) b \right),$$

so  $\nabla_a R = 0$  is two equations in two unknowns  $s, \rho$ . This leads to the choices for  $s$  and  $\rho$  made above. Once these choices are made,  $s(1 - r) = \rho^2$  and  $\rho > s$ . From this,

$$R(a, b) = R(0, b) = (rs^2 - s^2 + \rho^3)|b|^2 = \rho^2(\rho - s)|b|^2, \quad (8.8.23)$$

which is positive.

## 8.9 Stochastic Gradient Descent

★ under construction ★



# Chapter A

## Appendices

### A.1 SQL

Recall matrices (§2.1), datasets, CSV files, spreadsheets, arrays, dataframes are basically the same objects.

Databases are collections of tables, where a *table* is another object similar to the above. Hence

$$\text{matrix} = \text{dataset} = \text{CSV file} = \text{spreadsheet} = \text{table} = \text{array} = \text{dataframe} \quad (\text{A.1.1})$$

One difference is that each entry in a table may be a string, or code, or an image, not just a number. Nevertheless, every table has rows and columns; rows are usually called *records*, and columns are columns.

A *database* is a collection of several tables that may or may not be linked by columns with common data. Software that serves databases is a *database server*. Often the computer running this software is also called a *database server*, or a *server* for short. Databases created by a database server (software) are stored as files on the database server.

There are many varieties of database server software. Here we use *MariaDB*, a widely-used open-source database server. By using open-source software, one is assured to be using the “purest” form of the software, in the sense that proprietary extensions are avoided, and the software is compatible with the widest range of commercial variations.

Because database tables can contain millions of records, it is best to access a database server programmatically, using an *application programming interface*, rather than a *graphical user interface*. The basic API for inter-

acting with database servers is SQL (*structured query language*). SQL is a programming language for creating and modifying databases.

Any application on your laptop that is used to access a database is called an SQL *client*. The database server being accessed may be *local*, running on the same computer you are logged into, or *remote*, running on another computer on the internet. In our examples, the code assumes a local or remote database server is being accessed.

Because SQL commands are case-insensitive, by default we write them in lowercase. Depending on the SQL client, commands may terminate with semicolons or not. As mentioned above, data may be numbers or strings.

The basic SQL commands are

```
select from
limit
select distinct
where/not where <column>
where <column> = <data> and/or <column> = <data>
order by <column1>,<column2>
insert into table (<column1>,<column2>,...) \
    values (<data1>,<data2>,...)
is null
update <table> set <column> = <data> where ...
like <regex> (%, _, [abc], [a-f], [!abc])
delete from <table> where ...
select min(<column>) from <table> (also max, count, avg)
where <column> in/not in (<data array>)
between/not between <data1> and <data2>
as
join (left, right, inner, full)
create database <database>
drop database <database>
create table <table>
truncate <table>
alter table <table> add <column> <datatype>
alter table <table> drop column <column>
insert into <table> select
```

All the objects in (A.1.1) are also equivalent to a Python list-of-dicts. In this section we explain how to convert between the objects

$$\text{list-of-dicts} \iff \text{JSON string} \iff \text{dataframe} \iff \text{CSV file} \iff \text{SQL table} \quad (\text{A.1.2})$$

For all conversions, we use `pandas`. We begin describing a Python *list-of-dicts*, because this does not require any additional Python modules.

A Python *dictionary* or *dict* is a Python object of the form (prices are in cents)

```
item1 = {"dish": "Hummus", "price": 800, "quantity": 5}
```

This is an *unordered* listing of key-value pairs. Here the keys are the strings `dish`, `price`, and `quantity`. Keys need not be strings; they may be integers or any *immutable* Python objects. Since a Python list is mutable, a key cannot be a list. Values may be any Python objects, so a value may be a list. In a dict, values are accessed through their keys. For example, `item1[  
→ "dish"]` returns '`Hummus`'.

A *list-of-dicts* is simply a Python list whose elements are Python dicts, for example,

```
item2 = {"dish": "Avocado", "price": 900, "quantity": 2}  
L = [item1, item2]
```

Here `L` is a list and

```
len(L), L[0]["dish"]
```

returns

```
(2, 'Hummus')
```

In other words, `L` is a *list-of-dicts*,

```
L == [{"dish": "Hummus", "price": 800, "quantity": 5},  
→ {"dish": ... }]
```

returns True.

A list-of-dicts L can be converted into a string using the `json` module, as follows:

```
from json import *
s = dumps(L)
```

Now print L and print s. Even though L and s “look” the same, L is a list, and s is a string. To emphasize this point, note

- `len(L) == 2` and `len(s) == 99`,
- `L[0:2] == L` and `s[0:2] == '[{'`
- `L[8]` returns an error and `s[8] == ':'`

To convert back the other way, use

```
from json import *
L1 = loads(s)
```

Then `L == L1` returns `True`. Strings having this form are called *JSON strings*, and are easy to store in a database as VARCHARs (see Figure A.4).

The basic object in the Python module `pandas` is the *dataframe* (Figures A.1, A.2, A.4, A.5). The `pandas` module can convert a dataframe `df` to many, many other formats

```
df.to_dict(), df.to_csv(), df.to_excel(), df.to_sql(),
→ df.to_json(), ...
```

To convert a list-of-dicts to a dataframe is easy. The code

```
from pandas import *
df = DataFrame(L)
df
```

returns the dataframe in Figure A.1 (prices are in cents).

	dish	price	quantity
0	Hummus	800	5
1	Avocado	900	2

Figure A.1: Dataframe from list-of-dicts.

	dish	price
0	Hummus	800
1	Baba Ghanouj	800
2	Avocado	900
3	Muhammara	800
4	Mujaddara	800
...	...	...
73	Chicken Kebab, one dip, baklava	2300
74	Lahmbajeen Pizza, two dips	2400
75	Merguez Kebab and Moussaka	2600
76	Margherita Pizza, Chicken Wings, Baklava	2500
77	Any three pizzas	5000

78 rows × 2 columns

Figure A.2: Menu dataframe and SQL table.

To go the other way is equally easy. The code

```
L1 = df.to_dict('records')
L == L1
```

returns True. Here the option '`'records'`' returns a list-of-dicts; other options returns a dict-of-dicts or other combinations.

To convert a CSV file into a dataframe, use the code

```
menu_df = read_csv("menu.csv")
menu_df
```

This returns Figure A.2 (prices are in cents).

To go the other way, to convert the dataframe `df` to the CSV file `menu1`  
 $\hookrightarrow$  `.csv`, use the code

```
df.to_csv("menu1.csv")
df.to_csv("menu2.csv", index=False)
```

The option `index=False` suppresses the index column, so `menu2.csv` has two columns, while `menu1.csv` has three columns. Also useful is the method `.to_excel`, which returns an excel file.

Now we explain how to convert between a dataframe and an SQL table. What we have seen so far uses only the module `pandas`. To convert to SQL, we need two more modules, `sqlalchemy` and `pymysql`.

The module `sqlalchemy` allows us to connect to a database server from within Python, and the module `pymysql` is the code necessary to complete the connection to our version of database server. For example, if we are connecting to an Oracle database server, we would use the module `cx-Oracle` instead of `pymysql`.

In Python, the standard module installation method is to use `pip`. To install `sqlalchemy` and `pymysql`, *type within jupyter*:

```
pip install sqlalchemy
pip install pymysql
```

To connect using `sqlalchemy`, we first collect the connection data into one URI string,

```
protocol = "mysql+pymysql://"
credentials = "username:password"
server = "@servername"
port = ":3306"
uri = protocol + credentials + server + port
```

This string contains your database username, your database password, the database server name, the server port, and the protocol. If the database is "\rawa", the URI is

```
database = "/rawa"
uri = protocol + credentials + server + port + database
```

Using this uri, the connection is made as follows

```
from sqlalchemy import create_engine
engine = sqlalchemy.create_engine(uri)
```

(In `sqlalchemy`, a connection is called an “engine”.) After this, to store the dataframe `df` into a table `Menu`, use the code

```
df.to_sql('Menu', engine, if_exists='replace')
```

The `if_exists = 'replace'` option replaces the table `Menu` if it existed prior to this command. Other options are `if_exists='fail'` and `if_exists=append`. The default is `if_exists='fail'`, so

```
df.to_sql('Menu', engine)
```

returns an error if `Menu` exists.

To read a table into a dataframe, use for example the code

```
from sqlalchemy import text
query1 = text("select * from rawa.OrdersIn")
query2 = text("select * from rawa.OrdersIn where items
    → like '%Hummus%';")
connection = engine.connect()
df1 = read_sql(query1, connection)
df2 = read_sql(query2, connection)
```

Better Python coding technique is to place `read_sql` and `to_sql` in a `with` block, as follows

```
with engine.connect() as connection:  
    df = pd.read_sql(query, connection)  
    df.to_sql('Menu', engine)
```

One benefit of this syntax is the automatic closure of the connection upon completion. This completes the discussion of how to convert between dataframes and SQL tables, and completes the discussion of conversions between any of the objects in (A.1.2).



Figure A.3: Rawa restaurant.

As an example how all this goes together, here is a task:

Given two CSV files `menu.csv` and `orders.csv` downloaded from a restaurant website (Figure A.3), create three SQL tables `Menu`, `OrdersIn`, `OrdersOut`.

The two CSV files are (click)

[orders.csv](#)      and      [menu.csv](#).

The three SQL table columns are as follows (price, tip, tax, subtotal, total are in cents)

	orderId	created	customerId	items
0	1	6/29/19 2:37	1497	[{"dish": "Citrus Chicken", "price": 1000, "quantity": 8}]
1	5	7/1/19 14:30	1600	[{"dish": "Hummus", "price": 800, "quantity": 10}]
2	11	7/9/19 17:07	1704	[{"dish": "Hummus", "price": 800, "quantity": 1}, {"dish": "Baba G..."]
3	12	7/10/19 12:14	1431	[{"dish": "Hummus", "price": 800, "quantity": 9}]
4	13	7/10/19 18:50	1458	[{"dish": "Labne", "price": 800, "quantity": 1}, {"dish": "Garden"...}]
...	...	...	...	...
3965	3985	1/20/23 17:23	1787	[{"dish": "Hummus", "price": 800, "quantity": 1}, {"dish": "Chicke..."]
3966	3986	1/20/23 17:37	10	[{"dish": "Falafel", "price": 1300, "quantity": 1}, {"dish": "Chic..."]
3967	3987	1/20/23 19:16	1354	[{"dish": "Lentil", "price": 800, "quantity": 1}, {"dish": "Marghe..."]
3968	3988	1/21/23 12:06	152	[{"dish": "Mamyaldi Vegan", "price": 1700, "quantity": 2}]
3969	3989	1/21/23 17:00	1579	[{"dish": "Sampler Plate", "price": 1800, "quantity": 1}, {"dish": "..."}]

3970 rows × 4 columns

Figure A.4: OrdersIn dataframe and SQL table.

```

/* Menu */
dish      varchar
price     integer

/* ordersin */
orderid   integer
created   datetime
customerid integer
items     json

/* ordersout */
orderid   integer
subtotal  integer
tip       integer
tax       integer
total     integer

```

To achieve this task, we download the CSV files `menu.csv` and `orders.in.csv`, then we carry out these steps. (price and tip in `menu.csv` and

`orders.csv` are in cents so they are INTs.)

1. Read the CSV files into dataframes `menu_df` and `orders_df`.
2. Convert the dataframes into list-of-dicts `menu` and `orders`.
3. Create a list-of-dicts `OrdersIn` with keys `orderId`, `created`, `customerId`  
 $\hookrightarrow$  whose values are obtained from list-of-dicts `orders`.
4. Create a list-of-dicts `OrdersOut` with keys `orderId`, `tip` whose values are obtained from list-of-dicts `orders` (tips are in cents so they are INTs).
5. Add a key `items` to `OrdersIn` whose values are JSON strings specifying the items ordered in `orders`, using the prices in `menu` (these are in cents so they are INTs). The JSON string is of a list-of-dicts in the form discussed above `L = [item1, item2]` (see row 0 in Figure A.4).

Do this by looping over each `order` in the list-of-dicts `orders`, then looping over each `item` in the list-of-dicts `menu`, and extracting the quantity ordered of the item `item` in the order `order`.

6. Add a key `subtotal` to `OrdersOut` whose values (in cents) are computed from the above values.

Add a key `tax` to `OrdersOut` whose values (in cents) are computed using the Connecticut tax rate 7.35%. Tax is applied to the sum of subtotal and tip.

Add a key `total` to `OrdersOut` whose values (in cents) are computed from the above values (subtotal, tax, tip).

7. Convert the list-of-dicts `OrdersIn`, `OrdersOut` to dataframes `OrdersIn_df`  
 $\hookrightarrow$  , `OrdersOut_df`.
8. Upload `menu_df`, `OrdersIn_df`, `OrdersOut_df` to tables `Menu`, `OrdersIn`  
 $\hookrightarrow$  , `OrdersOut`.

The resulting dataframes `ordersin_df` and `ordersout_df`, and SQL tables `OrdersIn` and `OrdersOut`, are in Figures A.4 and A.5.

	orderId	tip	subtotal	tax	total
0	0	0	4000	294	4294
1	1	0	8000	588	8588
2	2	0	2200	161	2361
3	3	0	8800	646	9446
4	4	0	10400	764	11164
...	...	...	...	...	...
3985	3985	434	7800	605	8839
3986	3986	0	3000	220	3220
3987	3987	1275	8200	696	10171
3988	3988	180	3400	263	3843
3989	3989	885	5800	491	7176

3990 rows × 5 columns

Figure A.5: OrdersOut dataframe and SQL table.

## Complete Code for the Task

```
# step 1
from pandas import *

protocol = "https://"
server = "math.temple.edu"
path = "/~hijab/teaching/csv_files/restaurant/"
url = protocol + server + path

menu_df = read_csv(url + "menu.csv")
orders_df = read_csv(url + "orders.csv")

# step 2
menu = menu_df.to_dict('records')
orders = orders_df.to_dict('records')

# step 3
OrdersIn = h
```

```
for r in orders:
    d = {}
    d["orderId"] = r["orderId"]
    d["created"] = r["created"]
    d["customerId"] = r["customerId"]
    OrdersIn.append(d)

# step 4
OrdersOut = h
for r in orders:
    d = {}
    d["orderId"] = r["orderId"]
    d["tip"] = r["tip"]
    OrdersOut.append(d)

# step 5
from json import *

for i,r in enumerate(OrdersIn):
    itemsOrdered = h
    for item in menu:
        dish = item["dish"]
        price = item["price"]
        if dish in orders[i]:
            quantity = orders[i][dish]
            if quantity > 0:
                d = {"dish": dish, "price": price,
→ "quantity": quantity}
                itemsOrdered.append(d)
    r["items"] = dumps(itemsOrdered)

# steps 6
for i,r in enumerate(OrdersOut):
    items = loads(OrdersIn[i]["items"])
    subtotal = sum([ item["price"]*item["quantity"] for item
→ in items ])
    r["subtotal"] = subtotal
    tip = OrdersOut[i]["tip"]
    tax = int(.0735*(tip + subtotal))
```

```
total = subtotal + tip + tax
r["tax"] = tax
r["total"] = total

# step 7
ordersin_df = DataFrame(OrdersIn)
ordersout_df = DataFrame(OrdersOut)

# step 8
from sqlalchemy import create_engine, text

# connect to the database
protocol = "mysql+pymysql://"
credentials = "username:password@"
server = "servername"
port = ":3306"
database = "/rawa"
uri = protocol + credentials + server + port + database

engine = create_engine(uri)

dtype1 = { "dish":sqlalchemy.String(60),
           → "price":sqlalchemy.Integer }

dtype2 = {
    "orderId":sqlalchemy.Integer,
    "created":sqlalchemy.String(30),
    "customerId":sqlalchemy.Integer,
    "items":sqlalchemy.String(1000)
}

dtype3 = {
    "orderId":sqlalchemy.Integer,
    "tip":sqlalchemy.Integer,
    "subtotal":sqlalchemy.Integer,
    "tax":sqlalchemy.Integer,
    "total":sqlalchemy.Integer
}
```

```

with engine.connect() as connection:
    menu_df.to_sql('Menu', engine,
                   if_exists = 'replace', index = False, dtype = dtype1)
    ordersin_df.to_sql("OrdersIn", engine,
                       index = False, if_exists = 'replace', dtype = dtype2)
    ordersout_df.to_sql("OrdersOut", engine,
                        index = False, if_exists = 'replace', dtype = dtype3)

```

## Moral of this section

In this section, all work was done in Python on a laptop, no SQL was used on the database, other than creating a table or downloading a table. Generally, this is an effective workflow:

- Use SQL to do big manipulations on the database (joining and filtering).
- Use Python to do detailed computations on your laptop (analysis).

Now we consider the following simple problem. The total number of orders is 3970. What is the total number of plates? To answer this, we loop through all the orders, summing the number of plates in each order. The answer is 14,949 plates.

```

from json import *
from pandas import *
from sqlalchemy import create_engine, text

protocol = "mysql+pymysql://"
credentials = "username:password@"
server = "servername"
port = ":3306"
database = "/rawa"
uri = protocol + credentials + server + port + database

engine = sqlalchemy.create_engine(uri)

connection = engine.connect()

```

```

query = text("select * from OrdersIn")
df = read_sql(query, connection)

num = 0

for item in df["items"]:
    plates = loads(item)
    num += sum( [ plate["quantity"] for plate in plates ] )

print(num)

```

A more streamlined approach is to use `map`. First we define a function whose input is a JSON string in the format of `df["items"]`, and whose output is the number of plates.

```

from json import *

def num_plates(item):
    dishes = loads(item)
    return sum( [ dish["quantity"] for dish in dishes ] )

```

Then we use `map` to apply to this function to every element in the series `df["items"]`, resulting in another series. Then we sum the resulting series.

```

num = df["items"].map(num_plates).sum()
print(num)

```

Since the total number of plates is 14,949, and the total number of orders is 4970, the average number of plates per order is 3.76.

## A.2 Minimizing Sequences

Several times in the text, we dealt with minimizing functions, most notably for the pseudo-inverse of a matrix (§2.3), for proper continuous functions (§7.5), and for gradient descent (§8.3).

Throughout, the technical foundations underlying the *existence* of mini-

mizers were ignored. In this section, which may safely be skipped, we review the foundational material supporting the existence of minimizers.

The first issue that must be clarified is the difference between the *minimum* and the *infimum*. In a given situation, it is possible that there is no minimum. By contrast, in any reasonable situation, there is always an infimum.

For example, since  $y = e^x$  is an increasing function, the minimum

$$\min_{0 \leq x \leq 1} e^x = \min\{e^x \mid 0 \leq x \leq 1\}$$

is  $y^* = e^0 = 1$ , and the minimizer, the location at which the minimum occurs, is  $x^* = 0$ . Here we have one minimizer.

For the function  $y = x^4 - 2x^2$  in Figure 7.5, the minimum over  $-2 \leq x \leq 2$  is  $y^* = -1$ , which occurs at the minimizers  $x^* = \pm 1$ . Here we have two minimizers.

On the other hand, if we attempt to minimize the function  $y = 1/x$  over the open interval  $1 < x < \infty$ , we have no minimizer, since  $1/x$  approaches 0 as  $x$  approaches  $\infty$ . Here we say there is an *infimum*, and we have

$$\inf_{1 < x < \infty} 1/x = \inf\{1/x \mid 1 < x < \infty\} = 0.$$

In this situation, the minimizer does not exist, but, since the values of  $1/x$  are arbitrarily close to 0, we say the infimum is 0. Since there is no minimizer, there is no minimum value. Also, even though 0 is the infimum, we do not say  $\infty$  is the “infimizer”, since  $\infty$  is not an actual number.



Let  $S$  be a collection of real numbers. A *lower bound* for  $S$  is a number  $b$  satisfying  $b \leq x$  for every  $x$  in  $S$ . For example, 0 is a lower bound for the closed interval  $0 \leq x \leq 1$ , and also 0 is a lower bound for the open interval  $0 < x < 1$ . Any number less than 0 is also a lower bound, for example,  $-1$  is a lower bound, in either case.

Not every collection  $S$  of numbers has a lower bound, for example the entire real line has no lower bound, since  $-\infty$  is not a number. If  $S$  does have a lower bound, we say  $S$  is *bounded below*.

If  $S$  has a lower bound  $m$  that is in  $S$ , then we say  $m$  is the *minimum* of  $S$ . If  $S$  is a finite set, then  $S$  has a minimum. However, as we saw above, if

$S$  is infinite, a minimum need not exist. When the minimum exists, we write  $m = \min S$ .

If  $S$  is bounded below, then  $S$  has many lower bounds. The greatest among these lower bounds is the *infimum* of  $S$ . A foundational axiom for real numbers is that the infimum always exists. When  $m$  is the infimum of  $S$ , we write  $m = \inf S$ .

### Existence of Infima

Any collection  $S$  of real numbers that is bounded below has an infimum: There is a lower bound  $m$  for  $S$  that is greater than any other lower bound  $b$  for  $S$ .

For example, for  $S = [0, 1]$ ,  $\inf S = 0$  and  $\min S = 0$ , and, for  $S = (0, 1)$ ,  $\inf S = 0$ , but  $\min S$  does not exist. For both these sets  $S$ , it is clear that 0 is the infimum. The power of the axiom comes from its validity for *any* set  $S$  of scalars that is bounded below, no matter how complicated.

By definition, the infimum of  $S$  is the lower bound for  $S$  that is greater than any other lower bound for  $S$ . From this, if  $\min S$  exists, then  $\inf S = \min S$ .



A *sequence* is an infinite ordered listing  $x_1, x_2, \dots$  of vectors. An *error sequence* is a sequence of nonnegative scalars  $e_1, e_2, \dots$  that is decreasing

$$e_1 \geq e_2 \geq \dots \geq 0.$$

We say an error sequence *converges to zero* if

$$\inf_{n \geq 1} e_n = 0.$$

In this case, we write  $e_n \rightarrow 0$  as  $n \rightarrow \infty$ . Let's unpack this.

Suppose  $e_1 \geq e_2 \geq \dots$  is an error sequence converging to zero. Since 0 is the greatest lower bound of the set  $S = \{e_1, e_2, \dots\}$ , given any positive  $\epsilon > 0$ , there is a term  $e_N$  satisfying  $e_N < \epsilon$ . Since the sequence is decreasing, we conclude  $0 \leq e_n < \epsilon$  for  $n \geq N$ .

### Error Sequence

An error sequence  $e_1 \geq e_2 \geq \dots \geq 0$  converges to zero iff for any  $\epsilon > 0$ , there is an  $N > 0$  with

$$0 \leq e_n < \epsilon, \quad n \geq N.$$

Now let  $x_1, x_2, \dots$  be a sequence of vectors. We say the sequence *converges to  $x^*$*  or *approaches  $x^*$*  if there is an error sequence  $e_1, e_2, \dots$  converging to zero with

$$|x_n - x^*| \leq e_n \quad n \geq 1.$$

In this case, we write

$$\lim_{n \rightarrow \infty} x_n = x^*,$$

or we write  $x_n \rightarrow x^*$ .

Note this definition of convergence is consistent with the previous definition, since an error sequence  $e_1, e_2, \dots$  converges to zero (in the first sense) iff

$$\lim_{n \rightarrow \infty} e_n = 0$$

(in the second sense).



Let  $x_1, x_2, \dots$  be a sequence. A *subsequence* is a selection of terms

$$x_{n_1}, x_{n_2}, x_{n_3}, \dots, \quad n_1 < n_2 < n_3 < \dots$$

Here it is important that the indices  $n_1 < n_2 < n_3 < \dots$  be strictly increasing.

If a sequence  $x_1, x_2, \dots$  has a subsequence  $x'_1, x'_2, \dots$  converging to  $x^*$ , then we say the sequence  $x_1, x_2, \dots$  *subconverges* to  $x^*$ . For example, the sequence  $1, -1, 1, -1, 1, -1, \dots$  subconverges to 1 and also subconverges to  $-1$ , as can be seen by considering the odd-indexed terms and the even-indexed terms separately.

Note a subsequence of an error sequence converging to zero is also an error sequence converging to zero. As a consequence, if a sequence converges to  $x^*$ , then every subsequence of the sequence converges to  $x^*$ . From this

it follows that the sequence  $1, -1, 1, -1, 1, -1, \dots$  does not converge to anything; it bounces back and forth between  $\pm 1$ .



We say a set  $S$  is *bounded* if  $|x|^2 \leq b$  for all  $x$  in  $S$ , for some constant  $b$ . The scalar  $b$  is then a *bound* for  $S$ .

### Bounded Sequences Must Subconverge

Let  $x_1, x_2, \dots$  be a bounded sequence of vectors. Then there is a subsequence  $x'_1, x'_2, \dots$  converging to some  $x^*$ .

To see this, assume first  $x_1, x_2, \dots$  are scalars, and let  $x_1, x_2, \dots$  be a bounded sequence of numbers, say  $a \leq x_n \leq b$  for  $n \geq 1$ . Bisect the interval  $I_0 = [a, b]$  into two equal subintervals. Then at least one of the subintervals, call it  $I_1$ , has infinitely many terms of the sequence. Select  $x'_1$  in  $I_1$  and let  $x_1^*$  be the right endpoint of  $I_1$ .

Now bisect  $I_1$  into two equal subintervals. Then at least one of the subintervals, call it  $I_2$ , has infinitely many terms of the sequence. Select  $x'_2$  in  $I_2$  and let  $x_2^*$  be the right endpoint of  $I_2$ . Continuing in this manner, we obtain a subsubsequence  $x'_1, x'_2, \dots$  with  $x'_n$  in  $I_n$ , and a sequence  $x_1^*, x_2^*, \dots$

Since the intervals are nested

$$I_0 \supset I_1 \supset I_2 \supset \dots,$$

the sequence  $x_1^*, x_2^*, \dots$  is decreasing and

$$x^* = \inf_{n \geq 1} x_n^*$$

exists. Thus  $e_n = x_n^* - x^*$  is an error sequence converging to zero.

Since the length of  $I_n$  equals  $(b - a)/2^n$ ,

$$0 \leq x_n^* - x_n' \leq (b - a)2^{-n},$$

hence by the triangle inequality (2.2.4)

$$|x_n' - x^*| \leq |x_n' - x_n^*| + |x_n^* - x^*| \leq (b - a)2^{-n} + e_n.$$

Since  $(b - a)2^{-n} + e_n$  is an error sequence converging to zero, this establishes  $x_n' \rightarrow x^*$ .

Now let  $x_1, x_2, \dots$  be a sequence of vectors in  $\mathbf{R}^d$ , and let  $v$  be a vector; then  $x_1 \cdot v, x_2 \cdot v, \dots$  are scalars, so, from the previous paragraph, there is a subsequence  $x'_n \cdot v$  (depending on  $v$ ) converging to some  $x_v^*$ .

Let  $e_1, e_2, \dots, e_d$  be the standard basis in  $\mathbf{R}^d$ . By choosing  $v = e_1$ , there is a subsequence  $x'_1, x'_2, \dots$  such that the first features of  $x'_n$  converge. By choosing  $v = e_2$ , and focusing on the subsequence  $x'_1, x'_2, \dots$ , there is a sub-subsequence  $x''_1, x''_2, \dots$  such that the first and second features of  $x''_n$  converge. Continuing in this manner, we obtain a subsequence  $x_1^*, x_2^*, \dots$  such that the  $k$ -th feature of the subsequence converges to the  $k$ -th feature of a single  $x^*$ , for every  $1 \leq k \leq d$ . From this, it follows that  $x_n^*$  converges to  $x^*$ .



Let  $S$  be a set of vectors and let  $y = f(x)$  be a scalar-valued function bounded below on  $S$ ,  $f(x) \geq b$  for some number  $b$ , for all  $x$  in  $S$ . Then  $b$  is a lower bound for  $f(x)$  over  $S$ . By the above axiom, the infimum

$$m = \inf_S f(x) = \inf\{f(x) \mid x \text{ in } S\} \quad (\text{A.2.1})$$

must exist.

A *minimizer* is a vector  $x^*$  satisfying  $f(x^*) = m$ . As we saw above, a minimizer may or may not exist, and, when the minimizer does exist, there may be several minimizers.

A *minimizing sequence* for  $f(x)$  over  $S$  is a sequence  $x_1, x_2, \dots$  of vectors in  $S$  such that the corresponding values  $f(x_1), f(x_2), \dots$  are *decreasing* and converge to  $m = \inf_S f(x)$  as  $n \rightarrow \infty$ . In other words,  $x_1, x_2, \dots$  is a minimizing sequence for  $f(x)$  over  $S$  if

$$f(x_1) \geq f(x_2) \geq f(x_3) \geq \dots$$

and

$$\inf_S f(x) = \inf_{n \geq 1} f(x_n).$$

If there is a minimizer  $x^*$  in  $S$ , then  $\inf_S f(x) = \min_S f(x) = f(x^*)$ , and the sequence  $x^*, x^*, \dots$  is a minimizing sequence in  $S$ . However, in general, there may be no such minimizer.

### Existence of Minimizing Sequences

If  $S$  is a collection of vectors, and  $y = f(x)$  is bounded below on  $S$ , then there is a minimizing sequence  $x_1, x_2, \dots$  in  $S$ .

If there is a minimizer  $x^*$  in  $S$ , the sequence  $x^*, x^*, \dots$  is a minimizing sequence in  $S$ . Otherwise, if there is no minimizer in  $S$ , pick any  $x_0$  in  $S$ . Since  $m$  is the greatest lower bound for  $f(x)$ ,  $f(x_0)$  is not a lower bound, so there is  $x_1$  in  $S$  with

$$m < f(x_1) < (f(x_0) + m)/2,$$

or

$$0 < f(x_1) - m < (f(x_0) - m)/2.$$

Similarly, there is  $x_2$  with

$$0 < f(x_2) - m < (f(x_1) - m)/2.$$

Continuing in this manner, we have  $x_n$  with

$$0 < f(x_n) - m < (f(x_{n-1}) - m)/2.$$

Since this implies

$$0 < f(x_n) - m < 2^{-n}(f(x_0) - m),$$

this yields a minimizing sequence in  $S$ .

We note a subsequence of a minimizing sequence is also a minimizing sequence.



A function  $y = f(x)$  is *continuous* if  $f(x_n)$  approaches  $f(x^*)$  whenever  $x_n$  approaches  $x^*$ .

Now we can establish

### Existence of Minimizers

If  $f(x)$  is continuous on  $\mathbf{R}^d$  and  $S$  is a bounded set in  $\mathbf{R}^d$ , then there is a minimizer  $x^*$ ,

$$f(x^*) = \inf_{x \text{ in } S} f(x). \quad (\text{A.2.2})$$

In general, the minimizer  $x^*$  may lie outside the set  $S$ . To guarantee  $x^*$  belongs to  $S$ , typically one assumes an additional requirement, the closedness of  $S$ . In our applications of this result, this point is of no concern.

To establish the result, let  $m$  be as in (A.2.1), and let  $x_1, x_2, \dots$  be a minimizing sequence for  $f(x)$  in  $S$ . Then  $x_1, x_2, \dots$  is bounded, so by the previous result, there is a subsequence  $x'_1, x'_2, \dots$  converging to some  $x^*$ . Since  $x'_1, x'_2, \dots$  is also a minimizing sequence, and  $f(x)$  is continuous,

$$f(x^*) = \lim_{n \rightarrow \infty} f(x'_n) = \lim_{n \rightarrow \infty} f(x_n) = m.$$

This shows  $x^*$  is a minimizer for  $f(x)$ .

## A.3 Keras Training

★ under construction ★

This section trains classifier networks for the Iris and MNIST datasets using `keras`.

# Bibliography

- [1] Joshua Akey, *Genome 560: Introduction to Statistical Genomics*, 2008. <https://www.gs.washington.edu/academics/courses/akey/56008/lecture/lecture1.pdf>.
- [2] Christopher M. Bishop, *Pattern Recognition and Machine Learning*, Information Science and Statistics, Springer, 2006.
- [3] Sébastien Bubeck, *Convex Optimization: Algorithms and Complexity*, Foundations and Trends in Machine Learning, vol. 8, Now Publishers, 2015.
- [4] Harald Cramér, *Mathematical Methods of Statistics*, Princeton University Press, 1946.
- [5] A. Aldo Faisal Marc Peter Deisenroth and Cheng Soon Ong, *Mathematics for Machine Learning*, Cambridge University Press, 2020.
- [6] J. L. Doob, *Probability and Statistics*, Transactions of the American Mathematical Society **36** (1934), 759-775.
- [7] R. A. Fisher, *The conditions under which  $\chi^2$  measures the discrepancy between observation and hypothesis*, Journal of the Royal Statistical Society **87** (1924), 442-450.
- [8] Ian Goodfellow and Yoshua Bengio and Aaron Courville, *Deep Learning*, MIT Press, 2016.
- [9] Google, *Machine Learning*. <https://developers.google.com/machine-learning>.
- [10] Robert M. Gray, *Toepplitz and Circulant Matrices: A Review*, Foundations and Trends in Communications and Information Theory **2** (2006), no. 3, 155-239.
- [11] T. L. Heath, *The Works of Archimedes*, Cambridge University, 1897.
- [12] Omar Hijab, *A Note on Binary Classifiers*, Preprint (2024).
- [13] Nikolai Janakiev, *Classifying the Iris Data Set with Keras*, 2018. <https://janakiev.com/blog/keras-iris>.
- [14] Lily Jiang, *A Visual Explanation of Gradient Descent Methods*, 2020. <https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c>.
- [15] J. W. Longley, *An Appraisal of Least Squares Programs for the Electronic Computer from the Point of View of the User*, Journal of the American Statistical Association **62.319** (1967), 819-841.

- [16] David G. Luenberger and Yinyu Ye, *Linear and Nonlinear Programming*, Springer, 2008.
- [17] Ioannis Mitliagkas, *Theoretical principles for deep learning, lecture notes*, 2019. <http://mitliagkas.github.io/ift6085-dl-theory-class-2019/>.
- [18] Marvin Minsky and Seymour Papert, *Perceptrons, An Introduction to Computational Geometry*, MIT Press, 1988.
- [19] Yurii Nesterov, *Lectures on Convex Optimization*, Springer, 2018.
- [20] Roger Penrose, *A generalized inverse for matrices*, Proceedings of the Cambridge Philosophical Society **51** (1955), 406-413.
- [21] Boris Teodorovich Polyak, *Some methods of speeding up the convergence of iteration methods*, USSR Computational Mathematics and Mathematical Physics **4(5)** (1964), 1-17.
- [22] Karl Pearson, *On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling*, Philosophical Magazine Series 5 **50:302** (1900), 157-175.
- [23] Sebastian Raschka, *PCA in three simple steps*, 2015. [https://sebastianraschka.com/Articles/2015\\_pca\\_in\\_3\\_steps.html](https://sebastianraschka.com/Articles/2015_pca_in_3_steps.html).
- [24] Herbert Robbins and Sutton Monro, *A Stochastic Approximation Method*, The Annals of Mathematical Statistics **22** (1951), no. 3, 400 – 407.
- [25] Sheldon M. Ross, *Probability and Statistics for Engineers and Scientists, Sixth Edition*, Academic Press, 2021.
- [26] Mark J. Schervish, *Theory of Statistics*, Springer, 1995.
- [27] Stanford University, *CS224N: Natural Language Processing with Deep Learning*. <https://web.stanford.edu/class/cs224n>.
- [28] Irène Waldspurger, *Gradient Descent With Momentum*, 2022. [https://www.cerema.de.dauphine.fr/~waldspurger/tds/22\\_23\\_s1/advanced\\_gradient\\_descent.pdf](https://www.cerema.de.dauphine.fr/~waldspurger/tds/22_23_s1/advanced_gradient_descent.pdf).
- [29] Wikipedia, *Logistic Regression*, 2015. [https://en.wikipedia.org/wiki/Logistic\\_regression](https://en.wikipedia.org/wiki/Logistic_regression).
- [30] Stephen J. Wright and Benjamin Recht, *Optimization for Data Analysis*, Cambridge University, 2022.

# Python

\*<sup>9</sup>, 16  
`all`, 189  
`append`, 189  
`def.angle`, 24, 75  
`def.assign_clusters`, 189  
`def.backward_prop`, 361, 369,  
    413  
`def.ball`, 62  
`def.chi2_independence`, 323  
`def.confidence_interval`, 295,  
    307, 312, 315  
`def.delta_out`, 412  
`def.derivative`, 368  
`def.dimension_staircase`, 130  
`def.downstream`, 413  
`def.ellipse`, 51, 58  
`def.find_first_defect`, 128  
`def.forward_prop`, 360, 367, 408  
`def.gd`, 424  
`def.goodness_of_fit`, 320  
`def.H`, 347  
`def.hexcolor`, 10  
`def.incoming`, 366, 407  
`def.J`, 409  
`def.local`, 410  
`def.nearest_index`, 189  
`def.newton`, 417  
`def.num_plates`, 479  
`def.outgoing`, 366, 407  
`def.pca`, 182  
`def.pca_with_svd`, 182  
`def.plot_cluster`, 190  
`def.plot_descent`, 418  
`def.poly`, 444  
`def.project`, 121  
`def.project_to_ortho`, 122  
`def.pvalue`, 272  
`def.random_batch_mean`, 249  
`def.random_vector`, 189  
`def.tensor`, 32  
`def.train_nn`, 426  
`def.ttest`, 308  
`def.type2_error`, 301, 309  
`def.update_means`, 189  
`def.update_weights`, 426  
`def.zero_variance`, 108  
`def.ztest`, 299  
`diag`, 175  
`dict`, 467  
`display`, 150  
`enumerate`, 183  
`floor`, 167

```
import, 8
itertools.product, 62

join, 10
json.dumps, 468
json.loads, 468

keras
    datasets
        mnist.load_data, 4

lambda, 365
list, 7

matplotlib.pyplot.axes, 187
matplotlib.pyplot.contour, 58
matplotlib.pyplot.figure, 183
matplotlib.pyplot.grid, 7
matplotlib.pyplot.hist, 243
matplotlib.pyplot.imshow, 8, 9
matplotlib.pyplot.legend, 167
matplotlib.pyplot.meshgrid,
    58
matplotlib.pyplot.plot, 45
matplotlib.pyplot.scatter, 7
matplotlib.pyplot.show, 7
matplotlib.pyplot.stairs, 130
matplotlib.pyplot.subplot,
    183
matplotlib.pyplot.text, 168
matplotlib.pyplot.title, 347
matplotlib.pyplot.xlabel, 445

numpy.allclose, 83
numpyamax, 445
numpyamin, 445
numpy.arange, 58, 167
numpy.arccos, 24, 75
numpy.argsort, 182

numpy.array, 8, 65
numpy.column_stack, 86, 100
numpy.corrcoef, 54
numpy.cov, 47
numpy.cumsum, 180
numpy.degrees, 24
numpy.dot, 73
numpy.exp, 347
numpy.isclose, 158
numpy.linalg.eig, 144
numpy.linalg.eigh, 144, 180
numpy.linalg.inv, 84
numpy.linalg.matrix_rank, 128
numpy.linalg.norm, 21, 189
numpy.linalg.pinv, 121
numpy.linalg.svd, 175
numpy.linspace, 62
numpy.log, 347
numpy.mean, 14, 15
numpy.meshgrid, 62
numpy.outer, 323
numpy.pi, 347
numpy.random.binomial, 242
numpy.random.default_rng, 249
numpy.random.default_rng.
    → shuffle, 249
numpy.random.normal, 271
numpy.random.randn, 286
numpy.random.random, 45
numpy.reshape, 179
numpy.roots, 43
numpy.row_stack, 69
numpy.shape, 65
numpy.sqrt, 24

pandas.DataFrame, 468
pandas.DataFrame.drop, 72
pandas.DataFrame.to_csv, 470
```

pandas.DataFrame.to\_dict, 469  
pandas.DataFrame.to\_numpy, 72  
pandas.DataFrame.to\_sql, 471  
pandas.read\_csv, 443, 469  
pandas.read\_sql, 471  
  
random.choice, 10  
random.random, 15  
  
scipy.linalg.null\_space, 99,  
    100  
scipy.linalg.orth, 93  
scipy.linalg.pinv, 87  
scipy.spatial.ConvexHull, 373  
    simplices, 374  
scipy.special.comb, 218  
scipy.special.expit, 238  
scipy.special.softmax, 388  
scipy.stats.chi2, 276  
scipy.stats.norm, 263  
scipy.stats.t, 304, 307  
sklearn.datasets.load\_iris, 2  
sklearn.decomposition  
    .PCA, 184  
sklearn.preprocessing  
    .StandardScaler, 83  
sort, 180  
sqlalchemy.create\_engine, 471  
  
sqlalchemy.text, 471  
sympy.\*., 73  
sympy.diag, 72  
sympy.diagonalize, 149  
sympy.eigenvects, 149  
sympy.init\_printing, 149  
sympy.Matrix, 65  
sympy.Matrix.col, 70  
sympy.Matrix.cols, 70  
sympy.Matrix.columnspace, 92  
sympy.Matrix.eye, 71  
sympy.Matrix.hstack, 68, 87  
sympy.Matrix.inv, 84  
sympy.Matrix.nullspace, 98  
sympy.Matrix.ones, 71  
sympy.Matrix.rank, 134  
sympy.Matrix.row, 70  
sympy.Matrix.rows, 70  
sympy.Matrix.rowspace, 96  
sympy.Matrix.zeros, 71  
sympy.RootOf, 43  
sympy.shape, 65  
sympy.solve, 257  
sympy.symbols, 43  
  
tuple, 18  
  
zip, 186

492

*PYTHON*

# Index

- 1, 203, 387
- angle, 75, 136
- approaches, 482
- approximate equality, 167
- Archimedes, 39
- arcsine law, 331
- asymptotic equality, 198
- average, 11
- basis, 125
  - of eigenvectors, 148
  - of singular vectors, 172
  - orthonormal, 125, 137, 148
  - standard, 66
- Bayes theorem, 235
  - perceptron, 240
- binomial, 212
  - coefficient, 196, 213, 214
  - theorem, 212, 214
  - Newton's, 342
- cartesian plane, 17
- Cauchy-Schwarz inequality, 24, 75
- central limit theorem, 244, 263,  
265
- circle, 22
  - unit, 21
- coin-tossing, 229
- entropy, 347
  - relative, 350
- column space, 92
- columns, 68
  - orthonormal, 79
- combination, 194
- complex
  - conjugate, 36
  - division, 35, 37
  - hermitian product, 37
  - multiplication, 35, 36
  - numbers, 35
  - plane, 35
  - polar representation, 39
  - roots of unity, 40
- concave, 333, 343
- condition number, 452
- confidence, 268
  - interval, 294
  - level, 292
- contingency table, 322
- convex, 333
  - combination, 372
  - dual, 339, 385, 390, 394
  - function, 371
  - hull, 373, 436, 438, 439
  - set, 373

- strictly, 334
- correlation
  - coefficient, 53
  - matrix, 53, 83
- covariance, 46, 81
  - and correlation, 83
  - and variance, 50
  - biased, 47
  - ellipse, 50
  - inverse ellipse, 50
  - inverse ellipsoid, 158
  - matrix, 46
  - unbiased, 47
- CSV file, 70
- dataset, 1
  - attributes, 1
  - binary classification, 452
  - centered, 12
  - covariance, 46
  - dimension, 137
  - example, 1
  - features, 1
  - full-rank, 137
  - Iris, 1
  - mean, 44
  - projected, 49, 105, 123, 184
  - reduced, 49, 105, 123, 184
  - sample, 1
  - standardized, 53, 83
  - vectors or points, 13
- degree, 212
- derivative
  - definition, 327
  - directional, 351
  - logarithm, 339
  - partial, 351
  - second, 328
- convexity, 334
- descent
  - gradient, 417, 454, 455
  - heavy ball, 459
  - sequence, 419
  - with lookahead gradient, 461
  - with momentum, 459
- diagonalizable, 147
- diagonalization
  - eigen, 148
  - singular, 175
- dimension, 125
  - staircase, 130
- direct sum, 123
- distance formula, 20
- distribution
  - bernoulli, 234
  - binomial, 234
  - chi-squared, 274, 276
  - cumulative, 252
  - $F$ -, 317
  - normal, 263, 264
  - $T$ -, 304
  - $Z$ -, 263, 264
- dot product, 23, 73
- eigenspace, 158
- eigenvalue, 143
  - bottom, 156
  - clustering, 167
  - decomposition, 148
  - minimum variance, 156
  - projected variance, 154
  - top, 154
  - transpose, 145
- eigenvector, 142
- eigenvectors
  - best-aligned vector, 154

- is right singular vector, 176
- orthogonal, 146
- entropy, 344, 391
  - cross-, 395
  - relative, 348, 393
- epigraph, 375
- epoch, 400
- error
  - logistic, 433
  - mean square, 430
  - standard, 261
- Euler's constant, 221
- experiment, 240
- exponential
  - function, 223
  - series, 225
- factorial, 194
- full-rank
  - dataset, 137
  - matrix, 134
- function
  - bound, 379, 483
  - cumulative distribution, 252
  - error
    - information, 394, 429
    - mean, 400
    - mean square, 409, 429
  - level, 379
  - logistic, 238, 341
  - loss, 416, 429
  - moment generating, 256
    - chi-squared, 276
    - independence, 258
  - partition, 341, 387
  - probability density, 263
  - sigmoid, 238, 341
  - softmax, 388
- fundamental theorem of algebra, 43
- gradient, 352
  - weight, 414, 426
- graph, 198
  - bipartite, 209
  - complement, 205
  - complete, 200
  - connected, 202
  - cycle, 200, 202
  - directed, 198
  - edge, 198, 401
    - incoming, 401
    - outgoing, 401
  - isomorphism, 208
  - laplacian, 211
  - nodes, 198, 401
    - adjacent, 198
    - connected, 202
    - degree, 201
    - dominating, 201
    - hidden, 401
    - input, 401
    - isolated, 201
    - output, 401
  - order, 199
  - path, 202
  - regular, 202
  - simple, 199
  - size, 199
  - sub-, 200
  - undirected, 198
  - walk, 202
  - weighed, 198
  - weight
    - matrix, 401
- hyperplane, 105, 376

separating, 377  
 supporting, 378  
 tangent, 379  
 hypothesis  
     alternate, 297  
     null, 297  
     testing, 297  
 iff, 79, 145  
 incoming edge, 401  
 independence, 245  
 infimum, 480, 481  
 information, 346  
     cross-, 395  
     relative, 348, 392  
 inverse, 84  
     pseudo-, 114  
 Iris dataset, 1  
 iteration, 400  
 Jupyter, 4  
 law of large numbers, 244, 262,  
     289, 351  
 line-search, 456  
 linear  
     combination, 90  
     dependence, 97  
     independence, 97  
     system, 84, 151  
         homogeneous, 27, 98  
         inhomogeneous, 28  
         transformation, 133, 139  
 log-odds, 341  
 logit, 341  
 loss, 416, 429  
     cross-entropy, 433  
     information, 429  
     logistic, 433  
 mean, 400  
 mean square, 409, 429, 430  
 lower bound, 480  
 machine learning, 399  
 margin of error, 292  
 mass-spring system, 161  
 matrix, 66  
      $2 \times 2$ , 28  
     addition, 71  
     adjacency, 199, 203  
     augmented, 94  
     circulant, 167, 205  
         eigenvalues, 167  
     columns, 29  
     covariance, 46  
     dataset, 70  
     diagonal, 70  
     identity, 84  
     incidence, 211  
     inverse, 31, 84  
     nonnegative, 80  
     orthogonal, 136  
     positive, 80  
     projection, 118  
     rows, 28  
     scaling, 71  
     square, 70  
     symmetric, 31, 80  
     trace, 33  
     transpose, 29, 67  
     weight, 199, 401  
 maximizer, 339  
 mean, 11, 44, 253, 278  
     population, 253  
     sample, 261  
 minimizer, 480, 484  
     existence, 381

- global, 379
- residual, 382
- uniqueness, 382
- minimizers
  - existence, 486
- minimum, 480
- network, 366, 401
  - deep, 414
  - iteration, 426
  - neural, 401
    - layered, 414
    - training, 425
  - neuron, 366, 401
  - perceptron, 402
  - shallow, 414
    - dense, 414
  - trainability, 430
- Newton's method, 417
- norm, 21, 81
- null space, 98
- 1, 203, 387
- one-hot encoded, 395
- orthogonal, 75
  - complement, 101, 123
- orthonormal, 76
- outgoing edge, 401
- Pascal's triangle, 215
- perceptron, 240, 402
  - Bayes theorem, 403
  - parallel, 414
- permutation, 194
- point, 65
  - critical, 333, 355, 423
  - inflection, 334, 423
  - saddle, 333, 355
- point of best-fit, 44
- population, 9
- power of a test, 302
- principal axes, 57
- principal components, 148, 178
- probability, 240
  - binomial, 229
  - coin-tossing, 230
  - conditional, 230, 245
  - multinomial, 387
  - one-hot encoded, 432
  - strict, 432
- product
  - dot, 23, 73, 136
  - matrix-matrix, 77
  - matrix-vector, 76
  - tensor, 32, 79
- projection, 118
  - onto null space, 124
  - onto row space, 122
- propagation
  - back, 359, 361
    - chain, 361
    - network, 369
    - neural network, 413
  - forward, 358, 360
    - chain, 360
    - network, 367
    - neural network, 408
- proper, 380
- pseudo-inverse, 114
- Pythagoras theorem, 24
- Python, 4
  - installation, 4
- quadratic form, 33
- random variables, 246, 247
  - bernoulli, 234, 246

- correlation, 256
- identically distributed, 260
- independence, 258
- moments, 256
- standard, 255
- rank, 134
  - and eigenvalues, 150
  - and singular values, 172
  - column, 93
  - full-, 134
  - nonzero eigenvalues, 150
  - row, 96
- regression
  - linear, 430, 440, 441
    - convexity, 430
    - neural network, 430
    - properness, 431
    - trainability, 432
  - logistic, 433
    - convexity, 435
    - neural network, 433
    - one-hot encoded, 439
    - properness, 436
    - strict, 438
    - trainability, 438, 439
- regularization, 422
- residual, 110
  - minimizer, 111
    - minimum norm, 113
    - pseudo-inverse, 114
    - regression equation, 112
    - vanishing, 111
- row space, 96
- rows, 68
  - orthonormal, 79
- scaling
  - factor, 140
- principle, 62
- sequence, 481
  - convergent, 482
  - error, 481
  - minimizing, 484
  - sub-, 482
  - subconvergent, 482
- series
  - alternating, 217
  - exponential, 225
  - Taylor, 329, 330
- singular
  - value, 169
  - decomposition, 172
  - of pseudo-inverse, 177
- vector, 169
- vectors
  - left, 169
  - right, 169
- slope, 325
- space
  - column, 92
  - eigen-, 158
  - feature, 1, 65, 96
  - null, 98
  - row, 96
  - sample, 9
  - source, 133
  - sub-, 103
  - target, 133
  - vector, 11
- span, 91
- spherical coordinates, 62
- statistic, 13
- Stirling's approximation, 198
- sublevel set, 370
- sum
  - direct, 123

- of spans, 123
- of vectors, 66
- 
- t*-distribution, 304
- tangent
  - line, 325
  - parabola, 335
- test
  - goodness of fit, 319
  - independence, 323
  - $T$ , 308
  - $Z$ , 299
- trainability, 430
  - linear regression, 432
  - logistic regression, 438
    - one-hot encoded, 439
    - strict, 438
- transpose, 67
- triangle inequality, 76
- 
- unit circle, 21
- 
- variance, 46, 105, 254
  - population, 254
  - projected, 50, 105, 141, 146
  - sample, 282
  - total, 48
  - zero, 105
- vectors, 12, 17, 65
- 
- addition, 18, 66
- best aligned, 54
- cartesian, 18
- dimension, 65
- dot product, 23
- gradient, 352
  - downstream, 411
- incoming, 402
- length, 21, 74
- magnitude, 74
- norm, 21, 74
- orthogonal, 75
- orthonormal, 76, 100
- outgoing, 364, 402
- polar, 21
- projected, 119–122
- random, 278, 286
- reduced, 119–122
- scaling, 19
- shadow, 17
- span, 91
- standardized, 82
- subtraction, 20
- unit, 21, 74
- zero, 18, 66
- 
- weight, 401
  - gradient, 414, 426
  - matrix, 199





Dr. Omar Hijab obtained his doctorate in mathematics from the University of California at Berkeley. Currently he is Professor Emeritus at Temple University, Philadelphia, Pennsylvania, USA.