

HALIOTIS PUBLISHING

# Essential Math for Data Science

CALCULUS, STATISTICS, PROBABILITY  
THEORY AND LINEAR ALGEBRA

Hadrien Jean



---

# **Essential Math for Data Science**

## **CALCULUS, STATISTICS, PROBABILITY THEORY AND LINEAR ALGEBRA**

---

**Hadrien Jean**

**HALIOTIS PUBLISHING**

**Essential Math for Data Science**

by Hadrien Jean

Copyright ©2020 – Hadrien Jean

All rights reserved.

Published by Haliotis Publishings, 33 rue du Four, 75006, Paris, France.



Pour Zéphyr



# Contents

<b>Acknowledgments</b>	<b>1</b>
<b>Introduction</b>	<b>3</b>
0.1 Motivation . . . . .	3
0.1.1 What Math Do You Need? . . . . .	4
0.1.2 Teaching Method . . . . .	4
0.2 What you'll need . . . . .	6
0.2.1 A Bit of Math . . . . .	6
0.2.2 Some Python Experience . . . . .	6
0.2.3 Exposure to Numpy and Matplotlib . . . . .	6
<b>I Calculus</b>	<b>9</b>
<b>1 Calculus: Derivatives and Integrals</b>	<b>13</b>
1.1 Derivatives . . . . .	13
1.1.1 Introduction . . . . .	13
1.1.2 Mathematical Definition of Derivatives . . . . .	16
1.1.3 Derivatives of Linear And Nonlinear Functions . . . . .	18
1.1.4 Derivative Rules . . . . .	22
1.1.5 Partial Derivatives And Gradients . . . . .	26
1.2 Integrals and the Area Under the Curve . . . . .	29
1.2.1 Example . . . . .	29
1.2.2 Riemann Sum . . . . .	35
1.2.3 Mathematical Definition . . . . .	36
1.3 Hands-On Project: Gradient Descent . . . . .	37
1.3.1 Cost function . . . . .	38

## Contents

---

1.3.2	Derivative of the Cost Function . . . . .	42
1.3.3	Implementing Gradient Descent . . . . .	45
1.3.4	BONUS: MSE Cost Function With Two Parameters . . . . .	50
<b>II</b>	<b>Statistics and Probability</b>	<b>55</b>
<b>2</b>	<b>Statistics and Probability Theory</b>	<b>59</b>
2.1	Descriptive Statistics . . . . .	59
2.1.1	Mean, Variance and Standard Deviation . . . . .	59
2.1.2	Covariance and Correlation . . . . .	61
2.1.3	Covariance Matrix . . . . .	65
2.2	Random Variables . . . . .	66
2.2.1	Definitions and Notation . . . . .	67
2.2.2	Discrete and Continuous Random Variables . . . . .	70
2.3	Probability Distributions . . . . .	71
2.3.1	Probability Mass Functions . . . . .	71
2.3.2	Probability Density Functions . . . . .	74
2.4	Joint, Marginal, and Conditional Probability . . . . .	79
2.4.1	Joint Probability . . . . .	79
2.4.2	Marginal Probability . . . . .	81
2.4.3	Conditional Probability . . . . .	88
2.5	Cumulative Distribution Functions . . . . .	92
2.6	Expectation and Variance of Random Variables . . . . .	93
2.6.1	Discrete Random Variables . . . . .	93
2.6.2	Continuous Random Variables . . . . .	95
2.6.3	Variance of Random Variables . . . . .	95
2.7	Hands-On Project: The Central Limit Theorem . . . . .	96
2.7.1	Continuous Distribution . . . . .	96
2.7.2	Discrete Distribution . . . . .	97
<b>3</b>	<b>Common Probability Distributions</b>	<b>99</b>
3.1	Uniform Distribution . . . . .	99
3.2	Gaussian distribution . . . . .	100
3.2.1	Formula . . . . .	100
3.2.2	Parameters . . . . .	102
3.2.3	Requirements . . . . .	103
3.3	Bernoulli Distribution . . . . .	104

3.4	Binomial Distribution . . . . .	105
3.4.1	Description . . . . .	105
3.4.2	Graphical Representation . . . . .	108
3.5	Poisson Distribution . . . . .	109
3.5.1	Mathematical Definition . . . . .	109
3.5.2	Example . . . . .	110
3.5.3	Bonus: Deriving the Poisson Distribution . . . . .	114
3.6	Exponential Distribution . . . . .	117
3.6.1	Derivation from the Poisson Distribution . . . . .	118
3.6.2	Effect of $\lambda$ . . . . .	119
3.7	Hands-on Project: Waiting for the Bus . . . . .	119
<b>4</b>	<b>Bayesian Statistics and Information Theory</b>	<b>125</b>
4.1	Bayes' Theorem . . . . .	125
4.1.1	Mathematical Formulation . . . . .	125
4.1.2	Example . . . . .	127
4.1.3	Bayesian Interpretation . . . . .	127
4.1.4	Bayes' Theorem with Distributions . . . . .	128
4.2	Likelihood . . . . .	129
4.2.1	Introduction and Notation . . . . .	130
4.2.2	Finding the Parameters of the Distribution . . . . .	132
4.2.3	Maximum Likelihood Estimation . . . . .	135
4.3	Information Theory . . . . .	138
4.3.1	Shannon Information . . . . .	138
4.3.2	Entropy . . . . .	141
4.3.3	Cross-Entropy . . . . .	147
4.3.4	Kullback-Leibler Divergence (KL Divergence) . . . . .	151
4.4	Hands-On Project: Bayesian Inference . . . . .	152
4.4.1	Bayesian Inference . . . . .	152
4.4.2	Project . . . . .	153
<b>III</b>	<b>Linear Algebra</b>	<b>163</b>
<b>5</b>	<b>Scalars and Vectors</b>	<b>167</b>
5.1	What are Vectors? . . . . .	167
5.1.1	Geometric and Coordinate Vectors . . . . .	168
5.1.2	Vector Spaces . . . . .	173

## Contents

---

5.1.3	Special Vectors . . . . .	174
5.2	Operations and Manipulations on Vectors . . . . .	175
5.2.1	Scalar Multiplication . . . . .	175
5.2.2	Vector Addition . . . . .	177
5.2.3	Transposition . . . . .	178
5.3	Norms . . . . .	179
5.3.1	Definitions . . . . .	180
5.3.2	Common Vector Norms . . . . .	181
5.3.3	Norm Representations . . . . .	185
5.4	The Dot Product . . . . .	186
5.4.1	Definition . . . . .	187
5.4.2	Geometric interpretation: Projections . . . . .	190
5.4.3	Properties . . . . .	192
5.5	Hands-on Project: Regularization . . . . .	193
5.5.1	Introduction . . . . .	193
5.5.2	Effect of Regularization on Polynomial Regression . . . . .	196
5.5.3	Differences between $L^1$ and $L^2$ Regularization . . . . .	201
<b>6</b>	<b>Matrices and Tensors</b>	<b>211</b>
6.1	Introduction . . . . .	211
6.1.1	Matrix Notation . . . . .	212
6.1.2	Shapes . . . . .	213
6.1.3	Indexing . . . . .	214
6.1.4	Main Diagonal . . . . .	216
6.1.5	Tensors . . . . .	216
6.1.6	Frobenius Norm . . . . .	218
6.2	Operations and Manipulations on Matrices . . . . .	218
6.2.1	Addition and Scalar Multiplication . . . . .	218
6.2.2	Transposition . . . . .	221
6.3	Matrix Product . . . . .	223
6.3.1	Matrices with Vectors . . . . .	223
6.3.2	Matrices Product . . . . .	226
6.3.3	Transpose of a Matrix Product . . . . .	229
6.4	Special Matrices . . . . .	231
6.4.1	Square Matrices . . . . .	232
6.4.2	Diagonal Matrices . . . . .	232
6.4.3	Identity Matrices . . . . .	233
6.4.4	Inverse Matrices . . . . .	235

6.4.5	Orthogonal Matrices . . . . .	238
6.4.6	Symmetric Matrices . . . . .	240
6.4.7	Triangular Matrices . . . . .	241
6.5	Hands-on Project: Image Classifier . . . . .	242
6.5.1	Images as Multi-dimensional Arrays . . . . .	242
6.5.2	Data Preparation . . . . .	245
<b>7</b>	<b>Span, Linear Dependency, and Space Transformation</b>	<b>249</b>
7.1	Linear Transformations . . . . .	249
7.1.1	Intuition . . . . .	249
7.1.2	Linear Transformations as Vectors and Matrices . . . . .	250
7.1.3	Geometric Interpretation . . . . .	251
7.1.4	Special Cases . . . . .	256
7.2	Linear combination . . . . .	259
7.2.1	Intuition . . . . .	259
7.2.2	All combinations of vectors . . . . .	262
7.2.3	Span . . . . .	264
7.3	Subspaces . . . . .	264
7.3.1	Definitions . . . . .	264
7.3.2	Subspaces of a Matrix . . . . .	266
7.4	Linear dependency . . . . .	268
7.4.1	Geometric Interpretation . . . . .	268
7.4.2	Matrix View . . . . .	269
7.5	Basis . . . . .	270
7.5.1	Definitions . . . . .	270
7.5.2	Linear Combination of Basis Vectors . . . . .	272
7.5.3	Other Bases . . . . .	273
7.5.4	Vectors Are Defined With Respect to a Basis . . . . .	274
7.6	Special Characteristics . . . . .	275
7.6.1	Rank . . . . .	275
7.6.2	Trace . . . . .	277
7.6.3	Determinant . . . . .	280
7.7	Hands-On Project: Span . . . . .	282
<b>8</b>	<b>Systems of Linear Equations</b>	<b>287</b>
8.1	System of linear equations . . . . .	288
8.1.1	Row Picture . . . . .	289
8.1.2	Column Picture . . . . .	290

## Contents

---

8.1.3	Number of Solutions . . . . .	293
8.1.4	Representation of Linear Equations With Matrices . . . . .	298
8.2	System Shape . . . . .	300
8.2.1	Overdetermined Systems of Equations . . . . .	300
8.2.2	Underdetermined Systems of Equations . . . . .	301
8.3	Projections . . . . .	303
8.3.1	Solving Systems of Equations . . . . .	304
8.3.2	Projections to Approximate Unsolvable Systems . . . . .	304
8.3.3	Projections Onto a Line . . . . .	308
8.3.4	Projections Onto a Plane . . . . .	312
8.4	Hands-on Project: Linear Regression Using Least Squares Approximation . . . . .	315
8.4.1	Linear Regression Using the Normal Equation . . . . .	316
8.4.2	Food Data . . . . .	321
8.4.3	Bonus: Relationship Between Least Squares and the Normal Equation . . . . .	324
<b>9</b>	<b>Eigenvectors, Eigenvalues, and Eigendecomposition</b>	<b>329</b>
9.1	Eigenvectors and Eigenvalues . . . . .	329
9.2	Change of Basis . . . . .	333
9.2.1	Linear Combinations of the Basis Vectors . . . . .	334
9.2.2	The Change of Basis Matrix . . . . .	336
9.2.3	Example: Changing the Basis of a Vector . . . . .	340
9.3	Linear Transformations in Different Bases . . . . .	343
9.3.1	Transformations . . . . .	343
9.3.2	Transformation Matrix in Another Basis . . . . .	345
9.3.3	Interpretation . . . . .	348
9.4	Eigendecomposition . . . . .	348
9.4.1	First Step: Change of Basis . . . . .	349
9.4.2	Eigenvectors and Eigenvalues . . . . .	349
9.4.3	Diagonalization . . . . .	351
9.4.4	Eigendecomposition of Symmetric Matrices . . . . .	351
9.5	Hands-On Project: Principal Component Analysis . . . . .	352
9.5.1	Under the Hood . . . . .	352
9.5.2	Making Sense of Audio . . . . .	360
<b>10</b>	<b>Singular Value Decomposition</b>	<b>373</b>
10.1	Non-square Matrices . . . . .	374

10.1.1	Different Input and Output Spaces . . . . .	374
10.1.2	Specifying the Bases . . . . .	375
10.2	Expression of the SVD . . . . .	379
10.2.1	Notation . . . . .	379
10.2.2	Singular Vectors and Singular Values . . . . .	380
10.2.3	Finding the Singular Vectors and the Singular Values .	382
10.2.4	Summary . . . . .	389
10.3	Geometry of the SVD . . . . .	390
10.3.1	Two-Dimensional Example . . . . .	390
10.3.2	Comparison with Eigendecomposition . . . . .	394
10.3.3	Three-Dimensional Example . . . . .	396
10.3.4	Summary . . . . .	401
10.4	Low-Rank Matrix Approximation . . . . .	401
10.4.1	Full SVD, Thin SVD and Truncated SVD . . . . .	402
10.4.2	Decomposition into Rank One Matrices . . . . .	404
10.5	Hands-On Project: Image Compression . . . . .	405
<b>Conclusion</b>		<b>411</b>
<b>A Appendix A. Graphical Representation of Equations</b>		<b>413</b>
A.1	Plotting Equations . . . . .	413
A.1.1	Two Variables . . . . .	413
A.1.2	More Than Two Variables . . . . .	417
A.2	Slope And Intercept . . . . .	419
A.2.1	Slope . . . . .	419
A.2.2	$y$ -Intercept . . . . .	420
<b>B Appendix B. Mathematical Notation</b>		<b>423</b>
B.1	Greek Letters . . . . .	423
B.2	Calculus . . . . .	424
B.3	Dataset . . . . .	424
B.4	Probability . . . . .	424
B.5	Information Theory . . . . .	426
B.6	Sets . . . . .	427
B.7	Linear Algebra . . . . .	427

## Contents

---

# Acknowledgments

I am so thankful to all the people that helped me to create this book. I received a lot of messages from readers about early released samples of the book, sending some comments or just showing support. I have to say that this helped me a lot throughout the writing process. Thank you all for that!

This work also benefited from a lot of amazing people who took the time to read the book with great care and give me their technical reviews. I am especially grateful to Tai-Danae Bradley, William A. Huber, Josh Starmer, Davy Wai, and Gilbert Strang.

Many thanks as well to Sarah Grey for the end-to-end reviews and to Jessica Haberman for the discussions about the scope of the book.

Big thanks to people from Le Wagon who gave me great reviews, feedback and comments. In particular, Bernard Le Moullec, Gabrielle Prat, Pierre-François Renard. I am also incredibly thankful to Sébastien Saunier for his support and his unfailing ability to solve problems and help people.

I am also grateful to my friends and family for their feedbacks and support, especially Daniel Jean and Samuel Recht.

And I am infinitely grateful to Hélène for everything else, like her infinite patience and day-to-day backing.

## Contents

---

# Introduction

## 0.1 Motivation

With the increasing demand for data scientists, more people than ever are entering the field. Though it is possible to learn the job without an advanced math background, improving your mathematical theoretical knowledge will allow you to be more proficient. Instructional math resources exist, but few approach the topics from a data science perspective, focusing on the math useful in the field and at the right level of difficulty. That's what this book does.

In data science, each dataset and each problem is different. Even though it is possible to use complex algorithms and tools to work with data, it can be tough to use them on real-world datasets without understanding what is under the hood. For that reason, building a solid math intuition helps you choose the right tool for the right job and to tune the pipeline's building blocks from data to insight. It helps you un-black-box the methods, pull the methods out of the black box and examine them; you can solve problems or ask and answer the right questions. It allows you to transfer your experience and creativity to brand new problems.

This book is aimed at prospective or young data scientists and machine learning scientists who don't necessarily have a mathematics background. It is also for developers going into data science or data scientists who need a refresher on mathematics. This book is also for the large number of people who jump into data science with a top-down approach: starting to do real-world projects and then diving into the theoretical background with the big picture in mind.

## Contents

---

By the end of this book you will be able to read and write mathematical notation as used in data science, machine learning, and deep learning literature to communicate ideas. This will allow you to convert problems under mathematical form, which in turn will allow you to use code and data science/machine learning tools to solve it.

It will also give you solid foundations to understand more advanced or targeted resources that you might find inaccessible because of the math. A large number of machine learning or data science text books indeed assume that the reader has some math background. This book will provide you with all you need to tackle these resources and unlock a great way of thinking.

You will also be able to use Python and Jupyter Notebooks to plot data, represent equations, visualize space transformations, perform descriptive statistics and preliminary observation on a dataset and manipulate vectors, matrices and tensors as necessary to use machine learning/deep learning libraries like Tensorflow or Keras.

### 0.1.1 What Math Do You Need?

The book is designed to give you a general math background with an emphasis on linear algebra, targeting the fields of data science and machine learning.

In this journey, you'll start in part I by learning the basics of calculus needed for data science and machine learning, including derivatives, important-to-understand algorithms such as gradient descent, and integrals, showing you how to calculate area under curves.

In part II, you'll enter the field of statistics and probability with descriptive statistics, probability theory, Bayesian statistics, and information theory.

Finally, in part III, you'll build a solid understanding of linear algebra, starting from basics operations on vectors and matrices to advanced methods widely used in machine learning and data science like eigendecomposition and Singular Value Decomposition (SVD).

### 0.1.2 Teaching Method

The goal of this book is to show you that mathematics for data science can be approached in a very practical way. It is not a classic math textbook, exposing

theorems and proofs. It's designed to help you understand mathematical concepts, maybe as a self-learner, using programming as a way to understand theory. Learning by doing is a great way to understand concepts deeply.

### 0.1.2.1 Hands-on Projects

You'll see a lot of Python code and illustrations in this book. Furthermore, you'll find a "hands-on project" at the end of each chapter, applying what you learned to a concrete data science and machine learning problem.

Some of the hands-on projects are more difficult than others. In the following table, you'll see difficulty ratings for the mathematics and the code involved in each one.

Chapter	Title	Math and Concept	Code
02	Gradient Descent	Advanced	Intermediate
03	The Central Limit Theorem	Beginner	Beginner
04	Waiting for the Bus (Exponential Distribution)	Beginner	Intermediate
05	Bayesian inference	Advanced	Intermediate
06	Regularization	Advanced	Advanced
07	Image Classifier	Beginner	Beginner
08	Span	Beginner	Beginner
09	Linear Regression Using Least Squares Approximation	Intermediate	Beginner
10	PCA	Advanced	Advanced
11	Image Compression	Beginner	Beginner

### 0.1.2.2 Notebooks

If you got the bundle pack (book + access to the notebook repository), I recommend that you clone the notebooks from the Github repository and run the code along your reading. This is a great way to:

- Look at the variables in intermediate steps. Reading code, I often ask myself: "What is in this variable?" or "What is the shape of this array?". You'll see that this is especially useful for the linear algebra

- part. Running the notebooks, you can get all the information you need.
- Move around interactive plots running `%matplotlib notebook` in a separate cell of the notebook.
  - Look at the datasets and eventually run other analyses.

### 0.1.2.3 Code in the Book

You'll see that not all the code is displayed in the book: for the sake of clarity, I've used excerpts, with ellipses [...] to mark elided code. To see the code in full, you'll need to go to the notebooks on Github.

## 0.2 What you'll need

Here are a few more ingredients that will help you to make the most of this book.

### 0.2.1 A Bit of Math

This book assumes that you have been exposed to high school-level algebra, even if, in some cases, it offers a refresher on basic concepts before delving into more advanced topics.

### 0.2.2 Some Python Experience

Since the book uses code to help you build insight on math concepts, it would be great if you had some experience with Python. The code chunks are designed to be flat and easy to read, but they are not explained at the same level of detail as the math.

### 0.2.3 Exposure to Numpy and Matplotlib

Numpy is a Python library extensively used in data science and machine learning. You'll see that it bridges math and Python and is perfectly suited to implementing theoretical concepts. For instance, you'll manipulate vectors and matrices using Numpy arrays to get insights about linear algebra. I'll introduce the main things you'll need from Numpy, but some of the hands-on projects needs more advanced Numpy skills.

## Contents

---

Similarly, the plotting library Matplotlib is extensively used throughout the book. Some explanations are given about the Matplotlib code but a little exposure to this library is also assumed.<sup>1</sup>

---

<sup>1</sup>You can find great introductions to Numpy and Matplotlib, for instance: <https://cs231n.github.io/python-numpy-tutorial/>

## Contents

---

# **Part I**

# **Calculus**



---

In this part, you'll learn what you need of calculus for machine learning and data science. The main topics covered in this part are derivatives, partial derivatives, integrals and area under the curve. You will need these essential notions for more advanced parts of the book.



# Chapter 01

## Calculus: Derivatives and Integrals

Calculus is a branch of mathematics that gives tools to study rate of change of functions through two main areas: derivatives and integrals. In the context of machine learning and data science, you can for instance use derivatives to optimize the parameters of a model with gradient descent (as you'll see in the hands-on project at the end of this chapter). You might use integrals to calculate area under the curve (for instance, to evaluate performance of a model with the ROC curve, or to calculate probability from densities, as you'll see in Section 2.3.2 ).

### 1.1 Derivatives

#### 1.1.1 Introduction

The *derivative* of a function is related to its *rate of change*. The rate of change tells you how much the output of the function changes when a change is done to the input. It is calculated as the ratio between a change in the output and the corresponding change in the input.

Graphically, it is the slope of the tangent at a given point of the function.

Let's start with an example. You measure the distance traveled (in meters) of a moving train as a function of time (in seconds), as represented in Figure

1.1.

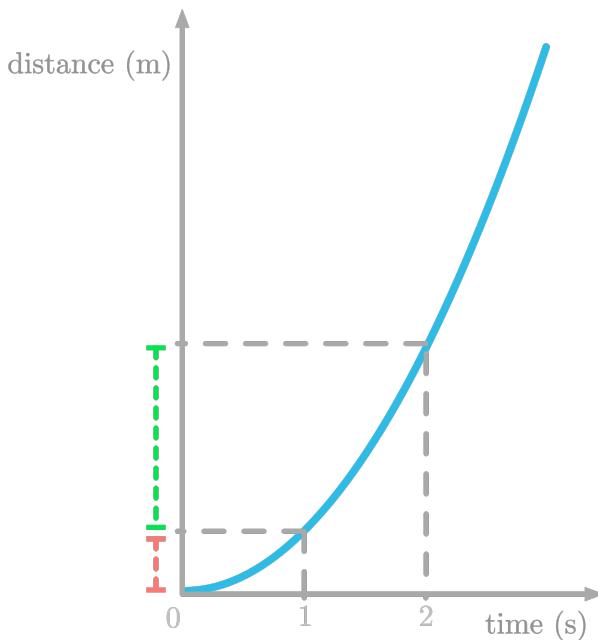
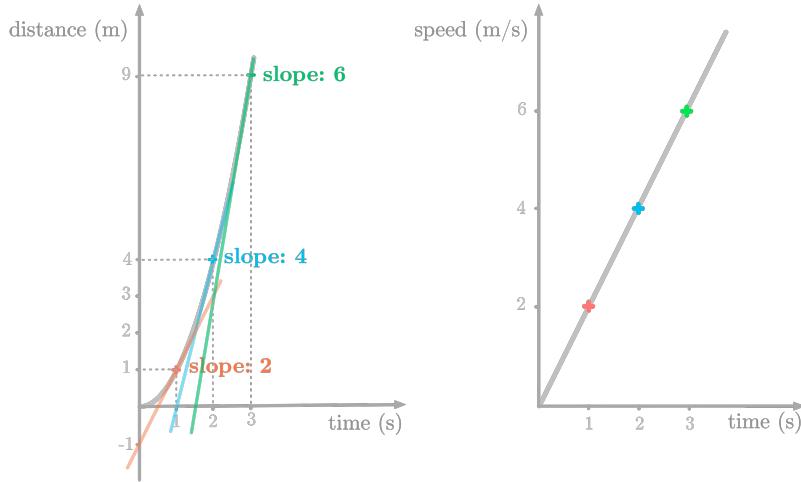


Figure 1.1: Example of a moving train. Distance is represented as a function of time.

Think about the rate of change of this function: it corresponds to the distance traveled in a specific unit of time. With the train example, it corresponds to another physical concept: speed.

Figure 1.1 shows that the distance traveled by the train in one unit of time is smaller at the beginning than at the end. A larger distance was traveled between one and two seconds (in green) in comparison to the distance traveled between zero and one second (in red): the speed increases with time.

Now, say that you record the speed of the train.



*Figure 1.2: The left panel shows the distance (y-axis) as a function of time (x-axis) and the slope of the tangent at different point in time. The right panel shows the speed (y-axis) as a function of time.*

Figure 1.2 shows again the distance as a function of time in the left panel and the speed as a function of time in the right panel.

Here is how these two functions are related. If you take the point at 1 second in the right panel (in red), you get a speed value of two meter per second: this is the slope of the tangent line at 1 second in the left panel (also in red). Similarly, any  $y$  value on the right figure gives you the slope of the tangent line for the corresponding  $x$  value in the left figure.

The slope in the left panel corresponds to the rate of change of the function at this point. For instance, if you take the red slope, you can see that in one unit of time, the distance traveled is 2 meters (for instance, between  $x = 0$  and  $x = 1$ , you go from  $y = -1$  to  $y = 1$ ).

Mathematically, the plot in the right figure shows the derivative of the function plotted in the left figure. The derivative of a function  $f$  with respect to  $x$  is the rate of change of  $f$  as a function of  $x$ . The derivative of  $f$  is another function which takes  $x$  as input and returns the slope of the tangent line of  $f$  at this value of  $x$ .

## 1.1.2 Mathematical Definition of Derivatives

### 1.1.2.1 Limits

You saw that the derivative at a point of a function is the slope of the tangent line at this point. The value of the slope corresponds to the *instantaneous rate of change* at this point, which is the rate of change at a specific moment. A change occurs between two points so the rate of change is calculated using two points. We consider that the distance between these two points, usually called  $\Delta x$ <sup>1</sup> and pronounced *delta x* approaches zero. Only in this case, the rate of change corresponds to the slope of the tangent at this point, and thus to the derivative of the function at this point.

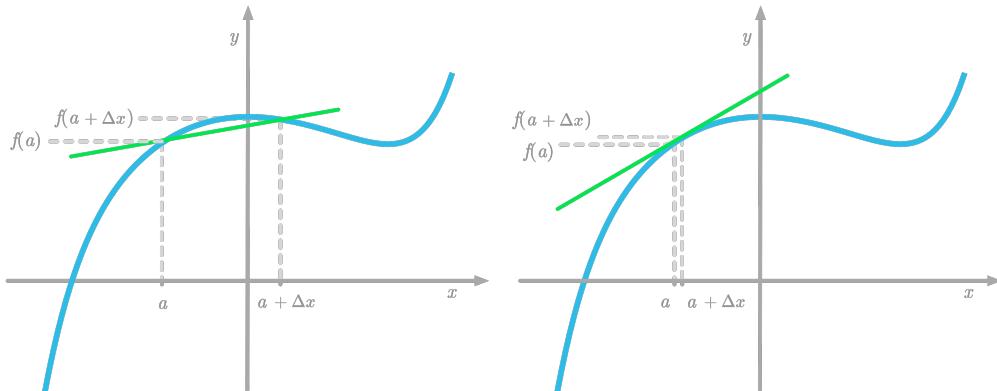


Figure 1.3: Comparison of the line corresponding to a large  $\Delta x$  (left) and a smaller  $\Delta x$  (right). You can see that the line approaches the tangent when  $\Delta x$  decreases.

Figure 1.3 shows the function  $f(x)$  (in blue) and two points on the  $x$ -axis:  $a$  and  $a + \Delta x$ . The value  $f(a)$  is the output of the function when the input is  $a$  and  $f(a + \Delta x)$  when the input is  $a + \Delta x$ .

In the right plot,  $\Delta x$  is small: when you decrease the difference between the two points used to calculate the rate of change the line approaches the tangent of the curve at  $a$ .

---

<sup>1</sup>You can find a summary of the Greek letters used in this book at the beginning of appendix B

Mathematically, the difference between the two points on the curve when  $\Delta x$  approaches zero is denoted as:

$$\lim_{\Delta x \rightarrow 0} f(x + \Delta x) - f(x)$$

This is the concept of *limit*: this expression is read as “the limit of  $f(x + \Delta x) - f(x)$  as  $\Delta x$  approaches zero”.

### 1.1.2.2 Calculating Derivatives

The rate of change is given by the slope of the tangent line.

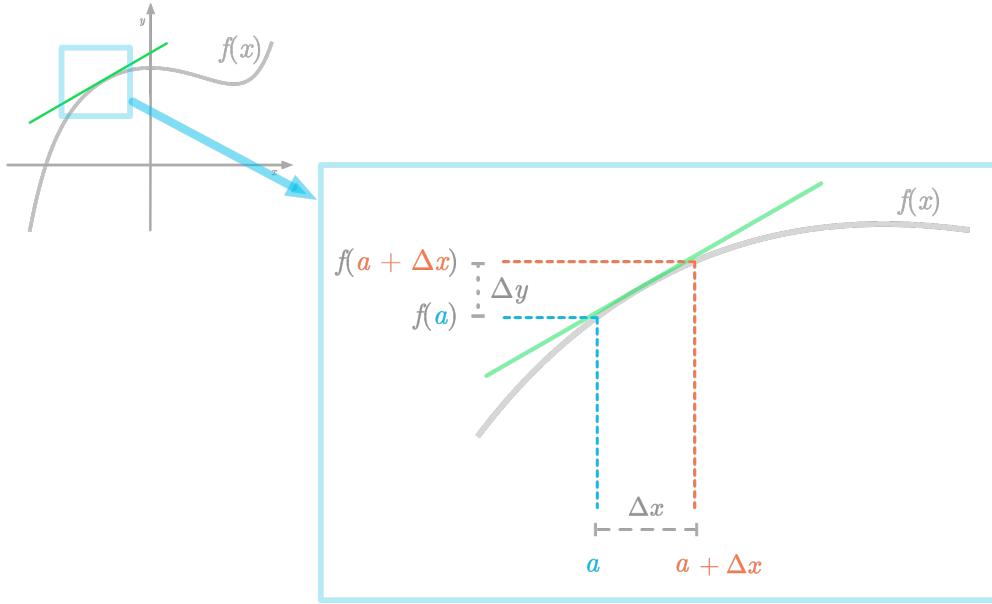


Figure 1.4: The slope of the green line is the ratio between  $\Delta y$  and  $\Delta x$ .

Figure 1.4 illustrates how the rate of change is calculated: it is the ratio between the change in the  $y$ -axis ( $\Delta y$ ) and the change in the  $x$ -axis ( $\Delta x$ ):

$$\frac{\Delta y}{\Delta x} = \frac{f(a + \Delta x) - f(a)}{\Delta x}$$

When  $\Delta x$  approaches zero, this is the *differentiation equation*, used to calculate the derivative of the function  $f$  with respect to  $x$ . It is denoted as:

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \quad (1.1)$$

### Notation

There are several kinds of mathematical notation that you can use to write a derivative.

You can use the prime symbol ': the derivative of the function  $f$  is for instance denoted as  $f'$ . This is called the *Lagrange notation*.

With the *Leibniz notation*, you write:

$$\frac{df(x)}{dx}$$

This refers to the derivative of  $f(x)$  with respect to  $x$  (the variable  $x$  is used for the differentiation). Going from  $\Delta x$  to  $dx$  means that you take an infinitely small difference with the limit. Note that this is not really a ratio, but a notation system<sup>a</sup>. For instance,

$$\frac{d(x^2 + 3)}{dx}$$

corresponds to the derivative of  $x^2 + 3$  with respect to  $x$ .

The Leibniz notation makes clear what is the variable over which you differentiate but the Lagrange notation is more compact. In this book, we'll use both notations.

---

<sup>a</sup>See this question on the difference about ratio as notation in derivatives:  
<https://math.stackexchange.com/questions/21199/is-frac-textrmdy-textrmdx-not-a-ratio>.

### 1.1.3 Derivatives of Linear And Nonlinear Functions

You can extract rules from the fact that a function derivative corresponds to its rate of change.

### 1.1.3.1 Constant Function

The rate of change of a function is equal to zero when  $y$  does not depend on  $x$  (when the value of  $y$  is the same for any  $x$ ). For instance, take the function  $f(x) = 4$ :

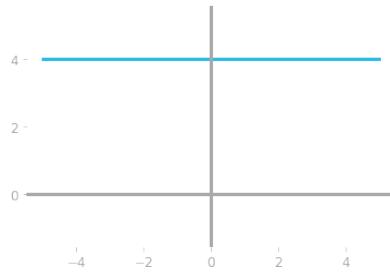


Figure 1.5: The constant function  $f(x) = 4$ .

As shown in Figure 1.5, the value of  $y$  is the same for any  $x$ . The slope of the function is equal to zero, so its derivative is equal to zero:

$$\frac{df(x)}{dx} = 0$$

The general rule is that the derivative of a constant  $a$  is zero:

$$\frac{da}{dx} = 0$$

This makes sense because constants doesn't impact the slope of a function.

### 1.1.3.2 Derivative of a Linear Function

Let's now consider a linear function that is not constant:  $f(x) = 3x + 1$ , as represented in Figure 1.6.

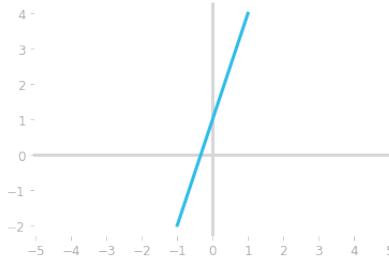


Figure 1.6: The linear function  $f(x) = 3x + 1$ .

To differentiate sums, you can use the *sum rule of differentiation*, which says that the derivative of the sum of two functions is equal to the sum of the derivatives. Mathematically, you have:

$$\frac{df(x) + g(x)}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}$$

In our example, the derivative of the term 1 is equal to zero because it is a constant, so the remaining part is  $3x$ .

As you can see in Figure 1.6, you go three steps on the  $y$ -axis when you go one step on the  $x$ -axis so the slope is three. The derivative of  $f(x)$  is thus:

$$\frac{df(x)}{dx} = 3$$

The general rule is that the derivative of functions with the form  $f(x) = ax + b$  is:

$$\frac{d(ax + b)}{dx} = a$$

With the derivative with respect to the variable  $x$  (as denoted with  $dx$ ), only the values multiplied by  $x$  have an effect on the slope.

### 1.1.3.3 Derivatives Of Nonlinear Functions

With nonlinear functions, you must characterize the slope of the tangent for each point on the curve because the rate of change depends on  $x$ .

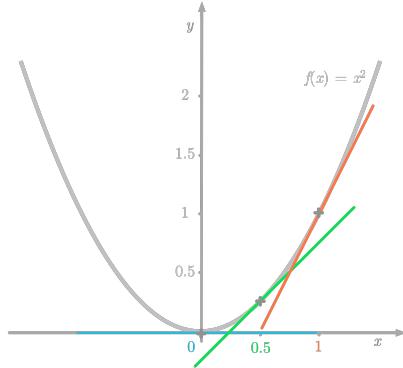


Figure 1.7: Slopes at different values of  $x$ .

Look for instance at the function  $f(x) = x^2$  in Figure 1.7: the slope of the tangent line is different for each value of  $x$ . For instance, the slope is zero at  $x = 0$  and increases when  $x$  increases. Similarly, when  $x$  is negative, the slope is negative. There is a relation between  $x$  and the slope of  $f$ .

Let's use the differentiation equation (see equation 1.1) to calculate the derivative of  $x^2$ :

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

$$\frac{dx^2}{dx} = \lim_{\Delta x \rightarrow 0} \frac{(x + \Delta x)^2 - x^2}{\Delta x}$$

$$= \lim_{\Delta x \rightarrow 0} \frac{x^2 + 2x\Delta x + \Delta x^2 - x^2}{\Delta x}$$

$$= \lim_{\Delta x \rightarrow 0} \frac{2x\Delta x + \Delta x^2}{\Delta x}$$

$$= \lim_{\Delta x \rightarrow 0} \frac{\Delta x(2x + \Delta x)}{\Delta x}$$

$$= \lim_{\Delta x \rightarrow 0} 2x + \Delta x$$

$$= 2x$$

The general rule is that derivatives of power functions are given by the following formula:

$$f(x) = x^p$$

$$\frac{df(x)}{dx} = px^{p-1}$$

For instance,  $\frac{dx^3}{dx} = 3x^2$ .

#### 1.1.4 Derivative Rules

It is often the case where the function you want to differentiate is made of multiple functions, multiplied or interleaved. A key idea behind differentiation is that you can use a small number of derivatives calculated with the limit

of the difference quotient (as you saw in the last section). You can usually calculate the derivative of a function by using rules such as addition, product, or composition, as you'll see here.

#### 1.1.4.1 Multiplication by Constant

Constants can be extracted from derivatives. For instance:

$$\frac{d(2x^2)}{dx} = 2 \left( \frac{dx^2}{dx} \right) = 2 \cdot 2x = 4x$$

More generally written:

$$\frac{d(af(x))}{dx} = a \frac{df(x)}{dx}$$

with  $a$  being a constant.

#### 1.1.4.2 Function Addition: The Sum Rule

The derivative of the sum of functions is equivalent to the sum of the derivatives of these functions:

$$\frac{d(f(x) + g(x))}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}$$

#### 1.1.4.3 Function Multiplication: The Product Rule

To differentiate the product of two functions, you must use the *product rule*.

Let's say that you want to differentiate the product of two functions  $f(x)$  and  $g(x)$ . You can't just take the product of the derivatives: you must use the following formula<sup>2</sup>:

$$\frac{d(f(x)g(x))}{dx} = \frac{df(x)}{dx} \cdot g(x) + f(x) \cdot \frac{dg(x)}{dx}$$

---

<sup>2</sup>You can refer to <https://www.khanacademy.org/math/ap-calculus-ab/ab-differentiation-1-new/ab-2-8/a/proving-the-product-rule> if you want the proof of the product rule.

Or more compactly with the Lagrange notation:

$$(fg)' = f'g + fg'$$

In plain English, this means that you add the derivative of  $f(x)$  multiplied by  $g(x)$  to the derivative of  $g(x)$  multiplied by  $f(x)$ .

**Example** Let's take the example of  $x^3$ . To illustrate the product rule, you can consider  $x^3$  as the product of  $x$  and  $x^2$ . This means that in our example,  $f(x) = x$  and  $g(x) = x^2$ . Let's apply the product rule:

$$\frac{d(x \cdot x^2)}{dx} = \frac{dx}{dx} \cdot x^2 + x \cdot \frac{dx^2}{dx}$$

Since the derivative of  $x$  is one and the derivative of  $x^2$  is  $2x$ , you have:

$$\begin{aligned} \frac{dx}{dx} \cdot x^2 + x \cdot \frac{dx^2}{dx} &= 1 \cdot x^2 + x \cdot 2x \\ &= x^2 + 2x^2 \\ &= 3x^2 \end{aligned}$$

#### 1.1.4.4 Function Composition: The Chain Rule

Composite function refers to the composition of two functions. As function composition in computer science, one function takes the output of the other as input. Consider for instance the functions  $f(x) = x^2$  and  $g(x) = 2x$ . You can compose them as  $f(g(x)) = f(2x) = (2x)^2$ .

To calculate derivatives of composite functions, you need to use the *chain rule*: multiply the derivative of the outer expression with the derivative of the inner expression:

$$\frac{df(g(x))}{dx} = \left( \frac{df(g(x))}{dg(x)} \right) \cdot \left( \frac{dg(x)}{dx} \right)$$

Note that the derivative of  $f(g(x))$  is with respect to  $g(x)$ .

With the Lagrange notation:

$$(f(g(x)))' = f'(g(x)) \cdot g'(x)$$

**Example** Let's differentiate the following composite function  $h(x)$ :

$$h(x) = f(g(x)) = (2x)^2$$

You have the functions  $f(x) = x^2$  and  $g(x) = 2x$ .

If you refer to the basic function derivatives<sup>3</sup>, you know that  $f'(x) = 2x$  and  $g'(x) = 2$ .

Let's calculate the derivative of  $h(x)$ , which is, according to the chain rule:

$$h'(x) = f'(g(x)) \cdot g'(x)$$

Since  $f(g(x)) = g(x)^2$ , you have  $f'(g(x)) = 2g(x)$ . In addition,  $g'(x) = 2$ , so you have:

$$h'(x) = 2g(x) \cdot 2 = 2 \cdot 2x \cdot 2 = 8x$$

The goal is to split composite functions into simpler functions and use the known derivatives of these functions.

The chain rule is important to understand artificial neural networks: the output of the network is calculated as the composition of multiple nested functions (*forward propagation*). To update the network parameters, the chain rule is used to calculate the derivative of the cost function and update the parameters accordingly (*backpropagation*).

---

<sup>3</sup>You can find a summary of the rules to calculate basic derivatives here: <https://www.mathsisfun.com/calculus/derivatives-rules.html>.

### 1.1.5 Partial Derivatives And Gradients

When a function has more than one variable, the concept of derivative is a bit different: you need to compute the slope separately for each variable.

For instance, in the physical world, you can imagine having a variable corresponding to the direction “east/west” and a variable to the direction “south/north”. It is possible to calculate the slope for each direction. This is the purpose of *partial derivatives*, called *partial* because you calculate the derivatives partially, variable per variable.

#### 1.1.5.1 Partial Derivatives

You can use the stylized cursive letter  $d$ ,  $\partial$ , that can be called “curly d”, to refer to partial derivatives.

Let’s take the following function as a first example:

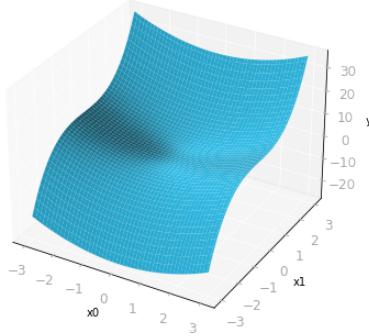
$$f(x_0, x_1) = x_0^2 + x_1^3$$

The function  $f$  takes the two variables  $x_0$  and  $x_1$  as input. Let’s represent this function graphically. You’ll create a three-dimensional plot because there are three variables (the dependent variable  $y$ , and the independent variables  $x_0$  and  $x_1$ ).

```
x0, x1 = np.meshgrid(np.linspace(-3,3,100), np.linspace(-3, 3, 100))
y = x0 ** 2 + x1 ** 3

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(x0, x1, y)
# [...] Add labels
```



*Figure 1.8: Three-dimensional representation of the function  $f(x_0, x_1) = x_0^2 + x_1^3$ .*

You can recognize in Figure 1.8 a mix between the function  $x^2$  and the function  $x^3$ . If you run this into a Jupyter Notebook, you can run `%matplotlib notebook` in a cell before to be able to move the shape and have a better understanding of the function.

Partial derivatives of  $f(x_0, x_1) = x_0^2 + x_1^3$  are derivatives with respect to each independent variable ( $x_0$  and  $x_1$ ). To calculate the partial derivative of a function with respect to a single variable, you consider all the other variables as constants (as they don't affect the slope of the tangent line in the direction you're considering).

Here is the partial derivative of  $f(x_0, x_1)$  with respect to  $x_0$  (considering  $x_1$  as a constant):

$$\begin{aligned} \frac{\partial}{\partial x_0} f(x_0, x_1) &= \frac{\partial}{\partial x_0} (x_0^2 + x_1^3) \\ &= 2x_0 \end{aligned}$$

Note that the denominator tells you that you differentiate with respect to  $x_0$ .

The partial derivative of  $f(x)$  with respect to  $x_1$ :

$$\begin{aligned}\frac{\partial}{\partial x_1} f(x_0, x_1) &= \frac{\partial}{\partial x_1} (x_0^2 + x_1^3) \\ &= 3x_1^2\end{aligned}$$

### 1.1.5.2 Gradient

If you calculate the derivative with respect to each variable of a function ( $f(x_0, x_1, \dots, x_n)$ ):

$$\frac{\partial}{\partial x_0} f(x_0, x_1, \dots, x_n)$$

$$\frac{\partial}{\partial x_1} f(x_0, x_1, \dots, x_n)$$

⋮

$$\frac{\partial}{\partial x_n} f(x_0, x_1, \dots, x_n)$$

and store all these partial derivatives in a vector (you'll learn more about vectors in chapter 5) named with the symbol  $\nabla$  ("nabla"). It is the gradient of  $f$ :  $\nabla f$  is pronounced here "gradient of  $f$ ", "grad  $f$ " or "del  $f$ ", and contains the partial derivatives of the function  $f$  with respect to each variable:

$$\nabla f = \begin{bmatrix} \frac{\partial}{\partial x_0} \\ \frac{\partial}{\partial x_1} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{bmatrix}$$

The gradient of  $f$  gives the slope corresponding to each variable. As an

analogy, if you are standing on a mountain, the gradient would tell you what is the slope in each direction.

## 1.2 Integrals and the Area Under the Curve

*Integration* is the inverse operation of differentiation. Take a function  $f(x)$  and calculate its derivative  $f'(x)$ , the *indefinite integral* (also called *antiderivative*) of  $f'(x)$  gives you back  $f(x)$  (up to a constant, as you'll soon see).<sup>4</sup>

You can use integration to calculate the *area under the curve*, which is the area of the shape delimited by the function, as shown in Figure 1.9.

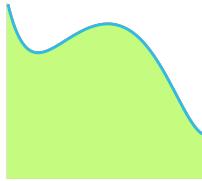


Figure 1.9: Area under the curve.

A *definite integral* is the integral over a specific interval. It corresponds to the area under the curve in this interval.

### 1.2.1 Example

You'll see through this example how to understand the relationship between the integral of a function and the area under the curve. To illustrate the process, you'll approximate the integral of the function  $g(x) = 2x$  using a discretization of the area under the curve.

#### 1.2.1.1 Example Description

Let's take again the example of the moving train. You saw that speed as a function of time was the derivative of distance as a function of time. These functions are represented in Figure 1.10.

---

<sup>4</sup>The link between derivatives and integrals is described by what is called the *fundamental theorem of calculus*.

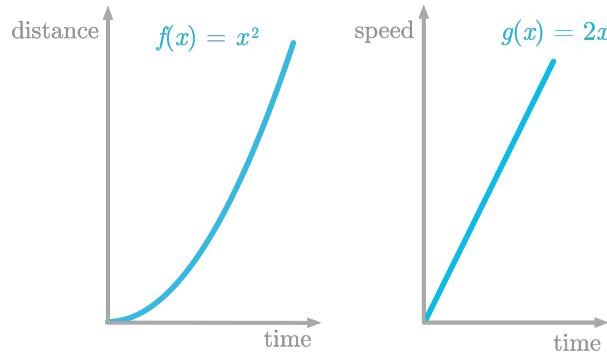


Figure 1.10: The left panel shows  $f(x)$  which is the distance as a function of time, and the right panel its derivative  $g(x)$ , which is the speed as a function of time.

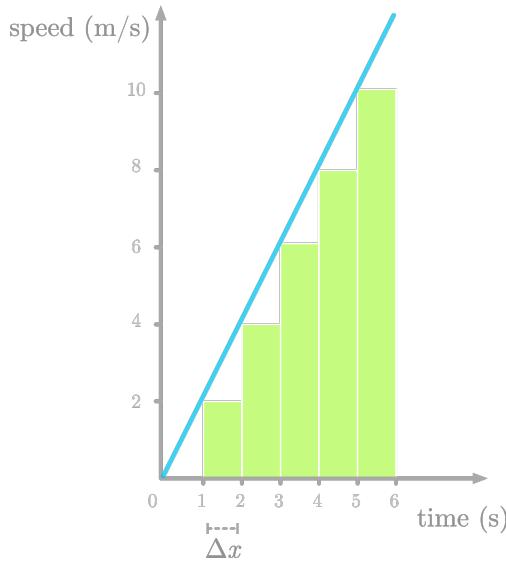
The function shown in the left panel of Figure 1.10 is defined as  $f(x) = x^2$ . Its derivative is defined as  $g(x) = 2x$ .

In this example, you'll learn how to find an approximation of the area under the curve of  $g(x)$ .

### 1.2.1.2 Slicing the Function

To approximate the area of a shape, you can use the slicing method: you cut the shape into small slices with an easy shape like rectangles, calculate the area of each of these slices and sum them.

You'll do exactly that to find an approximation of the area under the curve of  $g(x)$ .



*Figure 1.11: Approximation of the area under the curve by discretizing the area under the curve of speed as a function of time.*

Figure 1.11 shows the area under the curve of  $f'(x)$  sliced as one-second rectangles (let's call this difference  $\Delta x$ ). Note that we underestimated the area (look at the missing triangles), but we'll fix that later.

Let's try to understand the meaning of the slices. Take the first one: its area is defined as  $2 \cdot 1$ . The height of the slice is the speed at one second (the value is 2). So there are two units of speed by one unit of time for this first slice. The area corresponds to a multiplication between speed and time: this is a distance.

For instance, if you drive at 50 miles per hour (speed) for two hours (time), you traveled  $50 \cdot 2 = 100$  miles (distance). This is because the unit of speed corresponds to a ratio between distance and time (like miles *per* hour). You get:

$$\frac{\text{distance}}{\text{time}} \cdot \text{time} = \text{distance}$$

To summarize, the derivative of the distance by time function is the speed by

time function, and the area under the curve of the speed by time function (its integral) gives you a distance. This is how derivatives and integrals are related.

### 1.2.1.3 Implementation

Let's use slicing to approximate the integral of the function  $g(x) = 2x$ . First, let's define the function  $g(x)$ :

```
def g_2x(x):
    return 2 * x
```

As illustrated in Figure 1.11, you'll consider that the function is discrete and take a step of  $\Delta x = 1$ . You can create an  $x$ -axis with values from zero to six, and apply the function `g_2x()` for each of these values. You can use the Numpy method `arange(start, stop, step)` to create an array filled with values from `start` to `stop` (not included):

```
delta_x = 1
x = np.arange(0, 7, delta_x)
x

array([0, 1, 2, 3, 4, 5, 6])

y = g_2x(x)
y

array([ 0,  2,  4,  6,  8, 10, 12])
```

You can then calculate the slice's areas by iterating and multiplying the width ( $\Delta x$ ) by the height (the value of  $y$  at this point). of the slice. As you saw, this area (`delta_x * y[i-1]` in the code below) corresponds to a distance (the distance of the moving train traveled during the  $i$ th slice). You can finally append the results to an array (`slice_area_all` in the code below).

Note that the index of `y` is `i-1` because the rectangle is on the left of the  $x$  value you estimate. For instance, the area is zero for  $x = 0$  and  $x = 1$ .

```

slice_area_all = np.zeros(y.shape[0])
for i in range(1, len(x)):
    slice_area_all[i] = delta_x * y[i-1]
slice_area_all

array([ 0.,  0.,  2.,  4.,  6.,  8., 10.])
    
```

These values are the slice's areas.

To calculate the distance traveled from the beginning to the corresponding time point (and not corresponding to each slice), you can calculate the cumulative sum of `slice_area_all` with the Numpy function `cumsum()`:

```

slice_area_all = slice_area_all.cumsum()
slice_area_all

array([ 0.,  0.,  2.,  6., 12., 20., 30.])
    
```

These are the estimated values of the area under the curve of  $g(x)$  as a function of  $x$ . You know that the function  $g(x)$  is the derivative of  $f(x) = x^2$ , so you should get back  $f(x)$  by the integration of  $g(x)$ .

Let's plot our estimation and  $f(x)$ , which we'll call the "true function", to compare them:

```

plt.plot(x, x ** 2, label='True')
plt.plot(x, slice_area_all, label='Estimated')
    
```

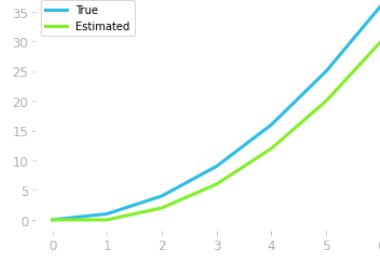


Figure 1.12: Comparison of the estimated and original function.

The estimation represented in Figure 1.12 shows that the estimation is not bad, but could be improved. This is because we missed all these triangles represented in red in Figure 1.13. One way to reduce the error is to take a smaller value for  $\Delta x$ , as illustrated in the right panel in Figure 1.13.

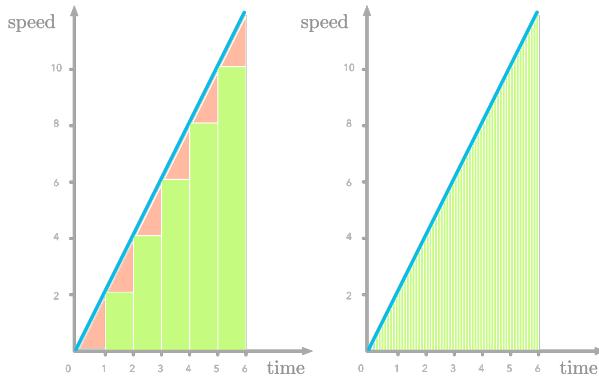


Figure 1.13: Missing parts in slices of the speed function (in red). The error is smaller with a smaller  $\Delta x$ .

Let's estimate the integral function with  $\Delta x = 0.1$ :

```
delta_x = 0.1
x = np.arange(0, 7, delta_x)
y = g_2x(x)
# [...] Calculate and plot slice_area_all
```

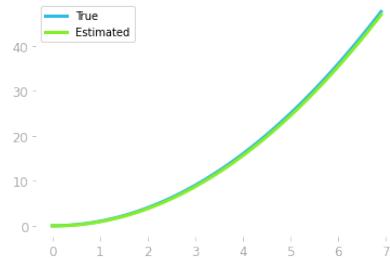
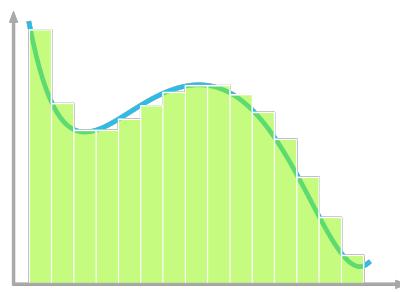


Figure 1.14: Smaller slice widths lead to a better estimation of the original function.

As shown in Figure 1.14, you recovered (at least, up to an additive constant) the original function whose derivative you integrated.

#### 1.2.1.4 Extension

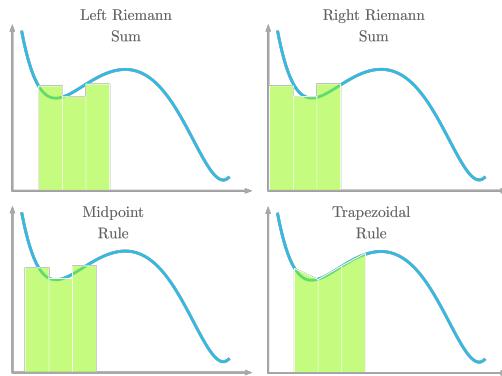
In our previous example, you integrated the function  $2x$ , which is a linear function, but the principle is the same for any continuous function (see Figure 1.15 for instance).



*Figure 1.15: The slicing method can be used with many linear or nonlinear function, including all continuous functions.*

#### 1.2.2 Riemann Sum

Approximating an integral using this slicing method is called a *Riemann sum*. Riemann sums can be calculated in different ways, as you can see in Figure 1.16.



*Figure 1.16: Four kinds of Riemann sums for integral approximation.*

As pictured in Figure 1.16, with the left Riemann sum, the curve is aligned with the left corner of the rectangle. With the right Riemann sum, the curve is aligned with the right corner of the rectangle. With the midpoint rule, the curve is aligned with the center of the rectangle. With the trapezoidal rule, a trapezoidal shape is used instead of a rectangle. The curve crosses both top corners of the trapezoid.

### 1.2.3 Mathematical Definition

In the last section, you saw the relationship between the area under the curve and integration (you got back the original function from the derivative). Let's see now the mathematical definition of integrals.

The integrals of the function  $f(x)$  with respect to  $x$  is denoted as follows:

$$\int f(x) dx$$

The symbol  $dx$  is called the *differential* of  $x$  and refers to the idea of an infinitesimal change of  $x$ . It is a difference in  $x$  that approaches 0. The main idea of integrals is to sum an infinite number of slices which have an infinitely small width.

The symbol  $\int$  is the integral sign and refers to the sum of an infinite number of slices.

The height of each slice is the value  $f(x)$ . The multiplication of  $f(x)$  and  $dx$  is thus the area of each slice. Finally,  $\int f(x) dx$  is the sum of the slice areas over an infinite number of slices (the width of the slices tending to zero). This is the *area under the curve*.

You saw in the last section how to approximate function integrals. But if you know the derivative of a function, you can retrieve the integral knowing that it is the inverse operation. For example, if you know that:

$$\frac{d(x^2)}{dx} = 2x$$

You can conclude that the integral of  $2x$  is  $x^2$ . However, there is a problem. If you add a constant to our function the derivative is the same because the

derivative of a constant is zero. For instance,

$$\frac{d(x^2 + 3)}{dx} = 2x$$

It is impossible to know the value of the constant. For this reason, you need to add an unknown constant to the expression, as follows:

$$\int 2x \, dx = x^2 + c$$

with  $c$  being a constant.

**Definite Integrals** In the case of *definite integrals*, you denote the interval of integration with numbers below and above the integral symbol, as follows:

$$\int_a^b f(x) \, dx$$

It corresponds to the area under the curve of the function  $f(x)$  between  $x = a$  and  $x = b$ , as illustrated in Figure 1.17.

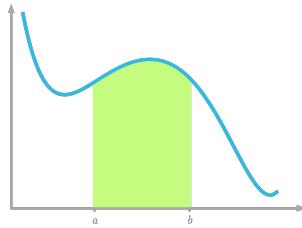


Figure 1.17: Area under the curve between  $x = a$  and  $x = b$ .

### 1.3 Hands-On Project: Gradient Descent

In this hands-on project, you'll learn the math behind an important method leveraging derivatives to optimize *cost functions* (that is, to find the parameters of the functions associated with the lower error): *gradient descent*.

You'll define a cost function used for linear regression. You'll learn to implement a prediction function, evaluate the error with a cost function, and use the derivative of this cost function to optimize the parameters of your model.<sup>5</sup>

### 1.3.1 Cost function

As humans, we rely on feedback to learn a new skill. You can try something, use the feedback to see how well or badly you performed, and iterate until the feedback is positive.

Training a model is similar. You first need a way to assess its performances. The *loss* is a value that tells you how badly the model performs for a given data sample: it is zero if the estimation is perfect and increases when the prediction is less good. Training the model means changing its parameters to reduce the loss as much as possible.

In this hands-on, you'll see the example of linear regression: data points are fitted with a line, as illustrated with in Figure 1.18.

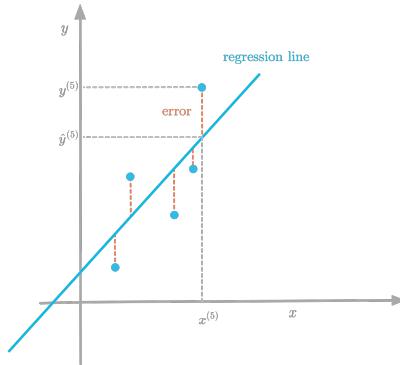


Figure 1.18: Data points and the regression line. Errors are represented in red: it is the difference between the estimated value (the point on the line) and the true value.

Figure 1.18 shows five data points and the regression line (the fit of a linear regression). The goal is to find the value of the parameter (such as the

---

<sup>5</sup>What you'll learn here applies to other algorithms, like neural networks. Using different prediction function, cost function, or update rules gives you the possibility to build various algorithms.

slope of the regression line) that fits the data well. The error of the model is the distance between the data points and the estimated values (every points on the regression line), represented in red on the figure. As you'll see in Section 5.3, there are multiple ways to calculate a distance. A common choice for linear regression is the Euclidean distance, which corresponds to the  $L^2$  loss, or *squared loss*.

Averaging this loss across all data samples, you get what is called the *Mean Squared Error*, or MSE. It corresponds to the average of the squared differences between estimation and true values across all data points. It is commonly used for linear regression.

### 1.3.1.1 Mathematical Definition of the Cost Function

The MSE loss is defined as follows:

$$L = (\hat{y} - y)^2$$

with  $\hat{y}$  (pronounced “y hat”) being the estimated value and  $y$  the true value. So the difference  $\hat{y} - y$  corresponds to the error made by the model.

Looking again at Figure 1.18, you can observe that some errors are positive (data points below the estimated value) and some are negative. Squaring the errors allows you to take both into account to evaluate the error.

#### Cost functions vs Loss Functions

Loss functions and cost functions are intimately related, but usually, loss functions refer to the error calculated on one training example, whereas the cost function is over the entire training dataset.

**Prediction Function** In the case of linear regression, the predicted value  $\hat{y}$  is calculated with a linear function (the assumption is that a line could fit our data well):

$$\hat{y} = ax + b$$

It is called the *prediction function*, *hypothesis function*, or simply the *model*. You can refer to appendix A for more details on the parameters of a line. The slope parameter is  $a$  (the parameter multiplied by  $x$ ) and the intercept is  $b$ . This prediction function takes  $x$  as input and returns an estimation  $\hat{y}$  according to your model's parameters ( $a$  and  $b$ ).

**Loss Function** If you replace  $\hat{y}$  in the loss function, you get:

$$L = (\hat{y} - y)^2 = ((ax + b) - y)^2$$

For now, we'll simplify the model and use only one parameter: the slope parameter  $\hat{y} = ax$ . Intuitively, this means that you force the line to pass through zero (because the  $y$ -intercept is  $b = 0$ ). The loss becomes:

$$L = (ax - y)^2$$

Conventionally, the model parameters are called  $\theta$  (the Greek letter “theta”), as in the following expression:

$$L = (\theta x - y)^2$$

**Cost Function** To calculate the Mean Squared Error, let's calculate the averaged loss across data samples. Mathematically, the cost function, usually denoted as  $J(\theta)$ , is written as:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (\theta \mathbf{x}^{(i)} - \mathbf{y}^{(i)})^2$$

with  $m$  being the number of training examples,  $\mathbf{x}^{(i)}$  the  $i$ th sample of the dataset, and  $\mathbf{y}^{(i)}$  the true value of the  $i$ th sample.

As you'll see in chapter 5, we'll denote vectors with lowercase, boldface letters. For now, consider these vectors as arrays of values. For instance,  $\mathbf{x}$  is an array with all data samples.

Dividing by  $2m$  instead of  $m$  is done only to simplify the final result of the cost function derivative.  $J(\theta)$  is thus proportional to the average of the squared

loss since you calculate the sum of the losses and divide by the number of data samples multiplied by two.

The Sigma notation ( $\sum$ ) means that you take the sum over every  $i$  training example until  $m$ .

### 1.3.1.2 Implementation

Let's write an implementation of the MSE (feel free to try to write it before looking at the solution: it is a good way to go from mathematical notation to code and the other way around):

```
def MSE(x, y, theta):
    m = y.shape[0]
    cost = (1 / (2 * m)) * np.sum((theta * x - y) ** 2)
    return cost
```

You can get the number of data samples with the shape of `y`: this array contains one value per data sample. The variable `x` is an array containing the data samples, and thus `theta * x - y` is also an array of the same size.

To summarize, the cost function takes  $x$ ,  $y$  and  $\theta$  as inputs and returns the cost.

Let's illustrate the cost function with the example of three observations.

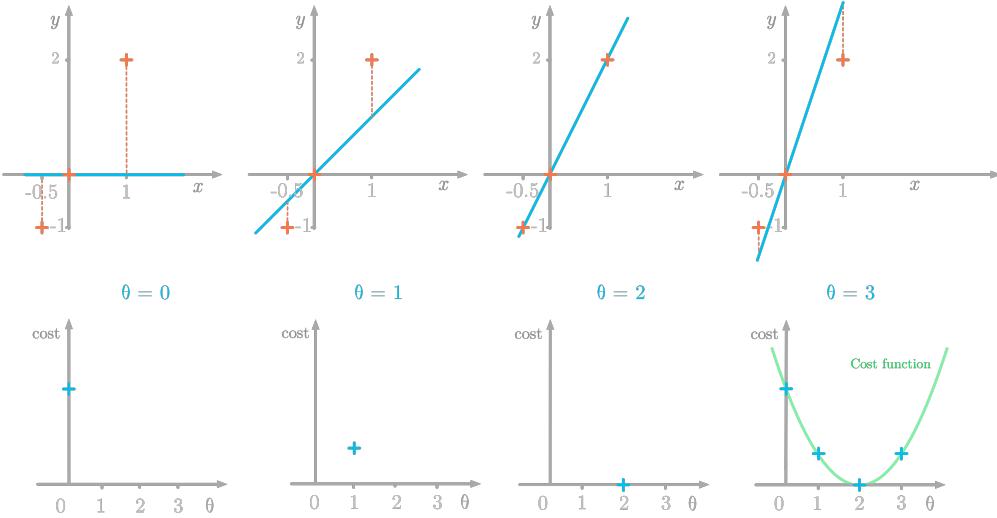


Figure 1.19: The cost function corresponds to the error of the fit as a function of the parameter  $\theta$  (here, the slope), as shown in the bottom figures.

Figure 1.19 illustrates four different values of  $\theta$  from left to right. The top figures show the three data points in red and the regression line with a slope of  $\theta$ . The bottom figures show the cost as a function of  $\theta$ . This cost is calculated as the error of the fit (the dotted red lines in the top figures). For instance, you can see that the cost is zero when  $\theta = 2$  because the fit is perfect.

When you consider the different values of  $\theta$ , you see the shape of the MSE the cost function, as in the bottom right panel. However, the goal of gradient descent is to avoid to calculate the cost for every parameters (or combinations of parameters, which would be computationally very difficult).

### 1.3.2 Derivative of the Cost Function

Since derivatives correspond to the slope of a function, it indicates the direction of  $x$  needed to maximize or minimize the function. If the function has multiple parameters, you need to know how to move in multiple directions and calculate the derivative with respect to each of these directions: the partial derivatives (see Section 1.1.5). The gradient is a vector containing the partial derivatives for all directions and gradient descent is the method used

to find the minimum of cost functions.

Let's differentiate the MSE cost function defined in the last section. You want to find:

$$\frac{dJ(\theta)}{d\theta} = \frac{d}{d\theta} \left( \frac{1}{2m} \sum_{i=1}^m (\theta x^{(i)} - y^{(i)})^2 \right)$$

The expression  $\frac{dJ(\theta)}{d\theta}$  corresponds to the derivative of the cost function  $J(\theta)$  with respect to  $d\theta$ .

First, you can get  $\frac{1}{2m}$  out from the derivative because of the multiplication by constant rule. The sum can also be extracted because of the sum rule. You get:

$$\frac{dJ(\theta)}{d\theta} = \frac{1}{2m} \sum_{i=1}^m \frac{d}{d\theta} ((\theta x^{(i)} - y^{(i)})^2)$$

You'll need the chain rule (Section 1.1.4.4) to differentiate this composite function. Let's use the Lagrange notation and create an intermediate function  $g(\theta)$  such as:

$$g(\theta) = \theta x - y$$

We omitted the index  $i$  for clarity. Let's also denote  $f(g(x))$  or simply  $f(g)$ :

$$f(g) = g^2$$

From the derivatives rules, the derivatives of these two functions are:

$$g'(\theta) = x$$

$$f'(g) = 2g$$

Using the chain rule from Section 1.1.4.4, you get:

$$(f(g))' = f'(g) \cdot g'$$

$$= 2g \cdot x$$

$$= 2(\theta x - y) \cdot x$$

So the derivative of the MSE cost function is defined as:

$$\begin{aligned}\frac{dJ(\theta)}{d\theta} &= \frac{1}{2m} \sum_{i=1}^m 2(\theta x^{(i)} - y^{(i)}) \cdot x^{(i)} \\ &= \frac{1}{m} \sum_{i=1}^m (\theta x^{(i)} - y^{(i)}) \cdot x^{(i)}\end{aligned}$$

The function  $\frac{dJ}{d\theta}$  takes the parameter  $\theta$  as input and returns the slope of the tangent of the cost function for this value of  $\theta$ . The slope tells you the direction that you must take to minimize the cost. Look back at the cost function pictured in green in Figure 1.19: for instance, the slope of the tangent line at  $\theta = 0$  is negative. It means that you need to increase the value of  $\theta$  to reach the minimum of the cost function. Conversely, the slope of the tangent is positive for  $\theta = 3$ , meaning that the value of  $\theta$  that minimizes the cost is smaller than three.

You can now implement the derivative of the cost function (try to do it yourself, and then look at the solution):

```
def MSE_derivative(x, y, theta):
    m = y.shape[0]
    cost_derivative = (1 / m) * np.sum((theta * x - y) * x)
    return cost_derivative
```

You give a value of  $\theta$  and arrays containing the  $x$  values and the  $y$  values, and the function returns the slope of the tangent of the cost function for this  $\theta$ .

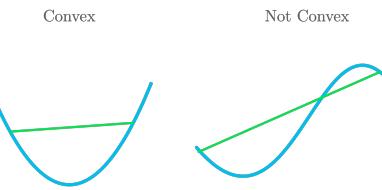
### 1.3.3 Implementing Gradient Descent

Now that you have the cost function derivative, you'll be able to optimize the parameter  $\theta$  (the slope of the fitting line).

Here are the steps:

- Start with the parameter (the slope) equals to 0.
- Calculate the derivative for this parameter.
- Update the parameter in the direction that decreases the cost.

You can then repeat the steps 1 and 2.



*Figure 1.20: The left figure shows a convex function: all lines between pairs of points are above the curve. The right panel shows a non-convex function with an example of line that lies below the curve.*

#### Convex Functions

This process can't work if your cost function is not *convex*. A convex function has the shape of a bowl and is defined as follows: if you draw a line between any pair of points in the curve, this line will be above or on the curve (see Figure 1.20).

#### 1.3.3.1 Dataset

You'll implement gradient descent on a dataset on red wine quality, where various chemical properties of wines are described<sup>6</sup>.

---

<sup>6</sup>The dataset comes from here: <https://archive.ics.uci.edu/ml/datasets/wine+quality>. The related paper is Cortez, Paulo, et al. “Modeling wine preferences by data mining from physicochemical properties.” Decision Support Systems 47.4 (2009): 547-553.

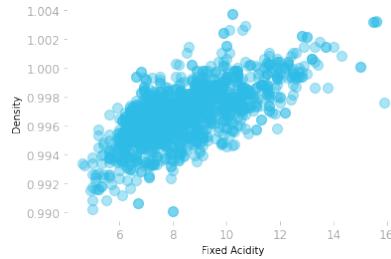
Let's load the data and have a look at the first rows and columns:

```
data = pd.read_csv("data/winequality-red.csv", sep=";")  
data.iloc[:5, :5]
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides
0	7.4	0.70	0.00	1.9	0.076
1	7.8	0.88	0.00	2.6	0.098
2	7.8	0.76	0.04	2.3	0.092
3	11.2	0.28	0.56	1.9	0.075
4	7.4	0.70	0.00	1.9	0.076

The goal is to implement linear regression to model the relationship between two features of this dataset: the fixed acidity of the wine and its density. Let's first look at the scatter plot of these variables:

```
plt.scatter(data['fixed acidity'], data['density'], alpha=0.4)  
# [...] Add axes, labels etc.
```



*Figure 1.21: Density of wines as a function of fixed acidity.  
Each point corresponds to a data observation.*

Let's create the variables `x` and `y` from the Pandas dataframe:

```
X = data['fixed acidity'].to_numpy().reshape(-1, 1)
```

```
X.shape
```

```
(1599, 1)
```

```
y = data['density'].to_numpy().reshape(-1, 1)
```

```
y.shape
```

```
(1599, 1)
```

You need to reshape the Numpy arrays to use them in the following Sklearn method.

You'll also standardize the data: transforms the variables to have a mean equals to zero and a standard deviation equals to one<sup>7</sup>:

```
from sklearn.preprocessing import StandardScaler  
  
standard_scaler = StandardScaler()  
X = standard_scaler.fit_transform(X)  
y = standard_scaler.fit_transform(y)
```

### 1.3.3.2 Derivative of the Cost Function

You can use your `MSE_derivative` function to calculate the derivative using these data. Let's try it with  $\theta = 0$ :

```
MSE_derivative(x=X, y=y, theta=0)
```

```
-0.6680472921189744
```

This negative value tells you that you need to increase  $\theta$  if you want to reduce the cost (you can still refer to Figure 1.19 and the last section).

### 1.3.3.3 Parameter Update

A large derivative value means that the slope is steep. In this case, you want to add or subtract a large value to approach the minimum rapidly. This is

---

<sup>7</sup>You'll find more details about the mean and the standard deviation in Section 2.1.

why you update the parameters by subtracting the value of the derivative to the parameter: a large value will largely change the parameter and a small value not so much.

In addition, a weight, called the *learning rate*, is applied to the derivative allowing you to choose how large are the steps done at each iteration. This is done by multiplying a scalar to the derivative (called `lr` for ‘learning rate’ in the following code):

```
lr = 0.01
theta = 0
theta = theta - lr * MSE_derivative(x=X, y=y, theta=theta)
theta
```

0.006680472921189744

This is the value of  $\theta$  after the update. Let’s iteratively update the parameters, store the cost and the parameter value and plot the cost as a function of the parameters:

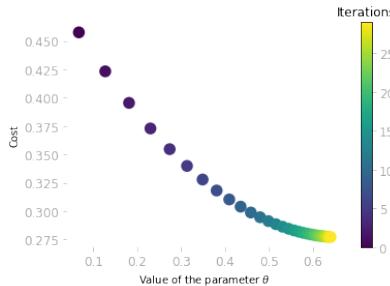
```
lr = 0.1
theta = 0

theta_all = []
cost_all = []

for i in range(30):
    theta = theta - lr * MSE_derivative(x=X, y=y, theta=theta)
    cost = MSE(X, y, theta)

    theta_all.append(theta)
    cost_all.append(cost)

plt.scatter(theta_all, cost_all, linewidth=1.5,
            c=np.arange(len(cost_all)))
```



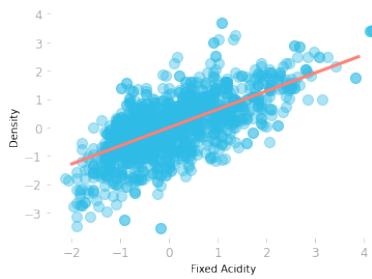
*Figure 1.22: Cost as a function of the parameter  $\theta$ . The cost decreases at each iteration.*

You can see in Figure 1.22 that the cost decreases at each iteration. Let's see what is the best parameter value that you got and use it to plot the corresponding regression line:

```
best_slope = theta_all[-1]
best_slope
```

```
0.6397279936234294
```

```
x_axis = np.arange(-2, 4, 0.1)
y_axis = best_slope * x_axis
plt.scatter(X, y, alpha=0.4, zorder=0)
plt.plot(x_axis, y_axis, c='#FF8177')
```



*Figure 1.23: Density as a function of fixed acidity with the regression line.*

You can see in Figure 1.23 that after 30 iterations you reach a slope that fits the data well.

Congratulation, you have learned about one of the core mathematical principles of machine learning: *gradient descent*. In machine learning, algorithms *learn* by minimizing the cost function. Instead of calculating the cost for all combination of parameters, derivatives tells you how to modify the parameter to decrease the cost.

### 1.3.4 BONUS: MSE Cost Function With Two Parameters

You saw in this section how to minimize a cost function for a single parameter model. You'll see now how to proceed with multiple parameters.

#### 1.3.4.1 The Cost Function

You'll calculate the gradient of the MSE cost function for linear regression using two parameters: the  $y$ -intercept of the line (that we'll denote  $\theta_0$ ) and its slope ( $\theta_1$ ).

With both parameters, the cost function is as follows:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2$$

You can see that the only difference with the cost function with one parameter is that the prediction function is  $\theta_0 + \theta_1 x^{(i)}$  instead of just  $\theta x^{(i)}$ .

#### 1.3.4.2 Partial Derivatives

With two parameters, you'll need to calculate the partial derivatives with respect to  $\theta_0$  and  $\theta_1$  independently. Here are the intermediate functions to apply the chain rule, as you had with a single parameter:

$$g(\theta_0, \theta_1) = \theta_0 + \theta_1 x - y$$

and

$$f(g) = g^2$$

You'll have different derivatives with respect to  $\theta_0$  and  $\theta_1$ :

$$\frac{\partial g}{\partial \theta_0} = 1$$

and

$$\frac{\partial g}{\partial \theta_1} = x$$

You also have:

$$\frac{\partial f(g)}{\partial g} = 2g$$

Let's calculate the partial derivatives of the cost function.

**With Respect to  $\theta_0$**  For the derivative with respect to  $\theta_0$ , you have:

$$\begin{aligned}\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} &= \frac{\partial f(g)}{\partial g} \cdot \frac{\partial g}{\partial \theta_0} \\ &= 2g \cdot 1 \\ &= 2(\theta_0 + \theta_1 x - y)\end{aligned}$$

so you have:

$$\begin{aligned}\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} &= \frac{1}{2m} \sum_{i=1}^m 2(\theta_0 + \theta_1 x^{(i)} - y^{(i)}) \\ &= \frac{1}{m} \sum_{i=1}^m \theta_0 + \theta_1 x^{(i)} - y^{(i)}\end{aligned}$$

**With Respect to  $\theta_1$**  Finally, for the derivative with respect to  $\theta_1$ , you have:

$$\begin{aligned}\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} &= \frac{\partial f(g)}{\partial g} \cdot \frac{\partial g}{\partial \theta_1} \\ &= 2g \cdot x \\ &= 2(\theta_0 + \theta_1 x - y) \cdot x\end{aligned}$$

and

$$\begin{aligned}\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} &= \frac{1}{2m} \sum_{i=1}^m 2(\theta_0 + \theta_1 x^{(i)} - y^{(i)}) \cdot x^{(i)} \\ &= \frac{1}{m} \sum_{i=1}^m (\theta_0 + \theta_1 x^{(i)} - y^{(i)}) \cdot x^{(i)}\end{aligned}$$

These functions allow you to update both parameters. You can calculate the derivative corresponding to each variable separately and update the parameters according to the slope exactly as you did previously.

## Conclusion

This concludes this chapter on calculus. You have learned about derivatives and integrals, and this hands-on project showed that these mathematical concepts are very important to understanding the core of machine learning.

You can often gain intuition as you learn by trying things out with code and by seeing multiple points of view: for instance, through plots, equations, and examples in the real world. Mathematical notation is just one more tool you'll have at your belt.



## Part II

# Statistics and Probability



---

One goal of data science is to find patterns in data. For instance, if you try to predict the price of an appartement, you can look at a lot of different appartements and their sell prices, and try to understand the link between the apartment features and prices. It is a common practice in science: using observations to understand systems.

The goal of probability is to deal with uncertainty. To make sense of data through machine learning, you have to deal with uncertainty. There are multiple possible sources of uncertainty. It can come from the data itself, from the data that you don't have, from the inherent stochasticity of the system you want to model, from the simplicity of your model, etc. For instance, if you build a model predicting who will develop cancer using medical imaging, you are likely to miss a lot of features that might be useful to do the task, like health records, occupational hazards or environmental exposure. There are so many variables that you can't measure, and that's where the uncertainty lies.

This is the reason why data science and machine learning needs mathematical frameworks to handle uncertainty. This is the goal of statistics and probability.



# Chapter 02

## Statistics and Probability Theory

Data have an inherent part of randomness due to generation or collection processes for instance. To model these data, you need to consider this randomness. The framework of probability theory helps you handle it mathematically.

This chapter will introduce you to the basic concepts of probability that will prepare you to handle, study or quantify uncertainty, and allow you to manipulate concepts like random variables and probability distribution functions.

### 2.1 Descriptive Statistics

It is crucial to summarize data samples into compact metrics to apprehend a dataset. Let's review the basics of descriptive statistics useful to give you a first glance at your data.

#### 2.1.1 Mean, Variance and Standard Deviation

##### 2.1.1.1 Mean

The *mean* of a vector, usually denoted as  $\bar{x}$ , is the mean of its elements, that is to say the sum of the components divided by the number of components. Mathematically, it can be expressed as:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

### 2.1.1.2 Variance and Standard Deviation

The *Variance* is a value describing how the data is spread around the mean. A dataset with a large variance means that data points are spread far away from the mean. A dataset with a small variance means that the data points are grouped closely around the mean.

The variance is the mean of the squared differences to the mean. Mathematically, the variance of a variable  $x$  (note that, in this context, we call variable a list of observations, and not a single value) is expressed with the following formula:

$$\text{Var}(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

with  $\text{Var}(x)$  being the variance of the variable  $x$ ,  $n$  the number of data samples,  $x_i$  the  $i$ th data sample and  $\bar{x}$  the mean of  $x$ .

You can use the function `np.var()` from Numpy to calculate it:

```
x = np.array([1, 2, 3, 4, 5, 6])
np.var(x)
```

2.916666666666665

The *standard deviation* is simply the square root of the variance. It is usually denoted as  $\sigma$ :

$$\sigma(x) = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$$

We square root the variance to go back to the units of the observations.

Both the variance and the standard deviation are *dispersion indicators*: they tell you if the observations are clustered around the mean.

Note also that the variance and the standard deviation are always positive (it is like a distance, measuring how far away the data points are from the mean):

$$\text{Var}(x) \geq 0$$

and

$$\sigma(x) \geq 0$$

### 2.1.2 Covariance and Correlation

As you saw in the last section, variance is expressed as differences from the mean. You can extend this idea and compare this difference between two variables. It leads to the idea of *covariance*. You'll also see how it relates to *correlation*.

#### 2.1.2.1 Covariance

The covariance between two variables tells if large values in one variable are associated with large values in the other and, conversely, if small values in one variable are associated with small values in the other.

It is calculated by multiplying the differences from the mean for both variables and dividing by the number of samples:

$$\text{Cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

With  $i$  being the current observation,  $x$  and  $y$  the two variables, and  $\bar{x}$  and  $\bar{y}$  the mean of  $x$  and  $y$  respectively.

A positive covariance means that the two variables varies in the same direction: the first is large when the second is large and small when the second is small.

A negative covariance means that large values of a variable are associated with small values of the other.

Finally, a covariance close to 0 means that there is no linear relationship between the variables.

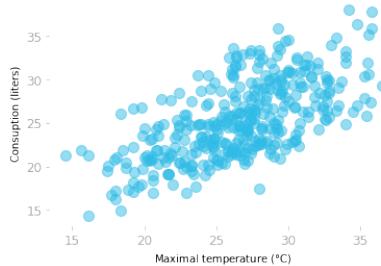
Let's look at a few examples with different situations.

**Positive covariance** Let's see an example of positive covariance between two variables: the beer consumption and the temperature. The beer dataset<sup>1</sup> shows the consumption of beer in São Paulo, Brazil for the year 2015.

Let's start by loading the data and plot the consumption as a function of the maximal temperature:

```
data_beer = pd.read_csv("https://raw.githubusercontent.com/hadrienj/" \
"essential_math_for_data_science/master/data/beer_dataset.csv")

plt.scatter(data_beer['Temperatura Maxima (C)'],
            data_beer['Consumo de cerveja (litros)'],
            alpha=0.5)
# [...] Add labels
```



*Figure 2.1: Example of variables with a positive covariance. The consumption of beer as a function of the temperature of the day. There are one point per day for the whole year 2015.*

You can see in Figure 2.1 that large values on the  $x$ -axis are associated with large values on the  $y$ -axis and small values on the  $x$ -axis are associated with small values on the  $y$ -axis. The consumption of beer increases when the temperature increases.

---

<sup>1</sup><https://www.kaggle.com/dongeorge/beer-consumption-sao-paulo>

Using the previous equation, you can calculate the covariance between the consumption of beer and the maximal temperature with:

```
x = data_beer['Temperatura Maxima (C)']
y = data_beer['Consumo de cerveja (litros)']

np.sum((x - x.mean()) * (y - y.mean())) / x.shape[0]
```

12.172649474197785

You can also use the Numpy function `np.cov()`:

```
np.cov(data_beer['Temperatura Maxima (C)'],
       data_beer['Consumo de cerveja (litros)'])

array([[18.63964745, 12.20609082],
       [12.20609082, 19.35245652]])
```

The function `np.cov()` returns a *covariance matrix* (more details in Section 2.1.3). The covariance is the value outside of the diagonal: again around 12.2. It is different than our result because it uses a correction named the *Bessel correction*<sup>2</sup> where the division is done by the number of samples minus one. This can be changed setting the parameter *ddof* (delta degree of freedom) to 0.

Note that since the covariance is calculated with the difference from the mean, this metric depends on the scale of the variables. For instance, large value for the variables leads to large covariance values.

**Negative covariance** Let's now consider happiness as a function of perception of corruption from the 2020 worldwide happiness report<sup>3</sup>. Let's load and plot the data:

```
data_hap = pd.read_csv("https://raw.githubusercontent.com/hadrienj/" \
"essential_math_for_data_science/master/data/happiness_2020.csv")
```

---

<sup>2</sup>The goal is to correct for an underestimation of the value. You can find more details here: <https://www.statisticshowto.com/bessels-correction/>

<sup>3</sup><https://worldhappiness.report/ed/2020/>

```
plt.scatter(data_hap["Perceptions of corruption"],
            data_hap["Ladder score"],
            alpha=0.5)
# [...] Add labels
```



Figure 2.2: Example of data with a negative covariance.

You can see in Figure 2.2 that happiness decreases when perception of corruption increases. Let's calculate the covariance:

```
np.cov(data_hap["Perceptions of corruption"], data_hap["Ladder score"])

array([[ 0.03068538, -0.08150217],
       [-0.08150217,  1.23714494]])
```

As expected, the value of the covariance is negative (around -0.08).

### 2.1.2.2 Correlation

The *correlation*, usually referring to the *Pearson's correlation coefficient*, is a normalized version of the covariance. It is scaled between -1 and 1, unlike the covariance, which can take any value between negative infinity and positive infinity. A correlation of -1 means that there is a perfect negative linear relationship and a correlation of 1 indicates a perfect positive linear relationship. A correlation of 0 indicates that the variables are uncorrelated (no linear relationship).

Mathematically, the correlation coefficient is calculated by dividing the covariance by the product of the standard deviations of the two variables. For instance, for two variables  $x$  and  $y$ :

$$\text{Corr}(x, y) = \frac{\text{Cov}(x, y)}{\sigma_x \sigma_y}$$

Let's take again the consumption of beer as a function of temperature, as plotted in Figure 2.1. You can calculate the correlation between these variables with Numpy function `np.corrcoef()`:

```
np.corrcoef(data_beer['Temperatura Maxima (C)'],
             data_beer['Consumo de cerveja (litros)'])

array([[1.          , 0.64267247],
       [0.64267247, 1.         ]])
```

Reading the value outside of the diagonal (which refer to the correlation of each variable with itself), you can see that the value obtained for the correlation coefficient is around 0.64.

Correlation is advantageous because you can compare it between pairs of variables since it not depends on the scale of the data.

### 2.1.3 Covariance Matrix

#### 2.1.3.1 Definition

When you previously used the function `np.cov()` from Numpy, it returned more than one value: this is what we call a *covariance matrix* or a *variance-covariance matrix*.

The covariance matrix is a way to structure variance and covariance between two or more variables. With  $n$  variables, the covariance matrix is a  $n$  by  $n$  matrix. The diagonal is filled with the variance of each variable, and the other cells correspond to the covariance between the pairs variables, as illustrated in Figure 2.3.

	$x$	$y$
$x$	Var( $x$ )	Cov( $x, y$ )
$y$	Cov( $y, x$ )	Var( $y$ )

Figure 2.3: Illustration of a covariance matrix of the variables  $x, y$ .

The cells showing the covariance of a variable with itself, like the covariance of the variable  $x$  with the variable  $x$ , correspond to the variance of this variable. Using the formula of the covariance and applying it to the same variable  $x$ , you have:

$$\begin{aligned}\text{Cov}(x, x) &= \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x}) \\ &= \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \\ &= \text{Var}(x)\end{aligned}$$

This shows why the diagonal of the covariance matrix corresponds to the variance of the variables.

## 2.2 Random Variables

Unlike the variables you're using in calculus, a *random variable* takes a value corresponding to the outcome of a random experiment (like a coin flip, for instance). The term *variable* tells us that the value is not fixed (it depends on the event), and the term *random* comes from the fact that the outcome is not fully determined: it can vary randomly.

More formally, a random variable is a function that takes an outcome as input and returns a value. For instance, if you flip a coin, the two possible outcomes are ‘heads’ and ‘tails’. An example of random variable would map ‘heads’ to 0 and ‘tails’ to 1. As another example, let’s say that you flip the coin 10 times, you could have a random variable that maps the sequence of outcomes you get to the number of times you get ‘heads’.

## 2.2.1 Definitions and Notation

### 2.2.1.1 Definitions

It is worth defining some terms in the context of probability theory. Feel free to refer to these definitions along your reading if needed.

A *random experiment*, or simply *experiment* describes a process that gives you uncertain results, as a coin flip, for instance.

The *outcome* of a random experiment is the result you obtain. If you flip a coin, the two possible outcomes are ‘heads’ and ‘tails’. If you pick a card in a deck of 52 playing cards, the 52 possible outcomes are the 52 playing cards. If you roll a six-sided die, the possible outcomes are 1, 2, 3, 4, 5, or 6.

The *sample space* is the set of all possible outcomes. For instance, the sample space of a dice-rolling experiment is  $\{1, 2, 3, 4, 5, 6\}$ . The curly braces ( $\{\}$ ) are used to denote sets (collections of things). In an experiment where you flip a coin two times, the sample space is  $\{\text{heads-heads}, \text{heads-tails}, \text{tails-heads}, \text{tails-tails}\}$ . The sample space is often denoted as  $S$ .

An *event* is a set of outcomes, that is, a subset of the sample space. Intuitively, it corresponds to a question you can ask about the outcome of a random experiment. For instance, if you roll a die, you can ask:

- ‘Is the outcome a 3?’
- ‘Is the outcome in the set  $\{1, 3, 6\}$ ?’

These questions correspond to individual events. An event has occurred when the outcome of the experiment is an element of the event.

The *probability* associated with an event tells you how likely it is to occur. Probabilities must satisfy the following rules:

- It must be a non-negative real number.
- It cannot exceed 1.
- The probability that one of the possible outcomes will occur is 1.
- For two mutually exclusive events, the probability that either of the events occurs is equal to the sum of the probability that each event occurs.

Roughly speaking, a probability of 1 means that the event will be always satisfied and a probability of 0 means that the event will never be satisfied.<sup>4</sup>

A *probability distribution function* (PDF) is a function that maps each outcome to its probability mass (for discrete variables) or to probability density (for continuous variable) to occur. You'll see more details about probability mass and density in Section 2.3.

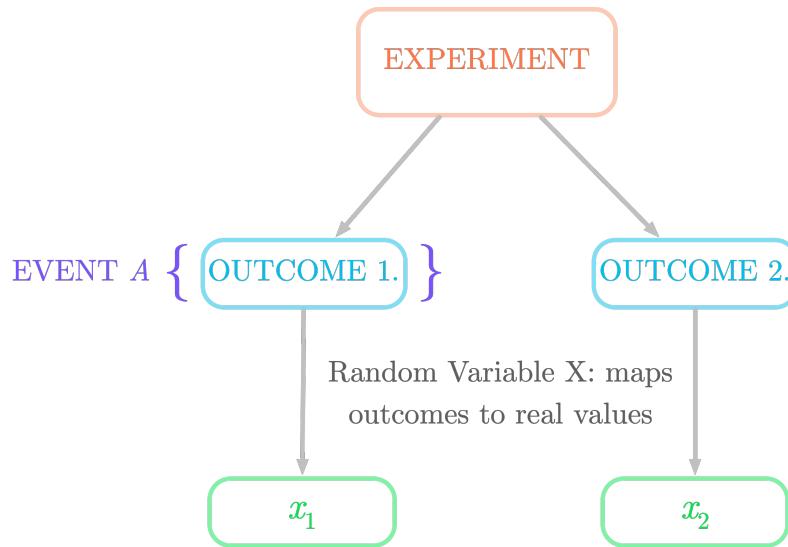
### 2.2.1.2 Notation

Now that we have defined the terms, let's see the notation.

We'll denote random variables with uppercase letters, such as  $X$ . The probability distribution associated with the random variable  $X$  is denoted as  $P(X)$ . The probability that this random variable takes the value  $x$  is denoted as  $P(X = x)$ , or simply  $P(x)$ .

---

<sup>4</sup>However, in the case of continuous random variables where the sample space is infinite, a probability of 1 means that the event is *almost* certain to occur and a probability of 0 means that the event will *almost* never take place.



*Figure 2.4: Summary of the relationship between the terms defined in the context of probability theory.*

Figure 2.4 summarizes how the terms are related between each other. An experiment is associated with various possible outcomes. These outcomes are mapped to real values by a random variable. There is thus a single random variable associated with the experiment. Still in Figure 2.4, the event  $A$  is a set of outcomes, here corresponding to a single outcome ("outcome 1.").

Let's take an example.

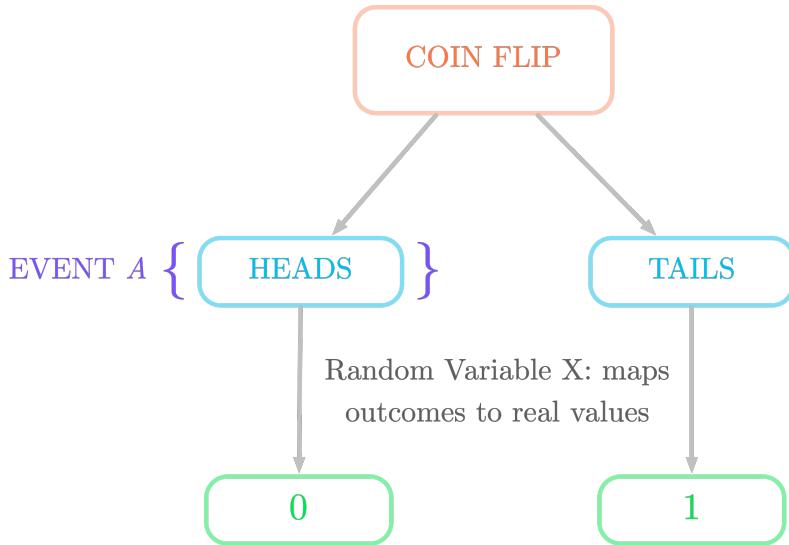


Figure 2.5: Example with a coin flip.

Figure 2.5 shows the case of a coin flip experiment. The two possible outcomes are ‘heads’ or ‘tails’. The random variable  $X$  maps these outcomes to real values, respectively 0 and 1. The event  $A$  corresponds to the following set of outcomes: {‘heads’}. This means that the probability that the outcome is ‘head’ can be denoted as:

$$P(X = 0) = P('heads') = P(A)$$

### 2.2.2 Discrete and Continuous Random Variables

It is important to distinguish *discrete random variables*, where the outcomes are in a *at most countable* list of possible values (meaning that this list can be infinite but each element can be associated to a natural number), and *continuous random variables*, which can take any value in an interval.

**Discrete Random Variables** Let’s say that you roll a six-sided die and the outcome is a number between 1 and 6. You can model this experiment with a random variable  $X$  which can return the following values as possible outcome: 1, 2, 3, 4, 5 or 6. In this example, the random variable is *discrete*: it takes discrete values between 1 and 6 (for instance, it can’t be 1.5).

**Continuous Random Variables** Continuous random variables can take any value in a range. For instance, if you calculate the heights of randomly selected people, the number of possible outcomes is infinite (theoretically, because in the real world, the measure would be discretized, for instance by the precision of the measurement).

## 2.3 Probability Distributions

*Deterministic* processes give the same results when they are repeated multiple times. This is not the case for random variables, which describe *stochastic* events, in which randomness characterizes the process.

This means that random variables can take various values. How can you describe and compare these values? One good way is to use the probability that each outcome will occur. The probability distribution of a random variable is a function that takes the sample space as input and returns probabilities: in other words, it maps possible outcomes to their probabilities.

In this section, you'll learn about probability distributions for discrete and continuous variables.

### 2.3.1 Probability Mass Functions

Probability functions of discrete random variables are called *probability mass functions* (or PMF). For instance, let's say that you're running a dice-rolling experiment. You call  $X$  the random variable corresponding to this experiment. Assuming that the die is fair, each outcome is *equiprobable*: if you run the experiment a large number of times, you will get each outcome approximately the same number of times. Here, there are six possible outcomes, so you have one chance over six to draw each number.

Thus, the probability mass function describing  $X$  returns  $\frac{1}{6}$  for each possible outcome and 0 otherwise (because you can't get something different than 1, 2, 3, 4, 5 or 6).

You can write  $P(X = 1) = \frac{1}{6}$ ,  $P(X = 2) = \frac{1}{6}$ , and so on.

### 2.3.1.1 Properties of Probability Mass Functions

Not every function can be considered as a probability mass function. A probability mass function must satisfy the following two conditions:

- The function must return values between 0 and 1 for each possible outcome:

$$0 \leq P(x) \leq 1$$

- The sum of probabilities corresponding to all the possible outcomes must be equal to 1:

$$\sum_{x \in S} P(x) = 1$$

The value of  $x$  can be any real number because values outside of the sample space are associated with a probability of 0. Mathematically, for any value  $x$  not in the sample space  $S$ ,  $P(x) = 0$ .

### 2.3.1.2 Simulation of the Dice Experiment

Let's simulate a die experiment using the function `np.random.randint(low, high, size)` from Numpy which draw  $n$  (`size`) random integers between `low` and `high` (excluded). Let's simulate 20 die rolls:

```
rolls = np.random.randint(1, 7, 20)
rolls

array([6, 3, 5, 3, 2, 4, 3, 4, 2, 2, 1, 2, 2, 1, 1, 2, 4, 6, 5, 1])
```

This array contains the 20 outcomes of the experiment. Let's call  $X$  the discrete random variable corresponding to the die rolling experiment. The probability mass function of  $X$  is defined only for the possible outcomes and gives you the probability for each of them.

Assuming the die is fair, you should have an *uniform distribution*, that is, equiprobable outcomes.<sup>5</sup>.

---

<sup>5</sup>`np.random.randint()` draw samples from an uniform distribution

Let's visualize the quantity of each outcome you got in the random experiment. You can divide by the number of trials to get the probability. Let's use `plt.stem()` from Matplotlib to visualize these probabilities:

```
val, counts = np.unique(rolls, return_counts=True)
plt.stem(val, counts/len(rolls), basefmt="C2-", use_line_collection=True)
```

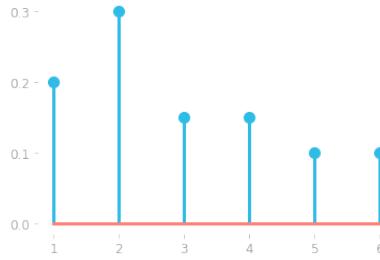


Figure 2.6: Probability mass function of the random variable  $X$  corresponding to a die rolling a six-sided die estimated from 20 rolls.

With a uniform distribution, the plot would have the same height for each outcome (since the height corresponds to the probability, which is the same for each outcome of a die throw). However, the distribution shown in Figure 2.6 doesn't look uniform. That's because you didn't repeat the experiment enough: the probabilities will stand when you repeat the experiment a large number of times (in theory, an infinite number of times).

Let's increase the number of trials:

```
throws = np.random.randint(1, 7, 100000)
val, counts = np.unique(throws, return_counts=True)
plt.stem(val, counts/len(throws), basefmt="C2-", use_line_collection=True)
```

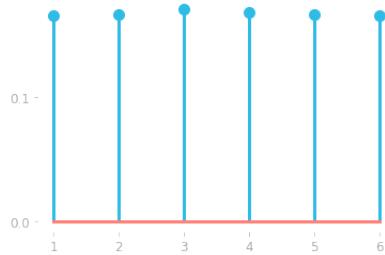


Figure 2.7: Probability mass function of the random variable  $X$  corresponding to a die rolling experiment estimated from 100,000 rolls.

With enough trials, the probability mass function showed in Figure 2.7 looks uniform. This underline the importance of the number of trials from a frequentist probability point of view.

### 2.3.2 Probability Density Functions

With continuous variables, there is an infinite number of possible outcomes (limited by the number of decimals you use). For instance, if you were drawing a number between 0 and 1 you might get an outcome of, for example, 0.413949834. The probability of drawing each number tends towards zero: if you divide something by a very large number (the number of possible outcomes), the result will be very small, close to zero. This is not very helpful in describing random variables.

It is better to consider the probability of getting a specific number within a range of values. The  $y$ -axis of probability density functions is not a probability. It is called a *probability density* or just *density*. Thus, probability distributions for continuous variables are called *probability density functions* (or PDF).

The integral of the probability density function over a particular interval gives you the probability that a random variable takes a value in this interval. This probability is thus given by the area under the curve in this interval (as you saw in Section 1.2).

### 2.3.2.1 Notation

Here, I'll denote probability density functions using a lowercase  $p$ . For instance, the function  $p(x)$  gives you the density corresponding to the value  $x$ .

### 2.3.2.2 Example

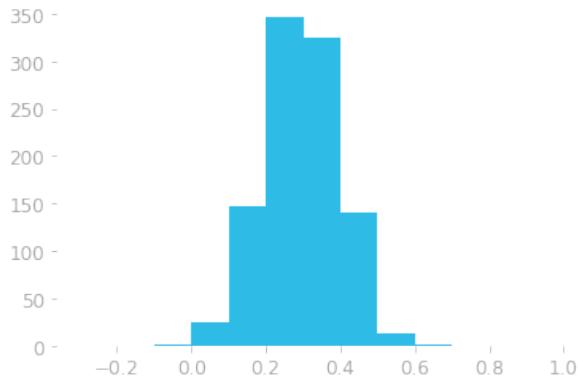
Let's inspect an example of probability density function. You can randomly draw data from a normal distribution using the Numpy function `np.random.normal` (you'll find more details about the normal distribution in Section 3.2).

You can choose the parameters of the normal distribution (the mean and the standard deviation) and the number of samples. Let's create a variable `data` with 1,000 values drawn randomly from a normal distribution with a mean of 0.3 and a standard deviation of 0.1.

```
np.random.seed(123)
data = np.random.normal(0.3, 0.1, 1000)
```

Let's look at the shape of the distribution using an histogram. The function `plt.hist()` returns the exact values for the  $x$ - and  $y$ -coordinates of the histogram. Let's store this in a variable called `hist` for latter use:

```
hist = plt.hist(data, bins=13, range=(-0.3, 1))
```



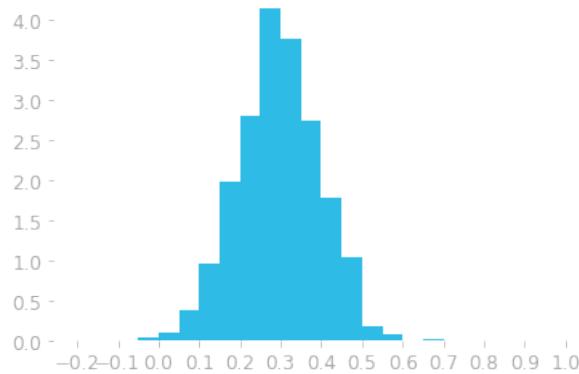
*Figure 2.8: Histogram of the data generated from a normal distribution. The x-axis is the value of the element in the vector and the y-axis the number of elements (count) that are in the corresponding range.*

## Histograms

*Histograms* show how values are distributed. It is a way to model a probability distribution using a finite number of values from the distribution. Since we're dealing with continuous distributions, this histogram corresponds to the number of values for specific intervals (the intervals depends on the parameter `bins` in the function `hist()`). For instance, Figure 2.8 shows that there are around 347 elements in the interval  $[0.2, 0.3]$ . Each bin corresponds to a width of 0.1, since we used 13 bins to represent data in the range -0.3 to 1.

Let's have a closer look at the distribution with more bins. You can use the parameter `density` to make the *y*-axis correspond to the probability density instead of the count of values in each bin:

```
hist = plt.hist(data, bins=24, range=(-0.2, 1), density=True)
```



*Figure 2.9: Histogram using 30 bins and density instead of counts.*

You can see in Figure 2.9 that there are more bins in this histogram (24 instead of 13). This means that each bin has now a smaller width. The *y*-axis is also on a different scale: it corresponds to the density, not the counter of values as before.

To calculate the probability to draw a value in a certain range from the density, you need to use the area under the curve. In the case of histograms, this is the area of the bars.

Let's take an example with the bar ranging from 0.2 to 0.25, associated with the following density:

```
print(f"Density: {hist[0][8].round(4)}")
print(f"Range x: from {hist[1][8].round(4)} to {hist[1][9].round(4)}")
```

```
Density: 2.8
Range x: from 0.2 to 0.25
```

Since there are 24 bins and the range of possible outcomes is from -0.2 to 1, each bar corresponds to a range of  $\frac{1-(-0.2)}{24} = \frac{1.2}{24} = 0.05$ . In our example, the height of the bar (the one from 0.2 to 0.25) is around 2.8, so the area of this

bar is  $2.8 \cdot 0.05 = 0.14$ . This means that the probability of getting a value between 0.2 and 0.25 is around 0.14, or 14%.

You saw that the sum of the probabilities must be equal to one, so the sum of the bar's areas should be equal to one. Let's check that: you can take the vector containing the densities (`hist[0]`) and multiply it by the bar width (0.05):

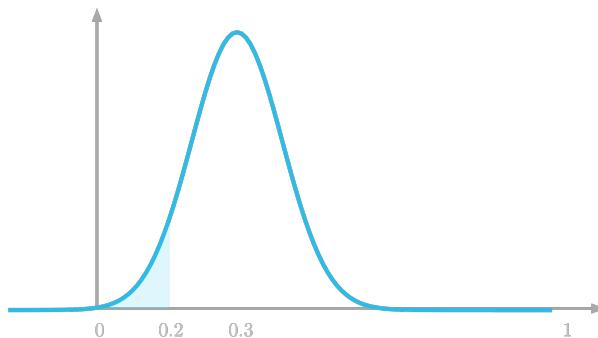
```
(hist[0] * 0.05).sum().round(4)
```

1.0

All good: the sum of the probabilities is equal to one.

### 2.3.2.3 From Histograms to Continuous Probability Density Functions

Histograms represent a binned version of the probability density function. Figure 2.10 shows a representation of the true probability density function. The blue shaded area in the figure corresponds to the probability of getting a number between 0 and 0.2 (the area under the curve between 0 and 0.2).



*Figure 2.10: The probability to draw a number between 0 and 0.2 is the highlighted area under the curve.*

### 2.3.2.4 Properties of Probability Density Functions

Like probability mass functions, probability density functions must satisfy some requirements. The first is that it must return only non negative values. Mathematically written:

$$p(x) \geq 0$$

The second requirement is that the total area under the curve of the probability density function must be equal to 1:

$$\int_{-\infty}^{\infty} p(x) dx = 1$$

In this part on probability distributions, you saw that probability mass functions are for discrete variables and probability density functions for continuous variables. Keep in mind that the value on the  $y$  axis of probability mass functions are probabilities, which is not the case for probability density functions. Look at the density values (for instance in Figure 2.9): they can be larger than one, which shows that they are not probabilities.

## 2.4 Joint, Marginal, and Conditional Probability

In some cases, you need to consider the probability that more than one event occurs. You'll see here how to handle this situation.

### 2.4.1 Joint Probability

*Joint probability* is the probability that two or more events occur.

#### 2.4.1.1 Notation

The *joint probability* that the random variable X takes the value  $x$  and that the random variable Y takes the value  $y$  is written:

$$P(X = x, Y = y)$$

You must consider the dependence between these events. So, let's first look at the concept of dependency between events.

### 2.4.1.2 Dependent and Independent Events

*Dependent events* are events that are related in the sense that the probability of one depends on the outcome of the other. For instance, if you randomly pick two balls from a bag without replacement, the probabilities associated with the second pick depend on the outcome of the first pick.

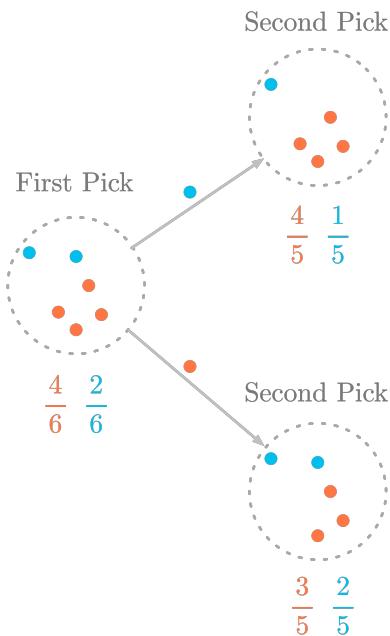


Figure 2.11: Examples of dependent events with a draw without replacement.

Figure 2.11 illustrates the following example: there are 4 red balls and 2 blue balls in the bag. The probability of choosing a red ball is  $\frac{4}{6}$  and  $\frac{2}{6}$  to choose a blue ball.

If you pick a red ball the first time, the probability of picking another red ball in the second draw is  $\frac{3}{5}$ ; that of picking a blue ball is  $\frac{2}{5}$  (as illustrated in the bottom right). However, if you get a blue ball at the first draw, the probability of picking a red ball on the second draw is  $\frac{4}{5}$  and  $\frac{1}{5}$  to pick another blue ball (top right). The first draw has an influence on the probability distribution of the second.

By contrast, *independent events* are events that don't interfere with each other. For instance, if you roll two dice, you don't expect that the outcome of one will affect the outcome of the other.

#### 2.4.1.3 Joint Probability of Independent and Dependent Events

**Independent Events** If two random variables X and Y are independent, their joint probability is equal to the product of the probability of each event. Mathematically, you can write:

$$P(X = x, Y = y) = P(X = x)P(Y = y)$$

**Dependent Events** For dependent events, you'll need to use rules from conditional probabilities (as we'll detail in Section 2.4.3) to calculate their joint probability.

#### 2.4.1.4 Example with Independent Events

For instance, if you throw a die and toss a coin, what is the probability to get a 6 and 'heads'?

To answer this question, you need to consider both events. The joint probability is calculated by multiplying the probability of each event together:

$$P(X = 6, Y = \text{'heads'}) = \frac{1}{6} \times \frac{1}{2} = \frac{1}{12}$$

Note that  $P(Y = \text{'heads'})$  is a shortcut because random variables maps outcomes to real values: it means that the experiment associated with the random variable Y has the outcome 'heads'.

## 2.4.2 Marginal Probability

In some cases, you'll have the joint probability of multiple events and you'll want to calculate the probability of one of the events individually, without considering the others. Such probabilities are called *marginal probabilities*.

In the preceding example of a coin toss and a die throw, the marginal probabilities are simply the probabilities of single events (respectively,  $\frac{1}{6}$  and  $\frac{1}{2}$ ).

		Throw a Die						
		1	2	3	4	5	6	
Flip a Coin	Heads	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{2}$
	Tails	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{2}$
Joint Probabilities								
$\frac{1}{6}$ $\frac{1}{6}$ $\frac{1}{6}$ $\frac{1}{6}$ $\frac{1}{6}$ $\frac{1}{6}$								Marginal probabilities

Figure 2.12: Joint and marginal probability.

Figure 2.12 summarizes the joint and marginal probabilities for this example. Let's see the rule of this calculation.

#### 2.4.2.1 The Sum Rule

Marginal probabilities are obtained from joint probabilities by *marginalizing* the other variables using the *sum rule of probability*: you add the joint probabilities. Mathematically, it is defined as follows:

$$P(X = x) = \sum_y P(X = x, Y = y)$$

The marginal probability  $P(X = x)$  is the probability that  $X = x$  without considering the other variables. This is calculated as the sum of the probabilities that  $X = x$  and  $Y = y$ , for all values of  $y$ :  $\sum_y P(X = x, Y = y)$ .

Let's take again the case of a die throw and coin flip experiment. For instance, the marginal probability of  $X = 6$  is the sum of the following joint probabilities:  $P(X = 6, Y = \text{'heads'})$  and  $P(X = 6, Y = \text{'tails'})$ . It is thus:

$$\begin{aligned} \sum_y P(X = 6, Y = y) &= P(X = 6, Y = \text{'heads'}) + P(X = 6, Y = \text{'tails'}) \\ &= \frac{1}{12} + \frac{1}{12} = \frac{1}{6} \end{aligned}$$

Looking at Figure 2.12, the marginal probability is the sum of the elements in the column corresponding to  $X = 6$ .

#### 2.4.2.2 Marginal Probability Density Functions

In the case of continuous variables, you calculate the marginal probabilities by integrating the joint probability distribution with respect to one of the variables.

Let's visualize the joint probability functions of two continuous random variables,  $X$  and  $Y$ , to understand how it works. Since there are two variables, the density needs to be represented on a three-dimensional plot.

Note also that the joint probability distribution is constructed with the product of each probability density (you can refer to the code in the notebook). You'll see that the density is calculated as `z = gaussian(xy[:, 0], 0.4, 0.1) * gaussian(xy[:, 1], 0.6, 0.05)`.

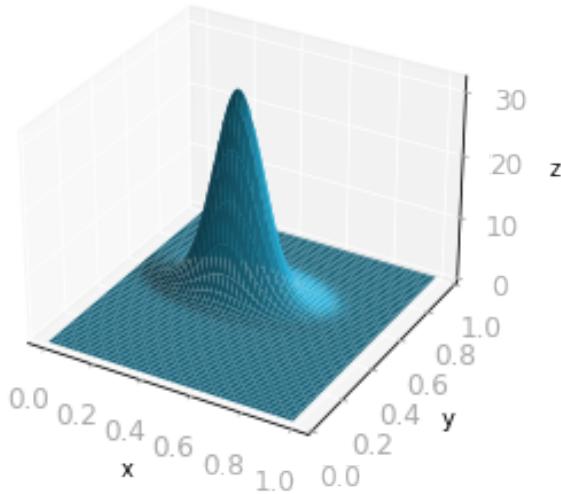


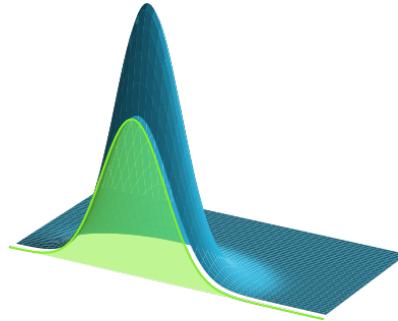
Figure 2.13: Joint probability distribution of the random variables  $X$  and  $Y$ .

Figure 2.13 shows the joint probability density function of  $X$  and  $Y$ . The total volume of this shape must be equal to one. This volume is calculated using a *double integral*<sup>6</sup>.

To find the marginal probability density of  $Y$ , you'll need to integrate with respect to  $X$ , meaning that you consider  $Y$  as a constant. Graphically, it corresponds to do slices on the  $x$ -axis at different values of  $Y$ .

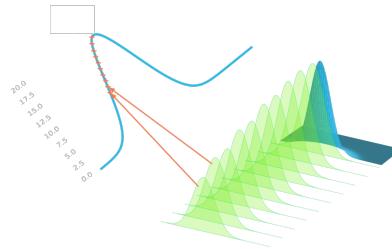
---

<sup>6</sup>You first integrate with respect to one of the variables, and then with respect to the other. You can find here a great resource about double integrals: <https://www.khanacademy.org/math/multivariable-calculus/integrating-multivariable-functions/double-integrals-a/a/double-integrals>



*Figure 2.14: Distribution of X at Y = 0.55.*

For instance, Figure 2.14 shows a slice at  $Y = 0.55$ . This area is one point of the probability density function of  $X$ .



*Figure 2.15: The area under the curve of each slice corresponds to a point in the probability density function.*

If you repeat this for all values of  $Y$ , you get the whole probability density distribution: the probability density function of  $Y$  is constructed as the areas of the slices, as illustrated in Figure 2.15.

To summarize, when you integrate with respect to  $X$ , denoted as  $\int p(x, y)dx$ , you get a function that gives the area of the slice for a specific value of  $Y$ . This area corresponds to one point on the probability density distribution of  $Y$ .

#### 2.4.2.3 Example

Let's check the previous statement by comparing the slice areas and the density function of  $Y$  with an example of the previous random variables  $X$

and Y.

Let's create the variable `z`, as in Figure 2.13:

```
x, y = np.mgrid[0:1:0.01, 0:1:0.01]
xy = np.column_stack([x.flat, y.flat])

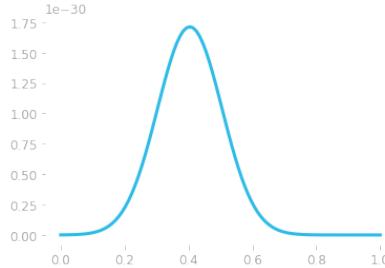
z = gaussian(xy[:, 0], mu_x, sigma_x) * gaussian(xy[:, 1], mu_y, sigma_y)
z = z.reshape(x.shape)

z.shape

(100, 100)
```

You can extract slices from `z` with for instance `z[:, 0]`, which is the first slice, corresponding to  $Y = 0$ . Slices are separated by 0.01. If you plot one of the slices, you get Figure 2.16:

```
x_axis = np.linspace(0, 1, z.shape[0])
plt.plot(x_axis, z[:, 0])
```



*Figure 2.16: The first slice.*

You want to calculate the area under the curve shown in Figure 2.16. To estimate it, simply calculate the sum of each value multiplied by the step (0.01 because you have one point every 0.01):

```
(z[:, 0] * 0.01).sum()
```

```
4.292658322616401e-31
```

This value is the area under the curve of the first slice.

Now that you've got the idea, let's do this for every slice:

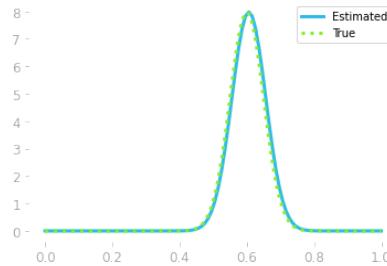
```
estimated_probability_density = np.zeros(x_axis.shape[0])

for i in range(z.shape[1]):
    slice_area = (z[:, i] * 0.01).sum()
    estimated_probability_density[i] = slice_area
```

Let's compare what you obtain with the true density distribution that you got as `density_y`:

```
density_y = gaussian(x_axis, mu_y, sigma_y)

plt.plot(x_axis, estimated_probability_density, label="Estimated")
plt.plot(x_axis, density_y, linestyle=':', label="True")
```



*Figure 2.17: Comparison of the estimated and true marginal probability functions.*

That's good: you can see in Figure 2.17 that these two density functions are quite identical, showing that the marginal probability density function can be obtained from the joint probability distribution.

#### 2.4.2.4 Conclusion

To go from the joint probability distribution to the marginal probability distribution, you calculate the integral of the joint probability distribution

$p(x, y)$  with respect to one of the variable, such as:

$$p(y) = \int p(x, y) dx$$

and

$$p(x) = \int p(x, y) dy$$

In the previous example, we approximated the integral by calculating the area under the curve using slices as you saw in Section 1.2.

### 2.4.3 Conditional Probability

In some cases, you will want to calculate the probability that an event will occur knowing that another event has already occurred. For instance, you can ask what is the probability to draw a pair of kings in a deck of playing cards. You need to calculate the probability of getting a king in the first draw, and then the probability to draw a second king. You want to know the probability of the second draw *under the condition* that you got a king in the first draw.

This is called *conditional probability*: the probability of an event given that another event has occurred.

#### 2.4.3.1 Notation

We refer to conditional probability using the vertical bar (“|”), as follows:

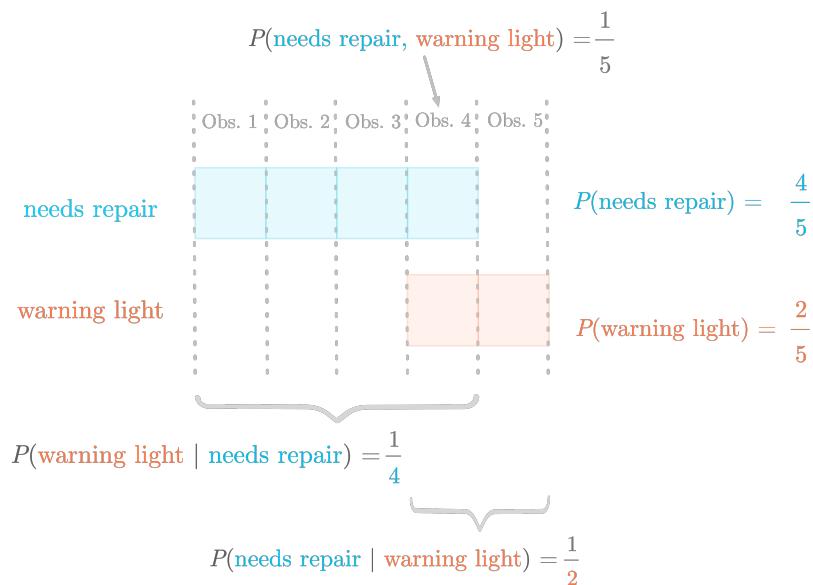
$$P(Y = y | X = x)$$

The conditional probability  $P(Y = y | X = x)$  is the probability that the random variable Y equals  $y$  *given that* the random variable X equals  $x$ . Note that this notation differs from  $P(Y = y, X = x)$ , which is the probability that both  $X = x$  and  $Y = y$ .

### 2.4.3.2 Example

Samantha's car sometimes shows a warning light when there is a problem, but it is defective: sometimes the warning light comes on even when there is no problem, and sometimes the light is off but the car needs repair.

Let's call  $P(\text{warning light})$  the probability of the event "the warning light is on" and  $P(\text{needs repair})$  the probability of the event "the car needs repair".



*Figure 2.18: The probability that the car needs repair is illustrated in blue and the probability that the warning light is on is illustrated in red, for five observations.*

Figure 2.18 illustrates the probabilities that the car needs repair (in blue) and the probability that the warning light is on (in red). The figure shows the data for five observations (the five columns from 'Obs. 1' to 'Obs. 5'). For each of them, Samantha looked at the status of the car (needs repair or not) and the warning light (on or off). A blue square tells that the car needed repair and a red square that the warning light was on. For instance, in the first observation, the car needed repair but the warning light was off (no red square).

You can see that  $P(\text{needs repair}) = \frac{4}{5}$  because there are four observations

where the car needed repair out of a total of five observations. Similarly,  $P(\text{warning light}) = \frac{2}{5}$  because there are two observations where the warning light was on. There is also an observation with a blue and a red square: the car needed repair and the warning light was on, so  $P(\text{needs repair, warning light}) = \frac{1}{5}$ .

Let's examine the conditional probabilities for this example. The statement  $P(\text{needs repair}|\text{warning light})$  is the probability that the car needed repair given that the warning light was on. You can think of this probability as follows: you take only the observations where the warning light was on ('Obs. 4' and 'Obs. 5'), and then you look at the proportion of observations where car needed repair. These conditions are met in one observation out of the two so  $P(\text{needs repair}|\text{warning light}) = \frac{1}{2}$ .

The same idea can be used to find  $P(\text{warning light}|\text{needs repair})$ . You take the observations where the car needed repair and consider the number of times the warning light was on. You can see that this is one observation out of four.

To summarize, conditional probability is the probability of a subset of the observations (depending on the condition). The link between conditional and joint probability is formalized in the next section.

#### 2.4.3.3 The Product Rule of Probability

For discrete random variables, the probability that  $X = x$  and  $Y = y$  (joint probability) equals the probability that  $X = x$  multiplied by the probability that  $Y = y$  given that  $X = x$  (conditional probability). Mathematically, you have:

$$P(X = x, Y = y) = P(Y = y|X = x)P(X = x)$$

This is the *product rule of probability*, also called the *chain rule of probability*. Equivalently, you can write:

$$P(Y = y|X = x) = \frac{P(X = x, Y = y)}{P(X = x)}$$

Let's see the logic behind these two formulations of the product rule using the previous example.

First, the joint probability  $P(\text{needs repair}, \text{warning light})$  is:

$$P(\text{needs repair}, \text{warning light}) = P(\text{warning light}|\text{needs repair})P(\text{needs repair})$$

The probability that the car needs repair and the warning light is on ( $P(\text{needs repair}, \text{warning light})$ ) will be smaller than  $P(\text{warning light}|\text{needs repair})$ . To consider all observations, you need to multiply by the proportion of observations where the car needs repair ( $P(\text{needs repair})$ ).

Let's see the second statement with the same example:

$$P(\text{warning light}|\text{needs repair}) = \frac{P(\text{needs repair}, \text{warning light})}{P(\text{needs repair})}$$

Similarly, to go from all observations to only the observations where the car needs repair, you need this time to divide by  $P(\text{needs repair})$ .

#### 2.4.3.4 More Events

We can generalize the product rule to more events. For instance, with three events  $A$ ,  $B$  and  $C$ , you would have:

$$P(A, B, C) = P(A|B, C)P(B, C)$$

This means that the probability that the events  $A$ ,  $B$  and  $C$  occurred is equal to the probability that  $A$  occurred given that  $B$  and  $C$  occurred multiplied by the probability that  $B$  and  $C$  occurred.

You can chain these probability products. Since the product rule says that:

$$P(B, C) = P(B|C)P(C)$$

You get:

$$P(A, B, C) = P(A|B, C)P(B|C)P(C)$$

## 2.5 Cumulative Distribution Functions

*Cumulative distribution functions* (CDFs) are an alternative way to describe a random variable. The CDF corresponds to the probability that a random variable will take a value less than or equal to the value  $x$ . Let's denote  $F_X(x)$  the CDF of the random variable X:

$$F_X(x) = P(X \leq x)$$

When  $x$  increases, the probability necessarily increases because all values smaller than  $x$  are considered (hence, the name *cumulative*). Thus, cumulative distribution functions are always non-decreasing going from 0 to 1.

Let's look at the following normal distribution (you can find more details on normal distribution in Section 3.2) and its CDF:

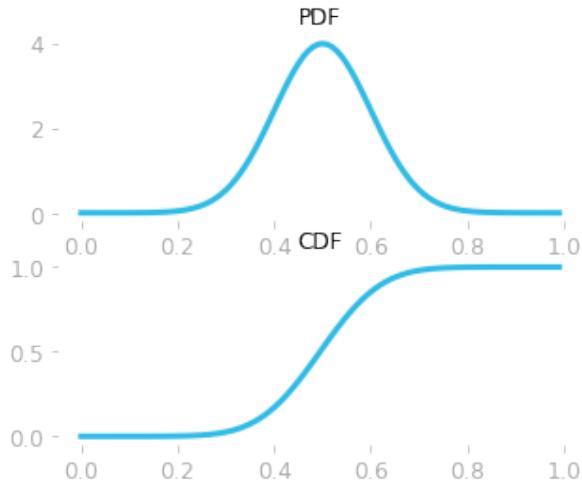


Figure 2.19: Probability density function (top panel) and cumulative density function (bottom panel) for a normal distribution with a mean of 0.5 and a standard deviation of 0.1.

Figure 2.19 shows a plot of the probability density function using a normal distribution of mean 0.5 and standard deviation 0.1. The cumulative distribution function is shown in the bottom panel. The PDF corresponds to the derivative of the CDF.

## 2.6 Expectation and Variance of Random Variables

The *expected value*, or *expectation*, of a random variable is the average value you'll get when you consider a large number of trials. The expected value of a random variable X is written as  $\mathbb{E}[X]$ .

### 2.6.1 Discrete Random Variables

For a discrete variable, the expected value corresponds to the sum of each outcome weighted by its probability:

$$\mathbb{E}[X] = \sum_{i=1}^n P(x_i)x_i$$

with X the random variable having  $n$  possible outcomes,  $x_i$  being the  $i$ th possible outcome that has a probability  $P(x_i)$  to occur.

#### 2.6.1.1 Example

For instance, let's take a random variable X that can take the following values  $x$ : 1, 5, 10 or 100. Let's say that the distribution is uniform: the outcomes are equiprobable, each with a probability of  $\frac{1}{4}$ . The expected value is:

$$\begin{aligned}\mathbb{E}[X] &= P(X = 1) \cdot 1 + P(X = 5) \cdot 5 + P(X = 10) \cdot 10 + P(X = 100) \cdot 100 \\ &= \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 5 + \frac{1}{4} \cdot 10 + \frac{1}{4} \cdot 100 \\ &= \frac{1}{4} + \frac{5}{4} + \frac{10}{4} + \frac{100}{4} = \frac{116}{4} = 29\end{aligned}$$

This means that if you repeat the experiment a large number of time, you'll get in average a value of 29.

You can simulate random trials with the function `np.random.choice()` and calculate the average of the outcomes. Let's plot the average value in a function of the number of trials:

```
np.random.seed(123)

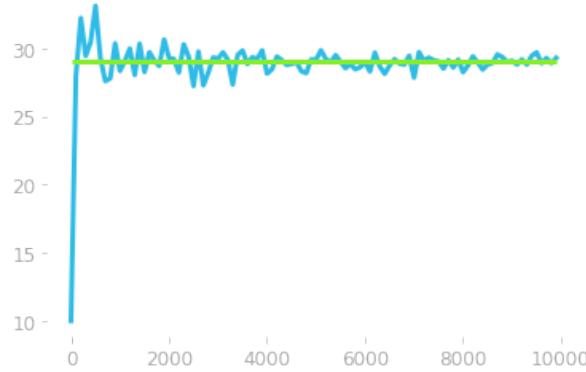
def run_trials(n_trials, choices):
    outcomes = np.zeros(n_trials)

    for i in range(n_trials):
        outcomes[i] = np.random.choice(choices)

    return outcomes.mean()

n_trials = np.arange(1, 10000, 100)
outcomes_average = np.zeros(n_trials.shape[0])
count = 0
for n_trial in n_trials:
    outcomes_average[count] = run_trials(n_trials=n_trial, choices=[1, 5,
    10, 100])
    count += 1

plt.plot(n_trials, outcomes_average, zorder=0)
plt.hlines(y=29, xmin=1, xmax=n_trials[-1], color='#84EE29')
```



*Figure 2.20: The average of the outcomes (y-axis) of a random experiment for different number of trials (from 1 to 10000, as shown in the x-axis). The expected value previously calculated is represented as a green horizontal line.*

Figure 2.20 shows that with enough trials, the result approaches the expected value you calculated (29).

## 2.6.2 Continuous Random Variables

For continuous variables, the idea is similar; you'll just need to integrate instead of summing the weighted outcomes.

$$\mathbb{E}[X] = \int p(x)x \, dx$$

## 2.6.3 Variance of Random Variables

The variance of a random variable is a metric describing the spread of the possible outcomes of the variable. You saw in Section 2.1.1 that the variance is the mean of the squared differences from the mean. With random variables, it is the expected value of the squared differences from the expected value:

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

The variance of a random variable gives information about the variability

of the samples, with a low value when the values agglomerate around the expected value and a large value when they are more spread.

## 2.7 Hands-On Project: The Central Limit Theorem

For this hands-on project, you'll need some knowledge about uniform and Gaussian distributions (you can refer respectively to Section 3.1 and Section 3.2).

The *central limit theorem* says that, if you consider multiple independent random variables drawn from a distribution with  $n$  large enough and average or sum their outcomes, the distribution of the results will be approximated by a Gaussian distribution.

### 2.7.1 Continuous Distribution

Let's implement an experiment to illustrate the central limit theorem with various number of random variables. At each of the 10000 trials of this experiment, you'll get random values from uniform distributions for each random variables and average these values. So you'll get one value per trial and look at the distribution of these values.

```
def mean_distribution_uniform(n_random_var):
    n_trials = 10000
    all_trials = np.zeros(n_trials)
    for i in range(n_trials):
        all_random_var = np.zeros(n_random_var)
        for random_var in range(n_random_var):
            all_random_var[random_var] = np.random.uniform(0, 1)
        all_trials[i] = all_random_var.mean()

    return all_trials
```

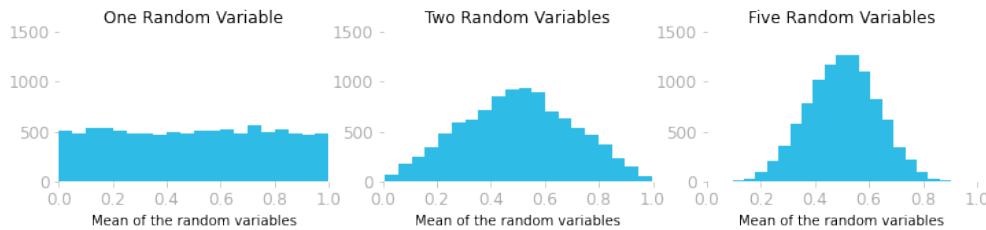
This function allows us to do the experiment with various number of random variables (the parameter `n_random_var`). Let's use it with one, two, and five random variables and look at the distribution of the results with histograms:

```

random_var_uniform_1 = mean_distribution_uniform(1)
random_var_uniform_2 = mean_distribution_uniform(2)
random_var_uniform_5 = mean_distribution_uniform(5)

f, axes = plt.subplots(1, 3, figsize=(12, 2))
axes[0].hist(random_var_uniform_1, bins=20)
axes[1].hist(random_var_uniform_2, bins=20)
axes[2].hist(random_var_uniform_5, bins=20)
# [...] Add titles, limits etc.

```



*Figure 2.21: Histograms showing the distribution of the mean values over 1, 2 or 5 random variables.*

Figure 2.21 shows that the distribution looks more and more like a Gaussian distribution as the number of random variables increases.

## 2.7.2 Discrete Distribution

The central limit theorem can also be observed with discrete variables. Let's say that we roll some dice and look at the sum of the outcomes (in the previous example, you looked at the mean of the outcomes: using the sum is equivalent). For instance, rolling two dice and getting 1 and 5 would lead to a sum of 6.

Let's start by implementing the new trial function:

```

def mean_distribution_dice(n_dice):
    n_trials = 100000
    all_trials = np.zeros(n_trials)
    for i in range(n_trials):
        all_dice = np.zeros(n_dice)

```

```

for dice in range(n_dice):
    all_dice[dice] = np.random.choice([1, 2, 3, 4, 5, 6])
all_trials[i] = all_dice.sum()

return all_trials

```

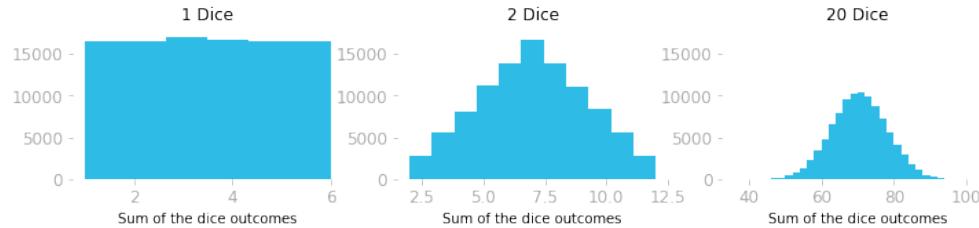
Now, you'll use this function to evaluate the distribution of the sums for 1, 2 and 20 dice:

```

random_var_dice_1 = mean_distribution_dice(1)
random_var_dice_2 = mean_distribution_dice(2)
random_var_dice_20 = mean_distribution_dice(20)

f, axes = plt.subplots(1, 3, figsize=(12, 2))
axes[0].hist(random_var_dice_1, bins=6)
axes[1].hist(random_var_dice_2, bins=11)
axes[2].hist(random_var_dice_20, bins=34)
# [...] Add titles, limits etc.

```



*Figure 2.22: Histograms showing the distribution of the sums of the outcomes in the dice experiment.*

You can see in Figure 2.22 that the same phenomenon shown in Figure 2.21 occurs when variables are drawn from a discrete distribution.

In the next chapter, you'll see more details about the uniform and Gaussian distribution.

# Chapter 03

## Common Probability Distributions

So far in this book, you've learned about different probability distributions, such as the normal and uniform distributions. In this chapter, you'll learn about important discrete and continuous probability distribution functions.

To specify the distribution of a random variable, we use the tilde symbol:  $\sim$ . For instance,  $X \sim \mathcal{P}$  means that a random variable  $X$  has a distribution  $\mathcal{P}$ .

### 3.1 Uniform Distribution

*Uniform distributions* describe random experiments where each possible outcome has the same probability of occurring.

For instance, rolling a die or flipping a coin corresponds to discrete uniform distributions.

Uniform distributions can also be continuous, as shown in Figure 3.1. With a continuous uniform distribution bounded between  $a$  and  $b$ , the probability of getting a value less than  $a$  or greater than  $b$  is 0. For the values between  $a$  and  $b$ , each interval of the same size is associated with the same probability.

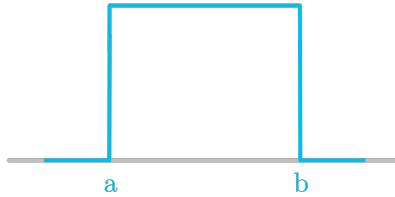


Figure 3.1: Continuous uniform distribution with bounds  $a$  and  $b$ .

## 3.2 Gaussian distribution

*Gaussian distributions*, also called *normal distributions* are one of the most important probability density functions. They are used to model the distribution of continuous random variables. Their shape looks like a bell (as shown in Figure 3.2): there is a peak of probability that decreases when you move off.

In the context of machine learning, Gaussian distributions are used in various algorithms like Gaussian Mixture Models (GMM) or Gaussian Processes.

### 3.2.1 Formula

The formula of the Gaussian distribution is described by the following equation:

$$\mathcal{N}(X = x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

First, the stylized  $N$  ( $\mathcal{N}$ ) is the symbol representing the Gaussian distribution.

The semicolon in  $\mathcal{N}(X = x; \mu, \sigma^2)$  indicates that the distribution is *parametrized* by the two parameters  $\mu$  (the mean of the distribution) and  $\sigma^2$  (the variance).<sup>1</sup> These parameters  $\mu$  and  $\sigma$  are not random variables but determine the distribution (the mean shifts the curve and the variance changes the width of the bell, as you'll see in this section).

---

<sup>1</sup>As usual, you can refer to appendix B for details about the notation. This notation is used for instance in Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.

The distribution is a function of  $x$ , which is an outcome of the random variable X. This is the input of the function.

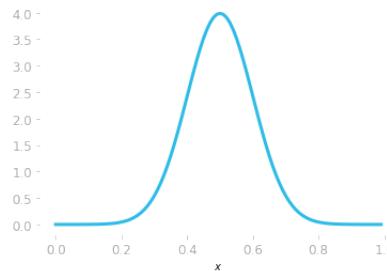
The part of the equation  $\frac{1}{\sqrt{2\pi\sigma^2}}$  is used to normalize the curve so that the total area under the curve is equal to one.

Let's implement the Gaussian function:

```
def gaussian(x, mu, sigma):
    return (1 / np.sqrt(2 * np.pi * sigma ** 2)) *\
        np.exp(-(1 / (2 * sigma ** 2)) * (x - mu) ** 2)
```

Let's plot a Gaussian function with a mean of 0.5 and a standard deviation of 0.1. You can create an array (`x_axis`) using `np.arange()` containing values that will be evaluated by the function (the output will be stored in `y_axis`).

```
x_axis = np.arange(0, 1, 0.01)
y_axis = gaussian(x_axis, 0.5, 0.1)
plt.plot(x_axis, y_axis)
```



*Figure 3.2: An example of Gaussian distribution, with a mean of 0.5 and a standard deviation of 0.1.*

Figure 3.2 shows a normal distribution with mean 0.5 and standard deviation 0.1.

### 3.2.2 Parameters

You saw that Gaussian distributions are parametrized by two parameters: the mean ( $\mu$ ) and the variance ( $\sigma^2$ ).

The width of the curve depends on the variance  $\sigma^2$ : a wide bell shape corresponds to a large variance.<sup>2</sup>

Let's see the effect of  $\sigma$ :

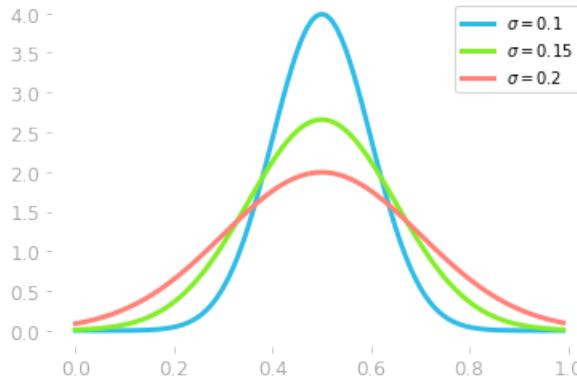


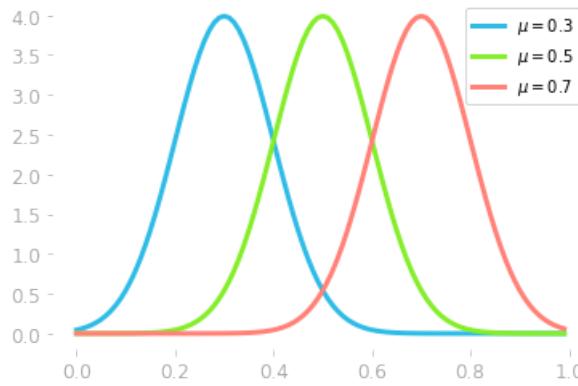
Figure 3.3: Effect of the standard deviation  $\sigma$  on the scale of the normal distribution.

Figure 3.3 shows the effect of the standard deviation on the shape of the normal distribution.

Here is an example of the effect of the mean  $\mu$ :

---

<sup>2</sup>In addition, the inverse of the variance  $\frac{1}{\sigma^2}$  is called the *precision*: a small precision corresponds to a wide bell shape



*Figure 3.4: Effect of the mean  $\mu$  on the normal distribution.*

Figure 3.4 shows that the mean shifts the curve without changing its shape.

### 3.2.3 Requirements

The Gaussian function satisfies the requirements of probability density distributions.

First, it returns only non-negative outputs:

$$\mathcal{N}(X = x; \mu, \sigma^2) \geq 0$$

Second, the total area under the curve (corresponding to the total probability) is equal to 1:

$$\int_{-\infty}^{+\infty} \mathcal{N}(x; \mu, \sigma^2) dx = 1$$

The normal distribution is widely used partly because a lot of random events are well described by it. One reason for that comes from the central limit theorem as you saw in the hands-on project in Section 2.7.

### 3.3 Bernoulli Distribution

The *Bernoulli distribution*, named after the mathematician Jacob Bernoulli, describes binary experiments where outcomes can only be 0 or 1 (such as a coin flip). This distribution has a single parameter  $\mu$  (you can read other names denoting this parameter, such as  $\phi$  and  $\theta$ ), such as  $0 \leq \mu \leq 1$ . This represents the probability that the outcome will be 1, so you have:

$$P(X = 1; \mu) = \mu$$

This means that the probability that the random variable  $X$  takes the value 1 is equal to  $\mu$  (because of the semicolon).

Similarly, the probability that the random variable  $X$  takes the value 0 corresponds to  $1 - \mu$ :

$$P(X = 0; \mu) = 1 - \mu$$

For instance, if you have a biased coin that lands ‘heads’ in 80% of the throws, and that a random variable  $X$  maps ‘heads’ to 1 and ‘tails’ to 0, you have  $\mu = 0.8$ . The probability to land ‘tails’ is then  $1 - \mu = 1 - 0.8 = 0.2$ , or 20%.

The Bernoulli probability distribution can encode both of these probabilities ( $P(X = 1; \mu)$  and  $P(X = 0; \mu)$ ) in a single expression:

$$\text{Bern}(X = x; \mu) = \mu^x(1 - \mu)^{1-x}$$

This is the Bernoulli probability distribution. Since a number raised at the power 0 is equal to 1, and doesn’t change when it is raised to the power of 1, you can check that it works for the two possible outcomes 0 and 1:

$$\text{Bern}(X = 0; \mu) = \mu^0(1 - \mu)^{1-0} = 1 \cdot (1 - \mu) = 1 - \mu$$

and

$$\text{Bern}(X = 1; \mu) = \mu^1(1 - \mu)^{1-1} = \mu$$

You can also check that the sum of the probabilities is equal to 1, as it should be for a probability distribution:

$$\begin{aligned} \sum_{x=0}^1 \text{Bern}(X = x; \mu) &= \text{Bern}(X = 0; \mu) + \text{Bern}(X = 1; \mu) \\ &= 1 - \mu + \mu \\ &= 1 \end{aligned}$$

## 3.4 Binomial Distribution

### 3.4.1 Description

You can use *Binomial distributions* when a binary experiment is ran multiple times. You can use the binomial distribution to answer questions like: “What is the probability of getting ‘heads’ twice if you flip a coin three times?” Bernoulli distributions are a special case of binomial distributions, when the experiment is ran once.

The binomial distribution is defined as follows:

$$\text{Bin}(m; N, \mu) = \binom{N}{m} \mu^m (1 - \mu)^{N-m}$$

The binomial distribution (denoted as  $\text{Bin}$ ) gives the probability of getting  $m$  positive outcome over  $N$  trials.

To understand this expression, let’s consider the example of multiple coin flips: you want to know the probability of getting ‘heads’ twice in three coin flip. Let’s say that the outcome ‘heads’ is encoded as 1 (positive trial) and ‘tails’ as 0. Since you ask for the probability to have two ‘heads’ and three trials,  $m = 2$  and  $N = 3$  in the formula. As with the Bernoulli distribution,  $\mu$  is the parameter corresponding to the probability to get the positive outcome (in our case, ‘heads’). Let’s say that the coin is fair, so  $\mu = 0.5$ .

First,  $\mu^m$  encodes the fact that you want  $m$  ‘heads’, each having the probability  $\mu$  of occurring. Since the trials are independent, you can multiply the

probabilities. In the example, the probability of getting two ‘heads’ is thus  $\mu \times \mu = \mu^2$ . More generally, the probability of getting  $m$  ‘heads’ is  $\mu^m$ .

Second,  $(1 - \mu)^{N-m}$  encodes the probabilities for the remaining trials. In the example, you still need to multiply by the probability of getting one ‘tails’ in the last trial: you don’t want three heads, but exactly two. This is expressed by the probability of getting ‘tails’  $(1 - \mu)$  raised to the power of the number of times you want ‘tails’, which is the number of remaining trials:  $N - m$ .

Note that with  $\mu^m(1 - \mu)^{N-m}$  you don’t consider the fact that multiple combinations are possible. For instance, if you get “heads”, “heads”, and “tails”, you have exactly two “heads”, but it is also possible to get “tails”, “heads”, and “heads”. This is the purpose of  $\binom{N}{m}$ . It is pronounced “ $N$  choose  $m$ ” and it is called the *binomial coefficient*. It refers to the number of possible combinations, with  $N$  being the number of trials (3 in our last example) and  $m$  the number of positive outcomes (in our example, 2).

It is calculated as follows:

$$\binom{N}{m} = \frac{N!}{(N - m)!m!} \text{ for } m \leq N$$

The exclamation point means *factorial* (for instance,  $N!$  is pronounced “ $N$  factorial”), which refers to multiplying the number by itself minus one until you reach one. For instance,

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Note that  $0! = 1$  by convention.

Let’s calculate the binomial coefficient for our last example:

```
def binomial_coeff(N, m):
    return np.math.factorial(N) / (np.math.factorial(N-m) *
                                    np.math.factorial(m))
```

```
binomial_coeff(N=3, m=2)
```

3.0

This means that there are three possible ways to get two ‘heads’ with three flips.

You can do it manually to understand this result. First, list the possible outcomes (‘heads’ is represented as “H” and ‘tails’ as “T.”): HHH, HHT, HTH, THH, HTT, THT, TTH, or TTT.

So there are 8 possibilities. Listing the combinations where you get exactly two heads (HHT, HTH and THH) shows that there are three possible combinations that met the requirements.

You can now implement the binomial function:

```
def binomial(m, N, mu):
    return binomial_coeff(N=N, m=m) * (mu ** m) * ((1 - mu) ** (N - m))
```

Note that this implementation is for illustration purpose but will overflow with larger values of  $N$  and  $m$ .

```
binomial(m=2, N=3, mu=0.5)
```

0.375

You can also use the function `binom.pmf` from the `scipy.stats` module.<sup>3</sup>

```
from scipy.stats import binom
binom.pmf(2, 3, 0.5)
```

0.375

You get  $0.375 = \frac{3}{8}$ , which corresponds also to what you calculated by listing the combinations manually.

---

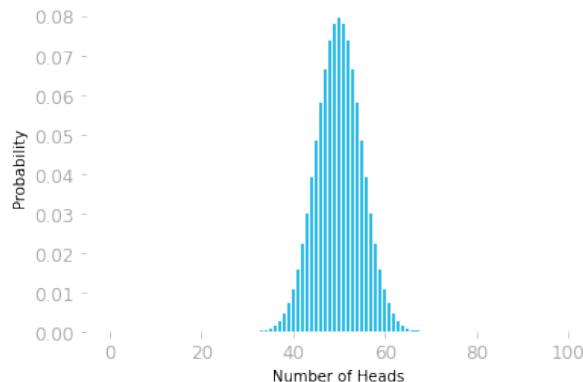
<sup>3</sup>See the documentation here: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.binom.html>. Note that the name of the parameters are different.

### 3.4.2 Graphical Representation

You can calculate the probability of getting various numbers of ‘heads’ for, say 100 trials, and plot the distribution.

```
N = 100
all_binomial_proba = np.zeros(N + 1)
for i in range(N + 1):
    all_binomial_proba[i] = binom.pmf(i, N, 0.5)

plt.bar(np.arange(len(all_binomial_proba)), all_binomial_proba, width=0.7)
# [...] Add axes
```



*Figure 3.5: Probability to get various numbers of  $N$  in 100 trials.*

Here is how you can interpret Figure 3.5: you have a large probability to get around 50 ‘heads’ (because in this example,  $\mu = 0.5$ ). This probability decreases for a smaller or larger number of ‘heads’. For instance, it is quite not probable to get 70 ‘heads’ over the 100 trials.

You can see that when the number of trials is large enough, the distribution looks like the normal distribution. This shows that the binomial distribution can be approximated by normal distributions for large  $N$ .

## 3.5 Poisson Distribution

The *Poisson distribution*, named after the French mathematician Denis Simon Poisson, is a discrete distribution function describing the probability that an event will occur a certain number of times in a fixed time (or space) interval. It is used to model count-based data, like the number of emails arriving in your mailbox in one hour or the number of customers walking into a shop in one day, for instance.

### 3.5.1 Mathematical Definition

Let's start with an example, Figure 3.6 shows the number of emails received by Sarah in intervals of one hour.

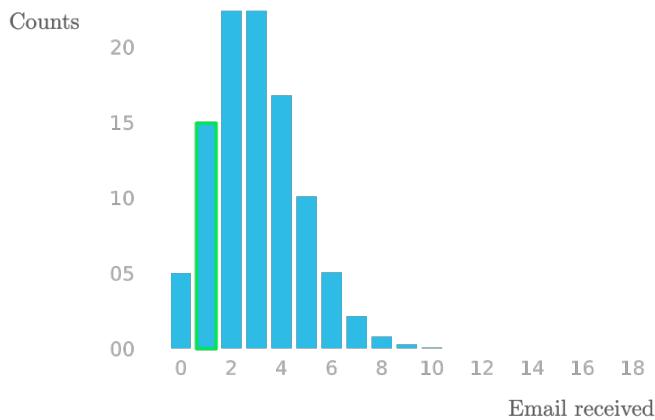


Figure 3.6: Emails received by Sarah in one-hour intervals for the last 100 hours.

The bar heights show the number of one-hour intervals in which Sarah observed the corresponding number of emails. For instance, the highlighted bar shows that there were around 15 one-hour slots where she received a single email.

The Poisson distribution is parametrized by the expected number of events  $\lambda$  (pronounced “lambda”) in a time or space window. The distribution is a function that takes the number of occurrences of the event as input (the integer called  $k$  in the next formula) and outputs the corresponding probability (the probability that there are  $k$  events occurring).

The Poisson distribution, denoted as Poi is expressed as follows:

$$\text{Poi}(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

for  $k = 0, 1, 2, \dots$

The formula of  $\text{Poi}(k; \lambda)$  returns the probability of observing  $k$  events given the parameter  $\lambda$  which corresponds to the expected number of occurrences in that time slot.

### Discrete Distributions

Note that both the binomial and the Poisson distributions are discrete: they give probabilities of discrete outcomes: the number of times an event occurs for the Poisson distribution and the number of successes for the binomial distribution. However, while the binomial calculates this discrete number for a discrete number of trials (like the number of coin tosses), the Poisson considers an infinite number of trials (each trial corresponds to a very small portion of time) leading to a very small probability associated with each event.

You can refer to Section 3.5.3 to see how the Poisson distribution is derived from the binomial distribution.

#### 3.5.2 Example

Priya is recording birds in a national park, using a microphone placed in a tree. She is counting the number of times a bird is recorded singing and wants to model the number of birds singing in a minute. For this task, she'll assume the independence of the detected birds.

Looking at the data of the last few hours, Priya observes that on average, there are two birds detected in an interval of one minute. So the value 2 could be a good candidate for the parameter of the distribution  $\lambda$ . Her goal is to know the probability that a specific number of birds will sing in the next minute.

Let's implement the Poisson distribution function from the formula you saw above:

```
def poisson_distribution(k, lambd):
    return (lambd ** k * np.exp(-lambd)) / np.math.factorial(k)
```

Remember that  $\lambda$  is the expected number of times a bird sings in a one-minute interval, so in this example, you have  $\lambda = 2$ . The function `poisson_distribution(k, lambd)` takes the value of  $k$  and  $\lambda$  and returns the probability to observe  $k$  occurrences (that is, to record  $k$  birds singing).

For instance, the probability of Priya observing 5 birds in the next minute would be:

```
poisson_distribution(k=5, lambd=2)
```

```
0.03608940886309672
```

The probability that 5 birds will sing in the next minute is around 0.036 (3.6%).

As with the binomial function, this will overflow for larger values of  $k$ . For this reason, you might want to use `poisson` from the module `scipy.stats`<sup>4</sup>, as follows:

```
from scipy.stats import poisson
poisson.pmf(5, 2)
```

```
0.03608940886309672
```

Let's plot the distribution for various values of  $k$ :

```
lambd=2
k_axis = np.arange(0, 25)
```

---

<sup>4</sup>See the doc here: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.poisson.html>

```
distribution = np.zeros(k_axis.shape[0])
for i in range(k_axis.shape[0]):
    distribution[i] = poisson.pmf(i, lambd)

plt.bar(k_axis, distribution)
# [...] Add axes, labels...
```

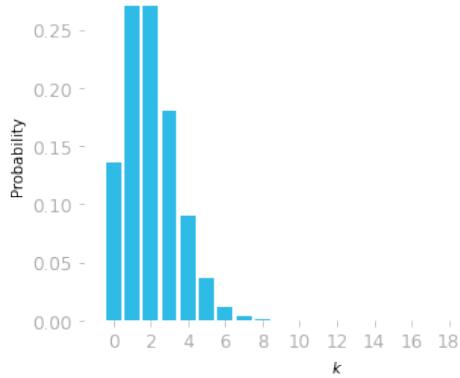


Figure 3.7: Poisson distribution for  $\lambda = 2$ .

The probabilities corresponding to the values of  $k$  are summarized in the probability mass function shown in Figure 3.7. You can see that it is most probable that Priya will hear one or two birds singing in the next minute.

Finally, you can plot the function for different values of  $\lambda$ :

```
f, axes = plt.subplots(6, figsize=(6, 8), sharex=True)

for lambd in range(1, 7):

    k_axis = np.arange(0, 20)
    distribution = np.zeros(k_axis.shape[0])
    for i in range(k_axis.shape[0]):
        distribution[i] = poisson.pmf(i, lambd)

    axes[lambd-1].bar(k_axis, distribution)
    axes[lambd-1].set_xticks(np.arange(0, 20, 2))
    axes[lambd-1].set_title(f"\u03bb: {lambd}")
```

```
# Add axes labels etc.
```

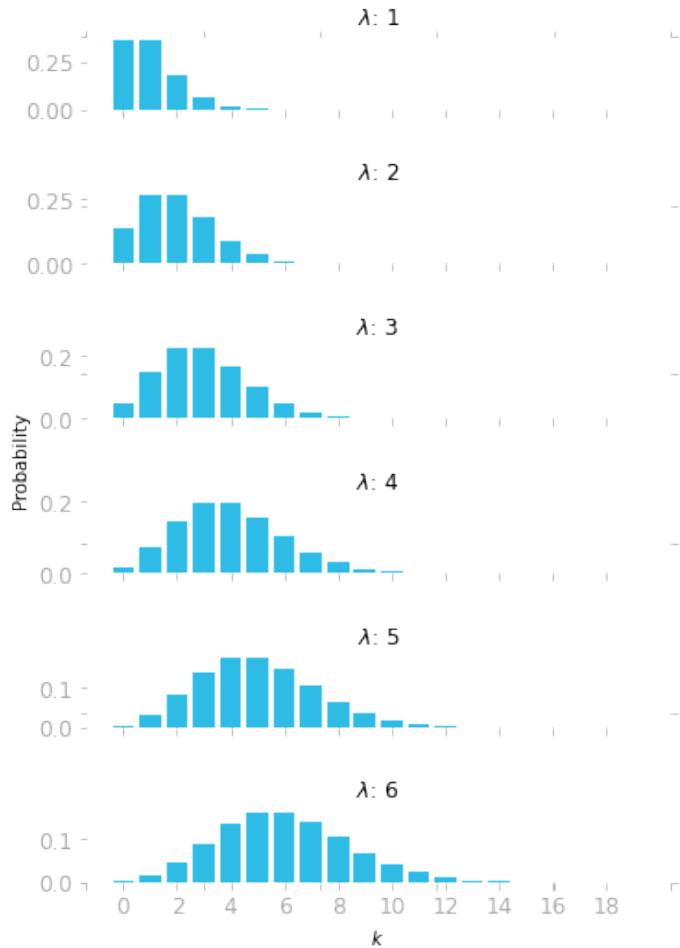


Figure 3.8: Poisson distribution for various values of  $\lambda$ .

Figure 3.8 shows the Poisson distribution for various values of  $\lambda$ , which looks a bit like a normal distribution in some cases. However, the Poisson distribution is discrete, not symmetric when the value of  $\lambda$  is low, and bounded to zero.

### 3.5.3 Bonus: Deriving the Poisson Distribution

Let's see how the Poisson distribution is derived from the Binomial distribution.

You saw in Section 3.4 that if you run a random experiment multiple times, the probability to get  $m$  successes over  $N$  trials, with a probability of a success  $\mu$  at each trial, is calculated through the binomial distribution:

$$\text{Bin}(k; N, \mu) = \binom{N}{k} \mu^k (1 - \mu)^{N-k}$$

#### 3.5.3.1 Problem Statement

How can you use the binomial formula to model the probability to observe an event a certain number of times *in a given time interval* instead of in a certain number of trials? There are a few problems:

1. You don't know  $N$ , since there is no specific number of trials, only a time window.
2. You don't know  $\mu$ , but you have the expected number of times the event will occur. For instance, you know that in the past 100 hours, you received an average of 3 emails per hour, and you want to know the probability of receiving 5 emails in the next hour.

Let's handle these issues mathematically<sup>5</sup>.

To address the first point, you can consider time as small discrete chunks. Let's call these chunks  $\epsilon$  (pronounced "epsilon"), as shown in Figure 3.9. If you consider each chunk as a trial, you have  $N$  chunks.



Figure 3.9: You can split the continuous time in segments of length  $\epsilon$ .

---

<sup>5</sup>You can find more details on the process in Morin, David J. Probability: For the Enthusiastic Beginner. Createspace Independent Publishing Platform, 2016. p.209

The estimation of a continuous time scale is more accurate when  $\epsilon$  is very small. If  $\epsilon$  is small, the number of segments  $N$  will be large. In addition, since the segments are small, the probability of success in each segment is also small.

To summarize, you want to modify the binomial distribution to be able to model a very large number of trials, each with a very small probability of success. The trick is to consider that  $N$  tends toward infinity (because continuous time is approximated by having a value of  $\epsilon$  that tends toward zero).

### 3.5.3.2 Update the Binomial Formula

Let's find  $\mu$  in this case and replace it in the binomial formula. You know the expected number of event in a period of time  $t$ , which we'll call  $\lambda$  (pronounced “lambda”). Since you split  $t$  into small intervals of length  $\epsilon$ , you have the number of trials:

$$N = t \cdot \epsilon$$

You have  $\lambda$  as the number of successes in the  $N$  trials. So the probability  $\mu$  to have a success in one trial is:

$$\mu = \frac{\lambda}{N}$$

Replacing  $\mu$  in the binomial formula, you get:

$$\binom{N}{k} \left(\frac{\lambda}{N}\right)^k \left(1 - \frac{\lambda}{N}\right)^{N-k}$$

Developing the expression, writing the binomial coefficient as factorials (as you did in Section 3.4), and using the fact  $a^{b-c} = a^b - a^c$ , you have:

$$\frac{N!}{(N-k)!k!} \left(\frac{\lambda}{N}\right)^k \left(1 - \frac{\lambda}{N}\right)^N \left(1 - \frac{\lambda}{N}\right)^{-k}$$

Let's consider the first element of this expression. If you state that  $N$  tends toward infinity (because  $\epsilon$  tends toward zero), you have:

$$\lim_{N \rightarrow +\infty} \frac{N!}{(N - k)!} = N^k$$

This is because  $k$  can be ignored when it is small in comparison to  $N$ . For instance, you have:

$$\frac{1,000,000!}{(1,000,000 - 3)!} = 1,000,000 \cdot 999,999 \cdot 999,998$$

which approximates  $1,000,000 \cdot 1,000,000 \cdot 1,000,000$

So you the first ratio becomes:

$$\frac{N!}{(N - k)!k!} = N^k \cdot \frac{1}{k!}$$

Then, using the fact<sup>6</sup> that  $\lim_{N \rightarrow +\infty} (1 + \frac{\lambda}{N})^N = e^\lambda$ , you have:

$$\lim_{N \rightarrow +\infty} \left(1 - \frac{\lambda}{N}\right)^N = e^{-\lambda}$$

Finally, since  $\left(1 - \frac{\lambda}{N}\right)$  tends toward 1 when  $N$  tends toward the infinity:

$$\lim_{N \rightarrow +\infty} \left(1 - \frac{\lambda}{N}\right)^{-k} = 1^{-k} = 1$$

Let's replace all of this in the formula of the binomial distribution:

---

<sup>6</sup>You can see this link for an example of proof: <https://math.stackexchange.com/questions/882741/limit-of-1-x-nn-when-n-tends-to-infinity>

$$\begin{aligned}
 & \lim_{N \rightarrow +\infty} \frac{N!}{(N-k)!k!} \left(\frac{\lambda}{N}\right)^k \left(1 - \frac{\lambda}{N}\right)^N \left(1 - \frac{\lambda}{N}\right)^{-k} \\
 &= \lim_{N \rightarrow +\infty} N^k \cdot \frac{1}{k!} \cdot \left(\frac{\lambda}{N}\right)^k \cdot e^{-\lambda} \cdot 1 \\
 &= \lim_{N \rightarrow +\infty} \frac{1}{k!} \cdot \left(N \frac{\lambda}{N}\right)^k \cdot e^{-\lambda} \\
 &= \frac{1}{k!} \cdot \lambda^k \cdot e^{-\lambda} \\
 &= \frac{\lambda^k e^{-\lambda}}{k!}
 \end{aligned}$$

This is the Poisson distribution, denoted as Poi:

$$\text{Poi}(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

for  $k = 0, 1, 2, \dots$

## 3.6 Exponential Distribution

You saw in Section 3.5 that a Poisson distribution describes the number of times an event occurs in a fixed interval of time. This number of occurrences corresponds to the parameter  $\lambda$ . A related measure, which is modeled by the *exponential distribution*, is the amount of time between two occurrences, where you still assume that the events are independent.

The probability density function of exponential distributions is mathematically expressed as:

$$\text{Exp}(x; \lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The value  $\lambda$  before the exponent tells you the starting point of the decay. You'll see that this expression can be derived from the Poisson distribution.

### 3.6.1 Derivation from the Poisson Distribution

The Poisson distribution describes the probability that the event will occur  $k$  times in one unit of time. Remember from Section 3.5 that the formula of the Poisson distribution is:

$$\text{Poi}(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

Exponential distribution, on the other hand, can describe the waiting time between events. For a specific waiting time, this can be reformulated as: "What is the probability that zero event will occur in this interval?" This corresponds to a Poisson distribution with  $k = 0$ :

$$\text{Poi}(0; \lambda) = \frac{\lambda^0 e^{-\lambda}}{0!} = e^{-\lambda}$$

So  $e^{-\lambda}$  is the probability that no event will occur in one unit of time. Since events are assumed to be independent, the probability that no event will occur in two units of time is  $e^{-\lambda} \cdot e^{-\lambda}$ . You can go further and calculate the probability that no event occurs in  $x$  unit of times, which would be  $e^{-\lambda x}$ .

This is called the *survival function* and tells you the probability that no event will occur for at least  $x$  time units. Similarly,  $1 - e^{-\lambda x}$  is the cumulative distribution function corresponding to the probability that an event will occur in at least  $x$  time units. Since probability distribution functions are derivatives of cumulative distribution function, the exponential probability distribution function is the derivative of  $1 - e^{-\lambda x}$  with respect to  $x$ , which is  $\lambda e^{-\lambda x}$ . That is the exponential distribution.

### 3.6.2 Effect of $\lambda$

Let's implement the exponential distribution function and visualize the effect of the expected number of events occurring in one unit of time  $\lambda$ :

```
def exponential(x, lambd):
    return lambd * np.exp(-lambd * x)

x_axis = np.arange(0, 3, 0.01)
plt.plot(x_axis, exponential(x_axis, lambd=1), label="$\lambda: 1$")
plt.plot(x_axis, exponential(x_axis, lambd=3), label="$\lambda: 3$")
plt.plot(x_axis, exponential(x_axis, lambd=5), label="$\lambda: 5$")
```

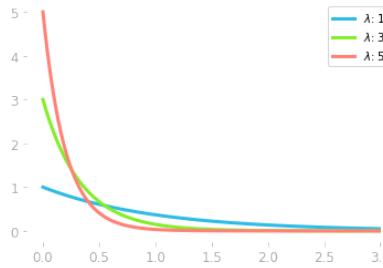


Figure 3.10: Exponential distribution for various values of  $\lambda$ .

Figure 3.10 shows the effect of the parameter  $\lambda$  on the decay of the exponential distribution: you can see that a large  $\lambda$  corresponds to a fast decay.

## 3.7 Hands-on Project: Waiting for the Bus

Jamal wants to use the exponential distribution to model how long he would wait the bus. He takes the bus timetables of the bus route he's interested in, and focuses on a 12-hour time interval (from 8 am to 8 pm). He reads that there are 48 buses in this time period, corresponding to an average of 4 buses per hour.

We'll create an array with random samples drawn from a uniform distribution to simulate the buses arriving at Jamal's stop.

```
t = 12
n_obs = 48
points_in_time = np.random.uniform(0, t, n_obs)
points_in_time

array([ 8.35763023,  3.43367202,  2.72221744, ...,  3.00546438,
       5.79641117, 11.82671743])
```

The variable `points_in_time` contains our simulation of the time of arrivals between 0 (corresponding to 8 am in our example) and 12 (8 pm), so the time unit is one hour.

Let's represent the arrival times with a scatter plot:

```
plt.scatter(points_in_time, np.repeat(1, points_in_time.shape[0]), s=20)
```

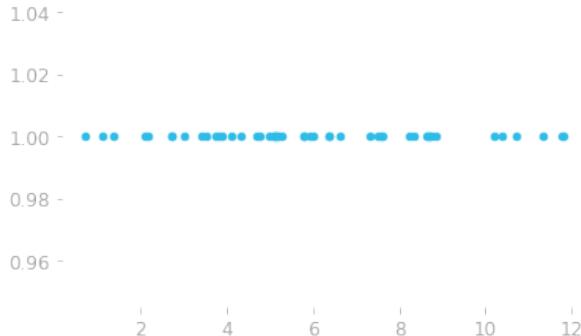


Figure 3.11: Time of arrival randomly drawn from a uniform distribution.

The time of arrival of the 48 buses between 8 am and 8 pm is illustrated in Figure 3.11.

The waiting time corresponds to the difference between two points (this is the waiting time between two buses). To model these waiting times, you need to calculate the time between each point. To do that, let's sort the array and calculate the differences between each point and the one before:

```
points_in_time = np.sort(points_in_time)
diff = points_in_time[1:] - points_in_time[:-1]
```

You can now look at the distribution of the waiting time.:

```
hist = plt.hist(diff, bins=10)
```

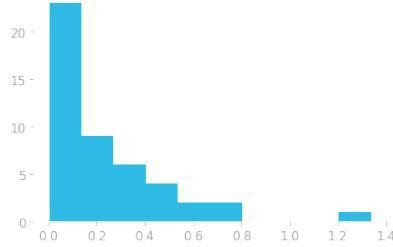


Figure 3.12: Distribution of the waiting times.

You can see in Figure 3.12 that the distribution of the waiting times is characterized by more values around 0 and a rapid decay. This shape is actually the shape of an exponential function: an exponential decay, mathematically expressed with a negative value in the exponent. It shows that an exponential function can describe the duration between occurrences of an event (with the events occurring at a constant rate and independently, as we assume for Poisson distributed events).

### More Observations

Jamal got more data and he now considers 100 days from 8 am to 8 pm (here, the data is still randomly generated with the function `np.random.uniform()`). You can use `density=True` in the `hist()` function of Matplotlib to have the density on the  $y$ -axis instead of the counts, and plot the exponential function as a line on the top of the histogram.

```
np.random.seed(123)
n_days = 100
```

```

lambd = 1

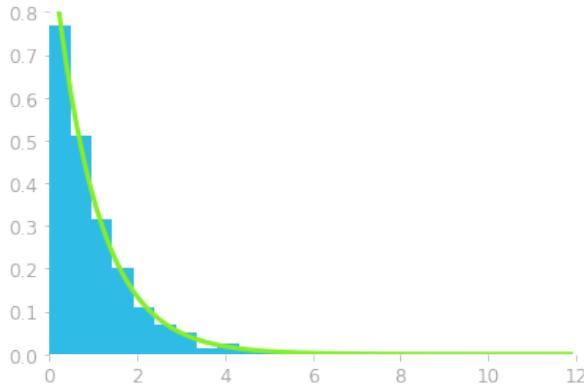
t = 12 * n_days
n_obs = int(t / lambd)

point_in_time = np.random.uniform(0, t, n_obs)
point_in_time = np.sort(point_in_time)

diff = point_in_time[1:] - point_in_time[:-1]
hist = plt.hist(diff, bins=25, density=True,
                 range=(0, t / n_days))

x_axis = np.arange(0, t / n_days, (t / n_days) / 100)
plt.plot(x_axis, lambd * np.exp(- lambd * x_axis))

plt.xlim(0, t / n_days)
plt.ylim(0, 0.8)
plt.show()
    
```



*Figure 3.13: Bus waiting time example with more observations.*

You can see in Figure 3.13 that the histogram corresponds to the exponential probability density function.

You saw in this hands-on project how exponential functions are good candidates to model events like waiting times. This concludes this chapter on common probability distributions: you learned about the uniform, the Gaussian, the Bernoulli, the binomial, the Poisson and the exponential distri-

butions. You'll need this knowledge about probability distributions in the next chapter, for instance to understand the notion of likelihood used to know the probability that some data comes from a specific distribution and parameters.



# Chapter 04

## Bayesian Statistics and Information Theory

Probability theory can be seen either with the frequentist point of view or with the Bayesian point of view. The frequentist point of view assumes that the experiment is repeated a large number of times. From the Bayesian point of view, probability is the belief that an event will occur. It expresses a degree of confidence.

This chapter will discuss how, in the context of machine learning, the Bayesian perspective is useful for quantifying the uncertainty associated with the parameters of a given model.

### 4.1 Bayes' Theorem

There are situations where your beliefs about the probability of an event can be influenced by new knowledge. This idea of incorporating evidence to update the probability of an event is developed in the *Bayes' Theorem*.

#### 4.1.1 Mathematical Formulation

As a first step, let's see how the Bayes' theorem can be used to reverse conditional probability. Let's have two random variables, X and Y. You'll see that you can use the Bayes' theorem to go from  $P(Y = y|X = x)$  to the

reverse probability  $P(X = x|Y = y)$ .

To do that, the Bayes' Theorem uses the sum rule and the product rule (as you learned respectively in Section 2.4.2.1 and Section 2.4.3.3). Let's see the derivation:

The product rule tells you that:

$$P(Y = y|X = x) = \frac{P(X = x, Y = y)}{P(X = x)} \quad (4.1)$$

Similarly, the product rule tells you that the reverse probability of  $X = x$ , given that  $Y = y$  is expressed as:

$$P(X = x|Y = y) = \frac{P(Y = y, X = x)}{P(Y = y)}$$

which can also be written as follows:

$$P(Y = y, X = x) = P(X = x|Y = y)P(Y = y) \quad (4.2)$$

In addition, note that the probability of getting  $X = x$  and  $Y = y$  ( $P(X = x, Y = y)$ ) is equivalent to the probability of getting  $Y = y$  and  $X = x$  ( $P(Y = y, X = x)$ ):

$$P(Y = y, X = x) = P(X = x, Y = y)$$

So, replacing the equation 4.2 in the equation 4.1, you get the Bayes' Theorem:

### Bayes' Theorem

$$P(Y = y|X = x) = \frac{P(X = x|Y = y)P(Y = y)}{P(X = x)} \quad (4.3)$$

The Bayes' theorem can be used to go from  $P(Y = y|X = x)$  to the reverse probability  $P(X = x|Y = y)$ .

### 4.1.2 Example

Let's return to the example from Section 2.4.3.2 (you can refer to it for the details), illustrated in Figure 2.18.

You'll use Bayes' Theorem to calculate  $P(\text{needs repair}|\text{warning light})$ , which is the probability that the car needs repair given that the warning light is on. To do that, you need the marginal probability that the warning light is on ( $P(\text{warning light})$ ) and the reverse conditional probability, that the warning light is on given that the car needs repair ( $P(\text{warning light}|\text{needs repair})$ ).

$$\begin{aligned} P(\text{needs repair}|\text{warning light}) &= \frac{P(\text{warning light}|\text{needs repair})P(\text{needs repair})}{P(\text{warning light})} \\ &= \frac{\frac{1}{4} \cdot \frac{4}{5}}{\frac{2}{5}} = \frac{\frac{4}{20}}{\frac{2}{5}} = \frac{4}{20} \cdot \frac{5}{2} = \frac{20}{40} = \frac{1}{2} \end{aligned}$$

You can check in Figure 2.18 that the probability of the event “the car needs repair” given that “the warning light is on” is indeed  $\frac{1}{2}$ .

### 4.1.3 Bayesian Interpretation

You can think of the Bayes' Theorem as a way to integrate data to update your belief of the probability of an event.

In the last example, compare  $P(\text{needs repair})$  with  $P(\text{needs repair}|\text{warning light})$ : in both cases, this concerns the probability that the car needs repair, but with  $P(\text{needs repair}|\text{warning light})$ , you don't consider the other event. You can consider it as an update of your initial belief  $P(\text{needs repair})$  with knowledge about the warning light.

#### 4.1.3.1 Prior, Posterior and Likelihood

$$\text{Posterior} = \frac{\text{Likelihood} \cdot \text{Prior}}{\text{Normalization Constant}}$$
$$P(\text{needs repair} | \text{warning light}) = \frac{P(\text{warning light} | \text{needs repair}) \cdot P(\text{needs repair})}{P(\text{warning light})}$$

Figure 4.1: Terms of the Bayes' Theorem.

As shown in Figure 4.1, the probability  $P(\text{needs repair})$  is called the *prior probability*, often shortened to just the *prior*. The prior probability is in this example your belief about the event ‘the car needs repair’ without knowing anything about the warning light.

When you have information about the warning light, you can update your belief and change your idea of the probability that the car needs repair. With the Bayesian interpretation, this probability ( $P(\text{needs repair} | \text{warning light})$ ) is called the *posterior probability* or simply the *posterior*. It corresponds to the prior updated with your knowledge about the other variable.

The reverse probability is called the *likelihood*. In the example, it tells you how likely it is that the warning light would turn on if the car needs repair ( $P(\text{warning light} | \text{needs repair})$ ).

Finally,  $P(\text{warning light})$  is a marginal probability, sometimes called the *marginal likelihood* or the *evidence*. It plays the role of a normalization constant, insuring that the posterior density integrates to 1.

#### 4.1.4 Bayes' Theorem with Distributions

In the previous section, you saw that you can incorporate data to convert prior to posterior probabilities. In this case, you manipulated events, like the event “the car needs repair”, associated with a single probability.

It is also possible to use probability distributions for priors and posteriors. This allows you to incorporate uncertainty in the process because instead of choosing a single value, you choose a distribution describing how plausible different values are.

For instance, in the previous example, let's say that the probability of the prior "the car needs repair" ( $P(\text{needs repair}) = \frac{4}{5}$ ) was a guess. You could instead use a distribution like a Gaussian distribution with a mean of  $\frac{4}{5}$ . You can choose a large value for the standard deviation of the distribution if you are uncertain of your guess.

#### 4.1.4.1 Modeling

How can Bayes' Theorem be used in the context of machine learning? Let's say that you want to find the parameters  $\boldsymbol{\theta}$  of a function to fit data samples.

Note that  $\boldsymbol{\theta}$  has a bold typeface; that's because it is a vector of parameters.<sup>1</sup>. I'll refer to the data as  $\mathbf{x}$ , also a vector containing multiple observations. Applying Bayes' Theorem, you get:

$$P(\boldsymbol{\theta}|\mathbf{x}) = \frac{P(\mathbf{x}|\boldsymbol{\theta})P(\boldsymbol{\theta})}{P(\mathbf{x})}$$

We're making initial assumptions about the parameters through the prior probability distribution  $P(\boldsymbol{\theta})$ .

Then, the prior distribution is updated by the likelihood, which is the conditional probability of observing the data given the parameters. The likelihood is mathematically expressed as  $P(\mathbf{x}|\boldsymbol{\theta})$  and is viewed as a function of  $\boldsymbol{\theta}$ . It tells you how plausible is the data for different sets of parameters. As you'll see in further details in Section 4.2, it is not a probability distribution and it doesn't necessarily satisfy the requirements of probability distributions. For instance, its integral with respect to  $\boldsymbol{\theta}$  is not necessarily equal to 1.

The left hand side of the equation is the updated distribution of  $\boldsymbol{\theta}$ , that is, the posterior distribution.

## 4.2 Likelihood

The *likelihood function* describes how likely data samples comes from a distribution with specific distribution parameters. It is a measure of how well a probability distribution fits data samples.

---

<sup>1</sup>For now, consider vectors as lists of values. You'll see more details about them in the next chapter.

The likelihood is a function of the distribution parameters. For instance, if you have some data and make the assumption that it is normally distributed, the likelihood function gives you the plausibility of the distribution as a function of the mean and standard deviation (the distribution parameters of the Gaussian function).

### 4.2.1 Introduction and Notation

#### 4.2.1.1 Notation

You might encounter a lot of different mathematical notations for likelihood functions in your readings, which can be confusing.<sup>2</sup> For clarifying this, you can refer to the summary of the notations in the Appendix appendix B.

In this book, I'll use the following notation<sup>3</sup>:

$$\mathcal{L}_x(\boldsymbol{\theta})$$

It refers to the likelihood that the data samples  $\mathbf{x}$  have the distribution with parameters  $\boldsymbol{\theta}$ .

Note that, in the context of the Bayes' Theorem, the notation  $p(\mathbf{x}|\boldsymbol{\theta})$  is generally used. This is because the parameter  $\boldsymbol{\theta}$  are actually random variables (as you'll see in Section 4.2.3.3).

#### 4.2.1.2 What's the Difference Between Probability and Likelihood?

In statistics, the terms *likelihood* and *probability* refer to different things.

Probability expresses uncertainty about the data when you know the distribution, while likelihood expresses uncertainty about the distribution when you have some data.

---

<sup>2</sup>For instance, see this: <https://stats.stackexchange.com/questions/284816/why-do-people-use-mathcall-thetax-for-likelihood-instead-of-px-theta>. You can also read the following post to have more details about likelihood and conditional probabilities: <https://stats.stackexchange.com/questions/224037/wikipedia-entry-on-likelihood-seems-ambiguous>

<sup>3</sup>I follow here the notation used in Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong. Mathematics for machine learning. Cambridge University Press, 2020. See p.266.

Look at Figure 4.2.

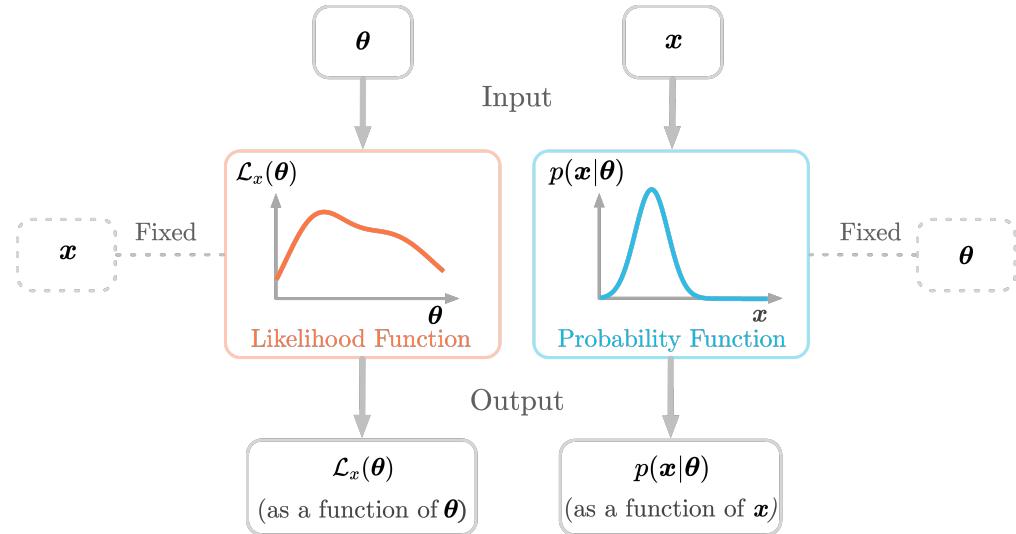


Figure 4.2: Comparison of probability and likelihood.

Figure 4.2 summarizes the differences between probability functions and likelihood functions.

Probability functions (illustrated in blue) describe a probability mass (for discrete random variables) or a probability density (for continuous random variables) as a function of the possible outcomes of a random variable X.

Thus, probability functions take possible outcomes of X (the values  $x$ ) as input and return a probability mass or density according to fixed distribution parameters  $\theta$ .

For instance, if your random variable describes apartment prices, the probability function will give you the probability mass or density to see an apartment, or a set of apartments, with specific prices  $x$ , according to the parameters of the distribution.

Likelihood functions (illustrated in red) correspond to how well a distribution fits the observations. It is a function of the distribution parameters (the inputs of the function are the parameters  $\theta$ ) and gives you the likelihood for a fixed set of observations. Still with the apartment example, you would have

a set of apartment prices and you would ask: “How likely it is to observe these apartment prices if the true distribution had the parameters  $\theta$ ?”.<sup>4</sup>

### 4.2.2 Finding the Parameters of the Distribution

You can use the likelihood function to find the parameters of a distribution that fit the data well. Let’s see how it works.

#### 4.2.2.1 Likelihood from a Single Data Sample

Let’s start with a single data sample. Assuming that this data sample comes from a Gaussian distribution, we’ll compare different values of  $\mu$  with a fixed  $\sigma = 0.1$ .

```
def gaussian(x, mu, sigma):
    return (1 / np.sqrt(2 * np.pi * sigma ** 2)) * np.exp(-(1 / (2 * sigma
        ** 2)) * (x - mu) ** 2)

x_axis = np.arange(0, 1, 0.01)
random_points = np.random.normal(0.3, 0.1, 10)

sigma = 0.1
#[...] Plot the first data sample and the gaussian distribution...
```

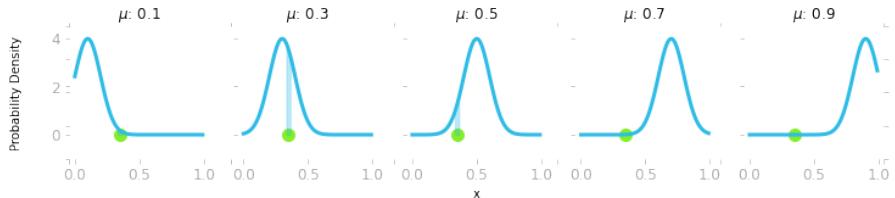


Figure 4.3: Probability density function for different  $\mu$  and  $\sigma = 0.1$  in blue and a single data sample ( $x \approx 0.35$ ) in green.

Figure 4.3 show distributions using various parameters  $\mu$ . With the single data

---

<sup>4</sup>For more details about the difference between probability and likelihood, I recommend the following link: <https://stats.stackexchange.com/questions/2641/what-is-the-difference-between-likelihood-and-probability>.

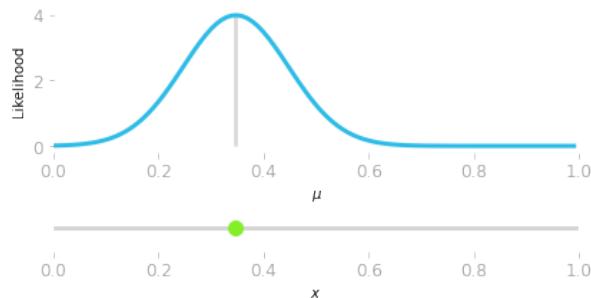
sample (in green), what value of  $\mu$  corresponds to the most likely distribution?

If you look at the first distribution (left panel,  $\mu = 0.1$ ), you can note that the probability of getting a value around our data sample is quite low (density around 0.19). In comparison, the second plot ( $\mu = 0.3$ ) shows that there is a high probability density (around 3.57) of getting a value corresponding to the data sample.

This means that, when there is a single data sample, the likelihood of one distribution corresponds to the probability density of the data sample for a specific set of parameters.

Let's calculate the likelihood for various values of  $\mu$  and plot what you obtain:

```
# Calculate densities for various mu
all_densities = []
for mu in x_axis:
    all_densities.append(gaussian(random_points[0], mu, sigma))
# [...] Plot `all_densities` and `random_points`
```



*Figure 4.4: The likelihood function. The likelihood of obtaining our single data sample (in green) as a function of  $\mu$ . The more plausible value for  $\mu$  is around 0.35 (which corresponds to the value of the data sample).*

Figure 4.4 represents the likelihood for each value of  $\mu$ . As you might expect, the distribution with a mean corresponding to the value of the data sample (0.35) is the more likely to generate the data sample.

In the next section, you'll see how to calculate the likelihood with multiple data samples.

#### 4.2.2.2 More Data Samples

What should you do if you have more data samples?

Let's consider an example in which the data samples are independently drawn from the same distribution. The value of one data sample doesn't depend on the values of the other data samples: we say that the data samples are *independent and identically distributed* (IID).

Independence implies that the probability to observe all data samples corresponds to the joint probability of observing the data samples separately. Remember from Section 2.4.1 that you can calculate the joint probability of two independent events by multiplying their individual probabilities:  $P(x, y) = P(x)P(y)$ .

You can thus calculate the probability of observing all data samples by multiplying the probabilities to observe  $N$  individual samples. You can write:

$$\mathcal{L}_x(\mu, \sigma^2) = \prod_{n=1}^N \mathcal{N}(x_n; \mu, \sigma^2) \quad (4.4)$$

This is the likelihood function. The symbol  $\Pi$  (the Greek letter “capital Pi”) is called the *product notation* and denotes a repeated product. It is the equivalent of  $\Sigma$  for product. As you saw in Section 3.2,  $\mathcal{N}$  refers to the Gaussian distribution, parameterized by  $\mu$  and  $\sigma^2$ .

You can understand equation Section 4.4.2.2 as follows: for each data sample  $x_n$  from the first ( $x_1$ ) to the  $N$ th ( $x_N$ ), you calculate the density using a Gaussian distribution that have a mean  $\mu$  and a variance  $\sigma^2$ . You multiply each of these densities together to get the likelihood for these values of  $\mu$  and  $\sigma^2$ . As you saw earlier, the likelihood is a function of the distribution parameters: you give distribution parameters as input and it returns the likelihood that the data samples are coming from a distribution with these parameters.

You can use this formula to calculate the likelihood to observe the data samples for specific distribution parameters (here,  $\mu$  and  $\sigma$ ).

Let's calculate the likelihood using 10 data samples. As shown in the equation, the likelihood of getting a specific distribution given that you observed these data samples is calculated as the product of all probability densities of  $x$  for each value of  $\mu$  (still with  $\sigma = 0.1$ ). Let's see what you obtain.

```
# Calculate densities for various mu
all_densities = []
for mu in x_axis:
    density = 1
    for i in range(n_points):
        density *= gaussian(random_points[i], mu, sigma)
    all_densities.append(density)
# [...] Plot the likelihood function
```

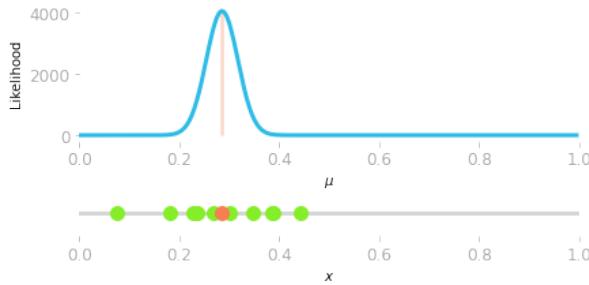


Figure 4.5: Likelihood function (in blue) corresponding to 10 data samples (in green) as a function of  $\mu$ .

You can see in Figure 4.5 that many data samples are represented. It shows that the best value of  $\mu$  changes when you add data samples. Note that the maximum likelihood value corresponds to the mean of the data samples.

### 4.2.3 Maximum Likelihood Estimation

*Maximum likelihood estimation* (MLE) is aimed at finding the parameters of a distribution that fits the data well, which is the maximum value of the likelihood function. For instance, it can be used to optimize a model's parameters in a supervised learning procedure where the targets are considered as conditional probabilities given the data.

#### 4.2.3.1 Negative Log-Likelihood

You saw in the previous section that the likelihood is calculated by multiplying the probability densities of each independent data sample. The issue is that, with a lot of data samples, we end up multiplying a large number of small probabilities. This can lead to a loss of numerical precision.<sup>5</sup>

Fortunately, there is a way to simplify this calculation. Logarithm functions have the property to convert products into sums ( $\log(xy) = \log(x) + \log(y)$ ). This is why it is easier to calculate the logarithm of the likelihood. For instance, to calculate the likelihood of two data samples, you can do:

$$\mathcal{L}_x(\mu, \sigma^2) = \mathcal{N}(x_0; \mu, \sigma^2) \cdot \mathcal{N}(x_1; \mu, \sigma^2)$$

Where  $\mathbf{x}$  is a vector containing all the data samples and  $x_0$  and  $x_1$  are individual data samples.

Using the logarithm, you can do:

$$\log(\mathcal{L}_x(\mu, \sigma^2)) = \log(\mathcal{N}(x_0; \mu, \sigma^2)) + \log(\mathcal{N}(x_1; \mu, \sigma^2))$$

For this reason, maximizing the log of the likelihood function is easier.

Note that this is possible only because the log function is *monotonically increasing*: that is, the value of  $\boldsymbol{\theta}$  maximizing  $\mathcal{L}_x(\boldsymbol{\theta})$  also maximizes  $\log(\mathcal{L}_x(\boldsymbol{\theta}))$ .

In addition, we generally use the negative log likelihood, because optimizers usually minimize functions. Minimizing the negative log likelihood is equivalent to maximizing the log likelihood.

#### 4.2.3.2 Derivative of the Log-Likelihood

One way to find the minimum value of the negative log likelihood function is to differentiate it with respect to the parameters you're trying to optimize (as you saw in Section 1.3).

---

<sup>5</sup>Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006., p.26.

With the maximum likelihood for the Gaussian distribution, you're trying to estimate the best values for  $\mu$  and  $\sigma^2$ . The solution<sup>6</sup> happens to be the sample mean and the sample variance.<sup>7</sup>

#### 4.2.3.3 Maximum A Posteriori Estimation

In some situations, you have hypotheses about the distribution of the parameters you're optimizing. Leveraging the Bayesian approach, you can use prior knowledge to modify the choice of the maximum point in the likelihood function using the *maximum a posteriori estimation* (MAP). The Latin term *a posteriori* refer to knowledge that use empirical evidence. This contrasts with maximum likelihood, which is a frequentist method.

In this context, the influence of the prior can be seen as regularization<sup>8</sup> (for more details about regularization, you can refer to the hands-on project in Section 5.5).

MAP uses a prior probability distribution of the parameters you're trying to optimize ( $\boldsymbol{\theta}$ ). You want to update the distribution of  $\boldsymbol{\theta}$  using your data  $\mathbf{x}$ . This is the posterior distribution and Bayes' Theorem can be used to calculate it. The prior distribution is  $p(\boldsymbol{\theta})$  and  $p(\mathbf{x}|\boldsymbol{\theta})$  is the likelihood. you have:

$$p(\boldsymbol{\theta}|\mathbf{x}) = \frac{p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{x})}$$

Note that in the context of Bayes' Theorem,  $\boldsymbol{\theta}$  and  $\mathbf{x}$  are both random variables, thus it makes sense to refer to the likelihood with  $p(\mathbf{x}|\boldsymbol{\theta})$ .<sup>9</sup>

Since the purpose of MAP is to find the best value of  $\boldsymbol{\theta}$  (associated with the maximum of the posterior distribution), you can remove  $p(\mathbf{x})$ . The same value of  $\boldsymbol{\theta}$  maximizes  $\frac{p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{x})}$  and  $p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})$ .

---

<sup>6</sup>If you want details on the calculation of the derivative of the negative log likelihood for the Gaussian distribution, you can read <http://jrmeyer.github.io/machinelearning/2017/08/18/mle.html>

<sup>7</sup>Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006., p.27

<sup>8</sup>More details in Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016., p.139

<sup>9</sup>More details in this thread: <https://stats.stackexchange.com/a/224418/42330>

To write the proportional relationship between the two statements, you use the  $\propto$  symbol, which means “proportional to”. You can’t use the equal sign anymore, since you removed  $p(\mathbf{x})$ :

$$p(\boldsymbol{\theta}|\mathbf{x}) \propto p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})$$

The standard deviation of the prior distribution affects the weight given to it: a large standard deviation means that you don’t have much confidence into the prior and that it is given less weight.

## 4.3 Information Theory

The field of *information theory* studies the quantification of information in signals. In the context of machine learning, some of these concepts are used to characterize or compare probability distributions. The ability to quantify information is also used in the decision tree algorithm, to select the variables associated with the maximum information gain. The concepts of entropy and cross-entropy are also important in machine learning because they lead to a widely used loss function in classification tasks: the cross-entropy loss or log loss.

### 4.3.1 Shannon Information

#### 4.3.1.1 Intuition

The first step to understanding information theory is to consider the concept of the quantity of information associated with a random variable. In information theory, this quantity of information is denoted as  $I$  and is called the *Shannon information*, *information content*, *self-information*, or *surprisal*. The main idea is that likely events convey less information than unlikely events (which are thus more *surprising*). For instance, if a friend from Los Angeles, California tells you: “It is sunny today”, this is less informative than if she tells you: “It is raining today”. For this reason, it can be helpful to think of the Shannon information as the amount of surprise associated with an outcome. You’ll also see in this section why it is also a quantity of information, and why likely events are associated with less information.

### 4.3.1.2 Units of Information

Common units to quantify information are the *nat* and the *bit*. These quantities are based on logarithm functions. The word *nat*, short for *natural unit of information* is based on the natural logarithm, while the bit, short for “binary digit”, is based on base-two logarithms. The bit is thus a rescaled version of the nat. The following sections will mainly use the bit and base-two logarithms in formulas, but replacing it with the natural logarithm would just change the unit from bits to nats.

Bits represent variables that can take two different states (0 or 1). For instance, 1 bit is needed to encode the outcome of a coin flip. If you flip two coins, you’ll need at least two bits to encode the result. For instance, 00 for HH, 01 for HT, 10 for TH, and 11 for TT. You could use other codes, such as 0 for HH, 100 for HT, 101 for TH, and 111 for TT. However, this code uses a larger number of bits on average (considering that the probability distribution of the four events is uniform, as you’ll see)

Let’s take an example to see what a bit describes. Erica sends you a message containing the result of three coin flips, encoding ‘heads’ as 0 and ‘tails’ as 1. There are 8 possible sequences, such as 001, 101, etc. When you receive a message of one bit, it divides your uncertainty by a factor of 2. For instance, if the first bit tells you that the first roll was ‘heads’, the remaining possible sequences are 000, 001, 010, and 011. There are only 4 possible sequences instead of 8. Similarly, receiving a message of two bits will divide your uncertainty by a factor of  $2^2$ ; a message of three bits, by a factor of  $2^3$ , and so on.

Note that we talk about “useful information”, but it is possible that the message is redundant and convey less information with the same number of bits.

### 4.3.1.3 Example

Let’s say that we want to transmit the result of a sequence of eight tosses. You’ll allocate one bit per toss. You thus need eight bits to encode the sequence. The sequence might be for instance “00110110”, corresponding to HHTTHHTH (four “heads” and four “tails”).

However, let’s say that the coin is biased: the chance to get “tails” is only 1

over 8. You can find a better way to encode the sequence. One option is to encode the index of the outcomes “tails”: it will take more than one bit, but ‘tails’ occurs only for a small proportion of the trials. With this strategy, you allocate more bits to rare outcomes.

This example illustrates that more predictable information can be compressed: a biased coin sequence can be encoded with a smaller amount of information than a fair coin. This means that Shannon information depends on the probability of the event.

#### 4.3.1.4 Mathematical Description

Shannon information encodes this idea and converts the probability that an event will occur into the associated quantity of information. Its characteristics are that, as you saw, likely events are less informative than unlikely events and also that information from different events is additive (if the events are independent).

Mathematically, the function  $I(x)$  is the information of the event  $X = x$  that takes the outcome as input and returns the quantity of information. It is a monotonically decreasing function of the probability (that is, a function that never increases when the probability increases). Shannon information is described as:

$$I(x) = -\log_2 P(x) \quad (4.5)$$

The result is a lower bound on the number of bits, that is, the minimum amount of bits needed to encode a sequence with an optimal encoding.

The logarithm of a product is equal to the sum of the elements:  $\log_2(ab) = \log_2(a) + \log_2(b)$ . This property is useful to encode the additive property of the Shannon information. The probability of occurrence of two events is their individual probabilities multiplied together (because they are independent, as you saw in Section 2.4.1.2):

$$I(x, y) = -\log_2 P(x, y) = -(\log_2 P(x) + \log_2 P(y))$$

This means that the information corresponding to the probability of occurrence of two events  $P(x, y)$  equals the information corresponding to  $P(x)$  added

to the information corresponding to  $P(y)$ . The information of independent events adds together.

Let's plot this function for a range of probability between 0 and 1 to see the shape of the curve:

```
plt.plot(np.arange(0.01, 1, 0.01), -np.log2(np.arange(0.01, 1, 0.01)))
```

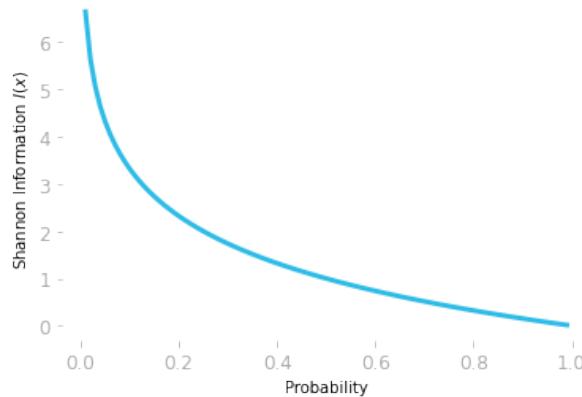


Figure 4.6: The quantity of information is given by the negative logarithm of the probability.

As you can see in Figure 4.6, the negative logarithm function encodes the idea that a very unlikely event (probability around 0) is associated with a large quantity of information and a likely event (probability around 1) is associated with a quantity of information around 0.

Since you used a base-two logarithm `np.log2()`, the information  $I(x)$  is measured in *bits*.

### 4.3.2 Entropy

You saw that Shannon information gives the amount of information associated with a single probability. You can also calculate the amount of information of a discrete distribution with the *Shannon entropy*, also called *information entropy*, or simply *entropy*.

#### 4.3.2.1 Example

Consider for instance a biased coin, where you have a probability of 0.8 of getting ‘heads’.

1. Here is your distribution: you have a probability of 0.8 of getting ‘heads’ and a probability of  $1 - 0.8 = 0.2$  of getting ‘tails’.
2. These probabilities are respectively associated with a Shannon information of:

$$-\log_2 0.8 \approx 0.32$$

and

$$-\log_2 0.2 = 2.32$$

3. Landing ‘heads’ is associated with an information around 0.32 and landing ‘tails’ to 2.32. However, you don’t have the same number of ‘heads’ and ‘tails’ in average, so you must weight the Shannon information of each probability with the probability itself. For instance, if you want to transmit a message with the results of, say, 100 trials, you’ll need around 20 times the amount of information corresponding to ‘tails’ and 80 times the amount of information corresponding to ‘heads’. You get:

$$0.8 \cdot (-\log_2 0.8) \approx 0.26$$

and

$$0.2 \cdot (-\log_2 0.2) = 0.46$$

4. The sum of these expressions gives you:

$$0.8 \cdot (-\log_2 0.8) + 0.2 \cdot (-\log_2 0.2) = 0.26 + 0.46 = 0.72$$

The average number of bits required to describe a series of events from this distribution is 0.72 bits.

To summarize, you can consider the entropy as a summary of the information associated with the probabilities of the discrete distribution:

1. You calculate the Shannon information of each probability of your distribution.
2. You weight the Shannon information with the corresponding probability.
3. You sum the weighted results.

#### 4.3.2.2 Mathematical Formulation

The entropy is the expectation of the information with respect to the probability distribution. Remember from Section 2.6 that the expectation is the mean value you'll get if you draw a large number of samples from the distribution:

$$\mathbb{E}[X] = \sum_{i=1}^n P(x_i)x_i \quad (4.6)$$

with the random variable  $X$  having  $n$  possible outcomes,  $x_i$  being the  $i$ th possible outcome corresponding to a probability of  $P(x_i)$ . The expected value of the information of a distribution corresponds to the average of the information you'll get.

Following the formula of the expectation (equation 4.6) and the Shannon information (equation 4.5), the entropy of the random variable  $X$  is defined as:

$$H(X) = \mathbb{E}[I(x)] = - \sum_x P(x) \log_2 P(x)$$

The entropy gives you the average quantity of information that you need to encode the states of the random variable  $X$ .

Note that the input of the function  $H(X)$  is the random variable  $X$  while  $I(x)$  denotes the Shannon information of the event  $X = x$ . You can also refer to the entropy of the random variable  $X$  which is distributed with respect to  $P(x)$  as  $H(P)$ .

### 4.3.2.3 Illustration

Let's take an example: as illustrated in Figure 4.7 in the bottom panel, you have a discrete distribution with four possible outcomes, associated with probabilities 0.4, 0.4, 0.1, and 0.1, respectively. As you saw previously, the information is obtained by log transforming the probabilities (top panel). This is the last part of the entropy formula:  $\log_2 P(x)$ .

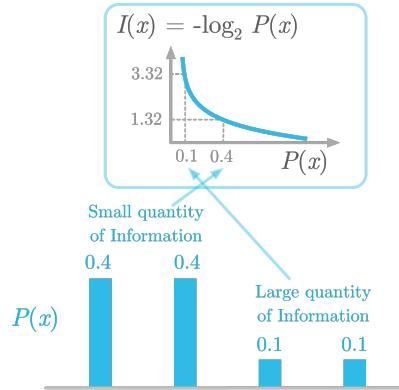


Figure 4.7: Illustration of the entropy as the weighted sum of the Shannon information.

Each of these transformed probabilities is weighted by the corresponding raw probability. If an outcome occurs frequently, it will give more weight into the entropy of the distribution. This means that a low probability (like 0.1 in Figure 4.7) gives a large amount of information (3.32 bits) but has less influence on the final result. A larger probability (like 0.4 in Figure 4.7) is associated with less information (1.32 bits as shown in Figure 4.7) but has more weight.

### 4.3.2.4 Binary Entropy Function

In the example of a biased coin, you calculated the entropy of a Bernoulli process (more details about the Bernoulli distribution in Section 3.3). In this special case, the entropy is called the *binary entropy function*.

To characterize the binary entropy function, you'll calculate the entropy of a biased coin described by various probability distributions (from heavily biased in favor of “tails” to heavily biased in favor of “heads”).

Let's start by creating a function to calculate the entropy of a distribution that takes an array with the probabilities as input and returns the corresponding entropy:

```
def entropy(P):
    return - np.sum(P * np.log2(P))
```

You can also use `entropy()` from `scipy.stats`, where you can specify the base of the logarithm used to calculate the entropy. Here, I have used the base-two logarithm.

Let's take the example of a fair coin, with a probability of 0.5 of landing 'heads'. The distribution is thus 0.5 and  $1 - 0.5 = 0.5$ . Let's use the function we just defined to calculate the corresponding entropy is:

```
p = 0.5
entropy(np.array([p, 1 - p]))
```

1.0

The function calculates the sum of `P * np.log2(P)` over each element of the array that you use as input. Using an array as input As you saw in the previous section, you can expect a lower entropy for a biased coin. Let's plot the entropy for various coin biases, from a coin landing only as 'tails' to a coin landing only as 'heads':

```
x_axis = np.arange(0.01, 1, 0.01)
entropy_all = []
for p in x_axis:
    entropy_all.append(entropy([p, 1 - p]))

# [...] plots the entropy
```

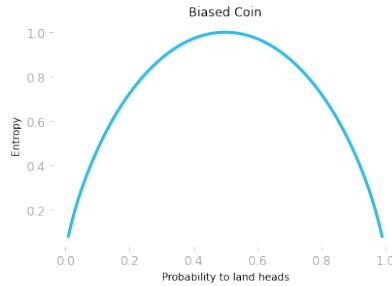


Figure 4.8: Entropy as a function of the probability to land “heads”.

Figure 4.8 shows that the entropy increases until you reach the more uncertain condition: that is, when the probability of landing ‘heads’ equals the probability of landing ‘tails’.

#### 4.3.2.5 Differential Entropy

The entropy of a continuous distribution is called *differential entropy*. It is an extension of the entropy for discrete distribution, but it doesn’t satisfy the same requirements. The issue is that values have probability tending to zero with continuous distributions, and encoding this would require a number of bits tending to infinity.

It is defined as:

$$H(P) = - \int p(x) \log_2 p(x) dx$$

Differential entropy can be negative. The reason is that, as you saw in Section 2.3.2, continuous distributions are not probabilities but probability densities, meaning that they don’t satisfy the requirements of probabilities. For instance, they are not constrained to be lower than 1. This has the consequence that  $p(x)$  can take positive values larger than 1 and  $\log_2 p(x)$  can take positive values (leading to negative values because of the negative sign).<sup>10</sup>

---

<sup>10</sup>You can find more details on differential entropy in Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006., p.53.

### 4.3.3 Cross-Entropy

The concept of entropy can be used to compare two probability distributions: this is called the *cross-entropy* between two distributions, which measures how much they differ.

The idea is to calculate the information associated with the probabilities of a distribution  $Q(x)$ , but instead of weighting according to  $Q(x)$  as with the entropy, you weight according to the other distribution  $P(x)$ . Note that you compare two distributions concerning the same random variable X.

You can also consider cross-entropy as the expected quantity of information of events drawn from  $P(x)$  when you use  $Q(x)$  to encode them.

This is mathematically expressed as:

$$H(P, Q) = - \sum_x P(x) \log_2 Q(x)$$

Let's see how it works.

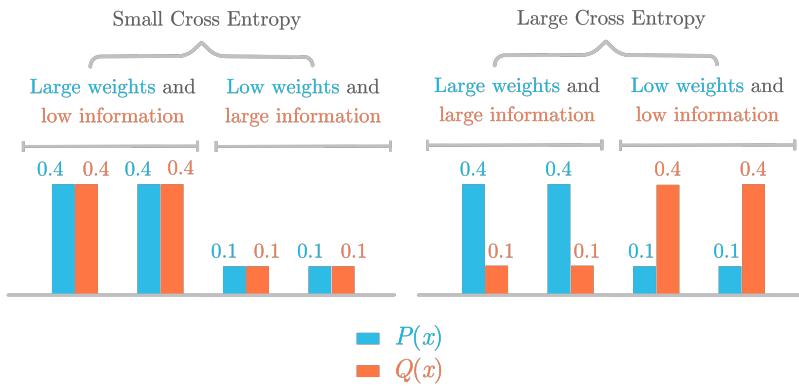


Figure 4.9: Illustration of the cross-entropy as the Shannon information of  $Q(x)$  weighted according to the distribution of  $P(x)$ .

Figure 4.9 shows two different situations to illustrate the cross-entropy. On the left, you have two identical distributions  $P(x)$  (in blue) and  $Q(x)$  (in red). Their cross-entropy is equal to the entropy because the information of  $Q(x)$  is weighted according to the distribution of  $P(x)$ , which is similar to  $Q(x)$ .

However, in the right panel,  $P(x)$  and  $Q(x)$  are different. This results in a larger cross-entropy, because probabilities associated with a large quantity of information have a small weight, while probabilities associated with a small quantity of information have large weights.

The cross-entropy can't be smaller than the entropy. Still in the right panel, you can see that, when the probability  $Q(x)$  is larger than  $P(x)$  (and thus associated with a lower amount of information), it is counterbalanced by the low weights (resulting in low weights and low information). These low weights will be compensated with larger weights in other probabilities from the distribution (resulting in large weights and large information).

Note also that both distributions  $P(x)$  and  $Q(x)$  must have the same *support* (that is the same set of values that the random variable can take associated with positive probabilities).

To summarize, the cross-entropy is minimum when the distributions are identical. As you'll see in Section 4.3.4, this property makes the cross-entropy a useful metric. Note also that the result is different according to the distribution you choose as a reference:  $H(P, Q) \neq H(Q, P)$ .

#### 4.3.3.1 Cross-Entropy as a Loss Function

In machine learning, cross-entropy is used as a loss function called the *cross-entropy loss* (also called the *log loss*, or the *logistic loss*, because it is used in logistic regression).

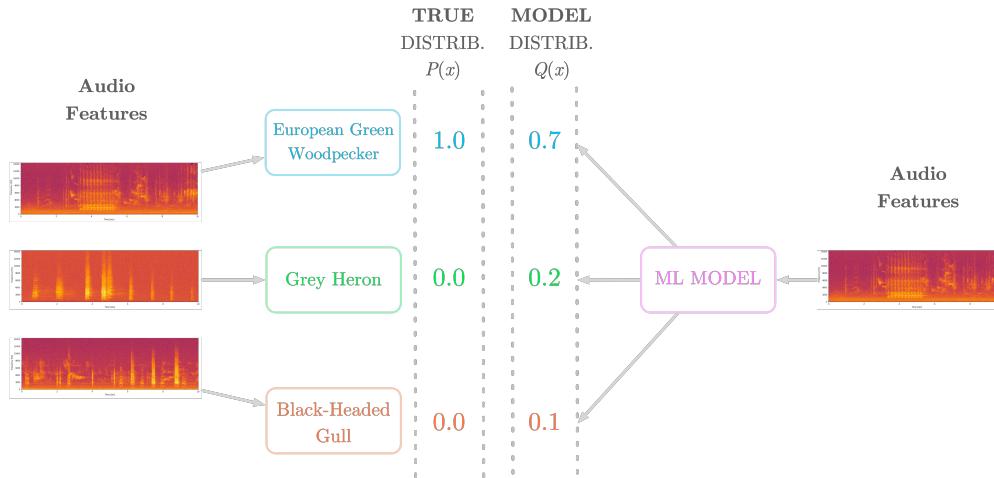


Figure 4.10: Cross-entropy can be used to compare the true distribution (probability of 1 for the correct class and 0 otherwise) and the distribution estimated by the model.

Say you want to build a model that classifies three different bird species from audio samples. As illustrated in Figure 4.10, the audio samples are converted in features (here spectrograms) and the possible classes (the three different birds) are *one-hot encoded*, that is, encoded as 1 for the correct class and 0 otherwise. Furthermore, the machine learning model outputs probabilities for each class.

To learn how to classify the birds, the model needs to compare the estimated distribution  $Q(x)$  (given by the model) and the true distribution  $P(x)$ . The cross-entropy loss is computed as the cross-entropy between  $P(x)$  and  $Q(x)$ .

Figure 4.10 shows that the true class corresponding to the sample you consider in this example is “European Green Woodpecker”. The model outputs a probability distribution and you’ll compute the cross-entropy loss associated with this estimation. Figure 4.11 shows both distributions.

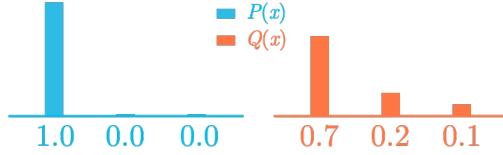


Figure 4.11: Comparison of the true distribution  $P(x)$  and the estimated distribution  $Q(x)$ .

Let's manually calculate the cross-entropy between these two distributions:

$$\begin{aligned} H(P, Q) &= - \sum_x P(x) \log Q(x) \\ &= -(1.0 \log 0.7 + 0.0 \log 0.2 + 0.0 \log 0.1) \\ &= -\log 0.7 \end{aligned}$$

The natural logarithm is used in the cross-entropy loss instead of the base-two logarithm, but the principle is the same. In addition, note the use of  $H(P, Q)$  instead of  $H(Q, P)$  because the reference is the distribution  $P$ .

Since you one-hot encoded the classes (1 for the true class and 0 otherwise), the cross-entropy is simply the negative logarithm of the estimated probability for the true class.

#### 4.3.3.2 Binary Classification: Log Loss

In machine learning, the cross-entropy is widely used as a loss for binary classification: the log loss.

Since the classification is binary, the only possible outcomes are  $y$  (the true label corresponds to the first class) and  $1 - y$  (the true label corresponds to the second class). Similarly, you have the estimated probability of the first class  $\hat{y}$  and the estimated probability of the second class  $1 - \hat{y}$ .

From the formula of the cross-entropy,  $\sum_x$  corresponds here to the sum over the two possible outcomes ( $y$  and  $1 - y$ ). You have:

$$\begin{aligned} H(P, Q) &= - \sum_x P(x) \log Q(x) \\ &= -(y \log(\hat{y}) + (1-y) \log(1-\hat{y})) \end{aligned}$$

which is the formula of the log loss.

#### 4.3.4 Kullback-Leibler Divergence (KL Divergence)

You saw that the cross-entropy is a value that depends on the similarity of two distributions, with the smaller cross-entropy value corresponding to identical distributions. You can use this property to calculate the *divergence* between two distributions: you compare their cross-entropy with the situation where the distributions are identical. This divergence is called the *Kullback-Leibler divergence* (or simply the *KL divergence*), or the *relative entropy*.

Intuitively, the KL divergence is the supplemental amount of information associated with the encoding of the distribution  $Q(x)$  compared to the true distribution  $P(x)$ . It tells you how different the two distributions are.

Mathematically, the KL divergence between two distributions  $P(x)$  and  $Q(x)$ , denoted as  $D_{KL}(P||Q)$ , is expressed as the difference between the cross-entropy of  $P(x)$  and  $Q(x)$  and the entropy of  $P(x)$ :

$$D_{KL}(P||Q) = H(P, Q) - H(P) \geq 0$$

Replacing with the expressions of the cross-entropy and the entropy, you get:

$$\begin{aligned} D_{KL}(P||Q) &= H(P, Q) - H(P) \\ &= - \sum_x P(x) \log_2 Q(x) - \left( - \sum_x P(x) \log_2 P(x) \right) \\ &= \sum_x P(x) \log_2 P(x) - \sum_x P(x) \log_2 Q(x) \end{aligned}$$

The KL divergence is always non-negative. Since the entropy  $H(P)$  is identical to the cross-entropy  $H(P, P)$ , and because the smallest cross-entropy

is between identical distributions ( $H(P, P)$ ),  $H(P, Q)$  is necessarily larger than  $H(P)$ . In addition, the KL divergence is equal to zero when the two distributions are identical.

However, the cross-entropy is not symmetrical. Comparing a distribution  $P(x)$  to a distribution  $Q(x)$  can be different than comparing a distribution  $Q(x)$  to  $P(x)$  – which implies that you can't consider the KL divergence to be a distance.

## 4.4 Hands-On Project: Bayesian Inference

*Statistical inference* is the process of finding the probability distribution of the population from which the data samples were drawn. This means that you want to know about the population from the data samples. *Bayesian inference* is statistical inference using Bayes' Theorem.

### 4.4.1 Bayesian Inference

As you saw in Section 4.2.3, you can use Maximum Likelihood Estimation (MLE) or Maximum A Posteriori estimation (MAP) to estimate the parameters of a distribution. With Bayesian inference, you estimates these parameters using Bayes' Theorem. The parameters are described through a distribution instead of a *point estimate* (the single maximum value) with MLE and MAP.<sup>11</sup> The posterior distribution is useful in describing the uncertainty associated with the estimation.

As with MAP, a prior distribution is used to change the distribution of the estimated parameters with elements from outside of the data. You want to calculate the posterior probability from the likelihood and the prior distribution. The posterior is obtained using Bayes' Theorem (see Section 4.1.4):

$$P(\boldsymbol{\theta}|\mathbf{x}) = \frac{P(\mathbf{x}|\boldsymbol{\theta})P(\boldsymbol{\theta})}{P(\mathbf{x})}$$

with  $\boldsymbol{\theta}$  the vector of parameters,  $\mathbf{x}$  the data samples,  $P(\boldsymbol{\theta}|\mathbf{x})$  being the posterior distribution,  $P(\mathbf{x}|\boldsymbol{\theta})$  the likelihood,  $P(\boldsymbol{\theta})$  the prior distribution,

---

<sup>11</sup>You can also calculate a standard error in addition to the point estimate with MLE and MAP, giving you a distribution as well.

and  $P(\mathbf{x})$  the marginal distribution. While the posterior can be calculated analytically, this is not always possible.<sup>12</sup>

### 4.4.2 Project

Let's take a practical example of Bayesian inference and visualize the effect of the prior distribution.<sup>13</sup> You'll generate some data samples from a known normal distribution. You'll estimate the parameters of this distribution (the mean  $\mu$  and the standard deviation  $\sigma$ , since it is a normal distribution) through Bayesian inference.

To simplify the example, let's assume that you already know that the true standard deviation  $\sigma_{\text{true}}$  is equal to 0.1. So the goal here is to estimate only the true mean.

Let's start by simulating some data:

```
n_obs = 10
sigma_true = 0.1
mu_true = 0.2

x_data = np.random.normal(mu_true, sigma_true, n_obs)
```

#### 4.4.2.1 Prior

Now, you can choose a prior distribution for  $\mu$  that encodes your best guess and your uncertainty about its value.

$$p(\mu) = \mathcal{N}(\mu; \mu_{\text{prior}}, \sigma_{\text{prior}}^2)$$

It is important to understand that this prior distribution is a function of  $\mu$ . You give a value of the parameter you want to test with Bayesian inference and

---

<sup>12</sup>When it is not, it still can be approximated with a method called Markov Chain Monte-Carlo, or MCMC. For more details, see for instance: Ravenzwaaij, Don, Pete Cassey, and Scott D. Brown. “A simple introduction to Markov Chain Monte–Carlo sampling.” *Psychonomic bulletin & review* 25.1 (2018): 143-154. It gives a nice introduction of MCMC

<sup>13</sup>This example is a practical illustration of Bayesian inference, inspired by the theoretical section of Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006., p.97. You can refer to it for further details.

it returns a probability density. The value  $\mu_{\text{prior}}$  and  $\sigma_{\text{prior}}^2$  are the parameters of the prior distribution, which represents your initial beliefs.

Let's choose a prior distribution with a mean 0.8 and standard deviation 0.1. You can try with other parameters, but this one is not too close to the true distribution, which is nice to visualize the effect on the posterior distribution.

```
mu_prior = 0.8
sigma_prior = 0.1
```

Now, you can create the prior using the Gaussian function as defined above:

```
def gaussian(x, mu, sigma):
    return (1 / np.sqrt(2 * np.pi * sigma ** 2)) *\
        np.exp(-(1 / (2 * sigma ** 2)) * (x - mu) ** 2)

x_axis = np.arange(0, 1.2, 0.01)
prior_y = gaussian(x_axis, mu_prior, sigma_prior)
```

Let's plot the prior:

```
plt.plot(x_axis, prior_y, label='prior')
# [...] Add legend and label
```

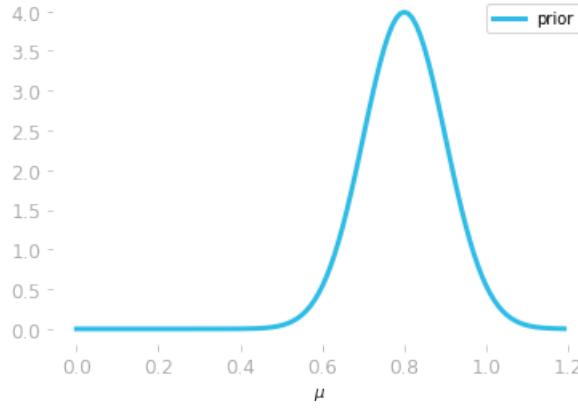


Figure 4.12: The prior distribution is a Gaussian distribution with mean 0.8 and standard deviation 0.1.

#### 4.4.2.2 Likelihood

The next step is to update the prior using the likelihood to calculate the posterior distribution. As you saw in Section 4.2.2.2, the likelihood is defined as:

$$\begin{aligned} p(\mathbf{x}; \mu) &= \prod_{n=1}^N \mathcal{N}(x_n; \mu, \sigma_{\text{true}}^2) \\ &= \prod_{n=1}^N \frac{1}{\sqrt{2\pi\sigma_{\text{true}}^2}} e^{-\frac{1}{2\sigma_{\text{true}}^2}(x_n - \mu)^2} \end{aligned}$$

You can note that the likelihood is as a function of  $\mu$  only and not  $\sigma_{\text{true}}^2$  because in this example, we assume that we know  $\sigma_{\text{true}}^2$ .

Let's calculate this product.

Since  $\sqrt{x} = x^{1/2}$ , you have:

$$\frac{1}{\sqrt{2\pi\sigma_{\text{true}}^2}} = \frac{1}{(2\pi\sigma_{\text{true}}^2)^{\frac{1}{2}}}$$

and thus:

$$\prod_{n=1}^N \frac{1}{\sqrt{2\pi\sigma_{\text{true}}^2}} = \frac{1}{(2\pi\sigma_{\text{true}}^2)^{\frac{N}{2}}}$$

In addition, from the laws of exponents,  $e^a \cdot e^b = e^{a+b}$ , so:

$$\begin{aligned} \prod_{n=1}^N e^{-\frac{1}{2\sigma_{\text{true}}^2}(x_n - \mu)^2} &= e^{\sum_{n=1}^N -\frac{1}{2\sigma_{\text{true}}^2}(x_n - \mu)^2} \\ &= e^{-\frac{1}{2\sigma_{\text{true}}^2} \sum_{n=1}^N (x_n - \mu)^2} \end{aligned}$$

So you have the likelihood function:

$$p(\mathbf{x}; \mu) = \frac{1}{(2\pi\sigma_{\text{true}}^2)^{\frac{N}{2}}} e^{-\frac{1}{2\sigma_{\text{true}}^2} \sum_{n=1}^N (x_n - \mu)^2}$$

This is the likelihood of observing the  $N$  data samples  $\mathbf{x}$  as a function of  $\mu$ . Like the prior distribution, the likelihood is a function of  $\mu$ .

Let's implement this function. It takes the data  $\mathbf{x}$  and  $\mu$  as inputs and returns the corresponding likelihood. Let's write this function in a way that allows you to pass an array of  $\mu$  values. It will return the likelihood for each of those values.

```
def likelihood(x, mu):
    sigma_true = 0.1
    N = x.shape[0]
    n_mu = mu.shape[0]
    likelihood = np.zeros(n_mu)
    for i in range(n_mu):
        likelihood[i] = (1 / ((2 * np.pi * sigma_true ** 2) ** (N / 2)))
    np.exp((-1 / (2 * sigma_true ** 2)) * np.sum((x - mu[i]) **
    2))
    return likelihood
```

You can try the likelihood function with a single value of  $\mu$ :

```
likelihood(x=x_data, mu=np.array([0.2]))
array([340.70708444])
```

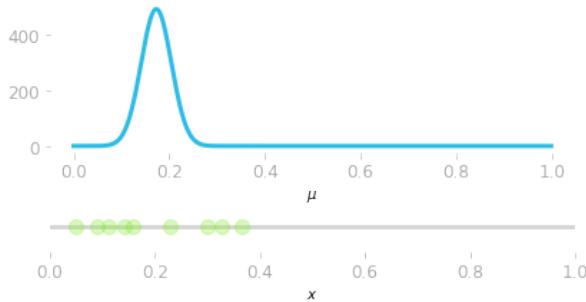
This result means that the likelihood that the distribution of the data is a Gaussian function with a mean of 0.2 and a standard deviation of 0.1 ( $\sigma_{\text{true}}^2$ ) is around 340.7.

You can use this function to calculate the likelihood for a range of values, and plot the likelihood function:

```
mu_axis = np.arange(0, 1, 0.001)
likelihood_y = likelihood(x=x_data, mu=mu_axis)

f, axes = plt.subplots(2, 1, figsize=(6, 3),
                      gridspec_kw={'height_ratios': [3, 1]})

axes[0].plot(mu_axis, likelihood_y)
axes[1].scatter(x_data, np.repeat(0, x_data.shape[0]), c="#84EE29",
                 alpha=0.3)
# [...] Add axes, label, etc.
```



*Figure 4.13: The likelihood function and the corresponding data samples.*

You can see in Figure 4.13 the likelihood function obtained for a range of potential values of  $\mu$  (`mu_axis`). The axes on the plot should remind you that

the likelihood curve is a function of  $\mu$ . The data samples are also represented, showing that the peak of the likelihood function corresponds to the sample mean.

#### 4.4.2.3 Posterior

When the prior and the likelihood are Gaussian distributions, the posterior is also a Gaussian distribution. It is thus possible to mathematically express the posterior as a Gaussian:

$$p(\mu|\mathbf{x}) = \mathcal{N}(\mu; \mu_{\text{post}}, \sigma_{\text{post}}^2)$$

This means that the posterior distribution  $p(\mu|\mathbf{x})$  is still a function of  $\mu$  and parametrized by the mean ( $\mu_{\text{post}}$ ) and the variance ( $\sigma_{\text{post}}^2$ ).

The posterior can be analytically calculated. We'll take for granted that the values of the parameters are calculated as follows<sup>14</sup>:

$$\mu_{\text{post}} = \frac{\sigma_{\text{true}}^2}{N\sigma_{\text{prior}}^2 + \sigma_{\text{true}}^2}\mu_{\text{prior}} + \frac{N\sigma_{\text{prior}}^2}{N\sigma_{\text{prior}}^2 + \sigma_{\text{true}}^2}\mu_{\text{ML}}$$

$$\frac{1}{\sigma_{\text{post}}^2} = \frac{1}{\sigma_{\text{prior}}^2} + \frac{N}{\sigma_{\text{true}}^2}$$

Here,  $N$  is the number of observations (`N = x.shape[0]` in the posterior function implemented below) and  $\mu_{\text{ML}}$  the maximum likelihood solution (corresponding to the value of  $\mu$  associated with the largest likelihood). As you saw in Section 4.2.3, the maximum likelihood solution is equal to the sample mean (the mean of the data samples: `mu_ML = x.mean()` in the posterior function below).

Let's see how to interpret the formula of the mean ( $\mu_{\text{post}}$ ) and the standard deviation ( $\sigma_{\text{post}}^2$ ) of the posterior.

If you look at the mean of the posterior,  $\mu_{\text{post}}$ , has two components: the mean of the prior ( $\mu_{\text{prior}}$ ) and the maximum likelihood solution ( $\mu_{\text{ML}}$ ). These two

---

<sup>14</sup>You can find more details in Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006., p. 98 and ex 2.38.

components are weighted by factors (`factor_mu_prior` and `factor_mu_ML` in the code below) that make the result close to the prior or close to the likelihood (and thus, the data). If the number of observations  $N$  is large, the weight of the maximum likelihood solution increases (because  $N$  is in the nominator for the weight of  $\mu_{ML}$ ), and thus, the relative prior weight decreases: the data has more influence. If  $N$  tends to 0, the maximum likelihood solution tends to zero, and thus, the estimated mean corresponds to the prior mean.

Concerning the estimated variance,  $\frac{1}{\sigma_{post}^2}$  corresponds to the precision (the inverse of the variance). You can see that the precision increases when  $N$  increases, which means that the width of the Gaussian distribution of the posterior will be smaller, leading a smaller uncertainty in the result.

Let's implement a function that calculates the posterior using these equations. This function must take the data and the parameters of the prior distribution  $\mu_{prior}$  and  $\sigma_{prior}$  as parameters. It will return the parameters of the posterior distribution (the mean and the standard deviation).

```
def posterior(x, mu_prior, sigma_prior):
    sigma_true = 0.1
    N = x.shape[0]
    mu_ML = x.mean()

    factor_mu_prior = sigma ** 2 / (N * sigma_prior ** 2 + sigma ** 2)
    factor_mu_ML = (N * sigma_prior ** 2) / (N * sigma_prior ** 2 + sigma ** 2)
    mu_N = factor_mu_prior * mu_prior + factor_mu_ML * mu_ML

    sigma_N = np.sqrt(1 / ((1 / (sigma_prior ** 2)) + (N / (sigma ** 2))))
    return mu_N, sigma_N
```

You have all the building blocks to look at the effect of the data and the prior on the estimated distribution. You'll calculate the likelihood from the data, use the prior to calculate the posterior, and then plot the prior and posterior distributions with the true mean, using different number of data samples.

```
mu_true = 0.2
sigma_true = 0.1

mu_prior = 0.8
sigma_prior = 0.1

x_axis = np.arange(0, 1, 0.001)
prior_y = gaussian(x_axis, mu_prior, sigma_prior)

# [...] Prepare axes etc.
all_obs = [1, 2, 3, 10, 50, 100]
x_data = np.random.normal(mu_true, sigma_true, all_obs[-1])

count = 0
for n_obs, ax in zip(np.repeat(all_obs, 2), axes):
    if count % 2 == 1:
        ax.scatter(x_data[:n_obs], np.repeat(0, n_obs), zorder=10,
                   alpha=0.2)
        # [...] Add axes, labels, etc.
    else:
        likelihood_y = likelihood(x=x_data[:n_obs], mu=mu_axis)
        mu_posterior, sigma_posterior = posterior(x_data[:n_obs],
                                                    mu_prior, sigma_prior)
        posterior_y = gaussian(x_axis, mu_posterior, sigma_posterior)

        ax.plot(x_axis, prior_y, label='prior')
        ax.plot(x_axis, posterior_y, label='posterior')

        ax.vlines(x=mu_true, ymin=0, ymax=15, color="#F57F53")
        # [...] Add axes, labels, etc.

    count += 1

# [...] Add common labels, etc.
```



*Figure 4.14: Prior (in blue) and posterior distributions (in green) using different number of data samples. The mean of the data samples is represented in red. The data samples are also represented (note that their values are not function of  $\mu$ , hence the separate axis).*

Figure 4.14 shows the effect of increasing the number of data samples on Bayesian inference. You can see that the prior is given less weight when you add data (the green curve move off the blue one).

Try to decrease the standard deviation of the prior and you'll see that the prior will be given more weight. It's like saying that you're more less uncertain about your prior.

Using a prior can be used to force the fit toward values that are not expressed in the data, and in this sense, it acts exactly as a regularization term (you'll

see more details on regularization in Section 5.5).

# **Part III**

# **Linear Algebra**



---

Machines only understand numbers. For instance, if you want to create a spam detector, you have first to convert your text data into numbers (for instance, through *word embeddings*). Data can then be stored in vectors, matrices, and tensors. For instance, images are represented as matrices of values between 0 and 255 representing the luminosity of each color for each pixel. It is possible to leverage the tools and concepts from the field of linear algebra to manipulate these vectors, matrices and tensors.

Linear algebra is the branch of mathematics that studies *vector spaces*. You'll see how vectors constitute vector spaces and how linear algebra applies linear transformations to these spaces. You'll also learn the powerful relationship between sets of linear equations and vector equations, related to important data science concepts like *least squares approximation*. You'll finally learn important matrix decomposition methods: *eigendecomposition* and *Singular Value Decomposition* (SVD), important to understand unsupervised learning methods like *Principal Component Analysis* (PCA).



# Chapter 05

## Scalars and Vectors

### 5.1 What are Vectors?

Linear algebra deals with *vectors*. Other mathematical entities in the field can be defined by their relationship to vectors: *scalars*, for example, are single numbers that *scale* vectors (stretching or contracting) when they are multiplied by them.

However, vectors refer to various concepts according to the field they are used in. In the context of data science, they are a way to store values from your data. For instance, take the height and weight of people: since they are distinct values with different meanings, you need to store them separately, for instance using two vectors. You can then do operations on vectors to manipulate these features without loosing the fact that the values correspond to different attributes.

You can also use vectors to store data samples, for instance, store the height of ten people as a vector containing ten values.

**Notation** We'll use lowercase, boldface letters to name vectors (such as  $\mathbf{v}$ ).<sup>1</sup> As usual, refer to the Appendix appendix B to have the summary of the notations used in this book.

---

<sup>1</sup>In other resources, you might also encounter a letter with an arrow, like  $\vec{v}$ , especially for handwriting, where boldface is not possible.

### 5.1.1 Geometric and Coordinate Vectors

The word *vector* can refer to multiple concepts. Let's learn more about geometric and coordinate vectors.

*Coordinates* are values describing a position. For instance, any position on earth can be specified by geographical coordinates (latitude, longitude, and elevation).

#### 5.1.1.1 Geometric Vectors

*Geometric vectors*, also called *Euclidean vectors*, are mathematical objects defined by their magnitude (the length) and their direction. These properties allow you to describe the displacement from a location to another.

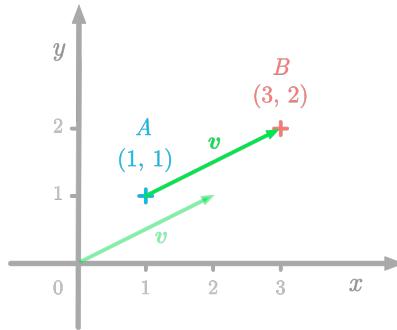


Figure 5.1: A geometric vector running from A to B.

For instance, Figure 5.1 shows that the point  $A$  has coordinates  $(1, 1)$  and the point  $B$  has coordinates  $(3, 2)$ . The geometric vector  $\mathbf{v}$  describes the displacement from  $A$  to  $B$ , but since vectors are defined by their magnitude and direction, you can also represent  $\mathbf{v}$  as starting from the origin.

## Cartesian Plane

In Figure 5.1, we used a coordinate system called the *Cartesian plane*. The horizontal and vertical lines are the *coordinate axes*, usually labeled respectively  $x$  and  $y$ . The intersection of the two coordinates is called the *origin* and corresponds to the coordinate 0 for each axis.

In a Cartesian plane, any position can be specified by the  $x$  and the  $y$  coordinates. The Cartesian coordinate system can be extended to more dimensions: the position of a point in a  $n$ -dimensional space is specified by  $n$  coordinates. The real coordinate  $n$ -dimensional space, containing  $n$ -tuples of real numbers, is named  $\mathbb{R}^n$ . For instance, the space  $\mathbb{R}^2$  is the two-dimensional space containing pairs of real numbers (the coordinates). In three dimensions ( $\mathbb{R}^3$ ), a point in space is represented by three real numbers.

### 5.1.1.2 Coordinate Vectors

*Coordinate vectors* are ordered lists of numbers corresponding to the vector coordinates. Since vector initial points are at the origin, you need to encode only the coordinates of the terminal point.

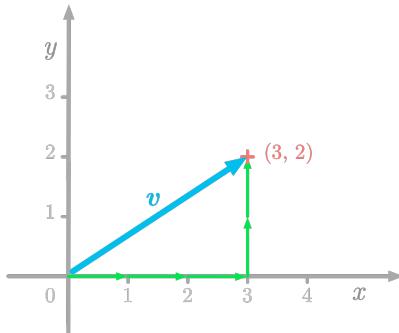


Figure 5.2: The vector  $v$  has coordinates  $(3, 2)$  corresponding to three units from the origin on the  $x$ -axis and two on the  $y$ -axis.

For instance, let's take the vector  $v$  represented in Figure 5.2. The corre-

sponding coordinate vector is as follows:

$$\mathbf{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Each value is associated with a direction: in this case, the first value corresponds to the  $x$ -axis direction and the second number to the  $y$ -axis.

$$\mathbf{v} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

This is a  
**component**  
or an **entry**  
of  $\mathbf{v}$

Figure 5.3: Components of a coordinate vector.

As illustrated in Figure 5.3, these values are called *components* or *entries* of the vector.

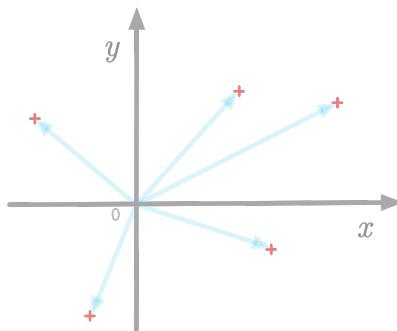


Figure 5.4: Vectors can be represented as points in the Cartesian plane.

In addition, as represented in Figure 5.4, you can simply represent the terminal point of the arrow: this is a scatter-plot.

**Indexing** *Indexing* refers to the process of getting a vector component (one of the values from the vector) using its position (its index).

Python uses zero-based indexing, meaning that the first index is zero. However mathematically, the convention is to use one-based indexing. I'll denote the component  $i$  of the vector  $\mathbf{v}$  with a subscript, as  $v_i$ , without bold font because the component of the vector is a scalar.

**Numpy** In Numpy, vectors are called *one-dimensional arrays*. You can use the function `np.array()` to create one:

```
v = np.array([3, 2])  
v
```

```
array([3, 2])
```

**More Components** Let's take the example of  $\mathbf{v}$ , a three-dimensional vector defined as follows:

$$\mathbf{v} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

As shown in Figure 5.5, you can reach the endpoint of the vector by traveling 3 units on the  $x$ -axis, 4 on the  $y$ -axis, and 2 on the  $z$ -axis.

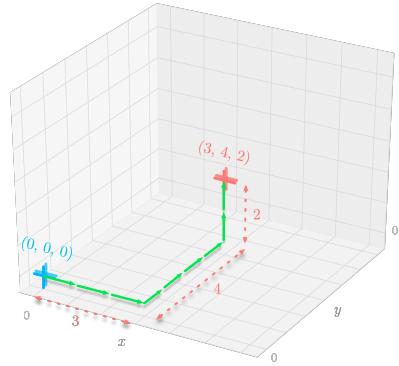


Figure 5.5: Three-dimensional representation of the origin at  $(0, 0, 0)$  and the point at  $(3, 4, 2)$ .

More generally, in a  $n$ -dimensional space, the position of a terminal point is described by  $n$  components.

#### 5.1.1.3 Dimensions

You can denote the dimensionality of a vector using the *set* notation  $\mathbb{R}^n$ . It expresses the *real coordinate space*: this is the  $n$ -dimensional space with real numbers as coordinate values.

For instance, vectors in  $\mathbb{R}^3$  have three components, as the following vector  $\mathbf{v}$  for example:

$$\mathbf{v} = \begin{bmatrix} 2.0 \\ 1.1 \\ -2.5 \end{bmatrix}$$

#### 5.1.1.4 Vectors in Data Science

In the context of data science, you can use coordinate vectors to represent your data.

You can represent data samples as vectors with each component corresponding to a feature. For instance, in a real estate dataset, you could have a vector

corresponding to an apartment with its features as different components (like the number of rooms, the location, etc.).

Another way to do it is to create one vector per feature, each containing all observations.

Storing data in vectors allows you to leverage linear algebra tools. Note that, even if you can't visualize vectors with a large number of components, you can still apply the same operations on them. This means that you can get insights about linear algebra using two or three dimensions, and then, use what you learn with a larger number of dimensions.

## 5.1.2 Vector Spaces

A *vector space* is a collection of vectors. You can use uppercase letters to refer to vector spaces, as  $V$  for instance. Mathematically, any vector in a vector space must satisfy the following statements (these rules are called *axioms*).

### 5.1.2.1 Axioms

For all  $\mathbf{u}, \mathbf{v}, \mathbf{w} \in V$  (the symbol  $\in$  is pronounced “in”, meaning that the vectors are in the space  $V$  and thus that the following axioms must hold for each vector in the space  $V$ ):

- $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$ .
- $(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$ .
- There exists an element  $0$  so that  $\mathbf{u} + 0 = \mathbf{u}$  for all  $\mathbf{u}$ .
- For all  $\mathbf{u}$ , there exists an element  $-\mathbf{u}$  so that:  $\mathbf{u} + (-\mathbf{u}) = 0$ .
- $(c + d)\mathbf{u} = c\mathbf{u} + d\mathbf{u}$ .
- $c(\mathbf{u} + \mathbf{v}) = c\mathbf{u} + c\mathbf{v}$ .
- $c(d\mathbf{u}) = (cd)\mathbf{u}$ .
- There exists an element  $1 \cdot \mathbf{u} = \mathbf{u}$ .

A set (that is, a collection of elements) that satisfies these axioms is called a vector space, and its elements are called vectors. You can imagine various vector spaces, the vectors only have to satisfy the axioms.

### 5.1.3 Special Vectors

There are some special vectors important to know in linear algebra and data science.

#### 5.1.3.1 Row and Columns Vectors

You can distinguish vectors according to their shape. Numbers are organized as a column in *column vectors*, as in the following vector  $\mathbf{u}$  :

$$\mathbf{u} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

They are organized as a row in *row vectors*, as in the following vector  $\mathbf{v}$ :

$$\mathbf{v} = [2 \ 1]$$

And what about this difference in Numpy's vectors? Let's create a vector with Numpy:

```
v_row = np.array([1, 2, 3])
v_row
```

```
array([1, 2, 3])
```

You can use the property `shape` to check the shape of the array:

```
v_row.shape
```

```
(3,)
```

You can see that there is only one dimension (the shape has one number). This is because Numpy doesn't distinguish between row and column vectors. To make this distinction, you need to create a matrix with one column or one row (as you'll see in chapter 6).

### 5.1.3.2 Unit Vectors

*Unit vectors* are vectors with a length of one. You'll see the concept of vector length in Section 5.3.

### 5.1.3.3 Zero Vectors

A *zero vector* is a vector such that adding it to another vector doesn't change this vector.

### 5.1.3.4 Orthogonal Vectors

Two vectors are called *orthogonal* when they run in perpendicular directions. You can see a geometric example of two orthogonal vectors in Figure 5.6.

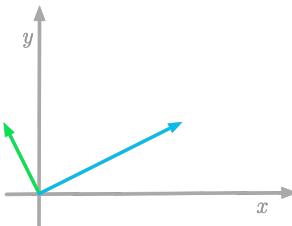


Figure 5.6: Two orthogonal vectors.

If the length of both orthogonal vectors is one (that is, if they are unit vectors), then they are called *orthonormal*.

## 5.2 Operations and Manipulations on Vectors

Addition and scalar multiplication are the two major operations in linear algebra.

### 5.2.1 Scalar Multiplication

Scalar multiplication is the operation of multiplying a vector with a scalar. When multiplied by a scalar, a vector gives another vector, which is a *scaled* version of the initial vector. It rescales the vector.

### 5.2.1.1 Example

Let's have the following vector  $\mathbf{v}$ :

$$\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

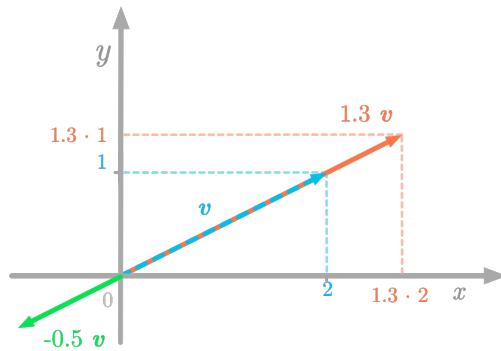


Figure 5.7: Multiplication of the vector  $\mathbf{v}$  by scalars (1.3 in red and -0.5 in green).

The vector  $\mathbf{v}$  is represented in Figure 5.7 (in blue) with two rescaled versions (multiplied by 1.3 and -0.5). You can also see that the scalar multiplication is done by multiplying each component of the vector by the scalar:

$$1.3\mathbf{v} = 1.3 \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.3 \cdot 2 \\ 1.3 \cdot 1 \end{bmatrix} = \begin{bmatrix} 2.6 \\ 1.3 \end{bmatrix}$$

You can do it using Numpy:

```
v = np.array([2, 1])
v
array([2, 1])
```

```
1.3 * v
```

```
array([2.6, 1.3])
```

### Fast computations

Numpy implements what is called *vectorized operations*, meaning that under the hood, it uses compiled C code, making the computations way faster than for loops.<sup>a</sup>

---

<sup>a</sup>You can for instance read [https://www.pythonlyoumeanit.com/Module\\_3\\_IntroducingNumpy/VectorizedOperations.html#Vectorized-Operations](https://www.pythonlyoumeanit.com/Module_3_IntroducingNumpy/VectorizedOperations.html#Vectorized-Operations)

## 5.2.2 Vector Addition

When you add two vectors, you get a third vector. Consider the two vectors  $\mathbf{u}$  and  $\mathbf{v}$  illustrated in Figure 5.8.

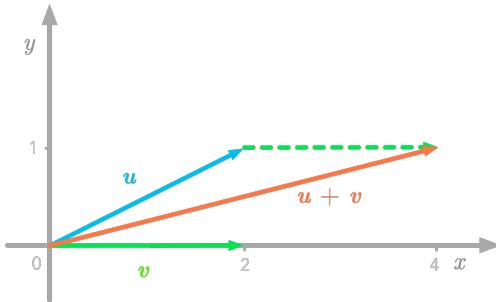


Figure 5.8: Adding the vectors  $\mathbf{u}$  and  $\mathbf{v}$  gives a new vector.

If you take  $\mathbf{v}$  and put it at the end of  $\mathbf{u}$  as represented with the dotted green line, you get the sum of the two vectors (the other way works as well:  $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$ ).

Let's consider the coordinates of these geometric vectors. The vectors  $\mathbf{u}$  and  $\mathbf{v}$  are defined as follows:

$$\mathbf{u} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

and

$$\mathbf{v} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

You can sum these vectors by adding their respective coordinates:

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 2+2 \\ 1+0 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

Let's do this vector addition with Numpy:

```
v1 = np.array([2, 1])
v2 = np.array([2, 0])
v1 + v2
```

```
array([4, 1])
```

### 5.2.3 Transposition

The *transposition* of a vector transforms a row vector into a column vector and vice versa. It is denoted as the superscript letter T. For instance,  $\mathbf{v}^T$  corresponds to the transpose of the vector  $\mathbf{v}$ :

$$\begin{bmatrix} x \\ y \end{bmatrix}^T = [x \ y]$$

and

$$\begin{bmatrix} x & y \end{bmatrix}^T = \begin{bmatrix} x \\ y \end{bmatrix}$$

With Numpy, the transpose of a vector is given by the simple letter `T`. However, since Numpy doesn't discriminate between row and column vectors, transposition will have no effect on one-dimensional arrays (but, as you'll see, it is useful with matrices):

### 5.3 Norms

A *norm* is a function that takes a vector and returns a single number and which satisfies the rules you'll see in this section. You can think of the norm of a vector as its length. It is denoted with double vertical bars:

$$\|\mathbf{u}\|$$

#### Norm and absolute values

Don't confuse the norm notation with the absolute values notation which uses single vertical bars ( $|a|$ ).<sup>a</sup>

---

<sup>a</sup>The similarity between these symbols comes from the fact that the absolute value can be considered as an special case of norm: <https://math.stackexchange.com/a/16167/484100>

It is common to use norms to evaluate the distance between two vectors  $\mathbf{u}$  and  $\mathbf{v}$ . You calculate the differences between them ( $\mathbf{w} = \mathbf{u} - \mathbf{v}$ ) and calculate the norm of the new vector  $\mathbf{w}$ . For this reason, norms are crucial in machine learning and deep learning. More specifically it is used in:

- Cost function: you can use the norm of a vector containing error values (for instance differences between true values and estimated values).
- Regularization: as you'll see in Section 5.5, you can use the norm of the vector containing the model parameters as a way to estimate how large

they are. Adding this term to the cost function helps avoiding large parameter values, which is known to reduce overfitting.

### 5.3.1 Definitions

There are various ways to calculate the length of a vector, and thus, multiple kinds of norms. You can call a mathematical entity a norm only if it satisfies the following rules:

1. Non-negativity: norms must be non-negative. If you interpret the norm of a vector as its length, you can see that there is no such things as negative length.
2. Zero-vector norm: the norm of vector is zero if and only if the vector is a zero-vector (vectors that don't modify other vectors when added to them, as you saw in Section 5.1.3.3).
3. Scalar multiplication: the norm of a vector multiplied by a scalar corresponds to the absolute value of this scalar multiplied by the norm of the vector. For instance, with the scalar  $k$  and the vector  $\mathbf{u}$ , you have  $\|k \cdot \mathbf{u}\| = |k| \cdot \|\mathbf{u}\|$ .
4. Triangle inequality: the norm of the sum of two vectors is less than or equal to the sum of their norms. You can write this mathematically as follows:

$$\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$$

As shown graphically in Figure 5.9, triangle inequality simply means that the shortest path between two points is a line.

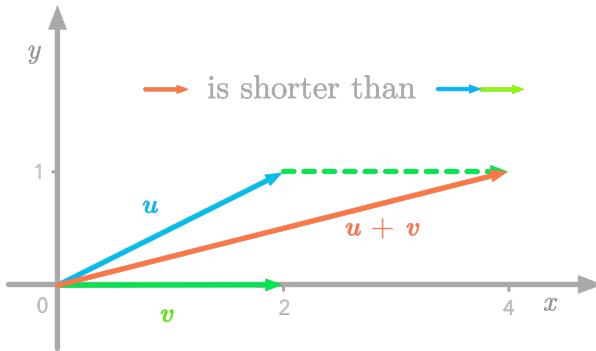


Figure 5.9: Illustration of the triangle inequality.

### 5.3.2 Common Vector Norms

In machine learning and deep learning, it is important to be able to compare vectors. Norms provide a way to do that and is used as a metric quantifying the difference between them. The kind of norm that you use change how these differences are weighted in the metric (for instance, if you consider the large differences as much as the small ones).

As you'll see, many norms fall into the category of *p-norms*, which are calculated as the sum of the absolute value of each component raised to the power of  $p$ . The result of this sum is then raised to the power of  $\frac{1}{p}$ . It may be easier to read the mathematical formula:

$$\|\mathbf{x}\|_p = \left( \sum_{i=1}^m |x_i|^p \right)^{\frac{1}{p}}$$

with  $\mathbf{x}$  a vector,  $m$  its number of components,  $i$  the index of the current vector component.

Different values of  $p$  give different norms. Let's see the more common.

#### 5.3.2.1 $L^2$ Norm

The  $L^2$  norm is extensively used in machine learning and deep learning. It is also called *weight decay* in the context of regularization of deep neural networks.

The vector length measured with the  $L^2$  norm corresponds to an *Euclidean distance*, that is, the physical distance in the real world, which is a consequence of the Pythagorean theorem. The formula is as follows:

$$\|\boldsymbol{x}\|_2 = \left( \sum_{i=1}^m |x_i|^2 \right)^{\frac{1}{2}} = \sqrt{\sum_{i=1}^m |x_i|^2} = \sqrt{\sum_{i=1}^m x_i^2}$$

with  $\boldsymbol{x}$  the vector,  $m$  its number of components, and  $i$  the index of current component. Note that you don't need to take the absolute value of coordinates since they are raised to the power of two.

You can use the function `np.linalg.norm` from Numpy with the parameter `ord` set to 2 to calculate the  $L^2$  norm:

```
u = np.array([2, 1])
np.linalg.norm(u, ord=2)
```

```
2.23606797749979
```

### 5.3.2.2 Squared L2 Norm

For computation reasons, the squared  $L^2$  norm can be preferred over the  $L^2$  norm. For instance, in the context of minimizing a cost function, squaring the function doesn't change its minimum. Squaring the  $L^2$  norm allows you to get rid of the square root in the formula.

$$\|\boldsymbol{x}\|_2^2 = \left( \sqrt{\sum_{i=1}^m x_i^2} \right)^2 = \sum_{i=1}^m x_i^2$$

The resulting norm function is the sum of the squared vector components.

In addition, as you'll see in Section 5.4 the squared  $L^2$  norm can be calculated using the dot product, which is computationally advantageous.

### Norm or cost function

The Mean Squared Error (MSE; as you used in Section 1.3.1) resembles the squared  $L^2$  norm. The difference is that you take the average of the squared errors with the MSE and the sum of the squared errors with the squared  $L^2$  norm.

With neural networks, you calculate the cost and its derivative (as you saw in Section 1.3), so it is advantageous to use cost functions that are easily calculated and simple to differentiate from a computing perspective. The squared  $L^2$  norm can be easily vectorized, which is highly desirable since it permits faster computations and parallelizations. At the hardware level, vectorized code can be optimized and computations can be ran in parallel<sup>2</sup>.

#### 5.3.2.3 $L^1$ Norm

The  $L^1$  norm is a function returning the sum of the absolute value of the components:

$$\|\mathbf{x}\|_1 = \left( \sum_{i=1}^m |x_i|^1 \right)^{\frac{1}{1}} = \sum_{i=1}^m |x_i|$$

with  $\mathbf{x}$  the vector,  $m$  its number of components, and  $i$  the index of the current component.

---

<sup>2</sup>Here is more details about why vectorized code is faster: <https://stackoverflow.com/questions/35091979/why-is-vectorization-faster-in-general-than-loops>

### Taxicab distance

The  $L^1$  norm is also called the *Manhattan distance* or the *taxicab distance* because of the displacement of a taxi in a street grid, like in Manhattan.

As illustrated in Figure 5.10, a taxi driver going from A to B would prefer to take the yellow diagonal path if she could. It is because the  $L^2$  norm represents the physical world. This is different with the  $L^1$  norm: the three paths have the same length. For instance, the  $L^1$  length of the diagonal vector is 6 (because it is the sum of its  $x$  and  $y$  component:  $3 + 3 = 6$ ), which is identical to the other paths.

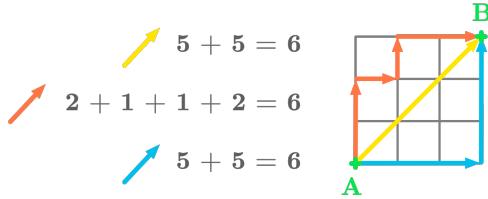


Figure 5.10: With the Manhattan distance, each path has the same length.

The  $L^1$  norm of  $\mathbf{u}$  can be calculated with Numpy as follows:

```
np.linalg.norm(u, ord=1)
```

3.0

#### 5.3.2.4 Max Norm

The  $L^\infty$ , or *max norm* (also called the *Chebyshev norm*) is a function returning the largest component of the vector.

### Weight penalties and weight constraints

Max norms are typically used as a constraint on the weights of neural networks<sup>a</sup>. Weight constraints differs from weight penalties: with weight penalties, a regularization term is added to the cost function, while with weight constraints, the weights are forced to be lower than a threshold.<sup>b</sup>

---

<sup>a</sup>For instance in association with dropout: Srivastava, Nitish, et al. “Dropout: a simple way to prevent neural networks from overfitting.” The journal of machine learning research 15.1 (2014): 1929-1958.

<sup>b</sup>Weight constraints allows you to use large learning rates. See for instance, Hinton, Geoffrey E., et al. “Improving neural networks by preventing co-adaptation of feature detectors.” arXiv preprint arXiv:1207.0580 (2012).

This norm is mathematically denoted as:

$$\|\mathbf{x}\|_\infty = \max_i |x_i|$$

It can be calculated with Numpy:

```
u = np.array([1, 0, 0, -1.532, 230, 0.23, 1.7])
np.linalg.norm(u, ord=np.inf)
```

230.0

### 5.3.3 Norm Representations

You can visualize the differences between norms by looking at the *unit circle*. The unit circle is a representation of all the unit vectors (the vectors with a norm equals to 1). So, the unit circle is a shape where every point has a distance of one from the center. According to the kind of distance you use ( $L^1$ ,  $L^2$ , or max norm), this shape is different.

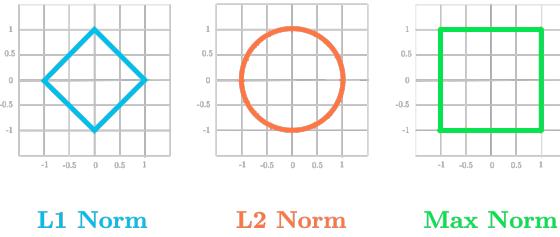


Figure 5.11: Comparison of the unit shapes corresponding to the  $L^1$ ,  $L^2$  and max norm.

Figure 5.11 shows that different norms give you different unit shapes: a diamond shape for the  $L^1$  norm, a circle for the  $L^2$  norm and a square for the max norm. When used as regularization, these shapes have an effect on how the weights are penalized.

For instance, the corners of the diamond correspond to the case where one of the vector components equals zero, which leads to the model sparsity associated with the  $L^1$  norm (you'll see more about this in Section 5.5). The circle corresponding to the  $L^2$  norm is what you expect in the physical world: every point on a circle has the same distance from the center. For the max norm, the maximum of the two components is one for each point on the unit shape.

## 5.4 The Dot Product

The *dot product* (referring to the dot symbol used to characterize this operation), also called *scalar product*, is an operation done on vectors. It takes two vectors, but unlike addition and scalar multiplication, it returns a single number (a scalar, hence the name). It is an example of a more general operation called the *inner product*.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = [1 \times 4 + 2 \times 5 + 3 \times 6] \\ = 32$$

*Figure 5.12: Illustration of the dot product.*

Figure 5.12 shows an illustration of how the dot product works. You can see that it corresponds to the sum of the multiplication of the components with the same index.

### 5.4.1 Definition

The dot product between two vectors  $\mathbf{u}$  and  $\mathbf{v}$ , denoted by the symbol  $\cdot$ , is defined as the sum of the product of each pair of components. More formally, it is expressed as:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^m u_i v_i$$

with  $m$  the number of components of the vectors  $\mathbf{u}$  and  $\mathbf{v}$  (they must have the same number of components), and  $i$  the index of the current vector component.

#### Dot Symbol

Note that the symbol of the dot product is the same as the dot used to refer to multiplication between scalars. The context (if the elements are scalars or vectors) tells you which one it is.

Let's take an example. You have the following vectors:

$$\mathbf{u} = \begin{bmatrix} 2 \\ 4 \\ 7 \end{bmatrix}$$

and

$$\mathbf{v} = \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix}$$

The dot product of these two vectors is defined as:

$$\mathbf{u} \cdot \mathbf{v} = \begin{bmatrix} 2 \\ 4 \\ 7 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix} = 2 \times 5 + 4 \times 1 + 7 \times 3 = 35$$

The dot product between  $\mathbf{u}$  and  $\mathbf{v}$  is 35. It converts the two vectors  $\mathbf{u}$  and  $\mathbf{v}$  into a scalar.

Let's use Numpy to calculate the dot product of these vectors. You can use the method `dot()` of Numpy arrays:

```
u = np.array([2, 4, 7])
v = np.array([5, 1, 3])
u.dot(v)
```

35

It is also possible to use the following equivalent syntax:

```
np.dot(u, v)
```

35

Or, with Python 3.5+, it is also possible to use the `@` operator:

```
u @ v
```

35

### Vector Multiplication

Note that the dot product is different from the *element-wise* multiplication, also called the *Hadamard product*, which returns another vector. The symbol  $\odot$  is generally used to characterize this operation. For instance:

$$\mathbf{u} \odot \mathbf{v} = \begin{bmatrix} 2 \\ 4 \\ 7 \end{bmatrix} \odot \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \cdot 5 \\ 4 \cdot 1 \\ 7 \cdot 3 \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \\ 21 \end{bmatrix}$$

#### 5.4.1.1 Dot Product and Vector Length

The squared  $L^2$  norm can be calculated using the dot product of the vector with itself ( $\mathbf{u} \cdot \mathbf{u}$ ):

$$\|\mathbf{u}\|_2^2 = \mathbf{u} \cdot \mathbf{u}$$

This is an important property in machine learning, as you saw in Section 5.3.2.2.

### 5.4.1.2 Special Cases

The dot product between two orthogonal vectors is equal to 0. In addition, the dot product between a unit vector and itself is equal to 1.

### 5.4.2 Geometric interpretation: Projections

How can you interpret the dot product operation with geometric vectors? You have seen in Section 5.2 the geometric interpretation of the addition and scalar multiplication of vectors, but what about the dot product?

Let's take the two following vectors:

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

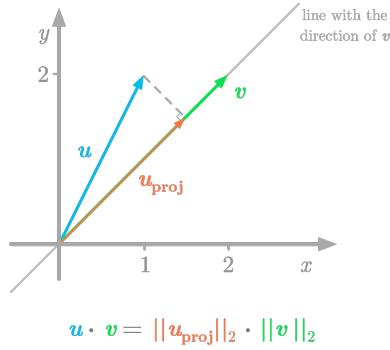
and

$$\mathbf{v} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

First, let's calculate the dot product of  $\mathbf{u}$  and  $\mathbf{v}$ :

$$\mathbf{u} \cdot \mathbf{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 2 \end{bmatrix} = 2 \cdot 1 + 2 \cdot 2 = 6$$

What is the meaning of this scalar? Well, it is related to the idea of projecting  $\mathbf{u}$  onto  $\mathbf{v}$ .



*Figure 5.13: The dot product can be seen as the length of  $\mathbf{v}$  multiplied by the length of the projection (the vector  $\mathbf{u}_{\text{proj}}$ ).*

As shown in Figure 5.13, the projection of  $\mathbf{u}$  on the line with the direction of  $\mathbf{v}$  is like the shadow of the vector  $\mathbf{u}$  on this line. The value of the dot product (6 in our example) corresponds to the multiplication of the length of  $\mathbf{v}$  (the  $L^2$  norm  $\|\mathbf{v}\|$ ) and the length of the projection of  $\mathbf{u}$  on  $\mathbf{v}$  (the  $L^2$  norm  $\|\mathbf{u}_{\text{proj}}\|$ ). You want to calculate:

$$\|\mathbf{u}_{\text{proj}}\|_2 \cdot \|\mathbf{v}\|_2$$

Note that the elements are scalars, so the dot symbol refers to a multiplication of these values. And you have:

$$\|\mathbf{v}\|_2 = \sqrt{2^2 + 2^2} = \sqrt{8}$$

The projection of  $\mathbf{u}$  onto  $\mathbf{v}$  is defined as follows (you can refer to Section 8.3.3 to see the mathematical details about the projection of a vector onto a line):

$$\mathbf{u}_{\text{proj}} = \frac{\mathbf{u}^T \mathbf{v}}{\mathbf{v}^T \mathbf{v}} \mathbf{v} = \frac{6}{8} \mathbf{v} = 0.75 \mathbf{v}$$

So the  $L^2$  norm of  $\mathbf{u}_{\text{proj}}$  is the  $L^2$  norm of 0.75 times  $\mathbf{v}$ :

$$\|\mathbf{u}_{\text{proj}}\|_2 = 0.75 \|\mathbf{v}\|_2 = 0.75 \cdot \sqrt{8}$$

Finally, the multiplication of the length of  $\mathbf{v}$  and the length of the projection is:

$$\|\mathbf{v}\|_2 \cdot \|\mathbf{u}_{\text{proj}}\|_2 = 0.75 \cdot \sqrt{8} \cdot \sqrt{8} = 0.75 \cdot 8 = 6$$

This shows that you can think of the dot product on geometric vectors as a projection. Using the projection gives you the same result as with the dot product formula.

Furthermore, the value that you obtain with the dot product tells you the relationship between the two vectors. If this value is positive, the angle between the vectors is less than 90 degrees, if it is negative, the angle is greater than 90 degrees, if it is zero, the vectors are orthogonal and the angle is 90 degrees.

### 5.4.3 Properties

Let's review some properties of the dot product.

**Distributive** The dot product is *distributive*. This means that, for instance, with the three vectors  $\mathbf{u}$ ,  $\mathbf{v}$  and  $\mathbf{w}$ , you have:

$$\mathbf{u} \cdot (\mathbf{v} + \mathbf{w}) = \mathbf{u} \cdot \mathbf{v} + \mathbf{u} \cdot \mathbf{w}$$

**Associative** The dot product is not *associative*, meaning that the order of the operations matters. For instance:

$$\mathbf{u} \cdot (\mathbf{v} \cdot \mathbf{w}) \neq (\mathbf{u} \cdot \mathbf{v}) \cdot \mathbf{w}$$

The dot product is not a binary operator: the result of the dot product between two vectors is not another vector (but a scalar).

**Commutative** The dot product between vectors is said to be *commutative*. This means that the order of the vectors around the dot product doesn't matter. You have:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$$

However, be careful, because this is not necessarily true for matrices.

## 5.5 Hands-on Project: Regularization

### 5.5.1 Introduction

In machine learning and deep learning, it happens that your model performs well on your training data but fails to generalize to new data. It can be that the training data are very well described by the model and that even the noise or the specificities of the dataset are considered.

This is called *overfitting*. Models that overfit have large parameter values<sup>3</sup>. You want to avoid that because large parameter values means hight sensitivity of the model, which leads to overfitting and unstable networks<sup>4</sup>.

One way to reduce overfitting is to increase the size of the training dataset to compensate this sensitivity. Another way is to constraint the parameters to be small to avoid too sensitive models. It gives machine learning algorithms a preference for a solution that may be suboptimal on the training set but that generalizes better.

The various strategies used to constraint model's parameters in order to improve the generalization of the performances (even if this means decrease the performance in the training set) are called *regularization*.

How does it work? The model learns the best parameter values according to the loss function. With regularization, you punish large parameter values by adding a extra term to the loss function. You construct this term to consider the values of the parameters. Large parameter values will increases the loss, and thus the model will try to fit the data while keeping the parameters small.

This extra term is constructed as the norm of the vector containing all the parameter values. You'll compare here two regularization methods using the

---

<sup>3</sup>As explained in Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006 on page 8.

<sup>4</sup>You can find more details here: <https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/>.

$L^1$  norm and the  $L^2$  norm.

Note that there is no one best regularization strategy but that it depends on the problem you're trying to solve. This is called the *no free lunch theorem*.<sup>5</sup>

### 5.5.1.1 Loss Functions and Regularization

Let's first clarify the difference between the norms used in loss functions and in regularization. It can be confusing at first to see norms like  $L^1$  or  $L^2$  used in these two contexts.

For cost functions, you calculate the norm of an “error vector”. What we loosely call an error vector here is a vector containing the errors of a model (the differences between the values estimated by the model and the true values). The length of this vector tells you the overall quality of the fit.

With regularization, you calculate the norm of the “parameter vector” (the vector that contains all parameter values). The norm of this vector tells you how large the overall parameter values are.

#### Which Norm to use?

Each norm for loss function is associated with pros and cons. For instance, a model using the  $L^1$  norm as a loss function has the advantage to be more robust to outliers in comparison to  $L^2$ , where the extreme values are weighted more than the low values (because of the squaring).

### 5.5.1.2 Regularization rate

You can modulate the importance of the regularization by an *hyperparameter* (a parameter not optimized during the training) which is multiplied by the regularization term. It is usually referred to as  $\alpha$  (a Greek letter pronounced “alpha”) and sometimes called *regularization parameter* or *regularization rate*.

A large value of  $\alpha$  implies that the solution found by the algorithm will be largely influenced by the regularization term, while a value of  $\alpha$  equals to

---

<sup>5</sup>See Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016., p.118

zero means that there is no regularization at all.

This hands-on project assume that you have some understanding of cost functions, more specifically to the MSE cost function for linear regression (you can refer to Section 1.3.1).

**Regularization Using the  $L^1$  Norm** In the case of regularization, the  $L^1$  norm tends to produce sparse weights (with a small number of non-zero values), meaning that some parameters will be weighted to zero. This can be considered as feature selection, as you can discard the corresponding parameters. Let's see how to create the regularized cost function using the MSE cost function (you can refer to Section 1.3.1 ).

You saw in Section 5.3.2.3 that the  $L^1$  norm of a vector  $\mathbf{x}$  is defined as the sum of the absolute values of the vector components:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^m |x_i|$$

with  $\mathbf{x}$  the vector,  $m$  the number of entries in this vector, and  $i$  the current index.

In the case of linear regression, you optimize two parameters (here called  $\theta_0$  and  $\theta_1$ ). You saw in Section 1.3.1 that the cost function is defined as:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2$$

This is for the case of a single feature  $x$ , leading to a model with two parameters (intercept and slope). The scalar  $m$  is the number of data samples. The errors are the differences between the prediction of the model ( $\theta_0 + \theta_1 x^{(i)}$ ) and the true values  $y^{(i)}$ .

To apply a  $L^1$  regularization to your model, you add the sum of the absolute values of the parameter vector to the cost function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2 + \alpha \sum_{i=1}^n |\theta_i|$$

With  $n$  being the number of parameters (here, two:  $\theta_0$  and  $\theta_1$ ). You can see that the extra term  $\sum_{i=1}^n |\theta_i|$  is multiplied by the regularization parameter  $\alpha$ .

**Regularization Using the  $L^2$  Norm** Similarly, you can regularize the MSE cost function using the squared  $L^2$  norm. We denote this regularization as the  $L^2$  regularization:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2 + \alpha \sum_{i=1}^n \theta_i^2$$

The principle is the same as with the  $L^1$  norm: you add a regularization term that will constraint the parameter values to be small. You'll see next the differences between  $L^1$  and  $L^2$  regularization.

### 5.5.2 Effect of Regularization on Polynomial Regression

To see the effect of regularization, let's see the case of *polynomial regression*, which is a form of regression combining the powers of the features to create new features. For instance, if your only feature is  $x$ , the degree-3 polynomials features are  $x^2$  and  $x^3$ . Instead of having the linear equation  $y = ax + b$ , you'll have  $y = ax + b + cx^2 + dx^3$ , meaning that you now have four parameters to optimize.

#### 5.5.2.1 Data

In this hands-on, you'll use polynomial regression to model the total number of cases of COVID-19 in France (Ile-de-France area) in March 2020<sup>6</sup>. Since the growth of the number of cases is (unfortunately) not linear, you'll use polynomials features to model the data.

Let's first load and plot the data:

```
data_covid = pd.read_csv("https://raw.githubusercontent.com/hadrienj/" \
    "essential_math_for_data_science/master/data/covid19.csv")
```

<sup>6</sup>The data comes from <https://www.data.gouv.fr/en/datasets/cas-confirmed-infection-au-covid-19-par-region/>

```
data_covid['days'] = data_covid["Date"].str.split("/").apply(lambda x: x[2]).astype(int)

plt.scatter(data_covid['days'], data_covid["Ile-de-France"])
```

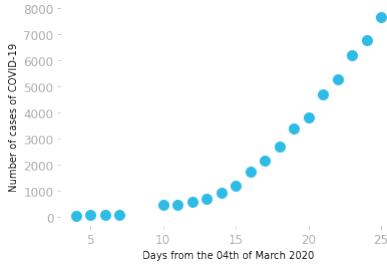


Figure 5.14: Number of cases of COVID19 in France (Ile-de-France area) in March 2020.

You can see in Figure 5.14 that the number of cases increases in a non-linear way. Few days are missing at the beginning but it is not important here.

Let's create the variables you need to do polynomial regression. The only feature is the number of day since the first day (the 04th of March), represented as integers. The number of cases is the dependent variable. You can use the function `reshape(-1, 1)` to have the features in the right shape for Sklearn (it doesn't want one-dimensional arrays):

```
X = data_covid['days'].to_numpy().reshape(-1, 1)
y = data_covid["Ile-de-France"].to_numpy()
```

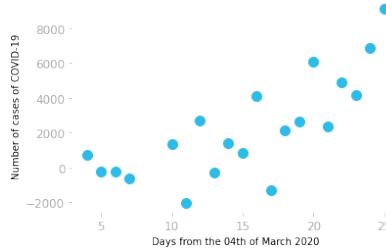
When you train a model that fits your training data very well, it can be the case that it is overfitting, meaning that even the noise of the dataset is taken into consideration, leading to lack of generalization to unseen data.

To observe this effect, and how regularization can fix it, let's add some random noise to the data. It simulates noisy measurements, and will allow you to see how overfitting can be encountered and handled. You can use `np.random.normal` to add a normally distributed noise to the data, however,

you'll also weight the noise we add to have more noise associated with the large value of `X`.

```
np.random.seed(1)
y_noise = y + (np.random.normal(0, 100, X.shape[0]) *
    X.flatten()).flatten()

plt.scatter(X, y_noise)
```



*Figure 5.15: Noisy data will allow you to see the effect of overfitting and regularization.*

Figure 5.15 shows the data after the addition of noise. It will allow you to visualize regularization effects, as you'll soon see.

Regularized models are often sensitive to differences in scale in the data, so you need to normalize the data. You can use standardization (it transforms the data such as the mean is equals to 0 and the standard deviation is equals to 1) using the `StandardScaler` from Sklearn:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

scaler.fit(X)
X_transformed = scaler.transform(X)
y_noise_transformed = scaler.transform(y_noise.reshape(-1, 1)).flatten()
y_transformed = scaler.transform(y.reshape(-1, 1)).flatten()
```

You now need to create the powers of the feature  $x$ . An easy way to do it is to use Sklearn `PolynomialFeatures` that adds all polynomial combinations. In this example, you have a single feature, and thus, there is no combination: it will add the additional features  $x^2$ ,  $x^3$  etc.

To observe overfitting, you'll use a degree-10 of polynomial features (high degree polynomials create complex models that can easily overfit):

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(10, include_bias=False)
X_poly = poly.fit_transform(X_transformed)
```

The option `include_bias=False` avoids the column of 1 created by `PolynomialFeatures` corresponding to the feature  $x^0$ . The variable `X_poly` contains now your new features. Here are, for instance, the two first rows:

```
X_poly[:2]
```

```
array([[ -1.74873638,    3.05807893,   -5.34777387,    9.35184673,
       -16.3539146 ,   28.59868542,   -50.01156162,   87.45703723,
      -152.93930272,  267.45052265],
       [-1.59119256,    2.53189377,   -4.02873054,    6.41048606,
       -10.20031774,   16.23066973,   -25.82612095,   41.09433157,
      -65.38899474,  104.04648209]])
```

The ten columns correspond respectively to the following features:  $x^0, x^1, x^2 \dots x^{10}$ . The first column  $x^0$  is thus a column filled with 1 (which is not included with `include_bias=False`) and the second column contains the values of  $x^1$  which is just  $x$ .

You'll now do a linear regression using the polynomial features as input. It will optimize the weight of each feature ( $x, x^2, x^3$  etc).

To see the effect of the regularization you'll use a linear model using a MSE cost function with  $L^2$  regularization, which is called *ridge regression*.

Let's use the class `Ridge` from Sklearn and iterate over different values of the regularization parameter  $\alpha$ . At each iteration, you'll fit a new model with the

corresponding value of  $\alpha$ , do the prediction on a range of values, and finally use these predictions to plot the regression curve.

```
from sklearn.linear_model import Ridge

X_axis = np.arange(-1.7, 1.65, 0.01)
X_axis_poly = poly.fit_transform(X_axis.reshape(-1, 1))

f, axes = plt.subplots(1, 5, figsize=(16, 4), sharey=True)

for alpha, ax in zip([0.00001, 0.001, 0.01, 1, 1e5], axes.flatten()):
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_poly, y_noise_transformed)
    y_axis_ridge = ridge.predict(X_axis_poly)

    ax.plot(X_axis_poly[:, 0], y_axis_ridge, c="#F57F53")
    ax.scatter(X_transformed, y_transformed,
               alpha=0.6, s=60, label="Data")
    ax.scatter(X_transformed, y_noise_transformed,
               alpha=0.6, s=60, label="Data with noise")
    ax.set_title(r"\$\\alpha\$" + f" = {alpha}")
    # [...] Figure titles, legend...
```

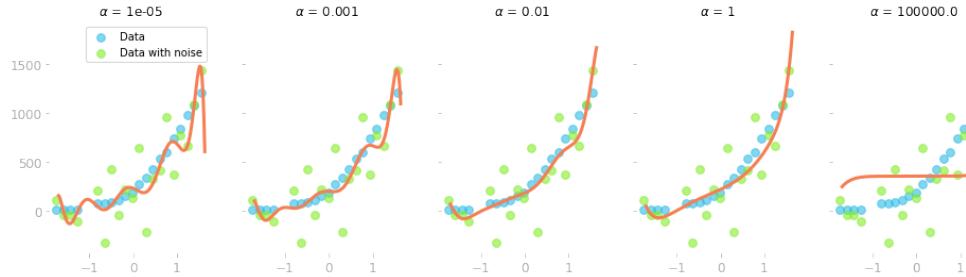


Figure 5.16: Effect of the regularization rate  $\alpha$ .

You can see the effect of regularization in Figure 5.16. Raw data is shown in blue and data with noise in green. The red curve is the regression curve.

In the first plot, the hyperparameter  $\alpha$  is very small, leading to almost no regularization. You can see that the model is overfitting because the oscillations fit also the noise. When you increase  $\alpha$ , the regularization is

more important and the curve is smoother. Finally, with  $\alpha = 1$ , the fit is simpler and ignore a good part of the noise. However, if the value of  $\alpha$  is too large, the model becomes too simple and the regression curve is almost an horizontal line.

You can see how the mathematical concept of norm explains how regularization works. As you see in Figure 5.16, adding a term with the norm of the parameter vector has the effect to constrain the model to be simpler, which often leads to reducing overfitting.

### 5.5.3 Differences between $L^1$ and $L^2$ Regularization

In this last part, you'll visualize how the parameters are optimized and how they are changing during the training (using gradient descent), and what are the differences between the  $L^1$  and the  $L^2$  norms.

In the previous section, you used the class `Ridge` from Sklearn, which uses the closed-form solution of the ridge regression expression (you'll learn more about it in Section 8.4). The other way to find the best parameters is to use gradient descent, as you saw in Section 1.3. You'll use this method to visualize how the parameters change from their initial value (0 because of *zero-initialization*) to the best combination during the training, and look at how the parameters update epoch by epoch.

#### 5.5.3.1 Creating the Polynomial Features

To visualize the parameters, you'll use degree-2 polynomials to be able to visualize the *parameter space* in two dimensions (one parameter in each axis).<sup>7</sup>

Let's create the features, as you did before:

```
poly = PolynomialFeatures(2, include_bias=False)
X_poly = poly.fit_transform(X_transformed)
```

Here are the first three rows of your features (the first column is  $x$  and the second  $x^2$ ):

---

<sup>7</sup>The intercept of the model is also learned, but it is not influenced by regularization so we can't use it for the purpose of this hands-on project.

```
X_poly[:3]  
  
array([[-1.74873638,  3.05807893],  
       [-1.59119256,  2.53189377],  
       [-1.43364874,  2.05534872]])
```

Let's start by finding the best parameters for an unregularized model using the class `LinearRegression`:

```
from sklearn.linear_model import LinearRegression  
  
reg_poly = LinearRegression()  
reg_poly.fit(X_poly, y_transformed)  
  
LinearRegression()
```

Note that we used the raw data and not the data with added noise. The parameters are stored in the attribute named `.coef_` and the intercept in `.intercept_`:

```
reg_poly.coef_  
  
array([382.47220015, 157.18454754])  
  
reg_poly.intercept_  
  
228.15975420975187
```

Keep in mind the mathematical meaning of this. It means that the model is mathematically defined as the following equation:

$$y = 382.47220015x + 157.18454754x^2 + 228.15975420975187$$

### 5.5.3.2 Using Gradient Descent

**Progression of Parameter Values Without Regularization** Now that you know the best parameters, let's use gradient descent to see how the parameters are learned. Let's start by initializing the class `SGDRegressor` (the Sklearn implementation of gradient descent) without regularization. This

will allow you to visualize the progression of the parameters from the initial values to the best combination.

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=10000, random_state=123,
                      learning_rate='constant', eta0=0.01, penalty=None)
```

By default, `SGDRegressor` uses a  $L^2$  regularization so you need to use `penalty=None` to have no regularization. The parameter `learning_rate` tells how the weights are updated and `eta0` corresponds to the initial value of the learning rate (with `learning_rate='constant'`, it will be the only value).

Now, you'll run epochs using `partial_fit`, which updates parameters only once. You can store parameter values at each epoch to be able to visualize the process of the training:

```
history_theta_0 = [0]
history_theta_1 = [0]
for i in range(30):
    sgd_reg.partial_fit(X_poly, y_transformed)

    history_theta_0.append(sgd_reg.coef_[0].copy())
    history_theta_1.append(sgd_reg.coef_[1].copy())
```

**Best Parameters with  $L^1$  and  $L^2$  and Various  $\alpha$**  Let's use  $L^1$  and  $L^2$  regularization with different values of the regularization rate  $\alpha$  to see its influence.

Let's start with  $L^1$  regularization:

```
alpha_theta_0_11 = []
alpha_theta_1_11 = []

all_alpha_11 = np.arange(0, 500, 20)
for alpha in all_alpha_11:
    sgd_reg_11 = SGDRegressor(max_iter=10000, verbose=0, random_state=123,
```

```
        learning_rate='constant', eta0=0.0001,
penalty='l1', alpha=alpha)
sgd_reg_l1.fit(X_poly, y_transformed)

alpha_theta_0_l1.append(sgd_reg_l1.coef_[0].copy())
alpha_theta_1_l1.append(sgd_reg_l1.coef_[1].copy())
```

The array `all_alpha_l1` contains various values of the regularization rate  $\alpha$  used in `SGDRegressor`. For each value of  $\alpha$ , a new model is created, fitted to the data, and the model coefficients are stored.

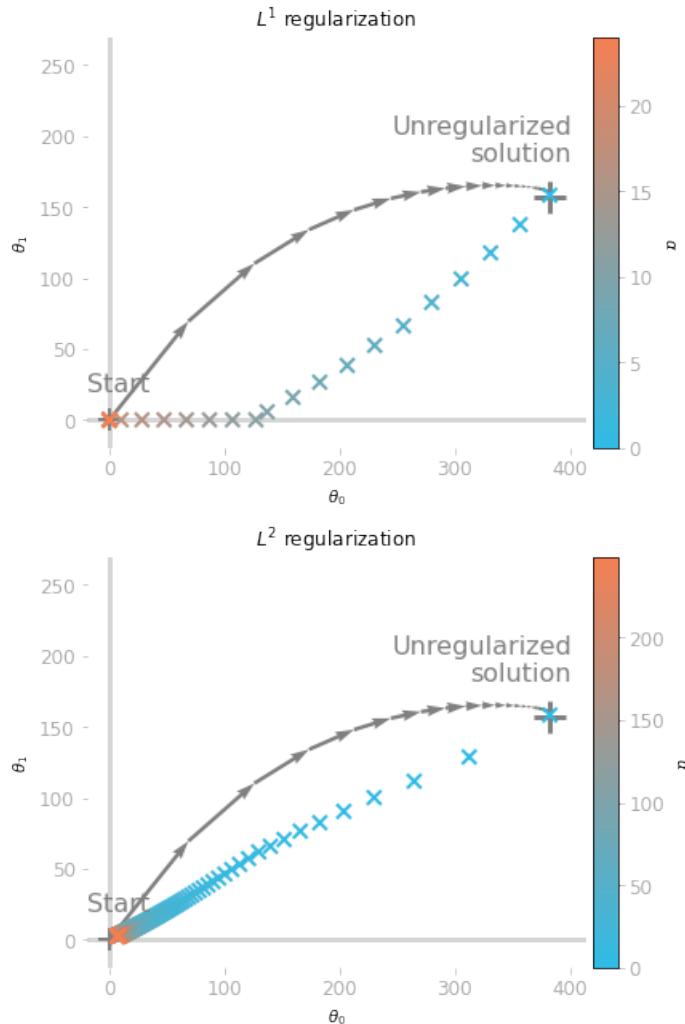
Let's do the same for  $L^2$  regularization:

```
alpha_theta_0_l2 = []
alpha_theta_1_l2 = []

all_alpha_l2 = np.arange(0, 50, 0.2)
for alpha in all_alpha_l2:
    sgd_reg_l2 = SGDRegressor(max_iter=10000, verbose=0, random_state=123,
                             learning_rate='constant', eta0=0.0001,
                             penalty='l2', alpha=alpha)
    sgd_reg_l2.fit(X_poly, y_transformed)

    alpha_theta_0_l2.append(sgd_reg_l2.coef_[0].copy())
    alpha_theta_1_l2.append(sgd_reg_l2.coef_[1].copy())
```

That's good, you can now plot all of how the parameters change for  $L^1$  and  $L^2$  regularization.



*Figure 5.17: Visualization of the effect of  $L^1$  (left) and  $L^2$  regularization in the parameter space ( $\theta_1$  as a function of  $\theta_0$ ). The gray arrows shows the path in the case of no regularization. The crosses correspond to solutions with regularization, and the color shows the amount of regularization (values of  $\alpha$ ).*

Figure 5.17 shows the parameters  $\theta_1$  (y-axis) as a function of  $\theta_0$  (x-axis) in the context of gradient descent. The learning path without regularization is shown as the gray arrows (each arrow corresponds to one epoch). They go

from the initial point (zero for both parameters) to the best solution when there is no regularization.

The solution found with gradient descent is different if you use regularization. The cross symbols correspond to the solutions found with a specific value of  $\alpha$  (this value is given by the color of the symbol). When  $\alpha$  equals zero (blue cross symbols), the regularization term equals zero, so you're in the case of no regularization and the point is close to the unregularized solution. When  $\alpha$  is larger, the regularization is stronger, until the solution approaches zero (red cross symbols) because the model wants only minimize the norm of the parameter vector.

You can then compare the positions of the cross symbols with  $L^1$  (left) and  $L^2$  (right). With  $L^1$  regularization, one of the two parameters ( $\theta_0$ ) is pulled toward zero when regularization gets stronger, and only then, the other parameter ( $\theta_1$ ) goes to zero. This shows a major characteristic of the  $L^1$  regularization: it enforces the sparsity of the parameters. This means that the parameters are constraints to be small with the strategy to set some of the parameters to zero instead of reducing all of them. This can be used as feature selection.

With  $L^2$  regularization, both  $\theta_0$  and  $\theta_1$  are similarly pulled toward zero when  $\alpha$  increases: it prefers parameters near but different than zero<sup>8</sup>.

### 5.5.3.3 Visualizing Regularized Cost Functions

To finish the hands-on project, you'll create contour plots of cost functions regularized with  $L^1$  and  $L^2$ . You'll see that the cost function depends heavily on the regularization parameter  $\alpha$ .

To create these plots, you need to calculate the cost for any combinations of parameters in the range you want to visualize. This is a job for the Numpy function `np.meshgrid`: you input one-dimensional arrays (vectors) corresponding to the ranges of  $\theta_0$  and  $\theta_1$  and it returns two two-dimensional arrays organized such that if you take pairs of values from them, you get all possible combinations.

---

<sup>8</sup>You can see this page of the machine learning crash course by Google to have an idea of the distribution of the parameters with the  $L^2$  regularization: <https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/lambda>.

```

theta_0_space = np.linspace(-300, 550, 100)
theta_1_space = np.linspace(-300, 550, 100)

theta_0_mesh, theta_1_mesh = np.meshgrid(theta_0_space, theta_1_space)

```

Next, you'll create the MSE cost function that takes the data (`x` and `y`), the weights (`theta_0`, `theta_1` and `intercept`), and the regularization parameter  $\alpha$ . Note that the computation is not fully vectorized to be easier to understand.

You'll also implement the  $L^1$  and  $L^2$  norms:

```

def J(X, y, theta_0, theta_1, intercept):
    m = X.shape[0]
    sum_cost = 0
    for i in range(m):
        sum_cost += (((intercept + theta_0 * X[i, 0] + theta_1 * X[i, 1]) - y[i])) ** 2

    return (1 / (2 * m)) * sum_cost

def l1(theta_0, theta_1, alpha):
    return alpha * (np.abs(theta_0) + np.abs(theta_1))

def l2(theta_0, theta_1, alpha):
    return alpha * (theta_0 ** 2 + theta_1 ** 2)

```

Finally, let's use the function `contourf` from Matplotlib to visualize the cost function associated with various values of  $\alpha$  for the  $L^1$  and  $L^2$  regularizations:

```

f, axes = plt.subplots(nrows=1, ncols=4, figsize=(17, 4), sharey=True)

alpha_values = [20, 150, 230, 500]

for alpha, ax in zip(alpha_values, axes.flatten()):
    cost_contour = J(X=X_poly, y=y_transformed, theta_0=theta_0_mesh,
                      theta_1=theta_1_mesh,
                      intercept=reg_poly.intercept_ + l1(theta_0_mesh,
                      theta_1_mesh, alpha=alpha)

    contour_plot = ax.contourf(theta_0_mesh, theta_1_mesh, cost_contour,
                               levels=10, zorder=0, cmap="inferno")

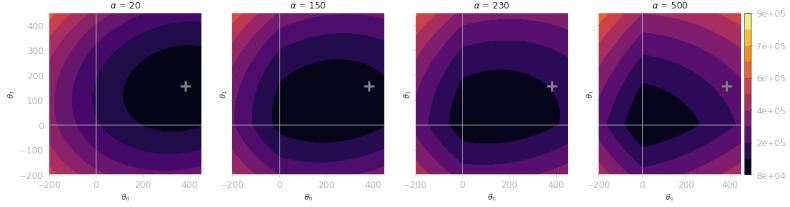
```

```

ax.scatter(reg_poly.coef_[0], reg_poly.coef_[1], c="gray", marker="+",
           s=200, zorder=0)

# [...] Add axes, limits, titles, etc.

```



*Figure 5.18: The MSE cost function with  $L^1$  regularization. The cost (dark colors corresponds to small costs) is represented for each pair of parameters  $\theta_0$  and  $\theta_1$ , and for different value of  $\alpha$ .*

You can see in Figure 5.18 that the cost function changes when the regularization parameter increases. It tends to have the shape of the  $L^1$  regularization term (you can look in Figure 5.19 at the  $L^1$  and  $L^2$  regularization terms alone).

```

theta_0_space_reg = np.linspace(-200, 200, 100)
theta_1_space_reg = np.linspace(-200, 200, 100)

theta_0_mesh_reg, theta_1_mesh_reg = np.meshgrid(theta_0_space_reg,
                                                theta_1_space_reg)

f, axes = plt.subplots(nrows=1, ncols=2, figsize=(9, 4), sharey=True)

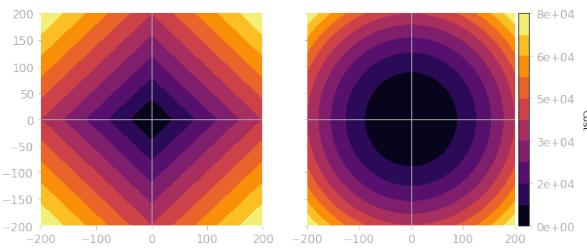
contour_plot = axes[0].contourf(theta_0_mesh_reg, theta_1_mesh_reg,
                                11(theta_0_mesh_reg, theta_1_mesh_reg, alpha=1),
                                levels=10, zorder=0, cmap="inferno")

contour_plot = axes[1].contourf(theta_0_mesh_reg, theta_1_mesh_reg,
                                12(theta_0_mesh_reg, theta_1_mesh_reg, alpha=1),
                                levels=10, zorder=0, cmap="inferno")

for ax in axes.flatten():

```

```
ax.set_xlim(-200, 200)
ax.set_ylim(-200, 200)
```



*Figure 5.19: Regularization terms:  $L^1$  in the left plot and  $L^2$  in the right plot. Dark colors correspond to a smaller cost.*

Figure 5.19 shows the contour plot of the regularization term alone, without cost function.

Let's now have a look at the  $L^2$  regularized MSE cost function:

```
f, axes = plt.subplots(nrows=1, ncols=4, figsize=(17, 4), sharey=True)

for alpha, ax in zip([0.01, 0.3, 1, 10.0], axes.flatten()):
    cost_contour = J(X=X_poly, y=y_transformed, theta_0=theta_0_mesh,
                      theta_1=theta_1_mesh,
                      intercept=reg_poly.intercept_ + l2(theta_0_mesh,
                      theta_1_mesh, alpha=alpha))

    ax.contourf(theta_0_mesh, theta_1_mesh, cost_contour, levels=10,
                zorder=0, cmap="inferno")
    ax.scatter(reg_poly.coef_[0], reg_poly.coef_[1], c="gray", marker="+",
               s=200, zorder=0)
# [...] Add axes, limits, titles, etc.
```

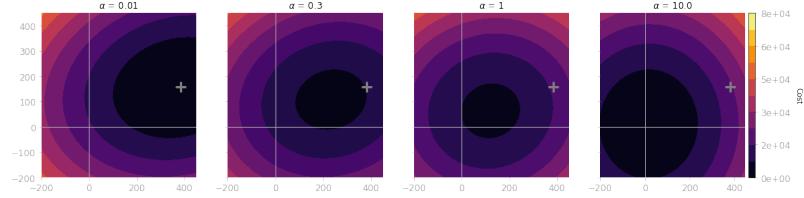


Figure 5.20: The MSE cost function with  $L^2$  regularization. The cost (dark colors corresponds to small costs) is represented for each pair of parameters  $\theta_0$  and  $\theta_1$ , and for different value of the regularization parameter  $\alpha$ .

Figure 5.20 shows again that regularized cost functions tends to the  $L^2$  term alone when the regularization parameter increases. However, unlike with  $L^1$ , the region where some parameters are equal to zero are not easily reached, leading to a decrease of all the parameter values.

That's great. I know there is a lot of things in this hands-on project and that it is clearly for people with some background in machine learning, so take some time on it. It shows you a practical case where understanding the mathematical concepts underlying a machine learning method can give you some insights on what is best in a specific situation.

# Chapter 06

## Matrices and Tensors

As you saw in chapter 5, vectors are a useful way to store and manipulate data. You can represent them geometrically as arrows, or as arrays of numbers (the coordinates of their ending points). However, it can be helpful to create more complicated data structures – and that is where matrices need to be introduced.

### 6.1 Introduction

As vectors, *matrices* are data structures allowing you to organize numbers. They are square or rectangular arrays containing values organized in two dimensions: rows and columns. You can think of them as a spreadsheet. Usually, you'll see the term *matrix* in the context of math and *two-dimensional array* in the context of Numpy.

#### Dimensions

In the context of matrices, the term *dimension* is different from dimensions of the geometric representation of vectors (the dimensions of the space). When we say that a matrix is a two-dimensional array, it means that there are two *directions* in the array: the rows and the columns.

### 6.1.1 Matrix Notation

In this book, I'll denote matrices with bold typeface and upper-case letters, like  $\mathbf{A}$ :

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}$$

The matrix  $\mathbf{A}$  has two rows and two columns but you can imagine matrices with any shape. More generally, if the matrix has  $m$  rows and  $n$  columns and contains real values, you can characterize it with the following notation:  $\mathbf{A} \in \mathbb{R}^{m \times n}$ .

You can refer to matrix entries with the name of the matrix with no bold font (because the entries are scalars) followed by the index for the row and the index for the column separated by a comma in subscript. For instance,  $A_{1,2}$  denotes the entry in the first row and the second column.

By convention, the first index is for the row and the second for the column. For instance, the entry 2 in the matrix  $\mathbf{A}$  above is located in the second row and the first column of the matrix  $\mathbf{A}$ , so it is denoted as  $A_{2,1}$  (remember from Section 5.1.1.2 that one-based indexing is generally used in mathematical notation).

You can write the matrix components as follows:

$$\mathbf{A} = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix}$$

### 6.1.2 Shapes

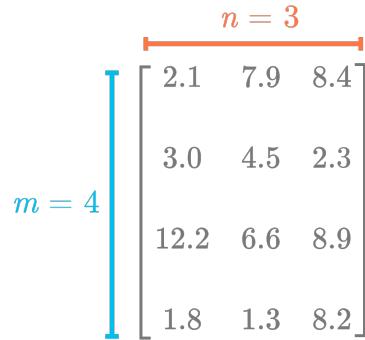


Figure 6.1: Matrices are two-dimensional arrays. The number of rows is usually denoted as  $m$  and the number of columns as  $n$ .

The *shape* of an array gives you the number of components in each dimension, as illustrated in Figure 6.1. Since this matrix is two-dimensional (rows and columns), you need two values to describe the shape (the number of rows and the number of columns in this order).

Let's start by creating a 2D Numpy array with the method `np.array()`:

```
A = np.array([[2.1, 7.9, 8.4],
             [3.0, 4.5, 2.3],
             [12.2, 6.6, 8.9],
             [1.8, 1., 8.2]])
```

Note that we use arrays in arrays (`[[ ]]`) to create the 2D array. This differs from creating a 1D array by the number of square brackets that you use.

Like with vectors, it is possible to access the *shape* property of Numpy arrays:

```
A.shape
```

```
(4, 3)
```

You can see that the *shape* contains two numbers: they correspond to the number of rows and columns respectively.

### 6.1.3 Indexing

To get a matrix entry, you need two indexes: one to refer to the row index and one to refer to the column index.

Using Numpy, the indexing process is the same that of vectors. You just need to specify two indexes. Let's take again the following matrix  $A$ :

```
A = np.array([[2.1, 7.9, 8.4],  
             [3.0, 4.5, 2.3],  
             [12.2, 6.6, 8.9],  
             [1.8, 1.3, 8.2]])
```

It is possible to get a specific entry with the following syntax:

```
A[1, 2]
```

2.3

`A[1, 2]` returns the component with the row index one and the column index two (with zero-based indexing).

To get a complete column, it is possible to use a colon:

```
A[:, 0]
```

```
array([ 2.1,  3. , 12.2,  1.8])
```

This returns the first column (index zero) because the colon says that we want the components from the first to the last rows. Similarly, to get a specific row, you can do:

```
A[1, :]
```

```
array([3. , 4.5, 2.3])
```

Being able to manipulate matrices containing data is an essential skill for data scientists. Checking the shape of your data is important to be sure that it is organized the way you want. It is also important to know the data shape you'll need to use libraries like Sklearn or Tensorflow.

## Default indexing

Note that if you specify a single index from a 2D array, Numpy considers that it is for the first dimension (the rows) and all the values of the other dimension (the columns) are used. For instance

```
A[0]
array([2.1, 7.9, 8.4])
```

which is similar to:

```
A[0, :]
array([2.1, 7.9, 8.4])
```

### 6.1.3.1 Vectors and Matrices

With Numpy, if the array is a vector (1D Numpy array), the shape is a single number:

```
v = np.array([1, 2, 3])
v.shape
(3,)
```

You can see that *v* is a vector. If it is a matrix, the shape has two numbers (the number of values in the rows and in the columns respectively). For instance:

```
A = np.array([[2.1, 7.9, 8.4]])
A.shape
(1, 3)
```

You can see that the matrix has a single row: the first number of the shape is 1. Once again, using two square brackets, `[[` and `]]`, allows you to create a

two-dimensional array (a matrix).

### 6.1.4 Main Diagonal

The *main diagonal* of a matrix is the diagonal starting at the index (1, 1) (the upper left corner).

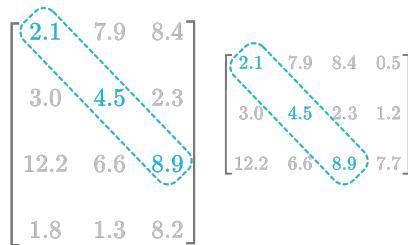


Figure 6.2: Main diagonal of rectangle matrices.

Figure 6.2 illustrates the main diagonal of rectangular matrices (with more columns than rows or more rows than columns).

The other diagonal (from the upper right corner to the lower left corner) is called the *anti-diagonal* or *counter-diagonal*.

### 6.1.5 Tensors

*tensors* are multi-dimensional arrays. This means that scalars, vectors or matrices are tensors. Arrays with more than two dimensions are also tensors. The number of indexes needed to get an entry (for instance, two for a matrix: rows and columns) is called the *rank* of the tensor, not to be confused with the rank of a matrix that you will learn about in Section 7.6.1). It is sometimes called *order* for this reason.

Scalar	Vector	Matrix	Rank n Tensor
1	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$	$\begin{bmatrix} & & & \\ & & 3 & 6 \\ & & 1 & 1 \\ & \dots & & \\ \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} & & & \end{bmatrix}$
- 0-D array - rank 0 tensor	- 1-D array - rank 1 tensor	- 2-D array - rank 2 tensor	- n-D array - rank n tensor

Figure 6.3: An example of a scalar, a vector, a matrix and a tensor.

The difference between scalars, vectors, matrices and tensors is summarized in Figure 6.3. Tensors of rank 0 are scalars, tensors of rank 1 are vectors, tensors of rank 2 are matrices.

### Tensors vs. Multi-dimensional Arrays

Unlike in pure mathematics or physics, the word *tensor* in machine learning, deep learning, or data science simply refers to a multi-dimensional array<sup>a</sup>.

For instance, as you'll see in Section 6.5, color images are represented as pixel values. Each pixel is indexed by:

- Its position on the left/right direction.
- Its position on the bottom/up direction.
- Its position in the color list.

For instance, the index [0, 0, 0] corresponds to the luminance of the upper left pixel for the red color (assuming that image is encoded with the RGB color model).

---

<sup>a</sup>You can find nice details about this difference here: <https://stats.stackexchange.com/questions/198061/why-the-sudden-fascination-with-tensors>

### 6.1.6 Frobenius Norm

The *Frobenius norm* is a  $L^2$  norm (that you have seen in Section 5.3) applied to a matrix. The matrix is first flattened (converted to a one-dimensional vector) and the  $L^2$  norm is calculated on the resulting vector. The Frobenius norm of the matrix  $\mathbf{A}$  is denoted as  $\|\mathbf{A}\|_F$ . Each component of the matrix is squared and the square root of the sum is calculated:

$$\|\mathbf{A}\| = \sqrt{\sum_{i,j} A_{i,j}^2}$$

## 6.2 Operations and Manipulations on Matrices

It is important to know how to manipulate matrices in data science and machine learning. In this section, you will see how to use addition, scalar multiplication and transposition with matrices.

### 6.2.1 Addition and Scalar Multiplication

#### 6.2.1.1 Addition

As with vectors, you can add matrices by adding each component with the same index:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} + \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \\ B_{3,1} & B_{3,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} + B_{1,1} & A_{1,2} + B_{1,2} \\ A_{2,1} + B_{2,1} & A_{2,2} + B_{2,2} \\ A_{3,1} + B_{3,1} & A_{3,2} + B_{3,2} \end{bmatrix}$$

Figure 6.4: Addition of two matrices.

With Numpy, the addition of matrices is done simply by using the `+` operator, like with scalars:

```
A = np.array([[2, 7],  
             [3, 4],
```

```

        [8, 2]])
B = np.array([[3, 1],
              [4, 5],
              [7, 6]])
A + B

```

```

array([[ 5,  8],
       [ 7,  9],
       [15,  8]])

```

To add matrices, you need to ensure that they have the same shape. The resulting matrix will also have the same shape:

```
(A + B).shape
```

```
(3, 2)
```

### Check your Shapes

Systematically check the shape of your data, it is a good way to follow the processing and be sure that there is no issue.

#### 6.2.1.2 Broadcasting

With Numpy, addition can sometimes work even if matrices have different shapes. Look at the following example:

```

A = np.array([[2, 7],
              [3, 4],
              [8, 2]])

```

```

B = np.array([[7],
              [1],
              [3]])

```

The matrix  $A$  has 3 by 2 entries and the matrix  $B$  is 3 by 1. However, you

can still perform addition with Numpy:

```
A + B
```

```
array([[ 9, 14],  
       [ 4,  5],  
       [11,  5]])
```

Since  $\mathbf{B}$  has only one column, it is added to each column of  $\mathbf{A}$ . This is called *broadcasting*. When possible, the smaller matrix will be augmented by duplication to match the shape required for the operation. In our example, the single column vector  $\mathbf{B}$  was duplicated to match the two columns of  $\mathbf{A}$ . The version of  $\mathbf{B}$  would look like:

$$\begin{bmatrix} 7 & 7 \\ 1 & 1 \\ 3 & 3 \end{bmatrix}$$

Broadcasting is useful, but you should also be cautious since it can lead to unexpected results if you're not aware of your data shapes.

### 6.2.1.3 Scalar Multiplication

Another major operation is *scalar multiplication* and refers to multiplying a matrix by a scalar. In this case, each entry is multiplied by this scalar:

$$c\mathbf{A} = c \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} = \begin{bmatrix} c \cdot A_{1,1} & c \cdot A_{1,2} \\ c \cdot A_{2,1} & c \cdot A_{2,2} \\ c \cdot A_{3,1} & c \cdot A_{3,2} \end{bmatrix}$$

Let's take the example of the following matrix  $\mathbf{A}$ :

```
A = np.array([[2, 7],  
             [3, 4],  
             [8, 2]])
```

Let's multiply  $A$  by a scalar (here -2):

```
-2 * A
```

```
array([[-4, -14],  
       [-6, -8],  
       [-16, -4]])
```

As you have seen in chapter 5, scalar multiplication works the same with matrices than with vectors.

## 6.2.2 Transposition

Transposition is a core manipulation in data science and machine learning. It allows you to exchange the rows and the columns of a matrix.

For instance, if your dataset is structured with the observations as the rows but you want to use a machine learning framework where observations must be the columns, you'll need to transpose it.

### 6.2.2.1 From Vector Transposition

Transposing a vector converts a column vector into a row vector or the opposite. This comes from considering vectors as matrices with only one row (row vector) or one column (column vector). As you'll see, the general rule to transpose matrices is to flip the cells through the main diagonal.

It is possible to transpose a Numpy vector with the `T` attribute of arrays only if its number of dimension is larger than one. It is not possible to exchange rows and columns if there is a single dimension, in which case the transposition gives the original vector.

### 6.2.2.2 Matrix Transposition

The transposition of vectors can be extended to matrices if you consider one column at a time. For each column, the values below the main diagonal are

exchanged with the values on the right of the main diagonal. In other words, for each component of the main diagonal of the matrix, the corresponding row becomes the corresponding column.

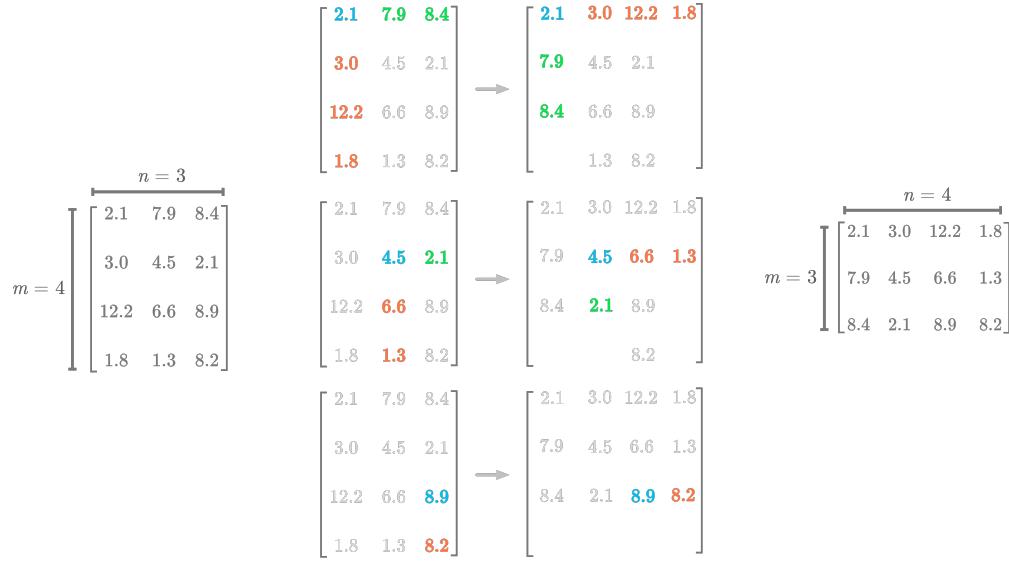


Figure 6.5: Steps of matrix transposition.

Figure 6.5 illustrates these steps. You can see that a 4 by 3 matrix becomes a 3 by 4 matrix because rows and columns are exchanged. More generally, a  $m$  by  $n$  matrix becomes a  $n$  by  $m$  matrix. In the middle panel, you can see the steps corresponding to each of the three diagonal values, where each row and column are exchanged.

Let's do this transposition with Numpy:

```
A = np.array([[2.1, 7.9, 8.4],  
             [3.0, 4.5, 2.1],  
             [12.2, 6.6, 8.9],  
             [1.8, 1.3, 8.2]])  
  
A.T  
  
array([[ 2.1,   3. ,  12.2,   1.8],  
       [ 7.9,   4.5,   6.6,   1.3],  
       [ 8.4,   2.1,   8.9,   8.2]])
```

### 6.2.2.3 Properties of Transposition

Finally, let's see some properties of the matrix transposition.

The transpose of  $\mathbf{A} + \mathbf{B}$  is:

$$(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$

The transpose of a scalar multiplication corresponds to the scalar multiplication of the transpose of the matrix:

$$(c\mathbf{A})^T = c\mathbf{A}^T$$

However, you'll see the transpose of a matrix product in Section 6.3.3.

## 6.3 Matrix Product

You learn about the dot product in Section 5.4. The equivalent operation for matrices is called the *matrix product*, or *matrix multiplication*. It takes two matrices and returns another matrix. This is a core operation in linear algebra.

### 6.3.1 Matrices with Vectors

The simpler case of matrix product is between a matrix and a vector (that you can consider as a matrix product with one of them having a single column).

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 3 + 2 \cdot 4 \\ 5 \cdot 3 + 6 \cdot 4 \\ 7 \cdot 3 + 8 \cdot 4 \end{bmatrix} = \begin{bmatrix} 11 \\ 39 \\ 53 \end{bmatrix}$$

Figure 6.6: Steps of the product between a matrix and a vector.

Figure 6.6 illustrates the steps of the product between a matrix and a vector. Let's consider the first row of the matrix. You do the dot product between the vector (the values 3 and 4 in red) and the row you're considering (the

values 1 and 2 in blue). You multiply the values by pairs: the first value in the row with the first in the column vector ( $1 \cdot 3$ ), and the second in the row with the second in the vector ( $2 \cdot 4$ ). It gives you the first component of the resulting matrix ( $1 \cdot 3 + 2 \cdot 4 = 11$ ).

You can see that the matrix-vector product relates to the dot product. It is like splitting the matrix  $A$  into three rows and applying the dot product (as in Section 5.4).

Let's see how it works with Numpy.

```
A = np.array([
    [1, 2],
    [5, 6],
    [7, 8]
])
v = np.array([3, 4]).reshape(-1, 1)
A @ v

array([[11],
       [39],
       [53]])
```

Note that we used the `reshape()` function to reshape the vector into a 2 by 1 matrix (the `-1` tells Numpy to guess the remaining number). Without it, you would end with a one-dimensional array instead of a two-dimensional array here (a matrix with a single column).

### 6.3.1.1 Weighting of the Matrix's Columns

There is another way to think about the matrix product. You can consider that the vector contains values that weight each column of the matrix.<sup>1</sup> It clearly shows that the length of the vector needs to be equal to the number of columns of the matrix on which the vector is applied.

---

<sup>1</sup>As explained by Gilbert Strang in his introduction to linear algebra that I recommend: Strang, G. "Introduction to linear algebra, 5th edn. Wellesley.", 2016.

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = 3 \cdot \begin{bmatrix} 1 \\ 5 \\ 7 \end{bmatrix} + 4 \cdot \begin{bmatrix} 2 \\ 6 \\ 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 3 + 2 \cdot 4 \\ 5 \cdot 3 + 6 \cdot 4 \\ 7 \cdot 3 + 8 \cdot 4 \end{bmatrix} = \begin{bmatrix} 11 \\ 39 \\ 53 \end{bmatrix}$$

*Figure 6.7: The vectors values are weighting the columns of the matrix.*

Figure 6.7 might help to visualize this concept. You can consider the vector values (3 and 4) as weights applied to the columns of the matrix. The rules about scalar multiplication that you saw earlier lead to the same results as before.

Using the last example, you can write the dot product between  $\mathbf{A}$  and  $\mathbf{v}$  as follows:

$$\begin{aligned} \mathbf{Av} &= \begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ &= 3 \begin{bmatrix} 1 \\ 5 \\ 7 \end{bmatrix} + 4 \begin{bmatrix} 2 \\ 6 \\ 8 \end{bmatrix} \end{aligned}$$

This is important because, as you'll see in more details in Section 7.2, it shows that  $\mathbf{Av}$  is a linear combination of the columns of  $\mathbf{A}$  with the coefficients being the values from  $\mathbf{v}$ .

### 6.3.1.2 Shapes

In addition, you can see that the shapes of the matrix and the vector must match for the dot product to be possible.

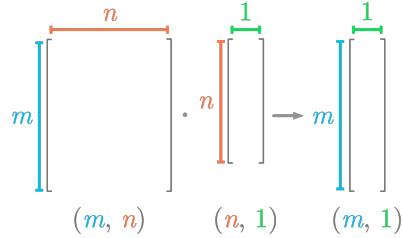


Figure 6.8: Shapes needed for the dot product between a matrix and a vector.

Figure 6.8 summarizes the shapes involved in the matrix-vector product and shows that the number of columns of the matrix must be equal to the number of rows of the vector.

### 6.3.2 Matrices Product

The *matrix product* is the equivalent of the dot product operation for two matrices. As you'll see, it is similar to the matrix-vector product, but applied to each column of the second matrix.

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 3 & 9 \\ 4 & 0 \end{bmatrix} = 3 \cdot \underbrace{\begin{bmatrix} 1 \\ 5 \\ 7 \end{bmatrix}}_{\substack{1 \cdot 3 + 2 \cdot 4 \\ 5 \cdot 3 + 6 \cdot 4 \\ 7 \cdot 3 + 8 \cdot 4}} + 4 \cdot \underbrace{\begin{bmatrix} 2 \\ 6 \\ 8 \end{bmatrix}}_{\substack{1 \cdot 9 + 2 \cdot 0 \\ 5 \cdot 9 + 6 \cdot 0 \\ 7 \cdot 9 + 8 \cdot 0}} = \begin{bmatrix} 11 & 9 \\ 39 & 45 \\ 53 & 63 \end{bmatrix}$$

Figure 6.9: Matrix product.

Figure 6.9 shows you an example of a matrix product. You can see that the resulting matrix has two columns, as the second matrix. The values of the first column of the second matrix (3 and 4) weight the two columns and the result fills the first column of the resulting matrix. Similarly, the values of the second column of the second matrix (9 and 0) weight the two columns and the result fills the second column of the resulting matrix.

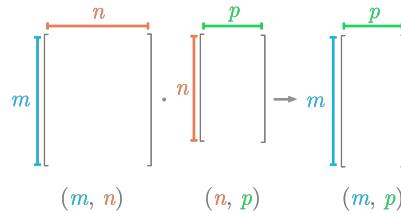
With Numpy, you can calculate the matrix product exactly as the dot product:

```
A = np.array([
    [1, 2],
    [5, 6],
    [7, 8],
])
B = np.array([
    [3, 9],
    [4, 0]
])
A @ B
```

```
array([[11,  9],
       [39, 45],
       [53, 63]])
```

### 6.3.2.1 Shapes

Like with the matrix-vector product and as illustrated in Figure 6.10, the number of columns of the first matrix must match the number of rows of the second matrix.



*Figure 6.10: Shapes must match for the dot product between two matrices.*

The resulting matrix has as many rows as the first matrix and as many columns as the second matrix.

Let's try it.

```
A = np.array([
    [1, 4],
    [2, 5],
    [3, 6],
])
```

```
B = np.array([
    [1, 4, 7],
    [2, 5, 2],
])
```

The matrices  $\mathbf{A}$  and  $\mathbf{B}$  have different shapes. Let's calculate their dot product:

```
A @ B
```

```
array([[ 9, 24, 15],
       [12, 33, 24],
       [15, 42, 33]])
```

You can see the result of  $\mathbf{A} \cdot \mathbf{B}$  is a 3 by 3 matrix. This shape comes from the number of rows of  $\mathbf{A}$  (3) and the number of columns of  $\mathbf{B}$  (3).

### 6.3.2.2 Matrix Product to Calculate the Covariance Matrix

You can calculate the covariance matrix (more details about the covariance matrix in Section 2.1.3) of a dataset with the product between the matrix containing the variables and its transpose. Then, you divide by the number of observations (or this number minus one for the Bessel correction). You need to be sure that the variables are centered around zero beforehand (this can be done by subtracting the mean).

Let's simulate the following variables  $x$ ,  $y$ , and  $z$ :

```
x = np.random.normal(10, 2, 100)
y = x * 1.5 + np.random.normal(25, 5, 100)
z = x * 2 + np.random.normal(0, 1, 100)
```

Using Numpy, the covariance matrix is:

```
np.cov([x, y, z])  
  
array([[ 4.0387007 ,  4.7760502 ,  8.03240398],  
       [ 4.7760502 , 32.90550824,  9.14610037],  
       [ 8.03240398,  9.14610037, 16.99386265]])
```

Now, using the matrix product, you first need to stack the variables as columns of a matrix:

```
X = np.vstack([x, y, z]).T  
X.shape  
  
(100, 3)
```

You can see that the variable `X` is a 100 by 3 matrix: the 100 rows correspond to the observations and the 3 columns to the features. Then, you center this matrix around zero:

```
X = X - X.mean(axis=0)
```

Finally, you calculate the covariance matrix:

```
(X.T @ X) / (X.shape[0] - 1)  
  
array([[ 4.0387007 ,  4.7760502 ,  8.03240398],  
       [ 4.7760502 , 32.90550824,  9.14610037],  
       [ 8.03240398,  9.14610037, 16.99386265]])
```

You get a covariance matrix similar to the one from the function `np.cov()`. This is important to keep in mind that the dot product of a matrix with its transpose corresponds to the covariance matrix.

### 6.3.3 Transpose of a Matrix Product

The transpose of the dot product between two matrices is defined as follows:

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

For instance, take the following matrices  $\mathbf{A}$  and  $\mathbf{B}$ :

```
A = np.array([
    [1, 4],
    [2, 5],
    [3, 6],
])
B = np.array([
    [1, 4, 7],
    [2, 5, 2],
])
```

You can check the result of  $(\mathbf{AB})^T$  and  $\mathbf{B}^T \mathbf{A}^T$ :

```
(A @ B).T
```

```
array([[ 9, 12, 15],
       [24, 33, 42],
       [15, 24, 33]])
```

```
B.T @ A.T
```

```
array([[ 9, 12, 15],
       [24, 33, 42],
       [15, 24, 33]])
```

This can be surprising at first that the order of the two vectors or matrices in the parentheses must change for the equivalence to be satisfied. Let's look at the details of the operation.

$$\begin{array}{ccc}
 \mathbf{A} & \mathbf{v} & \mathbf{Av} \\
 \left[ \begin{matrix} 1 & 2 \\ 5 & 6 \\ 7 & 8 \end{matrix} \right] \cdot \left[ \begin{matrix} 3 \\ 4 \end{matrix} \right] & \xrightarrow{\quad} & \left[ \begin{matrix} 11 \\ 39 \\ 53 \end{matrix} \right] \\
 \uparrow & & \downarrow \\
 [\mathbf{3} \ \mathbf{4}] \cdot \left[ \begin{matrix} 1 & 5 & 7 \\ 2 & 6 & 8 \end{matrix} \right] & \xrightarrow{\quad} & [11 \ 39 \ 53] \\
 \mathbf{v}^T & \mathbf{A}^T & (\mathbf{Av})^T
 \end{array}$$

Figure 6.11: You must change the order of the vector and the matrix to obtain the transpose of the matrix product.

Figure 6.11 shows that the transpose of a matrix product is equal to the product of the transpose if you change the order of the vector and matrix.

#### 6.3.3.1 More than two Matrices or Vectors

You can apply this property to more than two matrices or vectors. For instance,

$$(\mathbf{ABC})^T = \mathbf{C}^T \mathbf{B}^T \mathbf{A}^T$$

Keep this property in a corner of your mind. It explains many “cosmetic rearrangements” that you can encounter when matrices and vectors are manipulated. Trying these manipulations with code is a great way to learn.

To conclude, the matrix product is a key concept of linear algebra, and you will see in Section 7.4 how it relates to space transformation.

## 6.4 Special Matrices

There are few special matrices that are useful to know because they are required to understand more advanced linear algebra concepts.

### 6.4.1 Square Matrices

A *square matrix* has the same number of rows as columns ( $m = n$ ) resulting in a matrix with a square shape. For instance, let's consider the following square matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 7 & 4 \\ 2 & 1 & 2 \\ 6 & 9 & 12 \end{bmatrix}$$

The matrix  $\mathbf{A}$  is a square  $3 \times 3$  matrix. It has 3 rows and 3 columns.

### 6.4.2 Diagonal Matrices

A *diagonal matrix* is a matrix with zeros everywhere except in the main diagonal.

Diagonal matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

Figure 6.12: Example of a diagonal matrix.

Figure 6.12 shows an example of a 3 by 3 diagonal matrix.

Diagonal matrices can also be non square. For instance:

$$\mathbf{D} = \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 4.9 & 0 \\ 0 & 0 & 2.0 \\ 0 & 0 & 0 \end{bmatrix}$$

or

$$\mathbf{D} = \begin{bmatrix} 0.3 & 0 & 0 & 0 \\ 0 & 4.9 & 0 & 0 \\ 0 & 0 & 2.0 & 0 \end{bmatrix}$$

You'll often encounter diagonal matrices, for instance in the context of Singular Value Decomposition (or SVD), as you'll learn in chapter 10.

### 6.4.3 Identity Matrices

An *identity matrix* is a square matrix filled with ones in the main diagonal and zeros elsewhere.

Identity Matrix

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

Figure 6.13: Example of a 3 by 3 identity matrix.

Figure 6.13 shows an example of a 3 by 3 identity matrix. You can denote the  $n \times n$  identity matrix as  $\mathbf{I}_n$ . For instance:

$$\mathbf{I}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

One important property of identity matrices is that when they are multiplied to another matrix, the result is this other matrix unchanged. It is the equivalent to the multiplication by one with scalars. Mathematically, you can write:

$$\mathbf{A}\mathbf{I} = \mathbf{A}$$

The other way is also true with an identity matrix of the right shape. You would have:

$$\mathbf{I}\mathbf{A} = \mathbf{A}$$

Let's see how this is the case.

$$\begin{aligned}
 \mathbf{A}\mathbf{I} &= \begin{bmatrix} 1 & 7 & 4 \\ 2 & 1 & 2 \\ 6 & 9 & 12 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 \cdot 1 + 0 \cdot 2 + 0 \cdot 6 & 1 \cdot 7 + 0 \cdot 1 + 0 \cdot 9 & 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 12 \\ 0 \cdot 1 + 1 \cdot 2 + 0 \cdot 6 & 0 \cdot 7 + 1 \cdot 1 + 0 \cdot 9 & 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 12 \\ 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 6 & 0 \cdot 7 + 0 \cdot 1 + 1 \cdot 9 & 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 12 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 7 & 4 \\ 2 & 1 & 2 \\ 6 & 9 & 12 \end{bmatrix} \\
 &= \mathbf{A}
 \end{aligned}$$

Figure 6.14: The identity matrix product.

Take the example illustrated in Figure 6.14. It shows that you multiply each entry of the matrix by 1 and add the other entries multiplied by 0, resulting in the same matrix.

It is possible to create identity matrices with the function `np.eye()` from Numpy. It takes the number of rows and columns as a parameter. For instance, let's create a three by three identity matrix:

```
np.eye(3)

array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

#### 6.4.4 Inverse Matrices

A matrix  $\mathbf{A}$  is *invertible* if a matrix  $\mathbf{B}$  exists that gives the identity matrix when it is multiplied by  $\mathbf{A}$ . In this case,  $\mathbf{B}$  is said to be the *inverse* of  $\mathbf{A}$ .

The relationship between  $\mathbf{A}$  and  $\mathbf{B}$  is as follows:

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I}$$

The inverse of a matrix  $\mathbf{A}$  is denoted  $\mathbf{A}^{-1}$  (pronounced “A inverse”). So you can write:

$$\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

Only square matrices have an inverse. In addition, square matrices that are not invertible are called *singular matrices*.

Here is an example of a singular matrix:

```
A = np.array([
    [0, 0],
    [0, 1],
])
```

It is possible to calculate the inverse of a matrix using the Numpy function `np.linalg.inv()`.<sup>2</sup>

For instance, take the following matrix  $\mathbf{A}$ :

```
A = np.array([
    [1, -4, 5],
    [8, 5, -3],
    [3, 2, 12]
])
```

Calculate its inverse:

---

<sup>2</sup>If you’re interested in learning how to manually calculate the inverse of a matrix, you can refer for instance to Chapter 2.5 in Strang, Gilbert, et al. Introduction to linear algebra, 5th Edition. Wellesley, MA: Wellesley-Cambridge Press, 2016.

```
A_inv = np.linalg.inv(A)
A_inv

array([[ 0.13441955,  0.11812627, -0.02647658],
       [-0.21384929, -0.00610998,  0.08757637],
       [ 0.00203666, -0.02851324,  0.07535642]])
```

You can check that  $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ :

```
A @ A_inv

array([[ 1.00000000e+00,  6.93889390e-18,  2.77555756e-17],
       [-7.02563008e-17,  1.00000000e+00,  0.00000000e+00],
       [-5.20417043e-17, -2.77555756e-17,  1.00000000e+00]])
```

Note that, in the context of the diagonal values (ones), small numbers like `6.93889390e-18` can be neglected. You can use the function `np.allclose` from Numpy to check if two arrays are equal within a tolerance (that you can choose as a parameter):

```
np.allclose(np.eye(3), A @ A_inv)
```

`True`

Similarly, you can check that  $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ :

```
np.allclose(np.eye(3), A_inv @ A)
```

`True`

You'll see in Section 8.3.1 how to use the inverse of a matrix to solve systems of linear equations.

#### 6.4.4.1 Inverse of a Matrix Product

Similarly to the transposition of a product of two matrices, the inverse of the product of two matrices  $\mathbf{A}$  and  $\mathbf{B}$  is defined as:

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

#### 6.4.4.2 Inverse of a Diagonal Matrix

Consider a diagonal matrix  $\mathbf{D}$  with no zero on the diagonal:

$$\mathbf{D} = \begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & d_3 \end{bmatrix}$$

has the following inverse:

$$\mathbf{D}^{-1} = \begin{bmatrix} \frac{1}{d_1} & 0 & 0 \\ 0 & \frac{1}{d_2} & 0 \\ 0 & 0 & \frac{1}{d_3} \end{bmatrix}$$

This property makes the inverse of diagonal matrices computationally very easy to calculate.

#### 6.4.5 Orthogonal Matrices

As illustrated in Figure 6.15, an *orthogonal matrix* is a square matrix with orthonormal rows and columns<sup>3</sup>.

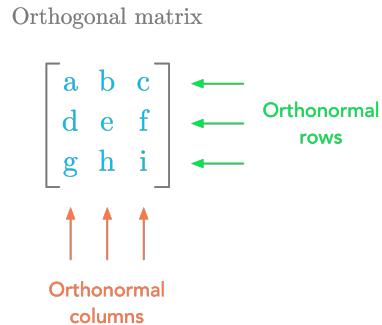


Figure 6.15: Illustration of an orthogonal matrix.

---

<sup>3</sup>Refer to Section 5.1.3.4 if you need a reminder about orthonormal vectors.

Any pairs of the matrix rows or columns are orthonormal vectors.<sup>4</sup> The first consequence is that the dot product between each row or between each column gives zero.

Since the vectors are also unit vectors, the dot product of a row or a column with itself gives one.

This results in the fact that the matrix product between an orthogonal matrix  $\mathbf{Q}$  and its transpose  $\mathbf{Q}^T$  gives the identity matrix. Mathematically, you have:

$$\mathbf{QQ}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}_n$$

It corresponds to the dot product of each row with itself for the diagonals of the resulting matrix. Since the rows are unit vectors, the diagonal are ones. In addition, the values outside of the diagonal are calculated with the dot product of pairs of rows, which results in zeros since the vectors are orthogonal.

Finally, a matrix is orthogonal if its transpose is equal to its inverse. You saw in Section 6.4.4 that:

$$\mathbf{QQ}^{-1} = \mathbf{I}$$

And you just saw that:

$$\mathbf{QQ}^T = \mathbf{I}$$

This implies that the transpose equals the inverse:

$$\mathbf{QQ}^T = \mathbf{QQ}^{-1}$$

$$\mathbf{Q}^T = \mathbf{Q}^{-1}$$

As diagonal matrices, you'll need orthogonal matrices in the context of matrix decomposition in chapter 10.

---

<sup>4</sup>Interestingly, orthogonality of rows implies orthogonality of columns and *vice versa*.

### 6.4.6 Symmetric Matrices

*Symmetric matrices* are matrices that are equal to their own transpose.

Symmetric matrix

$$\begin{bmatrix} 1 & 2 & 6 \\ 2 & 8 & 4 \\ 6 & 4 & 5 \end{bmatrix}$$

Figure 6.16: Illustration of a symmetric matrix.

This means that, as shown in Figure 6.16, they are symmetric with respect to the main diagonal.

Let's take the example of the following matrix  $\mathbf{S}$ :

$$\mathbf{S} = \begin{bmatrix} 0.4 & -2.3 & 7.5 \\ -2.3 & 1.9 & 4.3 \\ 7.5 & 4.3 & 1.0 \end{bmatrix}$$

Let's create the corresponding Numpy array and calculate its transpose:

```
S = np.array([
    [0.4, -2.3, 7.5],
    [-2.3, 1.9, 4.3],
    [7.5, 4.3, 1.0],
])
S.T

array([[ 0.4, -2.3,  7.5],
       [-2.3,  1.9,  4.3],
       [ 7.5,  4.3,  1. ]])
```

You can see that transposing the matrix  $\mathbf{S}$  doesn't modify it.

In addition, the product of a matrix with its transpose gives a symmetric matrix:

$$\mathbf{A}^T \mathbf{A} = \mathbf{S}$$

This is because, from the reverse order of the matrices that you saw in Section 6.3.3, you have:

$$(\mathbf{A}^T \mathbf{A})^T = \mathbf{A}^T (\mathbf{A}^T)^T = \mathbf{A}^T \mathbf{A}$$

Some matrices you saw earlier in this section are symmetric. For instance, square identity matrices are symmetric:

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

Other examples of symmetric matrices are correlation matrices or covariance matrices. You'll see in chapter 9 that symmetric matrices have useful decomposition properties.

#### 6.4.7 Triangular Matrices

*Triangular matrices* are matrices in which the non-zero values form a triangle shape, as illustrated in Figure 6.17.

Triangular matrices

$$\mathbf{U} = \begin{bmatrix} 5 & -2 & 4 \\ 0 & 2 & 9 \\ 0 & 0 & -4 \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} 3 & 0 & 0 \\ 2 & 7 & 0 \\ 8 & -3 & 5 \end{bmatrix}$$

Figure 6.17: Upper and lower triangular matrices.

*Upper triangular matrices* are matrices with zeros for all entries except for the upper triangle (up to the main diagonal). *Lower triangular matrices* have zeros except in the lower triangle (lower to the main diagonal).

As a fun fact, when upper triangular matrices are multiplied or added among themselves, they are still upper triangular matrices. This holds also for lower triangular matrices.

## 6.5 Hands-on Project: Image Classifier

In this hands-on project, you'll see how to manipulate images and put them into a shape understandable by machine learning and deep learning libraries. As a bonus, I have pre-trained a small neural network that you'll use on the data.

### 6.5.1 Images as Multi-dimensional Arrays

To use images as input of machine learning models, you need to encode them in a proper way. Color images are luminance values for each pixel and each color, so you can encode them as tensors with the following dimensions: height, width and color. Usually, color is encoded as three values per pixel (red, green and blue), leading to arrays that have a shape of (height, width, 3).

To give multiple images to a model you can stack the images: you get four-dimensional arrays (image, height, width, 3). You'll see here how it works in practice.

Let's use the CIFAR10 dataset<sup>5</sup> to play with the manipulation of images as multi-dimensional arrays. There are 60,000 color images that have 32 by 32 pixels, each of them is labelized as one of ten categories ("birds", "trucks" etc.) according to the content of the image.

You can load this dataset using the function `cifar10` from the deep learning library Keras:

---

<sup>5</sup>This is a classic dataset used for a lot of benchmarks. You can refer to the initial paper: Krizhevsky, Alex, and Geoffrey Hinton. "Learning multiple layers of features from tiny images.", 2009: 7.

```
from keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

You can see that the function `load_data()` returns multiple variables. The training set (`X_train`) is the data that you use to train your model and the test set (`X_test`) the data that you'll use to evaluate it. This separation is done to be sure that your model doesn't work only on the specific data you trained your model on, but generalizes to new data.<sup>6</sup>

Look at the shape of `X_train` that contains the images:

```
X_train.shape
```

```
(50000, 32, 32, 3)
```

The shape of `X_train` tells you first that the array has four dimensions. There are 50,000 images and each image is 32 by 32 pixels with three colors for each pixel. This means that there are  $32 \cdot 32 \cdot 3 = 3072$  values for each image. This is the number of features of the dataset and you can consider each of the 50,000 images as a data sample.

The labels of data are in `y_train` (50,000 labels) and `y_test` (10,000 labels). They are numbers between 0 and 9, each corresponding to a category like airplane, automobile, bird, etc.

You can now use what you learned on Numpy arrays to manipulate `X_train`. Let's start by displaying a few images:

```
fig, axes = plt.subplots(5, 5, figsize=(6, 6))
for i, ax in zip(np.arange(25), axes.flatten()):
    ax.imshow(X_train[i])
    # [...] Remove grid and axes
```

---

<sup>6</sup>For more details about training vs test set, you can for instance refer to the section “Testing and Validating” in Géron, Aurélien. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, 2019.

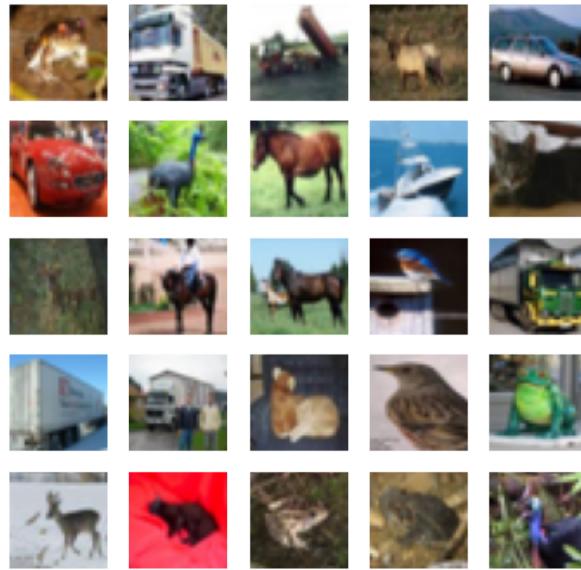


Figure 6.18: Images from the CIFAR10 dataset.

See how you can index the array (`X_train[i]`) as you would do with vectors or matrices? There is a single index for the four dimensions: when you provide a single value for the indexing, Numpy uses the first dimension. Here it returns the  $i$ th image. But you can also be more specific and write the four indexes as follows:

```
plt.imshow(X_train[0, :, :, :])
plt.show()
```



Figure 6.19: One image from the CIFAR10 dataset.

It returns the image with the index zero, and all the height pixels, all the width pixels, and all the colors using the colons.

The goal of this hands-on is to create a model that takes images as input and

returns the estimated class. I have used the 50,000 images from `X_train` and their labels in `y_train` to train a deep learning model. Then, you'll assess the model performance by predicting the class for each of the 10,000 images in `X_test` and check the results with the ground truth (`y_test`).

## 6.5.2 Data Preparation

### 6.5.2.1 Normalization

It is important that the features have the same scale before you use the data to train a model. Let's normalize the values to keep them between zero and one. Since the maximum luminance value is 255, you just need to divide by 255.

```
X_train_rescaled = X_train.astype(np.float32) / 255.0
X_test_rescaled = X_test.astype(np.float32) / 255.0
```

### 6.5.2.2 Encoding

You can encode categorical variables in multiple ways. The labels in CIFAR10 are image categories like ‘birds’, ‘airplanes’ and so one, and they must be numerically encoded. When you load the dataset with Keras, the encoding is what is called *label encoding*, meaning that each class corresponds to a number between 0 and 9. Look at `y_train`:

```
y_train

array([[6],
       [9],
       [9],
       ...,
       [9],
       [1],
       [1]], dtype=uint8)
```

However, Keras needs *one-hot encoding*, that is to say, one vector for each image, containing ten values (one per possible class) filled with zeros except for the category of the image. The position (index) in the vector corresponds to the category.

You can go from label encoding to one-hot encoding using the utility function `np_utils.to_categorical` from Keras:

```
from keras.utils import np_utils
n_classes = 10
y_train = np_utils.to_categorical(y_train, n_classes)
y_test = np_utils.to_categorical(y_test, n_classes)
```

Let's look at the new encoding:

```
y_train
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       ...,
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.]], dtype=float32)
```

```
y_train.shape
```

```
(50000, 10)
```

You can see that the shape of `y_train` is now 50,000 by 10 because there are ten values (nine zeros and a one) per image.

You saw in this example how to manipulate multi-dimensional arrays to put images in a shape digestible by machine learning libraries.

Just for the fun, let's add one more step and use what you did to classify images. It would be out of the scope of the book to cover the details of *Convolutional Neural Networks* (CNN), so I trained it so you can use it to do predictions.

```
from keras.models import Sequential
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.layers import Activation, Flatten, Dense, Dropout, Conv2D
from keras.layers.normalization import BatchNormalization
```

```
model = Sequential()
# [...] Create the model architecture with Keras
# load weights (model already trained)
model.load_weights("data/model.h5")
```

You can use the model to evaluate the test dataset:

```
scores = model.evaluate(X_test_rescaled, y_test, batch_size=128,
    verbose=1)
scores[1]

79/79 [=====] - 4s 50ms/step - loss: 0.4670 -
accuracy: 0.8507
```

0.8507000207901001

The accuracy of the model is around 85% correct.

The goal of this hands-on was to demonstrate that vectors, matrices and tensors, used as multi-dimensional arrays with Numpy, are the core of machine learning and deep learning. You saw that batches of images can be represented as 4D arrays and used under this form with libraries like Keras.



# Chapter 07

## Span, Linear Dependency, and Space Transformation

As you saw in chapter 5 and chapter 6, being able to manipulate vectors and matrices is critical to create machine learning and deep learning pipelines, for instance for reshaping your raw data before using it with machine learning libraries.

The goal of this chapter is to get you to the next level of understanding of vectors and matrices. You'll start seeing matrices, not only as operations on numbers, but also as a way to transform vector spaces. This conception will give you the foundations needed to understand more complex linear algebra concepts like matrix decomposition. You'll build up on what you learned about vector addition and scalar multiplication to understand linear combinations of vectors. You will also see subspace, span, linear dependency, that are major concepts of linear algebra used in machine learning and data science.

### 7.1 Linear Transformations

#### 7.1.1 Intuition

A *linear transformation* (or simply *transformation*, sometimes called *linear map*) is a mapping between two vector spaces: it takes a vector as input

and *transforms* it into a new output vector. A function is said to be linear if the properties of additivity and scalar multiplication are preserved, that is, the same result is obtained if these operations are done before or after the transformation. Linear functions are synonymously called linear transformations.

### Linear transformations notation

You can encounter the following notation to describe a linear transformation:  $T(\mathbf{v})$ . This refers to the vector  $\mathbf{v}$  transformed by  $T$ . A transformation  $T$  is associated with a specific matrix. Since additivity and scalar multiplication must be preserved in linear transformation, you can write:

$$T(\mathbf{v} + \mathbf{w}) = T(\mathbf{v}) + T(\mathbf{w})$$

and

$$T(c\mathbf{v}) = cT(\mathbf{v})$$

### 7.1.2 Linear Transformations as Vectors and Matrices

In linear algebra, the information concerning a linear transformation can be represented as a matrix. Moreover, every linear transformation can be expressed as a matrix<sup>1</sup>.

When you do the linear transformation associated with a matrix, we say that you *apply* the matrix to the vector. More concretely, it means that you calculate the matrix-vector product of the matrix and the vector. In this case, the matrix can sometimes be called a *transformation matrix*. For instance, you can apply a matrix  $\mathbf{A}$  to a vector  $\mathbf{v}$  with their product  $\mathbf{Av}$ .

---

<sup>1</sup>Gilbert Strang. Introduction to Linear Algebra, Fifth Edition. Wellesley, MA: Wellesley-Cambridge Press, 2016., p. 413

### Applying matrices

Keep in mind that, to apply a matrix to a vector, you *left multiply* the vector by the matrix: the matrix is on the left to the vector.

When you multiply multiple matrices, the corresponding linear transformations are combined in the order from right to left.

For instance, let's say that a matrix  $\mathbf{A}$  does a 45-degree clockwise rotation and a matrix  $\mathbf{B}$  does a stretching, the product  $\mathbf{BA}$  means that you first do the rotation and then the stretching.

This shows that the matrix product is:

- Not commutative ( $\mathbf{AB} \neq \mathbf{BA}$ ): the stretching then the rotation is a different transformation than the rotation then the stretching.
- Associative ( $\mathbf{A}(\mathbf{BC}) = ((\mathbf{AB})\mathbf{C})$ ): the same transformations associated with the matrices  $\mathbf{A}$ ,  $\mathbf{B}$  and  $\mathbf{C}$  are done in the same order.

A matrix-vector product can thus be considered as a way to transform a vector. You saw in Section 6.3 that the shape of  $\mathbf{A}$  and  $\mathbf{v}$  must match for the product to be possible.

#### 7.1.3 Geometric Interpretation

A good way to understand the relationship between matrices and linear transformations is to actually visualize these transformations. To do that, you'll use a grid of points in a two-dimensional space, each point corresponding to a vector (it is easier to visualize points instead of arrows pointing from the origin).

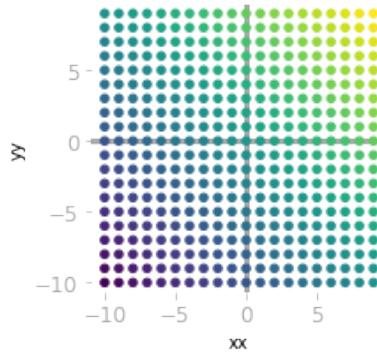
Let's start by creating the grid using the function `meshgrid()` from Numpy:

```
x = np.arange(-10, 10, 1)
y = np.arange(-10, 10, 1)
```

```
xx, yy = np.meshgrid(x, y)
```

The `meshgrid()` function allows you to create all combinations of points from the arrays `x` and `y`. Let's plot the scatter plot corresponding to `xx` and `yy`.

```
plt.scatter(xx, yy, s=20, c=xx+yy)  
# [...] Add axis, x and y witht the same scale
```



*Figure 7.1: Each point corresponds to the combination of  $x$  and  $y$  values.*

You can see the grid in Figure 7.1. The color corresponds to the addition of `xx` and `yy` values. This will make transformations easier to visualize.

#### 7.1.3.1 The Linear Transformation associated with a Matrix

As a first example, let's visualize the transformation associated with the following two-dimensional square matrix.

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

Consider that each point of the grid is a vector defined by two coordinates ( $x$  and  $y$ ).

Let's create the transformation matrix  $\mathbf{T}$ :

```
T = np.array([
    [-1, 0],
    [0, -1]
])
```

First, you need to structure the points of the grid to be able to apply the matrix to each of them. For now, you have two 20 by 20 matrices (`xx` and `yy`) corresponding to  $20 \cdot 20 = 400$  points, each having a  $x$  value (matrix `xx`) and a  $y$  value (`yy`). Let's create a 2 by 400 matrix with `xx` flatten as the first column and `yy` as the second column.

```
xy = np.vstack([xx.flatten(), yy.flatten()])
xy.shape
```

```
(2, 400)
```

You have now 400 points, each with two coordinates. Let's apply the transformation matrix  $\mathbf{T}$  to the first two-dimensional point (`xy[:, 0]`), for instance:

```
T @ xy[:, 0]
```

```
array([10, 10])
```

You can similarly apply  $\mathbf{T}$  to each point by calculating its product with the matrix containing all points:

```
trans = T @ xy
trans.shape
```

```
(2, 400)
```

You can see that the shape is still (2, 400). Each transformed vector (that is, each point of the grid) is one of the column of this new matrix. Now, let's reshape this array to have two arrays with a similar shape to `xx` and `yy`.

```
xx_transformed = trans[0].reshape(xx.shape)
yy_transformed = trans[1].reshape(yy.shape)
```

Let's plot the grid before and after the transformation:

```
f, axes = plt.subplots(1, 2, figsize=(6, 3))
axes[0].scatter(xx, yy, s=10, c=xx+yy)
axes[1].scatter(xx_transformed, yy_transformed, s=10, c=xx+yy)
# [...] Add axis, x and y with the same scale
```

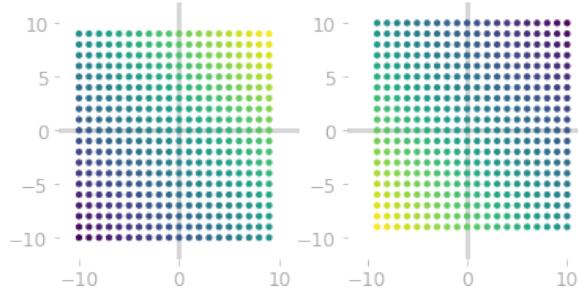


Figure 7.2: The grid of points before (left) and after (right) its transformation by the matrix  $\mathbf{T}$ .

Figure 7.2 shows that the matrix  $\mathbf{T}$  rotated the points of the grid.

### 7.1.3.2 Shapes of the Input and Output Vectors

In the previous example, the output vectors have the same number of dimensions than the input vectors (two dimensions).

You might notice that the shape of the transformation matrix must match the shape of the vectors you want to transform.

$$(2, 2) (2, 400) \longrightarrow (2, 400)$$

Figure 7.3: Shape of the transformation of the grid points by  $\mathbf{T}$ .

Figure 7.3 illustrates the shapes of this example. The first matrix with a shape (2, 2) is the transformation matrix  $\mathbf{T}$  and the second matrix with a shape (2, 400) corresponds to the 400 vectors stacked. As illustrated in blue, the number of rows of the  $\mathbf{T}$  corresponds to the number of dimensions of the output vectors. As illustrated in red, the transformation matrix must have the same number of columns than the number of dimensions of the matrix you want to transform.

More generally, the size of the transformation matrix tells you the input and output dimensions. A  $m$  by  $n$  transformation matrix transforms  $n$ -dimensional vectors to  $m$ -dimensional vectors.

### 7.1.3.3 Stretching and Rotation

Let's now visualize the transformation associated with the following matrix:

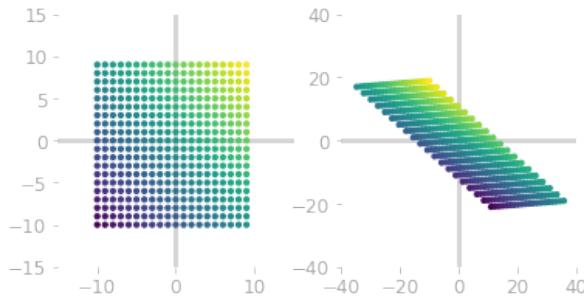
$$\mathbf{T} = \begin{bmatrix} 1.3 & -2.4 \\ 0.1 & 2 \end{bmatrix}$$

Let's proceed as in the previous example:

```
T = np.array([
    [1.3, -2.4],
    [0.1, 2]
])
trans = T @ xy

xx_transformed = trans[0].reshape(xx.shape)
yy_transformed = trans[1].reshape(yy.shape)

f, axes = plt.subplots(1, 2, figsize=(6, 3))
axes[0].scatter(xx, yy, s=10, c=xx+yy)
axes[1].scatter(xx_transformed, yy_transformed, s=10, c=xx+yy)
# [...] Add axis, x and y with the same scale
```



*Figure 7.4: The grid of points before (left) and after (right) the transformation by the new matrix  $\mathbf{T}$ .*

Figure 7.4 shows that the transformation is different from the previous rotation. This time, there is a rotation, but also a stretching of the space.

### Are these transformation linear?

You might wonder why these transformations are called “linear”. You saw that a linear transformation implies that the properties of additivity and scalar multiplication are preserved.

Geometrically, there is linearity if the vectors lying on the same line in the input space are also on the same line in the output space, and if the origin remains at the same location.

## 7.1.4 Special Cases

### 7.1.4.1 Inverse Matrices

Transforming the space with a matrix can be reversed if the matrix is invertible. In this case, the inverse  $\mathbf{T}^{-1}$  of the matrix  $\mathbf{T}$  is associated to a transformation that takes back the space to the initial state after  $\mathbf{T}$  has been applied.

Let’s take again the example of the transformation associated with the following matrix:

$$\mathbf{T} = \begin{bmatrix} 1.3 & -2.4 \\ 0.1 & 2 \end{bmatrix}$$

You'll plot the initial grid of point, the grid after being transformed by  $\mathbf{T}$ , and the grid after successive application of  $\mathbf{T}$  and  $\mathbf{T}^{-1}$  (remember that matrices must be left-multiplied):

```

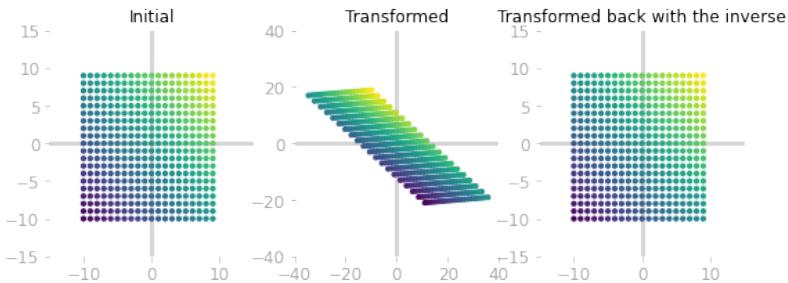
T = np.array([
    [1.3, -2.4],
    [0.1, 2]
])
trans = T @ xy

T_inv = np.linalg.inv(T)

un_trans = T_inv @ T @ xy

f, axes = plt.subplots(1, 3, figsize=(9, 3))
axes[0].scatter(xx, yy, s=10, c=xx+yy)
axes[1].scatter(trans[0].reshape(xx.shape), trans[1].reshape(yy.shape),
                 s=10, c=xx+yy)
axes[2].scatter(un_trans[0].reshape(xx.shape),
                 un_trans[1].reshape(yy.shape), s=10, c=xx+yy)

# [...] Add axis, x and y with the same scale
    
```



*Figure 7.5: Inverse of a transformation: the initial space (left) is transformed with the matrix  $\mathbf{T}$  (middle) and transformed back using  $\mathbf{T}^{-1}$  (right).*

As you can see in Figure 7.5, the inverse  $\mathbf{T}^{-1}$  of the matrix  $\mathbf{T}$  is associated with a transformation that reverses the one associated with  $\mathbf{T}$ .

Mathematically, the transformation of a vector  $\mathbf{v}$  by  $\mathbf{T}$  is defined as:

$$\mathbf{T}\mathbf{v}$$

To transform it back, you multiply by the inverse of  $\mathbf{T}$ :

$$\mathbf{T}^{-1}\mathbf{T}\mathbf{v}$$

### Order of the matrix products

Note that the order of the products is from right to left. The vector on the right of the product is first transformed by  $\mathbf{T}$  and then the result is transformed by  $\mathbf{T}^{-1}$ .

Since you saw in Section 6.4.4 that  $\mathbf{T}^{-1}\mathbf{T} = \mathbf{I}$ , you have:

$$\mathbf{T}^{-1}\mathbf{T}\mathbf{v} = \mathbf{I}\mathbf{v} = \mathbf{v}$$

meaning that you get back the initial vector  $\mathbf{v}$ .

#### 7.1.4.2 Non Invertible Matrices

The linear transformation associated with a singular matrix (that is a non invertible matrix, see more details in Section 6.4.4) can't be reversed. It can occur when there is a loss of information with the transformation. Take the following matrix:

$$\mathbf{T} = \begin{bmatrix} 3 & 6 \\ 2 & 4 \end{bmatrix}$$

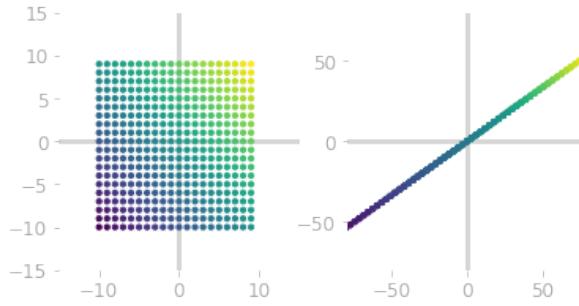
Let's see how it transforms the space:

```

T = np.array([
    [3, 6],
    [2, 4],
])
trans = T @ xy

f, axes = plt.subplots(1, 2, figsize=(6, 3))
axes[0].scatter(xx, yy, s=10, c=xx+yy)
axes[1].scatter(trans[0].reshape(xx.shape), trans[1].reshape(yy.shape),
    s=10, c=xx+yy)
# [...] Add axis, x and y with the same scale

```



*Figure 7.6: The initial space (left) is transformed into a line (right) with the matrix  $\mathbf{T}$ . Multiple input vectors land on the same location in the output space.*

You can see in Figure 7.6 that the transformed vectors are on a line. There are points that land on the same place after the transformation. Thus, it is not possible to go back. In this case, the matrix  $\mathbf{T}$  is not invertible: it is singular.

## 7.2 Linear combination

### 7.2.1 Intuition

Adding and multiplying vectors by scalars are two major operations in linear algebra. You saw that they must be preserved for a transformation to be called linear. The *linear combination* of two vectors corresponds to the addition of a scaled version of these vector.

Let's take an example with the following vectors  $\mathbf{u}$  and  $\mathbf{v}$ :

$$\mathbf{u} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

and

$$\mathbf{v} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

For instance, the operation  $3\mathbf{u} + 4\mathbf{v}$  is called a linear combination of the two vectors  $\mathbf{u}$  and  $\mathbf{v}$ . The values that scale each vector are sometimes called the *weights*. The linear combination of two vectors returns a new vector. Algebraically, you can calculate this linear combination as follows:

$$3\mathbf{u} + 4\mathbf{v} = 3 \begin{bmatrix} 2 \\ 3 \end{bmatrix} + 4 \begin{bmatrix} 3 \\ -1 \end{bmatrix} = \begin{bmatrix} 3 \cdot 2 \\ 3 \cdot 3 \end{bmatrix} + \begin{bmatrix} 4 \cdot 3 \\ 4 \cdot -1 \end{bmatrix} = \begin{bmatrix} 6 + 12 \\ 9 - 4 \end{bmatrix} = \begin{bmatrix} 18 \\ 5 \end{bmatrix}$$

The linear combination of  $\mathbf{u}$  and  $\mathbf{v}$  gives you a new vector  $(18, 5)$ .

#### 7.2.1.1 Matrix as Column Vectors

Another way to express a linear combination is to store the vectors in a matrix and the weights in a vector.

From the previous example, you can have a matrix  $\mathbf{A}$  containing the two vectors  $\mathbf{u}$  and  $\mathbf{v}$ :

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 3 & -1 \end{bmatrix}$$

The linear combination can be expressed as the product of the matrix  $\mathbf{A}$  and the weights applied to these vectors. You would have:

$$\mathbf{A} \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 3 & -1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 + 3 \cdot 4 \\ 3 \cdot 3 + (-1) \cdot 4 \end{bmatrix} = \begin{bmatrix} 18 \\ 5 \end{bmatrix}$$

In this perspective, the matrix-vector product corresponds to a linear combination of the matrix columns. The matrix  $\mathbf{A}$  contains two column vectors and the vector contains weights.

### 7.2.1.2 More than Two Dimensions

The preceding examples used vectors with two components ( $v_x$  and  $v_y$ ). However, linear combinations can be done with vectors with more components (higher dimensions).

Let's take an example with the following three-dimensional vectors  $\mathbf{u}$  and  $\mathbf{v}$ :

$$\mathbf{u} = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix}$$

and

$$\mathbf{v} = \begin{bmatrix} -2 \\ 0 \\ 1 \end{bmatrix}$$

For example,  $-2\mathbf{u} + 3\mathbf{v}$  is a linear combination of  $\mathbf{u}$  and  $\mathbf{v}$ , and is calculated as follows:

$$-2\mathbf{u} + 3\mathbf{v} = -2 \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix} + 3 \begin{bmatrix} -2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -6 \\ 2 \end{bmatrix} + \begin{bmatrix} -6 \\ 0 \\ 3 \end{bmatrix} = \begin{bmatrix} -2 - 6 \\ -6 + 0 \\ 2 + 3 \end{bmatrix} = \begin{bmatrix} -8 \\ -6 \\ 5 \end{bmatrix}$$

You can see that the idea is the same: these concepts work for a larger number of dimensions.

### 7.2.2 All combinations of vectors

You have seen examples of specific combinations of vectors like  $-2\mathbf{v} + 3\mathbf{w}$ . You take two vectors and their linear combination results in a third vector. The next step is to think about all the possible combinations for a set of vectors.

#### 7.2.2.1 One Vector

Let's start with a single vector  $\mathbf{u}$ . You can define linear combinations of this vector as:

$$a\mathbf{u}$$

The vector  $\mathbf{u}$  is weighted by a scalar  $a$ . If you represent the vector  $\mathbf{u}$  geometrically,  $a\mathbf{u}$  corresponds to a rescaling of  $\mathbf{u}$ . For this reason, all the combinations of the single vector  $\mathbf{u}$  corresponds to all possible scalings of  $\mathbf{u}$  (all possible values of  $a$ ).

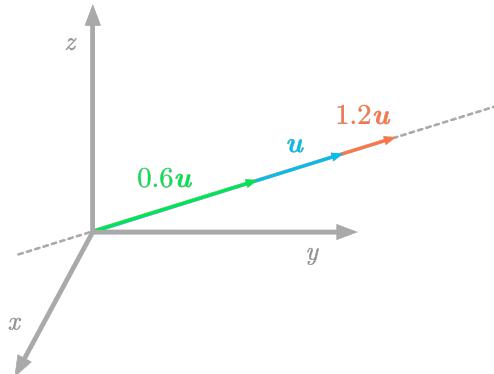


Figure 7.7: Rescaling of the vector  $\mathbf{u}$  with various values of  $a$ .

Figure 7.7 shows an example with a single three-dimensional vector  $\mathbf{u}$ . You can see that all combinations land on the same line (represented as a dotted gray line).

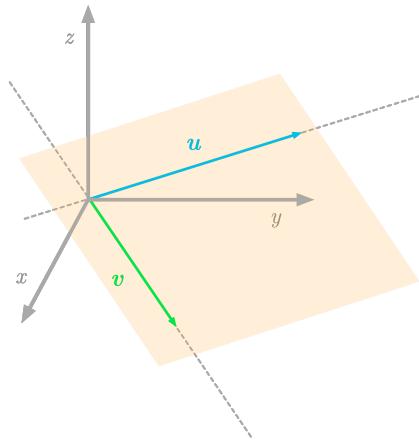
### 7.2.2.2 Two Vectors

You saw earlier that the linear combination of two vectors is algebraically defined by:

$$au + bv$$

If  $\mathbf{u}$  and  $\mathbf{v}$  are on the same line, then all linear combinations will stand on this same line, like in the case of a single vector. You don't gain information since the directions of both vectors are the same (more details in Section 7.4).

However, if they are not on the same line, all combinations will draw a plane.



*Figure 7.8: All linear combinations of two vectors with different directions draw a plane.*

Figure 7.8 illustrates the plane (in yellow) corresponding to all linear combinations of the two vectors  $\mathbf{u}$  and  $\mathbf{v}$ .

### 7.2.2.3 More Vectors

You can go further: what gives you all the linear combinations of three vectors? If the three vectors have the same direction, the linear combinations will still give you a line. If two of the three vectors have the same direction and the third another direction, you'll have a situation similar to linear combinations

of two vectors, and it will draw a plane. Finally, if the three vectors have different directions, all combinations will draw an hyperplane: that is, a plane with more than two dimensions.

### 7.2.3 Span

You saw that you can consider all vectors that can be created by linear combinations of two vectors: the resulting set of vectors is called the *span* of these two vectors.

When linear combinations of a vectors fill the entire space, we say that it spans the space.

The span of a set of vectors is the vector space of all vectors that can be obtained from the linear combinations of these vectors.

You'll see in Section 7.7 practical visualizations of the concept of span.

## 7.3 Subspaces

### 7.3.1 Definitions

A *subspace* is a subset of the initial space that is also a vector space. For instance and as shown in Figure 7.9, if you take a plane in  $\mathbb{R}^3$ , this is a subspace of  $\mathbb{R}^3$  (with the condition that it contains the origin, as you'll see).

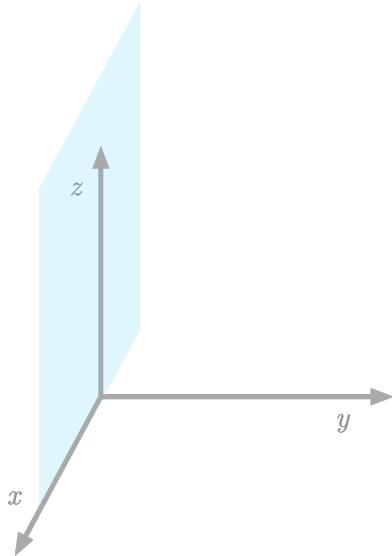


Figure 7.9: Example of a subspace in  $\mathbb{R}^3$ .

This doesn't mean that the subspace is  $\mathbb{R}^2$  because vectors in this subspace still have three components. This plane is a vector space inside  $\mathbb{R}^3$ .

More formally, two rules need to be fulfilled to characterize a set of vectors as a subspace of a vector space:

1. The addition of any two vectors in the subspace gives a vector in this same subspace.
2. The multiplication of a vector in the subspace with any scalar gives a vector in the same subspace.

This means that for any vectors  $\mathbf{v}$  and  $\mathbf{w}$  in the subspace and any scalar  $a$ , you should have  $\mathbf{v} + \mathbf{w}$  and  $a\mathbf{v}$  in the subspace.

We say that this set of vectors is *closed* under addition and scalar multiplication. This means that vectors resulting of linear combinations of vectors in the subspace stay in the subspace.

In addition, the second rule implies that if a vector  $\mathbf{v}$  is in the subspace, then  $0 \cdot \mathbf{v}$  has to be in the subspace as well. This means that the zero vector (see Section 5.1.3.3) has necessarily to be in the subspace.

Another example of a subspace of  $\mathbb{R}^3$  is a line passing through the origin  $(0,$

0, 0). Or, the whole  $\mathbb{R}^3$  space, that is also considered as a subspace of itself because the rules are fulfilled. Finally, the zero vector itself is a subspace of  $\mathbb{R}^3$ .

### Subspaces and subsets of a space

By definition, a subspace satisfies the rules introduced here, while a subset of a space doesn't have to.

## 7.3.2 Subspaces of a Matrix

### 7.3.2.1 Column Space

The *column space* of a matrix is the vector space corresponding to all linear combinations of the column vectors of this matrix.

Let's take the example of the following matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 0 \\ 3 & 2 \end{bmatrix}$$

$\mathbf{A}$  has the two column vectors  $\begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix}$  and  $\begin{bmatrix} 2 \\ 0 \\ 2 \end{bmatrix}$ .

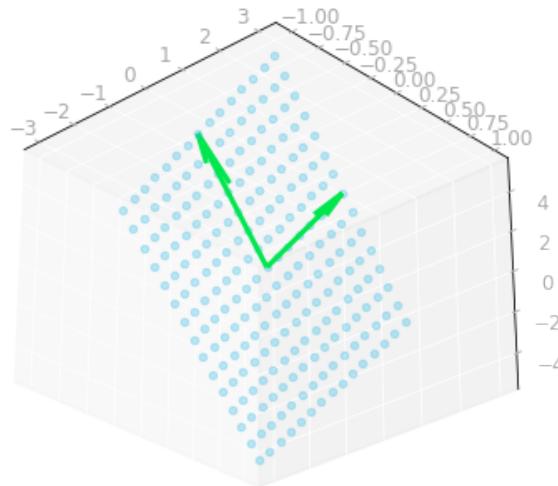
As you saw in Section 6.3.2, if you multiply this matrix with a vector  $\mathbf{x}$ , you have:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 2 \\ -1 & 0 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 0 \\ 2 \end{bmatrix}$$

with  $x_1$  and  $x_2$  being scalars. The new vector  $\mathbf{A}\mathbf{x}$  is a linear combination of the two column vectors with the weights  $x_1$  and  $x_2$ .

Now, consider all possible vectors  $\mathbf{x}$ : it gives you possible linear combinations of the columns of  $\mathbf{A}$ . This set of vectors is the column space of the matrix  $\mathbf{A}$ .

Let's visualize the column space of the matrix  $\mathbf{A}$ :



*Figure 7.10: The plane represented with the blue dots corresponds to linear combinations of the column vectors of  $\mathbf{A}$ . The column vectors are represented as green arrows.*

The plane represented in Figure 7.10 corresponds to the points that you can reach using linear combinations of the column vectors. This subspace is the column space of  $\mathbf{A}$ . You'll see in Section 8.3 how the column space of a matrix is useful to characterize systems of linear equations.

### 7.3.2.2 Row Space

Similarly to the column space, the *row space* is the space corresponding to every linear combinations of the matrix rows. You can also consider the row space of  $\mathbf{A}$  as the column space of  $\mathbf{A}^T$ .

### 7.3.2.3 Transformations

You can also encounter the term *image* or *range* in the context of transformations: the image or range of a transformation is the set of all possible outputs produced by this transformation.

A linear transformation corresponding to a matrix  $\mathbf{A}$  is associated with an *input space* and an *output space*. You'll learn more about that in chapter 9 and chapter 10.

## 7.4 Linear dependency

You have seen that the span of two vectors with the same direction is a line: the information gathered by the vector  $\mathbf{v}$  and  $\mathbf{w}$  is not different.

If both vectors have the same direction (they are on the same line), you can't access locations outside of this line with their linear combination. There is a *linear dependency* between these two vectors.

### Relationship with the inverse

Let's consider a  $m$  by  $n$  matrix  $\mathbf{A}$ : its columns are  $m$ -dimensional vectors. If these column vectors are independent, the matrix is invertible. If they are dependent, then the matrix is singular (more details on invertible and singular matrices in Section 6.4.4).

### 7.4.1 Geometric Interpretation

Let's see the geometric interpretation of linear dependency between vectors. Take the two vectors in a two-dimensional plane illustrated in Figure 7.11.

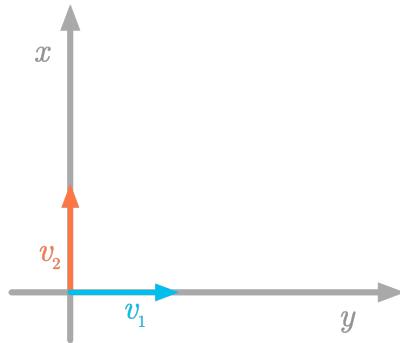


Figure 7.11: Two independent vectors in a two-dimensional space:  $v_1$  and  $v_2$ .

Can you find a linear combination of these two vectors that allows you to start from the origin and come back to the origin (excluding the multiplication of each vector by 0)? If yes, the vectors are dependent. In our example, this is not possible because they are independent. The reason is simply that if vectors are independent, you will go in two different directions, so one of the vector cannot cancel the other.

However, if vectors are dependent, for instance on the same line, you can start from the origin and come back to the origin.

Another way to conceive independence between two vectors is to ask if you can obtain one from the other.

### 7.4.2 Matrix View

Considering a matrix as the concatenation of multiple column vectors, the number of rows  $m$  corresponds to the dimensionality of each vector and the number of columns  $n$  to the number of vectors.

Any set that contains more vectors than dimensions cannot be linearly independent: a  $m$  by  $n$  matrix with  $m < n$  (more columns than rows) necessarily has dependent vectors. For instance, the maximum number of independent vectors in  $\mathbb{R}^2$  is two: you can't imagine a third direction. Similarly, the maximum number of independent vectors in  $\mathbb{R}^3$  is three, and so on.

Take for instance the following matrix  $A$ :

$$\mathbf{A} = \begin{bmatrix} 2 & 7 & -3 \\ 9 & 1 & 6 \end{bmatrix}$$

You can see it as three two-dimensional vectors, and these vectors cannot be linearly independent.

Conversely, if  $m > n$  (more rows than columns), the column vectors of the matrix can't span the space. For instance, two vectors can't span  $\mathbb{R}^3$ , even if they are linearly independent: you will have only two directions in a space where three directions are possible.

## 7.5 Basis

### 7.5.1 Definitions

The *basis* is a coordinate system used to describe vector spaces (sets of vectors). It is a reference that you use to associate numbers with geometric vectors. You'll for instance see in Section 9.2 that the concept of basis is important to understand eigendecomposition.

To be considered as a basis, a set of vectors must:

- Be linearly independent.
- Span the space.

Every vectors in the space is a unique combination of the basis vectors. The dimension of a space is defined to be the size of a basis set. For instance, there are two basis vectors in  $\mathbb{R}^2$  (corresponding to the  $x$  and  $y$  axis in the Cartesian plane), or three in  $\mathbb{R}^3$ .

As you saw in the last section, if the number of vectors in a set is larger than the dimensions of the space, they can't be linearly independent. If a set contains fewer vectors than number of dimensions, they can't span the whole space.

As you saw, vectors can be represented as arrows going from the origin to a point in space. The coordinates of this point can be stored in a list. The geometric representation of a vector in the Cartesian plane implies that we take a reference: the directions given by the two axes  $x$  and  $y$ .

*Basis vectors* are the vectors corresponding to this reference. In the Cartesian plane, the basis vectors are orthogonal unit vectors (length of one), generally denoted as  $\mathbf{i}$  and  $\mathbf{j}$ .

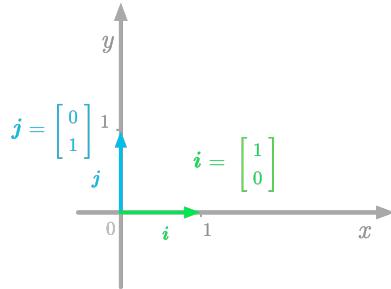


Figure 7.12: The basis vectors in the Cartesian plane.

For instance, in Figure 7.12, the basis vectors  $\mathbf{i}$  and  $\mathbf{j}$  point in the direction of the axis  $x$  and  $y$  respectively. These vectors give the standard basis. If you put these basis vectors into a matrix, you have the following identity matrix (see Section 6.4.3):

$$\mathbf{I}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Thus, the columns of  $\mathbf{I}_2$  span  $\mathbb{R}^2$ . In the same way, the columns of  $\mathbf{I}_3$  span  $\mathbb{R}^3$  and so on.

### Orthogonal basis

Basis vectors can be orthogonal because orthogonal vectors are independent. However, the converse is not necessarily true: non orthogonal vectors can be linearly independent and thus form a basis (but not a standard basis).

The basis of your vector space is very important because the values of the coordinates corresponding to the vectors depends on this basis. By the way,

you can choose different basis vectors, like in the ones in Figure 7.13 for instance.

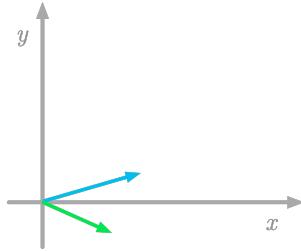


Figure 7.13: Another set of basis vectors.

Keep in mind that vector coordinates depend on an implicit choice of basis vectors.

### 7.5.2 Linear Combination of Basis Vectors

You can consider any vector in a vector space as a linear combination of the basis vectors.

For instance, take the following two-dimensional vector  $\mathbf{v}$ :

$$\mathbf{v} = \begin{bmatrix} 2 \\ -0.5 \end{bmatrix}$$

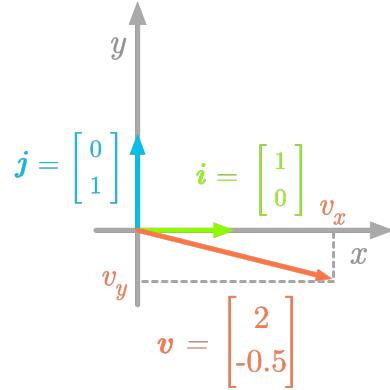


Figure 7.14: Components of the vector  $\mathbf{v}$ .

The components of the vector  $\mathbf{v}$  are the projections on the  $x$ -axis and on the  $y$ -axis ( $v_x$  and  $v_y$ , as illustrated in Figure 7.14). The vector  $\mathbf{v}$  corresponds to the sum of its components:  $\mathbf{v} = v_x \mathbf{i} + v_y \mathbf{j}$ , and you can obtain these components by scaling the basis vectors:  $v_x = 2\mathbf{i}$  and  $v_y = -0.5\mathbf{j}$ . Thus, the vector  $\mathbf{v}$  shown in Figure 7.14 can be considered as a linear combination of the two basis vectors  $\mathbf{i}$  and  $\mathbf{j}$ :

$$\mathbf{v} = 2\mathbf{i} - 0.5\mathbf{j}$$

$$= 2 \begin{bmatrix} 1 \\ 0 \end{bmatrix} - 0.5 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \cdot 1 \\ 2 \cdot 0 \end{bmatrix} - \begin{bmatrix} 0.5 \cdot 0 \\ 0.5 \cdot 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \\ -0.5 \end{bmatrix}$$

### 7.5.3 Other Bases

The columns of identity matrices are not the only case of linearly independent column vectors. It is possible to find other sets of  $n$  vectors linearly independent in  $\mathbb{R}^n$ .

For instance, let's consider the following vectors in  $\mathbb{R}^2$ :

$$\mathbf{v} = \begin{bmatrix} 2 \\ -0.5 \end{bmatrix}$$

and

$$\mathbf{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The vectors  $\mathbf{v}$  and  $\mathbf{w}$  are represented in Figure 7.15.

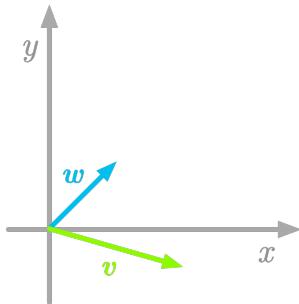


Figure 7.15: Another basis in a two-dimensional space.

From the definition above, the vectors  $\mathbf{v}$  and  $\mathbf{w}$  are a basis because they are linearly independent (you can't obtain one of them from combinations of the other) and they span the space (all the space can be reached from the linear combinations of these vectors).

It is critical to keep in mind that, when you use the components of vectors (for instance  $v_x$  and  $v_y$ , the  $x$  and  $y$  components of the vector  $\mathbf{v}$ ), the values are relative to the basis you chose. If you use another basis, these values will be different.

You'll see later that the ability to change the bases is fundamental in linear algebra and is key to understand eigendecomposition (chapter 9) or Singular Value Decomposition (chapter 10).

#### 7.5.4 Vectors Are Defined With Respect to a Basis

You saw that, to associate geometric vectors (arrows in the space) with coordinate vectors (arrays of numbers), you need a reference. This reference is the basis of your vector space. For this reason, a vector should always be defined with respect to a basis.

Let's take the following vector:

$$\mathbf{v} = \begin{bmatrix} 2 \\ -0.5 \end{bmatrix}$$

The values of the  $x$  and  $y$  components are respectively 2 and -0.5. The standard basis is used when not specified.

You could write  $\mathbf{I}\mathbf{v}$  to specify that these numbers correspond to coordinates with respect to the standard basis. In this case  $\mathbf{I}$  is called the *change of basis matrix*.

$$\mathbf{v} = \mathbf{I}\mathbf{v} = \begin{bmatrix} 2 \\ -0.5 \end{bmatrix}$$

You can define vectors with respect to another basis by using another matrix than  $\mathbf{I}$ . You'll see more about change of basis in Section 9.2.

## 7.6 Special Characteristics

In this last section, you'll see special matrix characteristics needed for more advanced linear algebra: the rank, the trace and the determinant of a matrix.

### 7.6.1 Rank

The *rank* of a matrix  $\mathbf{A}$  (denoted as  $\text{rank}(\mathbf{A})$ ) is the number of linearly independent columns or linearly independent rows. Matrices have always the same number of independent columns and independent rows. This implies that the rank of a matrix  $\mathbf{A}$  is equal to the rank of its transpose  $\mathbf{A}^T$ .

It is possible to use `np.linalg.matrix_rank()` to calculate the rank of a matrix with Numpy.

```
A = np.array([
    [0.4, -2.3, 7.5],
    [-2.3, 1.9, 4.3],
    [7.5, 4.3, 1.0],
])
```

```
np.linalg.matrix_rank(A)
```

3

```
np.linalg.matrix_rank(A.T)
```

3

The matrix  $A$  has rank 3, meaning that it has three linearly independent columns and three independent rows.

Let's take another example:

```
A = np.array([
    [1, 2, 5],
    [2, 4, 8],
    [3, 6, 2],
])
```

```
np.linalg.matrix_rank(A)
```

2

```
np.linalg.matrix_rank(A.T)
```

2

You can see that the first two columns of  $A$  are dependent (the second column corresponds to the first multiplied by two). This means that there are only two independent columns and thus the rank of the matrix is two.

A matrix with the highest possible rank (all columns are independent) is said to have a *full rank*. If the rank is lower, it is said to be *rank deficient*. A rank deficient square matrix is singular, meaning that it has no inverse (see Section 6.4.4).

### 7.6.2 Trace

The *trace* is the sum of the main diagonal values of a square matrix. For instance, take the following matrix  $\mathbf{A}$ :

$$\mathbf{A} = \begin{bmatrix} 5 & 4 & -9 \\ -2 & 3 & 3 \\ 0 & -1 & 7 \end{bmatrix}$$

The trace of the matrix  $\mathbf{A}$  is:

$$\text{Tr}(\mathbf{A}) = 5 + 3 + 7 = 15$$

More formally, for a square matrix  $\mathbf{A}$  of shape  $n \times n$ , you can write:

$$\text{Tr}(\mathbf{A}) = \sum_{i=1}^n \mathbf{A}_{i,i} = \mathbf{A}_{1,1}, \mathbf{A}_{2,2}, \dots, \mathbf{A}_{n,n}$$

The same index for rows and columns ( $i$ ) means that you sum all values from the main diagonal.

#### 7.6.2.1 Expression of the Frobenius Norm

A nice property of the trace operator is that you can use it to express the Frobenius norm of a matrix, as follows:

$$\|\mathbf{A}\|_F = \sqrt{\text{Tr}(\mathbf{A}\mathbf{A}^T)}$$

Let's take an example with the previous matrix  $\mathbf{A}$  and let's start by calculating  $\mathbf{A}\mathbf{A}^T$ :

```
A = np.array([
    [5, 4, -9],
    [-2, 3, 3],
    [0, -1, 7],
])
```

```
A @ A.T
```

```
array([[122, -25, -67],  
       [-25,  22,  18],  
       [-67,  18,  50]])
```

Let's now calculate  $\sqrt{\text{Tr}(\mathbf{A}\mathbf{A}^T)}$ :

```
np.sqrt(np.trace(A @ A.T))
```

```
13.92838827718412
```

Now let's calculate the Frobenius norm from the formula:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} \mathbf{A}_{i,j}^2}$$

You have:

```
np.sqrt(np.sum(A.flatten() ** 2))
```

```
13.92838827718412
```

If you look back at the calculation of  $\mathbf{A}\mathbf{A}^T$ , you can see in Figure 7.16 that each value of the diagonal is the sum of the squared values of the corresponding line. The trace is thus the total sum of the squared values.

$$\begin{aligned}
 \mathbf{A}\mathbf{A}^T &= \begin{bmatrix} 5 & 4 & -9 \\ -2 & 3 & 3 \\ 0 & -1 & 7 \end{bmatrix} \begin{bmatrix} 5 & -2 & 0 \\ 4 & 3 & -1 \\ -9 & 3 & 7 \end{bmatrix} \\
 &= \begin{bmatrix} 5 \cdot 5 + 4 \cdot 4 + -9 \cdot -9 & -2 \cdot 5 + 3 \cdot 4 + 3 \cdot -9 & 0 \cdot 5 + -1 \cdot 4 + 7 \cdot -9 \\ 5 \cdot -2 + 4 \cdot 3 + -9 \cdot 3 & -2 \cdot -2 + 3 \cdot 3 + 3 \cdot 3 & 0 \cdot -2 + -1 \cdot 3 + 7 \cdot 3 \\ 5 \cdot 0 + 4 \cdot -1 + -9 \cdot 7 & -2 \cdot 0 + 3 \cdot -1 + 3 \cdot 7 & 0 \cdot 0 + -1 \cdot -1 + 7 \cdot 7 \end{bmatrix} \\
 &= \begin{bmatrix} 122 & -25 & -67 \\ -25 & 22 & 18 \\ -67 & 18 & 50 \end{bmatrix}
 \end{aligned}$$

Figure 7.16: Using  $\mathbf{A}\mathbf{A}^T$  to calculate the Frobenius norm of  $\mathbf{A}$ .

Since the trace uses only the main diagonal of the matrix, you have:

$$\text{Tr}(\mathbf{A}) = \text{Tr}(\mathbf{A}^T)$$

### 7.6.2.2 Properties

The trace operator has a *cyclic* property: this means that the trace of the product of multiple matrices is equal to the trace of the matrices in another order (a cyclic order). Mathematically written:

$$\text{Tr}(\mathbf{AB}) = \text{Tr}(\mathbf{BA})$$

When there are more than two square matrices multiplied, you have:

$$\text{Tr}(\mathbf{ABC}) = \text{Tr}(\mathbf{CAB}) = \text{Tr}(\mathbf{BCA})$$

In addition, the trace operator has the following properties:

$$\text{Tr}(\mathbf{A} + \mathbf{B}) = \text{Tr}(\mathbf{A}) + \text{Tr}(\mathbf{B})$$

and

$$\text{Tr}(c\mathbf{A}) = c\text{Tr}(\mathbf{A})$$

with  $c$  being a scalar.

### 7.6.3 Determinant

The *determinant* is a function that associates a number to any square matrix. This number tells you information about the matrix. For instance, a matrix with a determinant equals to zero is singular (not invertible).

The determinant of the matrix  $\mathbf{A}$  is denoted as  $\det(\mathbf{A})$  or  $|\mathbf{A}|$ , like the absolute value symbol.

For a two by two matrix, the determinant is calculated by subtracting the product of the diagonals. Given the following matrix  $\mathbf{A}$ :

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The determinant of  $\mathbf{A}$  is  $ad - bc$ . For instance, the determinant of the following singular matrix:

$$\mathbf{A} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}$$

is  $(2 \cdot -2) - (4 \cdot -1) = 0$

#### 7.6.3.1 Geometric Point of View

You saw that you can consider matrices as linear transformations. The determinant of a matrix tells you something about this transformation: it

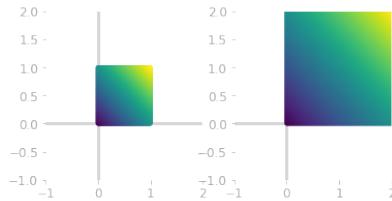
tells you how the area of a shape (or a volume in  $\mathbb{R}^3$ , or a hyper-volume in higher dimensions) changes when the matrix is applied.

Let's take an example with the two-dimensional unit vectors  $i$  and  $j$ : they form a one by one square, so the area of this *unit square* is one.

Now, take the following matrix:

$$\mathbf{T} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Let's visualize the effect of the matrix  $\mathbf{A}$  on the unit square:



*Figure 7.17: Visualization of the unit square (left panel) and the unit square after applying the matrix  $\mathbf{T}$ .*

You can see in Figure 7.17 that the area of the transformed unit square is  $2 \times 2 = 4$ . Let's calculate the determinant of the matrix  $\mathbf{T}$ :

$$\det(\mathbf{T}) = (2 \cdot 2) - (0 \cdot 0) = 4$$

The determinant of the matrix corresponds to the area of the transformed unit square. This means that the area of any shape is multiplied by 4 when the matrix  $\mathbf{T}$  is applied.

### 7.6.3.2 Positive vs Negative Determinant

A positive determinant corresponds to rotations and scalings while a negative determinant corresponds to a change in orientation (like mirror transformations).

### 7.6.3.3 Determinant Greater or Lower than 1

A determinant between 0 and 1 corresponds to a decrease of the area and a determinant greater than 1 to an increase of the area.

#### Determinant of Special Matrices

The determinant of orthogonal matrices is either -1 or 1.

The determinant of diagonal matrices is the product of the diagonal values.

## 7.7 Hands-On Project: Span

Let's visualize the span of two vectors. You will calculate linear combinations of these two vectors and look at the result. Let's start with the basis vectors  $i$  and  $j$  in a two-dimensional space:

$$i = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and

$$j = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Let's create the basis vectors:

```
i = np.array([0, 1])
j = np.array([1, 0])
```

Now, let's calculate linear combinations of the basis vectors:

```

x = np.arange(-10, 10, 1)
y = np.arange(-10, 10, 1)

span_x = np.zeros(x.shape[0] * y.shape[0])
span_y = np.zeros(x.shape[0] * y.shape[0])

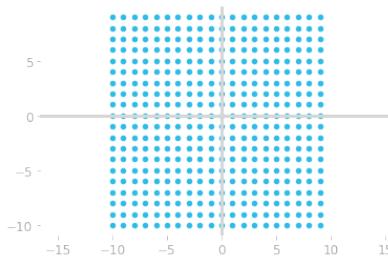
counter = 0
for ii in x:
    for jj in y:
        res = ii * i + jj * j
        span_x[counter] = res[0]
        span_y[counter] = res[1]

    counter += 1
    
```

Now, let's draw the results of these linear combinations.

```

plt.scatter(span_x, span_y, s=20)
# [...] Add axes and styles
    
```



*Figure 7.18: The span is represented as a grid of points resulting from linear combinations of the basis vectors.*

It is like in Section 7.1.3 but without using `meshgrid()` to emphasize how to calculate linear combinations of vectors.

Figure 7.18 shows that you can reach every points in the two-dimensional space with the combinations of the basis vectors. Note that the points only stop because of the finite number of values used in the calculations.

Let's try with another set of vectors (Figure 7.19):

$$\mathbf{v} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

and

$$\mathbf{w} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

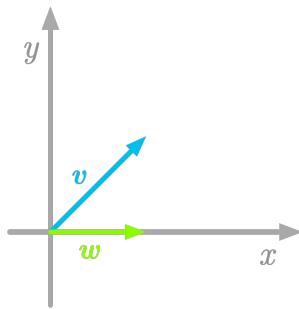


Figure 7.19: The two vectors  $\mathbf{v}$  and  $\mathbf{w}$ .

Similarly, let's calculate the span of  $\mathbf{v}$  and  $\mathbf{w}$ .

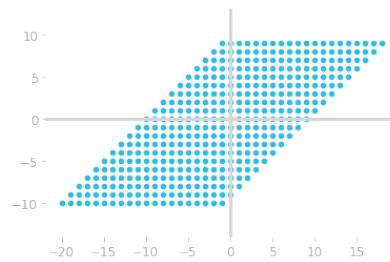


Figure 7.20: The span of the vectors  $\mathbf{v}$  and  $\mathbf{w}$ .

You can see in Figure 7.20 that the span of these non-orthogonal vectors also cover the entire two-dimensional plane.

Finally, you can try with the following vectors that have the same direction:

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

and

$$\mathbf{w} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

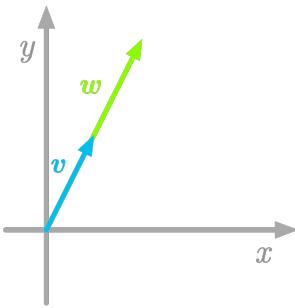


Figure 7.21: The two vectors  $\mathbf{v}$  and  $\mathbf{w}$ .

Let's represent their span:

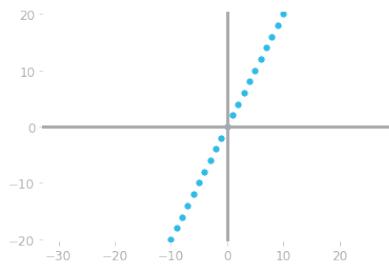


Figure 7.22: The span of two vectors that have the same direction.

For this last example, you can see that the span of  $\mathbf{v}$  and  $\mathbf{w}$  is a line and not a plane.



# Chapter 08

## Systems of Linear Equations

In this chapter, you'll be able to use what you learned about vectors (Section 5.1.1 and chapter 5) and matrices (chapter 6), and linear combinations (Section 7.2). This will allow you to convert data into systems of linear equations. At the end of this chapter, you'll see how you can use systems of equations and linear algebra to solve a linear regression problem (Section 8.4).

Linear equations are formalizations of the relationship between variables. Take the example of a linear relationship between two variables  $x$  and  $y$  defined by the following equation:

$$y = 2x + 1$$

You can represent this relationship in a Cartesian plane:

```
# create x and y vectors
x = np.linspace(-2, 2, 100)
y = 2 * x + 1
plt.plot(x, y)
# [...] Add axes and styles
```

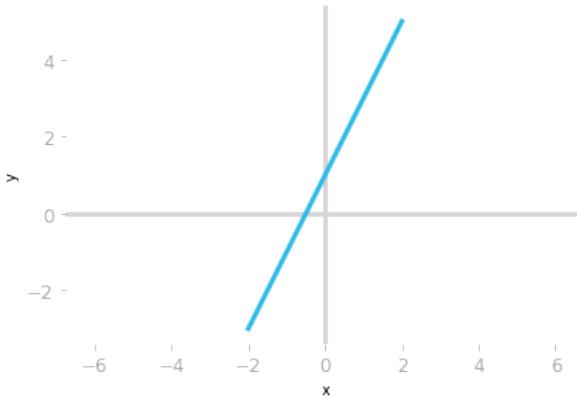


Figure 8.1: Plot of the equation  $y = 2x + 1$ .

Remember that each point on the line corresponds to a solution of this equation: if you replace  $x$  and  $y$  with the coordinates of a point on the line in this equation, the equality is satisfied. This means that there is an infinite number of solutions (every points in the line).

It is also possible to consider more than one linear equation using the same variables: this is a *system of equations*.

## 8.1 System of linear equations

A system of equations is a set of equations describing the relationship between variables. For instance, let's consider the following example:

$$\begin{cases} y &= 2x + 1 \\ y &= -0.5x + 3 \end{cases}$$

You have two linear equations and they both characterize the relationship between the variables  $x$  and  $y$ . This is a system with two equations and two variables (also called *unknowns* in this context).

You can consider systems of linear equations (each row of the system) as multiple equations, each corresponding to a line. This is called the *row picture*.

This is also possible to consider the system as different columns corresponding to coefficients scaling the variables. This is called the *column picture*. Let's see more details about these two pictures.

### 8.1.1 Row Picture

With the row picture, each row of the system corresponds to an equation. In the previous example, there are two equations describing the relationship between two variables  $x$  and  $y$ .

#### 8.1.1.1 Graphical Representation of the Row Picture

Let's represent the two equations graphically:

```
# create x and y vectors
x = np.linspace(-2, 2, 100)
y = 2 * x + 1
y1 = -0.5 * x + 3
plt.plot(x, y)
plt.plot(x, y1)
# [...]
```

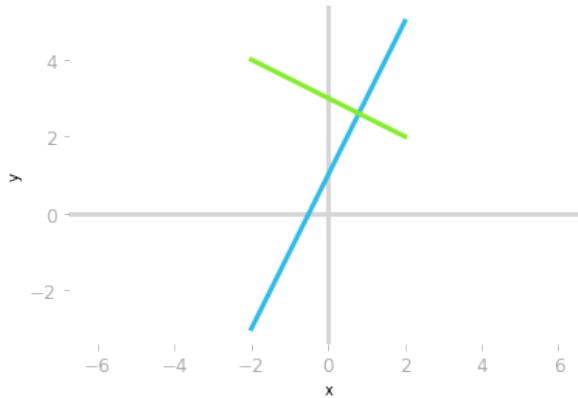


Figure 8.2: Representation of the two equations from our system.

Having more than one equation means that the values of  $x$  and  $y$  must satisfy

more equations. Remember that the  $x$  and  $y$  from the first equation are the same as the  $x$  and  $y$  from the second equation.

All points on the blue line satisfy the first equation and all points on the green line satisfy the second equation. This means that only the point on both lines satisfies the two equations. The system of equations is solved when  $x$  and  $y$  take the values corresponding to the coordinates of the line intersection.

In this example, this point has an  $x$ -coordinate of 0.8 and a  $y$ -coordinate of 2.6. If you replace these values in the system of equations, you have:

$$\begin{cases} 2.6 = 2 \cdot 0.8 + 1 \\ 2.6 = (-0.5) \cdot 0.8 + 3 \end{cases}$$

This is a geometrical way of solving the system of equation. The linear system is solved for  $x = 0.8$  and  $y = 2.6$ .

### 8.1.2 Column Picture

Viewing the system as columns is called the column picture: you consider your system as unknown values ( $x$  and  $y$ ) that scale vectors.

To better see this, let's rearrange the equations to have the variables on one side and the constants on the other side. For the first, you have:

$$y = 2x + 1$$

$$y - 2x = 1$$

and for the second:

$$y = -0.5x + 3$$

$$y + 0.5x = 3$$

You can now write the system as:

$$\begin{cases} y - 2x = 1 \\ y + 0.5x = 3 \end{cases}$$

You can now look at Figure 8.3 to see how to convert the two equations into a single *vector equation*.

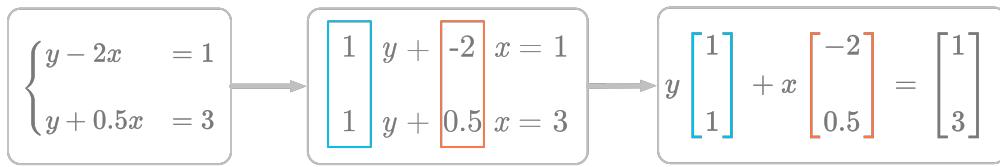


Figure 8.3: Considering the system of equations as column vectors scaled by the variables  $x$  and  $y$ .

In the right of Figure 8.3, you have the vector equation. There are two column vectors in the left-hand side and one column vector in the right-hand side. As you saw in Section 7.2, this corresponds to a linear combination of the following vectors:

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

and

$$\begin{bmatrix} -2 \\ 0.5 \end{bmatrix}$$

With the column picture, you replace multiple equations by a single vector equation. In this perspective, you want to find the linear combination of the left-hand side vectors that gives you the right-hand side vector.

The solution of the column picture is the same. Row and column pictures are just two different ways to consider the system of equations:

$$\begin{aligned}
 2.6 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 0.8 \begin{bmatrix} -2 \\ 0.5 \end{bmatrix} &= \begin{bmatrix} 2.6 \cdot 1 \\ 2.6 \cdot 1 \end{bmatrix} + \begin{bmatrix} 0.8 \cdot (-2) \\ 0.8 \cdot 0.5 \end{bmatrix} \\
 &= \begin{bmatrix} 2.6 \\ 2.6 \end{bmatrix} + \begin{bmatrix} -1.6 \\ 0.4 \end{bmatrix} \\
 &= \begin{bmatrix} 2.6 - 1.6 \\ 2.6 + 0.4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}
 \end{aligned}$$

It works: you got the right-hand side vector by replacing with the solution you found geometrically.

#### 8.1.2.1 Graphical Representation of the Column Picture

Let's represent the system of equations considering it as a linear combination of vectors. Let's take again the previous example:

$$y \begin{bmatrix} 1 \\ 1 \end{bmatrix} + x \begin{bmatrix} -2 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Figure 8.4 shows the graphical representation of the two vectors from the left-hand side (the vectors you want to combine, in blue and red in the picture) and the vector from the right-hand side of the equation (the vector you want to obtain from the linear combination, in green in the picture).

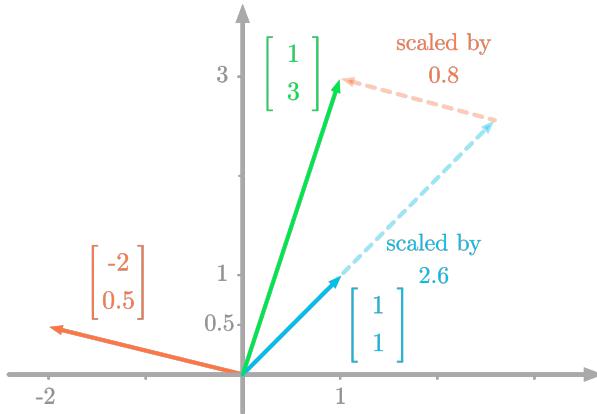


Figure 8.4: Linear combination of the vectors scaled by  $x$  and  $y$  gives the right-hand vector.

You can see in Figure 8.4 that you can reach the right-hand side vector by combining the left-hand side vectors. If you scale the vectors with the values 2.6 and 0.8, the linear combination gets you to the vector in the right-hand side of the equation.

### 8.1.3 Number of Solutions

In some linear systems, there is not a unique solution. Actually, linear systems of equations can have either:

- No solution.
- One solution.
- An infinite number of solutions.

Let's consider these three possibilities (with the row picture and the column picture) to see how it is impossible for a linear system to have more than one solution and less than an infinite number of solutions.

#### 8.1.3.1 Example 1. No Solution

Let's take the following linear system of equations, still with two equations and two variables:

$$\begin{cases} y = 2x + 1 \\ y = 2x + 3 \end{cases}$$

We'll start by representing these equations:

```
# create x and y vectors
x = np.linspace(-2, 2, 100)
y = 2 * x + 1
y1 = 2 * x + 3

plt.plot(x, y)
plt.plot(x, y1)
# [...] Add axes, styles...
```

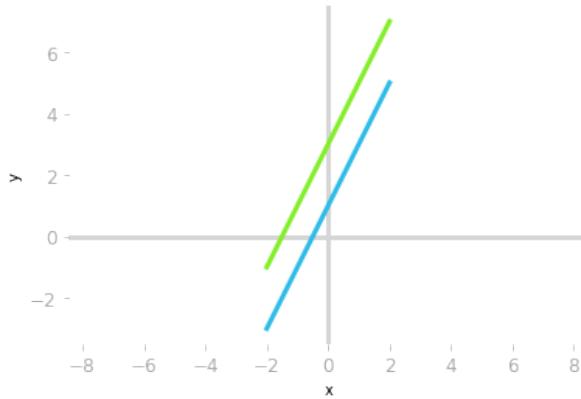


Figure 8.5: Parallel equation lines.

As you can see in Figure 8.5, there is no point that is on both the blue and green lines. This means that this system of equations has no solution.

You can also understand graphically why there is no solution through the column picture. Let's write the system of equations as follows:

$$\begin{cases} y - 2x = 1 \\ y - 2x = 3 \end{cases}$$

Writing it as a linear combination of column vectors, you have:

$$y \begin{bmatrix} 1 \\ 1 \end{bmatrix} + x \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

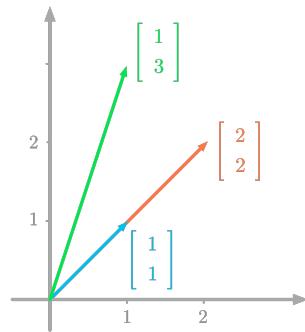


Figure 8.6: Column picture of a linear system with no solution.

Figure 8.6 shows the column vectors of the system. You can see that it is impossible to reach the end point of the green vector by combining the blue and the red vectors. The reason is that these vectors are linearly dependent (more details in Section 7.4). The vector to reach is outside of the span of the vectors you combine.

#### 8.1.3.2 Example 2. Infinite Number of Solutions

You can encounter another situation where the system has an infinite number of solutions. Let's consider the following system:

$$\begin{cases} y = 2x + 1 \\ 2y = 4x + 2 \end{cases}$$

```
# create x and y vectors
x = np.linspace(-2, 2, 100)
y = 2 * x + 1
y1 = (4 * x + 2) / 2

plt.plot(x, y)
plt.plot(x, y1, alpha=0.3)
# [...] Add axes, styles...
```

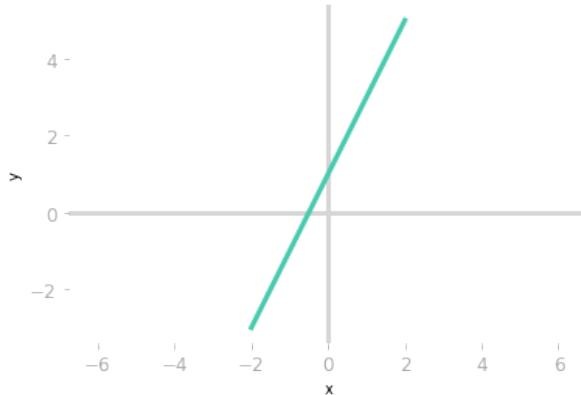


Figure 8.7: The equation lines are overlapping.

Since the equations are the same, an infinite number of points are on both lines and thus, there is an infinite number of solutions for this system of linear equations. This is for instance similar to the case with a single equation and two variables.

From the column picture perspective, you have:

$$\begin{cases} y - 2x &= 1 \\ 2y - 4x &= 2 \end{cases}$$

and with the vector notation:

$$y \begin{bmatrix} 1 \\ 2 \end{bmatrix} + x \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

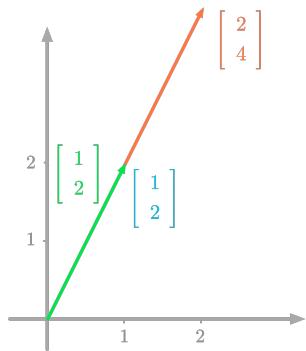


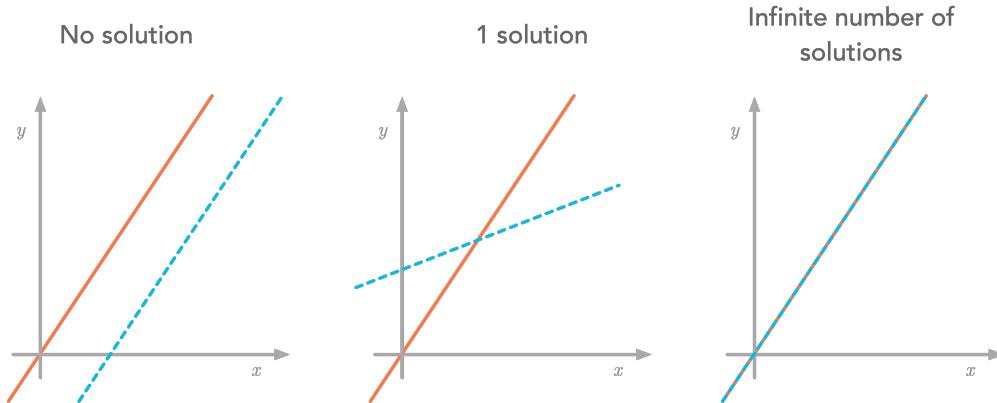
Figure 8.8: Column picture of a linear system with an infinite number of solutions.

Figure 8.8 shows the corresponding vectors graphically represented. You can see that there is an infinite number of ways to reach the end point of the green vector with combinations of the blue and red vectors.

Since both vectors go in the same direction, there is an infinite number of linear combinations allowing you to reach the right-hand side vector.

### 8.1.3.3 Summary

To summarize, you can have three possible situations, shown with two equations and two variables in Figure 8.9.



*Figure 8.9: Summary of the three situations for two equations and two variables.*

It is impossible to have two lines crossing more than once and less than an infinite number of times.

The principle holds for more dimensions. For instance, with three planes in  $\mathbb{R}^3$ , at least two can be parallel (no solution), the three can intersect (one solution), or the three can be superposed (infinite number of solutions).

#### 8.1.4 Representation of Linear Equations With Matrices

Now that you can write vector equations using the column picture, you can go further and use a matrix to store the column vectors.

Let's take again the following linear system:

$$y \begin{bmatrix} 1 \\ 1 \end{bmatrix} + x \begin{bmatrix} -2 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Remember from Section 7.2.1.1 that you can write linear combinations as a matrix-vector product. The matrix corresponds to the two column vectors from the left-hand side concatenated:

$$\begin{bmatrix} 1 & -2 \\ 1 & 0.5 \end{bmatrix}$$

And the vector corresponds to the coefficients weighting the column vectors of the matrix (here,  $x$  and  $y$ ):

$$\begin{bmatrix} y \\ x \end{bmatrix}$$

Your linear system becomes the following matrix equation:

$$\begin{bmatrix} 1 & -2 \\ 1 & 0.5 \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

#### 8.1.4.1 Notation

This leads to the following notation widely used to write linear systems:

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

with  $\mathbf{A}$  the matrix containing the column vectors,  $\mathbf{x}$  the vector of coefficients and  $\mathbf{b}$  the resulting vector, that we'll call the *target vector*. It allows you to go from calculus, where equations are considered separately, to linear algebra, where every pieces of the linear system is represented as vectors and matrices. This abstraction is very powerful and brings vector space theory to solve systems of linear equations.

With the column picture, you want to find the coefficients of the linear combination of the column vectors in the left-hand side of the equation. The solution exists only if the target vector is within their span.

## 8.2 System Shape

Now that you can represent linear systems of equations as matrix equations, you'll be able to leverage what you learn about matrices in this context.

The properties of the matrix  $\mathbf{A}$  in the system  $\mathbf{Ax} = \mathbf{b}$  gives some information about the system and the number of solutions depends of the shape of  $\mathbf{A}$ .

### 8.2.1 Overdetermined Systems of Equations

If  $\mathbf{A}$  has more rows than columns, the system has more equations than unknowns. It is called an *overdetermined system*. The consequence is that there is often no solution to such systems.

To see why, let's consider the case of a matrix  $\mathbf{A}$  of shape  $(3, 2)$ . With the row picture, you have three equations and two unknowns. Figure 8.10 shows an example of this system. There is no solution because the three lines will generally not intersect in a common point.

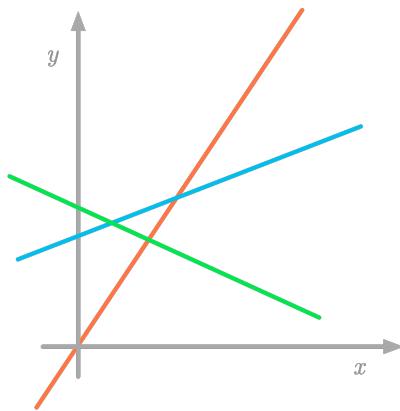


Figure 8.10: An overdetermined system with two unknowns and three equations. There is usually no solution.

Let's take another example: a matrix  $\mathbf{A}$  with a shape  $(4, 3)$ . The row picture, represented in Figure 8.11 shows four planes in three dimensions<sup>1</sup>.

---

<sup>1</sup>The planes were created with Geogebra. You can use the following link to move around the planes and get more insights: <https://www.geogebra.org/3d/pxvnabbx>

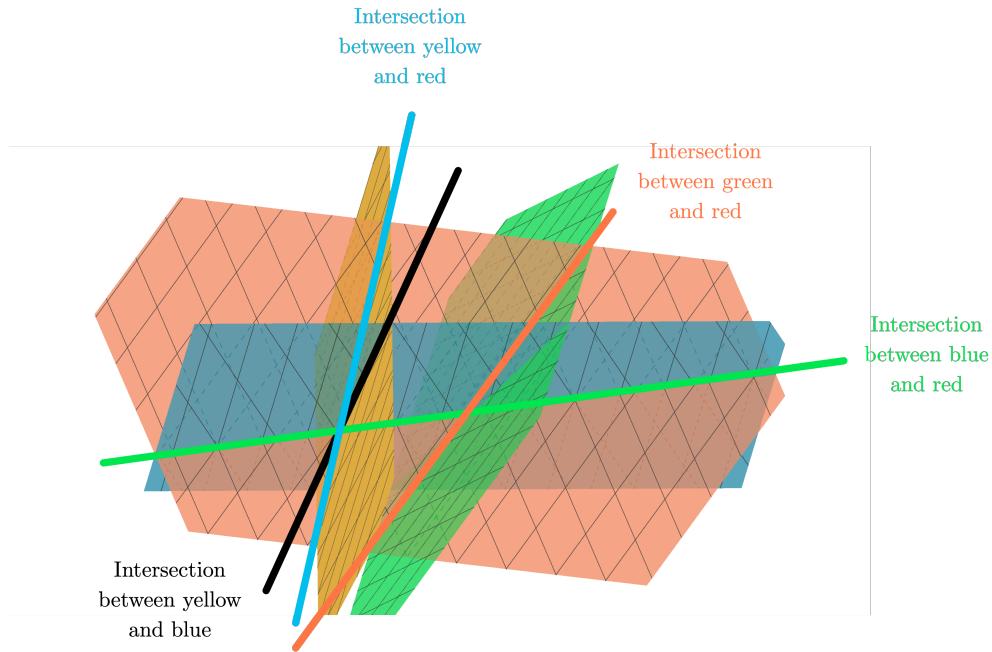


Figure 8.11: Three unknowns and four equations. There is usually no solution.

You can see by looking at the plane intersections in Figure 8.11 that there is no point at the intersection of all planes. This means that there is no solution in this system of linear equations.

The principle is the same for a larger number of dimensions.

### 8.2.2 Underdetermined Systems of Equations

When the system has fewer equations than unknowns ( $A$  has less rows than columns) it is called *underdetermined*. In this case, there is often an infinite number of solutions. In some particular cases, an underdetermined system can have no solution.

Let's take the following system as an example:

$$y = -3x - 4$$

There is a single equation and two unknowns, so the system is underdetermined. Since there is only one equation, all points on the line is a solution: there is an infinite number of points on the line, so there is an infinite number of solutions.

Let's take another example:

$$\begin{cases} z = -2x + 3y + 5 \\ z = 4x + y + 2 \end{cases}$$

In this system, there are two equations and three unknowns: it is underdetermined.

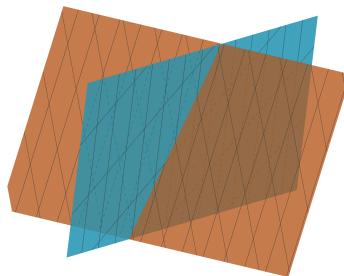


Figure 8.12: Two equations and three unknowns.

Figure 8.12 shows the graphical representation of the two planes in a three-dimensional space<sup>2</sup>. You can see that the intersection of the two planes is a line. This means that all points on this line is a solution to the system of equations: there is an infinite number of solutions.

Let's see another example showing a particular case of an underdetermined system of equations:

$$\begin{cases} z = x + y \\ z = x + y + 10 \end{cases}$$

---

<sup>2</sup>You can also play with it here: <https://www.geogebra.org/3d/dtsjjvsf>.

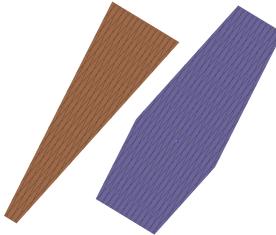


Figure 8.13: Two equations and three unknowns with no solution: planes are parallel.

Figure 8.13 shows a representation of the planes corresponding to these equations<sup>3</sup>. You can see that the two planes are parallel: there is no point at the intersection of these two planes, and thus, the system has no solution.

Even if overdetermined systems have often no solution and underdetermined systems have an infinite number of solutions, there are special situations where this is not the case.

### 8.3 Projections

You saw how to express a system of linear equations with the form  $\mathbf{Ax} = \mathbf{b}$ . However, how can you find a solution or approximate a solution when none exists? The matrix form of the system of equation sometimes allows you to express the solution using the inverse of the matrix  $\mathbf{A}$ .

You'll see that there is a solution if the target vector is in the column space of  $\mathbf{A}$  (more details about column space in Section 7.3.2). If it is not the case, you need to project the target vector to the column space to approximate a solution.

You'll learn how to do these projections and it will lead you to understand a major equation in machine learning and data science: the normal equation.

---

<sup>3</sup>you can play with it here: <https://www.geogebra.org/3d/nmdmvewt>.

### 8.3.1 Solving Systems of Equations

Finding the inverse of a matrix allows you to solve systems of linear equations. As you learned in Section 6.4.4, the multiplication of a matrix with its inverse gives the identity matrix ( $\mathbf{A}\mathbf{I} = \mathbf{A}$ ). Since a matrix multiplied by its inverse gives you the identity matrix ( $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ ), you can use the inverse to isolate an element in a matrix equation. If the matrix  $\mathbf{A}$  has an inverse, you can multiply each side of the linear system by  $\mathbf{A}^{-1}$ :

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

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

This is like with standard equations where you multiply or divide each side to isolate the variable you're interested in. However, be careful and remember that, with matrices and vectors, the order of the multiplication matters.

In the matrix equation, you know  $\mathbf{A}$  and  $\mathbf{b}$  and you want to find  $\mathbf{x}$ . If  $\mathbf{A}$  has an inverse: the equation is solved.

The matrix  $\mathbf{A}$  gives information about the system of equations. For instance, if  $\mathbf{A}$  is invertible (its inverse exists), you know that the set of equations has one and only one solution. In addition, as you learned in Section 7.6.3, a square matrix  $\mathbf{A}$  is singular if and only if its determinant is equal to zero, so if  $\mathbf{A}$  has a non-zero determinant, the system of linear equations has a single solution.

### 8.3.2 Projections to Approximate Unsolvable Systems

A linear system  $\mathbf{A}\mathbf{x} = \mathbf{b}$  is solvable only if the target vector  $\mathbf{b}$  is included in the column space of  $\mathbf{A}$  (the space corresponding to all possible linear combinations of the column vectors of the matrix, as you saw in Section 7.3). Solving the system  $\mathbf{A}\mathbf{x} = \mathbf{b}$  means finding a linear combination of the columns of  $\mathbf{A}$  equal to  $\mathbf{b}$ , the solution being  $\mathbf{x}$ .

However, some sets of linear equations have no solution. This is often the case with overdetermined system of equations (as you saw in Section 8.2),

where the matrix  $\mathbf{A}$  has more rows than column ( $m > n$ ).

Let's take an example with the following set of equations:

$$\begin{cases} y = 2x + 1 \\ y = x - 0.5 \\ y = -2x \end{cases}$$

```
# create x and y vectors
x = np.linspace(-2, 2, 100)
y = 2 * x + 1
y1 = x - 0.5
y2 = -2 * x

plt.plot(x, y)
plt.plot(x, y1)
plt.plot(x, y2)
# [...] Add axes, styles etc.
```

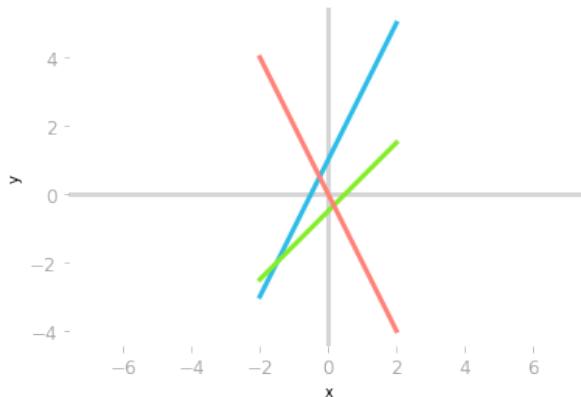


Figure 8.14: Row picture of a unsolvable system of equations.

You can see in Figure 8.14 that the three lines do not intersect in a common point. Writing the system with all variables in the left-hand side, you have:

$$\begin{cases} -2x + y = 1 \\ -x + y = 0.5 \\ 2x + y = 0 \end{cases}$$

The matrix form is  $\mathbf{A}\mathbf{x} = \mathbf{b}$ , with the following  $\mathbf{A}$ ,  $\mathbf{x}$  and  $\mathbf{b}$ :

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

$$\begin{bmatrix} -2 & 1 \\ -1 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \\ 0 \end{bmatrix}$$

Note that we used the letter  $x$  and  $y$  because we represented the equations in a Cartesian plane: the vector  $\mathbf{x}$  contains both values  $x$  and  $y$ .

Let's represent the column vectors of  $\mathbf{A}$  as geometric vectors. They are:

$$\begin{bmatrix} -2 \\ -1 \\ 2 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

We'll also visualize their span: the space containing all points reachable by linear combination of these two vectors. This space is the column space of  $\mathbf{A}$ .

Finally, let's represent the vector  $\mathbf{b}$  corresponding to the position you want to reach, which is defined as:

$$\begin{bmatrix} 1 \\ 0.5 \\ 0 \end{bmatrix}$$

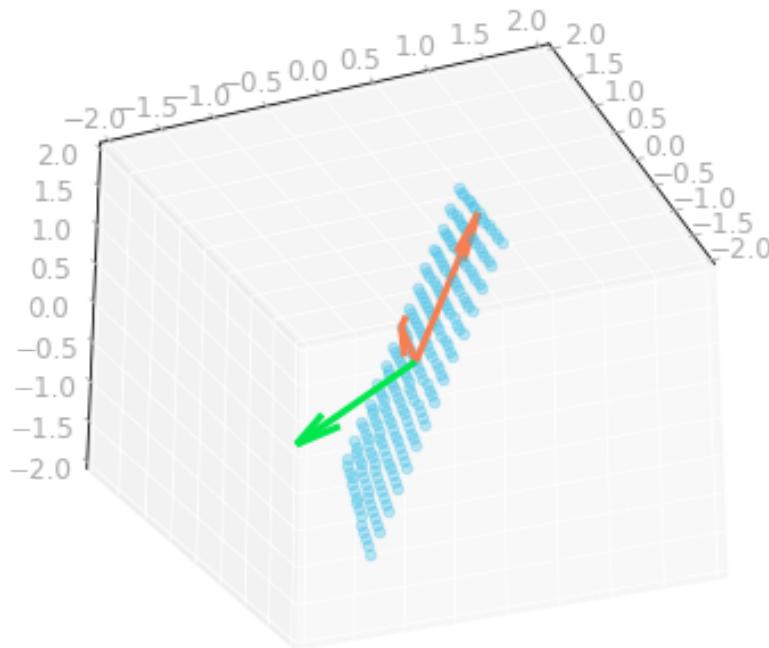


Figure 8.15: The target vector (green) is outside of the span of the column space (blue).

You can see in Figure 8.15 that the target vector you want to reach (in green)

is outside of the column space (the blue plane, corresponding to the span of the column vectors of  $\mathbf{A}$  represented in red). This shows that you can't reach the green vector by linear combinations of the red vectors.

However, you might want to approximate a solution when a true solution doesn't exist. A good candidate is the nearest point to the target vector which is into the plane. This is the green vector projected onto the column space of  $\mathbf{A}$ .

Let's see how projection works by looking at how you can project onto a line, and then onto a plane.

### 8.3.3 Projections Onto a Line

Let's see how to project a vector onto a line.

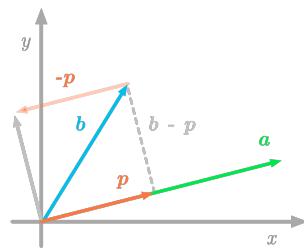


Figure 8.16: Projection onto a line: the vector  $\mathbf{p}$  is the projection of the vector  $\mathbf{b}$  onto the line passing by the vector  $\mathbf{a}$ .

Figure 8.16 shows that the projection line is perpendicular to the vector  $\mathbf{a}$  onto which you project. This projection is called *orthogonal projection*. Let's see how to find the vector  $\mathbf{p}$ .

First, you can see that the projection vector going from  $\mathbf{b}$  to the line passing by  $\mathbf{a}$  (dotted gray line in Figure 8.16) is defined as  $\mathbf{b} - \mathbf{p}$  (the gray solid vector in the figure; the starting point of the vector doesn't matter).

Furthermore, the vector  $\mathbf{p}$  is a scaled version of the vector  $\mathbf{a}$ . Mathematically, you can write:

$$\mathbf{p} = \hat{x}\mathbf{a}$$

with  $\hat{x}$  (pronounced “x hat”) the unknown value scaling the vector  $\mathbf{a}$  to reach the projection. You thus want to find  $\hat{x}$  from  $\mathbf{a}$  and  $\mathbf{b}$ . Since the vectors  $\mathbf{a}$  and  $\mathbf{b} - \mathbf{p}$  are orthogonal, you have:

$$\mathbf{a} \cdot (\mathbf{b} - \mathbf{p}) = 0$$

Replacing  $\mathbf{p}$ , you have:

$$\mathbf{a} \cdot (\mathbf{b} - \hat{x}\mathbf{a}) = 0$$

$$\mathbf{a} \cdot \mathbf{b} - \mathbf{a} \cdot \hat{x}\mathbf{a} = 0$$

$$-\mathbf{a} \cdot \hat{x}\mathbf{a} = -\mathbf{a} \cdot \mathbf{b}$$

$$\mathbf{a} \cdot \hat{x}\mathbf{a} = \mathbf{a} \cdot \mathbf{b}$$

$$\hat{x} = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a} \cdot \mathbf{a}}$$

You have a way to calculate  $\hat{x}$ . If you use this value to rescale the vector  $\mathbf{a}$ , you’ll get the projection vector  $\mathbf{p}$ . So, you can write:

$$\mathbf{p} = \hat{x}\mathbf{a} = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a} \cdot \mathbf{a}}\mathbf{a}$$

This formula allows you to calculate the projection of a vector onto another.

You can find more details about the projection onto a line in the introduction to linear algebra from Gilbert Strang<sup>4</sup>.

**Example** Let’s have the vector

---

<sup>4</sup>See Chapter 4.2, p. 207, in Strang, Gilbert, et al. Introduction to linear algebra, 5th Edition. Wellesley, MA: Wellesley-Cambridge Press, 2016.

$$\mathbf{a} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

and

$$\mathbf{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

You'll project the vector  $\mathbf{b}$  onto the vector  $\mathbf{a}$ . Plotting these vectors you have:

```
ax.quiver(0, 0, 1, 2, color="#2EBCE7", angles='xy', scale_units='xy',
          scale=1)
plt.text(0.2, 1, r'$\vec{b}$', color="#2EBCE7", size=18)

ax.quiver(0, 0, 3, 1, color="#00E64E", angles='xy', scale_units='xy',
          scale=1)
plt.text(1.5, 0.2, r'$\vec{a}$', color="#00E64E", size=18)
# [...] Add axes etc.
```

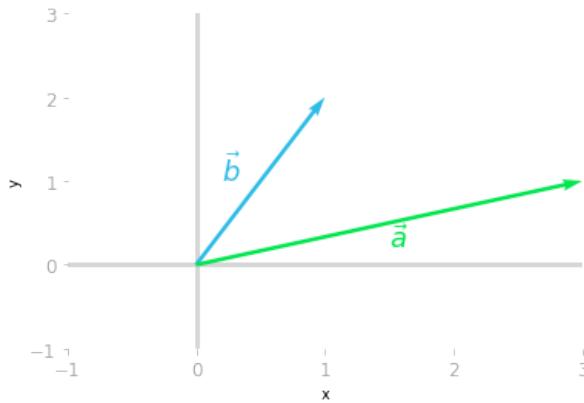


Figure 8.17: Representation of the vectors  $\mathbf{a}$  and  $\mathbf{b}$ .

The projection is  $p = \hat{x}\mathbf{a} = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a} \cdot \mathbf{a}}\mathbf{a}$ , so you have:

$$\frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a} \cdot \mathbf{a}} \mathbf{a} = \frac{\begin{bmatrix} 3 \\ 1 \\ 3 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}}{\begin{bmatrix} 3 \\ 1 \\ 3 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}} \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

$$= \frac{5}{10} \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1.5 \\ 0.5 \end{bmatrix}$$

Let's plot the projection.

```

ax.quiver(0, 0, 1, 2, color="#2EBCE7", angles='xy', scale_units='xy',
          scale=1)
plt.text(0.2, 1, r'$\vec{b}$', color="#2EBCE7", size=18)

ax.quiver(0, 0, 3, 1, color="#00E64E", angles='xy', scale_units='xy',
          scale=1)
plt.text(2, 0.3, r'$\vec{a}$', color="#00E64E", size=18)

ax.quiver(0, 0, 1.5, 0.5, color="#F57F53", angles='xy', scale_units='xy',
          scale=1)
plt.text(1.2, 0.1, r'$\vec{p}$', color="#F57F53", size=18)
# [...] Add axes etc.

```

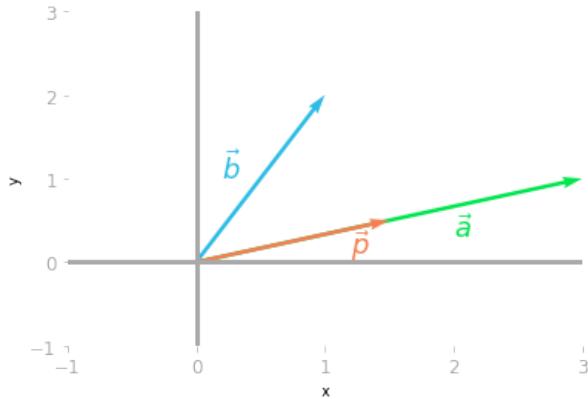


Figure 8.18: The vector  $\mathbf{p}$  is the projection of the vector  $\mathbf{b}$  onto the vector  $\mathbf{a}$ .

It works: you can see in Figure 8.18 that the vector  $\mathbf{p}$  is the projection of the vector  $\mathbf{b}$  onto the line corresponding to the vector  $\mathbf{a}$ .

### 8.3.4 Projections Onto a Plane

To approximate a solution for a system of equations, you can calculate the projection  $\mathbf{p}$  of the vector  $\mathbf{b}$  onto the plane corresponding to the column space of the matrix  $\mathbf{A}$ . You then solve the new equation  $\mathbf{A}\hat{\mathbf{x}} = \mathbf{p}$  which has a solution.

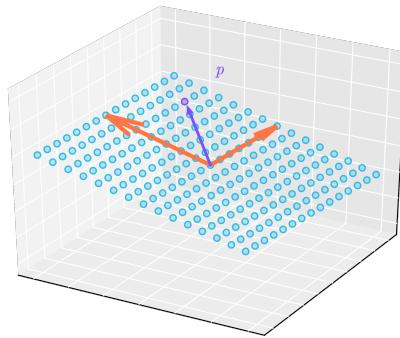


Figure 8.19: The projection vector  $\mathbf{p}$  is in the column space.

The example in Figure 8.19 shows that you can reach the projection (the purple dot) with linear combination of the column vectors (in red).

Let's start from the vector  $\mathbf{p}$ . Mathematically, it is a linear combination of the column vectors of  $\mathbf{A}$ :

$$\mathbf{p} = \hat{x}_1 \mathbf{a}_1 + \cdots + \hat{x}_n \mathbf{a}_n = \mathbf{A}\hat{\mathbf{x}}$$

with  $n$  being the number of columns of  $\mathbf{A}$ ,  $\mathbf{a}_1$  the first column vector,  $\mathbf{a}_n$  the  $n$ th column vector,  $\hat{x}_1$  the value scaling the first column vector, and  $\hat{x}_n$  the value scaling the  $n$ th column vector.

Calculating the projection onto a plane is very similar to the projection onto a line. You also want to find  $\hat{\mathbf{x}}$  (which is a vector of weights instead of a scalar) such as the vector  $\mathbf{p}$  is the closest vector to  $\mathbf{b}$  belonging to the column space. As shown in Figure 8.20, the vector from  $\mathbf{b}$  to  $\mathbf{p}$  (in gray) is perpendicular to the plane.

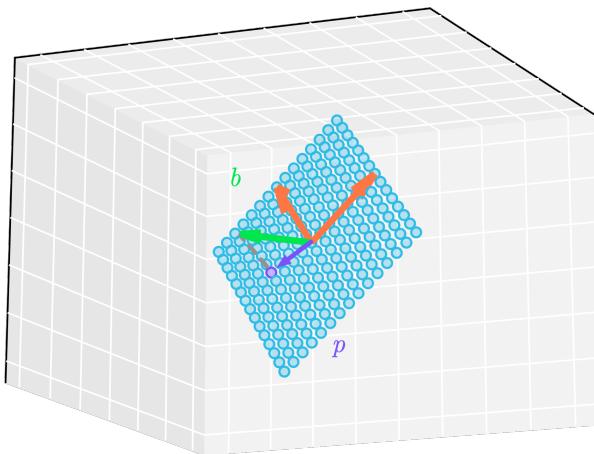


Figure 8.20: Projection of the vector  $\mathbf{b}$  onto the column space.

This means that the dot product between this line and the plane is zero:  $\mathbf{b} - \mathbf{p}$  is perpendicular to every vectors  $\mathbf{a}_1$  to  $\mathbf{a}_n$ :

$$\begin{cases} \mathbf{a}_1 \cdot (\mathbf{b} - \mathbf{p}) = 0 \\ \vdots \\ \mathbf{a}_n \cdot (\mathbf{b} - \mathbf{p}) = 0 \end{cases}$$

The vectors  $\mathbf{a}_1$  to  $\mathbf{a}_n$  are the column vectors from  $\mathbf{A}$ . You can write the equations as matrix product (you need to transpose the vectors  $\mathbf{a}_1$  to  $\mathbf{a}_n$  to have a working matrix product):

$$\begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} [\mathbf{b} - \mathbf{p}] = [0]$$

The matrix containing the vectors  $\mathbf{a}_1$  to  $\mathbf{a}_n$  can be written as:

$$\begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} = [\mathbf{a}_1 \ \cdots \ \mathbf{a}_n]^T = \mathbf{A}^T$$

So you have:

$$\mathbf{A}^T(\mathbf{b} - \mathbf{p}) = 0$$

and since  $p = \mathbf{A}\hat{\mathbf{x}}$ , you can replace and write:

$$\mathbf{A}^T(\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}) = 0$$

$$\mathbf{A}^T\mathbf{b} - \mathbf{A}^T\mathbf{A}\hat{\mathbf{x}} = 0$$

This leads to the famous equation called the *normal equation*:

$$\mathbf{A}^T \mathbf{A} \hat{\mathbf{x}} = \mathbf{A}^T \mathbf{b}$$

The advantage of this equation in comparison to  $\mathbf{A}\mathbf{x} = \mathbf{b}$  is that, if the column vectors of the matrix  $\mathbf{A}$  are independent (see Section 7.4), then  $\mathbf{A}^T \mathbf{A}$  is invertible. Thus, you can get  $\hat{\mathbf{x}}$  by multiplying both sides with the inverse of  $\mathbf{A}^T \mathbf{A}$ : this leads to cancel  $\mathbf{A}^T \mathbf{A}$ , allowing you to find  $\hat{\mathbf{x}}$ :

$$\mathbf{A}^T \mathbf{A} \hat{\mathbf{x}} = \mathbf{A}^T \mathbf{b}$$

$$(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{A} \hat{\mathbf{x}} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$$

$$\hat{\mathbf{x}} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$$

You can use this method to approximate a solution when a system of linear equations has no solution, which is the case for almost all overdetermined systems.

You saw that understanding the column picture of systems of linear equations allows you to write these systems under the matrix form. Then, you saw that you can use projections to approximate a solution to unsolvable systems.

In the next section, you'll see that this is the core of *Least Squares Approximation*.

## 8.4 Hands-on Project: Linear Regression Using Least Squares Approximation

You can use least square approximation to find a line fitting data points. In this hands-on project, you'll learn how to go from data to a matrix equation, which will allow you to use linear algebra on your data and approximate a solution for linear regression problems.

### 8.4.1 Linear Regression Using the Normal Equation

The normal equation is an equation used to find an analytical solution for linear regression, as an alternative to gradient descent.

Let's say that you have two-dimensional data on which you want to fit a line. The goal is to find the parameters of this best-fitting line. Let's call the slope  $\theta_0$  (pronounced "theta zero"), and the  $y$ -intercept  $\theta_1$  (pronounced "theta one").

In this first part, you'll take only a few data points to easily understand how you can convert a data problem into a linear algebra equation.

Take the following three two-dimensional data points:

- A: (0, 0)
- B: (1, 2)
- C: (2, 1)

Let's plot these data points. You have a  $x$  vector with the values 0, 1 and 2, and a  $y$ -vector with the values 0, 2 and 0.

```
x = np.array([0, 1, 2])
y = np.array([0, 2, 1])
plt.scatter(x, y)
# [...] Add axes, styles etc.
```

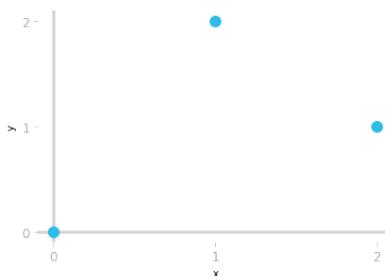


Figure 8.21: Scatter plot of the example data points.

The goal is to find the line passing by these data points. The equation of this line is:

$$\mathbf{y} = \theta_0 \mathbf{x} + \theta_1$$

The first step is to understand how this problem is related to systems of linear equations.

Start with an hypothetical perfect solution: let's say that there is a line passing by all the data points (even if it is not possible). In this case, the first data sample is on the line, meaning that the  $x$  and  $y$  values of this data point (which are 0 and 0) satisfy the equation of your line  $\mathbf{y} = \theta_0 \mathbf{x} + \theta_1 \mathbf{b}$ .

Replacing these values in the equation, you have:

$$0 = \theta_0 \cdot 0 + \theta_1$$

If you proceed identically with the second point, you have:

$$2 = \theta_0 \cdot 1 + \theta_1$$

and for the third point:

$$1 = \theta_0 \cdot 2 + \theta_1$$

With these equations, you have something that resembles a system of equations. You can see that the unknowns  $\theta_0$  and  $\theta_1$  are the parameters of the line. You can write the system as follows:

$$\begin{cases} 0 &= 0 \cdot \theta_0 + \theta_1 \\ 2 &= 1 \cdot \theta_0 + \theta_1 \\ 1 &= 2 \cdot \theta_0 + \theta_1 \end{cases}$$

Or under the matrix form:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

Note that you need to add a column of 1 to represent the  $y$ -intercept.

Also, remember that, since there is no line passing by all the data points, this system has no solution. You'll need to find an approximation.

Let's keep the notation  $\mathbf{A}\mathbf{x} = \mathbf{b}$  with  $A$  being the coefficients,  $\mathbf{x}$  the unknowns ( $\theta_0$  and  $\theta_1$ ) and  $\mathbf{b}$  the solution:

You have

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

and

$$\mathbf{b} = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

Let's write  $\mathbf{A}$  and  $\mathbf{b}$  in Numpy:

```
A = np.array([
    [0, 1],
    [1, 1],
    [2, 1]
])

b = np.array([
    [0],
    [2],
    [1]
])
```

You can now use the normal equation to approximate a solution. As you saw, you have

$$\mathbf{A}^T \mathbf{A} \hat{\mathbf{x}} = \mathbf{A}^T \mathbf{b}$$

and thus:

$$\hat{\mathbf{x}} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$$

The columns of  $\mathbf{A}$  should be linearly independent, and thus  $(\mathbf{A}^T \mathbf{A})^{-1}$  should exist.

### Moore-Penrose inverse

The expression  $(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T$  is one way to calculate what is called the *Moore-Penrose inverse*, or *pseudoinverse* of the matrix  $\mathbf{A}$ . The pseudoinverse is a generalization of the inverse. It is denoted as  $\mathbf{A}^+$ .

Another way to calculate the pseudoinverse is to calculate the Singular Value Decomposition (SVD) of  $\mathbf{A}$  (that you'll see in chapter 10), which is the method used by the Numpy function `np.linalg.inv()` (as documented here: <https://numpy.org/doc/stable/reference/generated/numpy.linalg.pinv.html#numpy.linalg.pinv>).

Let's calculate  $\hat{x}$ :

$$\hat{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$$

$$\hat{x} = \begin{bmatrix} 0.5 & -0.5 \\ -0.5 & 0.83 \end{bmatrix} \begin{bmatrix} 0 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

```
x_hat = np.linalg.inv(A.T @ A) @ A.T @ b
x_hat

array([[0.5],
       [0.5]])
```

This means that the best line has a slope of 0.5 and an intercept of 0.5. Let's plot this line along with the data points:

```
x = np.array([0, 1, 2])
y = np.array([0, 2, 1])

x1 = np.linspace(0, 3, 10)
y1 = 0.5 * x1 + 0.5

plt.scatter(x, y)
plt.plot(x1, y1, c="#F57F53")
# [...] Add axes, styles...
```

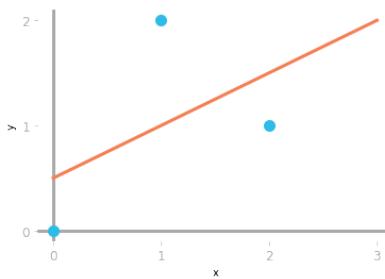


Figure 8.22: Regression line using the normal equation.

The line in Figure 8.22 looks good. You just implemented a linear regression using the normal equation.

This shows an example of how linear algebra concepts as independence, linear combinations, matrix product, column space, inverse of a matrix etc., can be used to deeply understand a very common method: linear regression using the normal equation.

### 8.4.2 Food Data

Now, that you saw how to go from data to matrix equations, let's use least squares approximation with real data: the CIQUAL dataset on food composition<sup>5</sup>.

You'll use the normal equation to model the relationship between the amount of phosphorus and zinc in vegetables and legumes.

Let's start by loading the data:

```
data = pd.read_csv("data/ciqual.csv", sep=";", encoding="latin9",
na_values=['-', 'NaN'])
```

Let's remove non-numeric characters in the data and convert the columns to numeric type.

```
data = data.replace("< ", "", regex=True).replace('traces', 0)
cols = data.loc[:, data.columns != 'alim_ssssgrp_nom_eng'].columns
data[cols] = data[cols].apply(pd.to_numeric, errors='coerce')
```

Now, you'll select only part of the dataset you'll need for the linear regression. Then, remove the missing values:

```
food = ['vegetables. raw',
'vegetables. cooked', 'vegetables. dried or dehydrated',
'legumes. cooked', 'legumes. raw', 'legumes. dried',
```

---

<sup>5</sup>French Agency for Food, Environmental and Occupational Health & Safety. ANSES-CIQUAL French food composition table version 2017. <https://ciqual.anses.fr/>

```
'fresh fruits']

data = data[['alim_ssssgrp_nom_eng', "Phosphorus (mg/100g)", "Zinc
(mg/100g)']]
data = data[(data['alim_ssssgrp_nom_eng'].isin(food))]

data = data.dropna()

data
```

	alim_ssssgrp_nom_eng	Phosphorus (mg/100g)	Zinc (mg/100g)
308	vegetables. raw	44.4	0.460
309	vegetables. raw	10.0	0.070
310	vegetables. raw	32.7	0.220
311	vegetables. raw	85.6	0.650
312	vegetables. raw	37.5	0.610
...	...	...	...
659	fresh fruits	19.0	0.130
660	fresh fruits	17.0	0.080
661	fresh fruits	9.8	0.050
662	fresh fruits	15.8	0.086
663	fresh fruits	32.6	0.220

271 rows × 3 columns

Now, let's visualize a scatter plot of the amount of zinc as a function of the amount of phosphorus (Figure 8.23):

```
plt.scatter(data["Phosphorus (mg/100g)"], data["Zinc (mg/100g)"])
# [...] Add labels
```

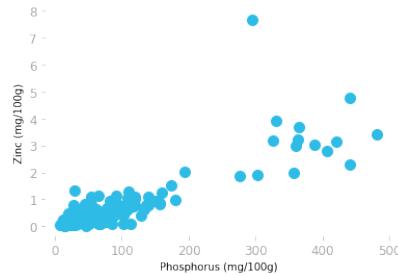


Figure 8.23: Amount of zinc as a function of the amount of phosphorus.

You can then create the matrix and vectors corresponding to the equation  $\mathbf{Ax} = \mathbf{b}$  (the function `to_numpy()` allows you to convert Pandas Series to Numpy arrays):

```
x = data["Phosphorus (mg/100g)"].to_numpy()
b = data["Zinc (mg/100g)"].to_numpy()

A = np.array([x, np.ones(x.shape[0])]).T
A
```

```
array([[44.4,  1. ],
       [10. ,  1. ],
       [32.7,  1. ],
       ...,
       [ 9.8,  1. ],
       [15.8,  1. ],
       [32.6,  1. ]])
```

Now, use the normal equation to calculate  $\hat{x}$ :

```
x_hat = np.linalg.inv(A.T @ A) @ A.T @ b
x_hat
```

```
array([ 0.0084948 , -0.05865873])
```

You can check that it works by plotting the regression line:

```
x_axis = np.arange(0, 500)
y_line = x_hat[0] * x_axis + x_hat[1]

plt.scatter(data["Phosphorus (mg/100g)"].to_numpy(), data["Zinc (mg/100g)"].to_numpy())
plt.plot(y_line, c="#F57F53")
# [...] Add labels
```

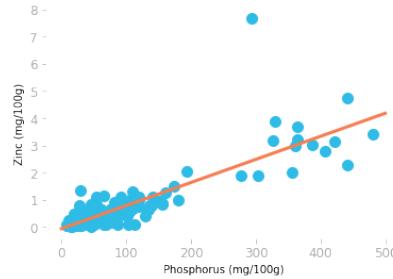


Figure 8.24: Regression line representing the relationship between amount of zinc and phosphorus.

You can see in Figure 8.24 that the line fits the data well.

### 8.4.3 Bonus: Relationship Between Least Squares and the Normal Equation

Why does the normal equation corresponds to the best solution of least squares approximation? The goal of the least squares method is to minimize the squared errors (the differences between the estimations given by the regression line and the true values). You saw that the matrix form of this problem is  $\mathbf{Ax} = \mathbf{b}$ , so you want to minimize the squared difference between  $\mathbf{Ax}$  and  $\mathbf{b}$ . Mathematically, you can write:

$$\operatorname{argmin}((\mathbf{Ax} - \mathbf{b})^2)$$

Remember that the sum of the squared  $L^2$  norm of a vector  $\mathbf{x}$  can be calculated with  $\mathbf{x}^T \mathbf{x}$  (see Section 5.4), so you have:

$$(\mathbf{A}\mathbf{x} - \mathbf{b})^2 = (\mathbf{A}\mathbf{x} - \mathbf{b})^T(\mathbf{A}\mathbf{x} - \mathbf{b})$$

You can consider this expression as a function of  $\mathbf{x}$ , since you want to find the values of  $\mathbf{x}$  that minimize it. As you saw in Section 1.3, you can minimize a function by finding the point where the derivative is equal to zero:

$$\frac{d}{dx}(\mathbf{A}\mathbf{x} - \mathbf{b})^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = 0$$

Let's start by developing what is inside the derivative. Using what you learned about the transposition properties in Section 6.2.2.3 and Section 6.3.3, you can start with the following arrangement:

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

So you have:

$$\frac{d}{dx}(\mathbf{A}\mathbf{x} - \mathbf{b})^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = 0$$

$$\frac{d}{dx}(\mathbf{x}^T \mathbf{A}^T - \mathbf{b}^T)(\mathbf{A}\mathbf{x} - \mathbf{b}) = 0$$

Then, you can develop as follows:

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

Since you'll differentiate with respect to  $\mathbf{x}$ , you can remove the terms that don't contain any  $\mathbf{x}$ :

$$\mathbf{x}^T \mathbf{A}^T \mathbf{A}\mathbf{x} - \mathbf{x}^T \mathbf{A}^T \mathbf{b} - \mathbf{b}^T \mathbf{A}\mathbf{x}$$

Now, look at the shape of  $\mathbf{b}^T \mathbf{A}\mathbf{x}$ . The vector  $\mathbf{b}$  is  $(m, 1)$ , so  $\mathbf{b}^T$  is  $(1, m)$ . The matrix  $\mathbf{A}$  is  $(m, n)$  and the vector  $\mathbf{x}$  is  $(n, 1)$ . So, the shape  $\mathbf{b}^T \mathbf{A}\mathbf{x}$  is:

$$(1, m)(m, n)(n, 1)$$

$$(1, n)(n, 1)$$

$$(1, 1)$$

This means that the shape of  $\mathbf{b}^T \mathbf{A} \mathbf{x}$  is 1: it is a scalar. You can use the fact that a scalar is equal to its transpose:

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

As you saw in Section 6.3.3, the transposition of matrix product is defined as:

$$(\mathbf{ABC})^T = \mathbf{C}^T (\mathbf{AB})^T = \mathbf{C}^T \mathbf{B}^T \mathbf{A}^T$$

So you have:

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

Replacing into the expression you want to minimize:

$$\mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{A}^T \mathbf{b} - \mathbf{b}^T \mathbf{A} \mathbf{x}$$

$$= \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{A}^T \mathbf{b} - \mathbf{x}^T \mathbf{A}^T \mathbf{b}$$

$$= \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{x}^T \mathbf{A}^T \mathbf{b}$$

We'll not detail the differentiation of this expression<sup>6</sup>. It results in:

$$\frac{d}{dx} \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{x}^T \mathbf{A}^T \mathbf{b} = 2\mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{A}^T \mathbf{b}$$

---

<sup>6</sup>You can find more details here: <https://eli.thegreenplace.net/2015/the-normal-equation-and-matrix-calculus/>.

Here it is. You want to set this expression to zero to minimize the error:

$$2\mathbf{A}^T \mathbf{A}\mathbf{x} - 2\mathbf{A}^T \mathbf{b} = 0$$

$$2\mathbf{A}^T \mathbf{A}\mathbf{x} = 2\mathbf{A}^T \mathbf{b}$$

$$\mathbf{A}^T \mathbf{A}\mathbf{x} = \mathbf{A}^T \mathbf{b}$$

This is the normal equation. You can interpret the best approximation obtained with this formula as a projection of the target vector into the column space of  $\mathbf{A}$ , or as a minimization of the distance between the solution and the approximation.



# Chapter 09

## Eigenvectors, Eigenvalues, and Eigendecomposition

*Matrix decomposition*, also called *matrix factorization* is the process of splitting a matrix into multiple pieces. In the context of data science, you can for instance use it to select part of the data, aimed at reducing dimensionality without losing much information (as for instance in Principal Component Analysis, that you'll see in Section 9.5). Some operations are also more easily computed on the matrices resulting from the decomposition.<sup>1</sup>

In this chapter you'll learn about the eigendecomposition of a matrix. One way to understand it is to consider it as a special change of basis. You'll first learn about eigenvectors and eigenvalues and then you'll see the concept of change of basis. The main idea is to consider the eigendecomposition of a matrix  $\mathbf{A}$  as a change of basis where the new basis vectors are the eigenvectors.

### 9.1 Eigenvectors and Eigenvalues

You know from Section 7.1 that you can consider matrices as linear transformations. This means that if you take any vector  $\mathbf{u}$  and apply the matrix  $\mathbf{A}$  to it, you obtain a transformed vector  $\mathbf{v}$ .

---

<sup>1</sup>For instance, raising matrices to power a high power, see No Bullshit Guide to Linear Algebra: Savov, Ivan, ch 7.1.

Take the example of:

$$\mathbf{u} = \begin{bmatrix} 1.5 \\ 1 \end{bmatrix}$$

and

$$\mathbf{A} = \begin{bmatrix} 1.2 & 0.9 \\ 0 & -0.4 \end{bmatrix}$$

If you apply  $\mathbf{A}$  to the vector  $\mathbf{u}$  (with the matrix-vector product), you get a new vector:

$$\mathbf{v} = \mathbf{A}\mathbf{u}$$

$$= \begin{bmatrix} 1.2 & 0.9 \\ 0 & -0.4 \end{bmatrix} \begin{bmatrix} 1.5 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1.2 \cdot 1.5 + 0.9 \cdot 1 \\ 0 \cdot 1.5 + -0.4 \cdot 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2.7 \\ -0.4 \end{bmatrix}$$

Let's draw the initial and transformed vectors:

```
u = np.array([1.5, 1])
A = np.array([
    [1.2, 0.9],
    [0, -0.4]
```

```
])
v = A @ u
```

```
plt.quiver(0, 0, u[0], u[1], color="#2EBCE7", angles='xy',
           scale_units='xy', scale=1)
plt.quiver(0, 0, v[0], v[1], color="#00E64E", angles='xy',
           scale_units='xy', scale=1)
# [...] Add axes, styles, vector names
```

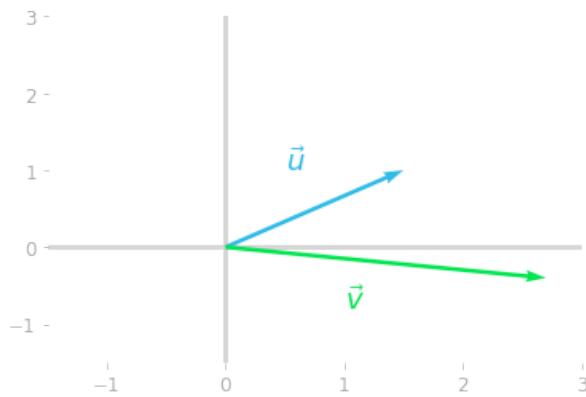


Figure 9.1: Transformation of the vector  $\mathbf{u}$  by the matrix  $\mathbf{A}$  into the vector  $\mathbf{v}$ .

Note that, as you can expect, the transformed vector  $\mathbf{v}$  doesn't run in the same direction as the initial vector  $\mathbf{u}$ . This change of direction characterizes most of the vectors you can transform by  $\mathbf{A}$ .

However, take the following vector:

$$\mathbf{x} = \begin{bmatrix} -0.4902 \\ 0.8715 \end{bmatrix}$$

Let's apply the matrix  $\mathbf{A}$  to the vector  $\mathbf{x}$  to obtain a vector  $\mathbf{y}$ :

```

x = np.array([-0.4902, 0.8715])
y = A @ x
plt.quiver(0, 0, x[0], x[1], color="#2EBCE7", angles='xy',
           scale_units='xy', scale=1)
plt.quiver(0, 0, y[0], y[1], color="#00E64E", angles='xy',
           scale_units='xy', scale=1)
# [...] Add axes, styles, vector names
    
```

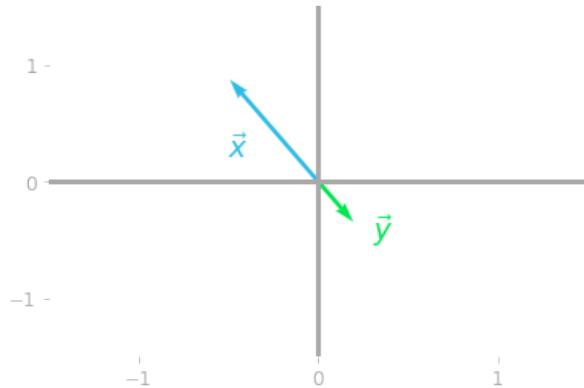


Figure 9.2: Transformation of the special vector  $\mathbf{x}$  by the matrix  $\mathbf{A}$ .

You can see in Figure 9.2 that the vector  $\mathbf{x}$  has a special relationship with the matrix  $\mathbf{A}$ : it is rescaled (with a negative value), but both the initial vector  $\mathbf{x}$  and the transformed vector  $\mathbf{y}$  are on the same line.

The vector  $\mathbf{x}$  is an *eigenvector* of  $\mathbf{A}$ . It is only scaled by a value, which is called an *eigenvalue* of the matrix  $\mathbf{A}$ . An eigenvector of the matrix  $\mathbf{A}$  is a vector that is contracted or elongated when transformed by the matrix. The eigenvalue is the scaling factor by which the vector is contracted or elongated.

Mathematically, the vector  $\mathbf{x}$  is an eigenvector of  $\mathbf{A}$  if:

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

with  $\lambda$  (pronounced “lambda”) being the eigenvalue corresponding to the eigenvector  $\mathbf{x}$ .

## Eigenvectors

Eigenvectors of a matrix are nonzero vectors that are only rescaled when the matrix is applied to them<sup>a</sup>. If the scaling factor is positive, the directions of the initial and the transformed vectors are the same, if it is negative, their directions are reversed.

<sup>a</sup>You can find nice interactive examples here: <https://textbooks.math.gatech.edu/ila/eigenvectors.html>.

### Number of eigenvectors

An  $n$ -by- $n$  matrix has, at most,  $n$  linearly independent eigenvectors. However, each eigenvector multiplied by a nonzero scalar is also an eigenvector. If you have:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

Then:

$$\mathbf{A}c\mathbf{v} = \lambda c\mathbf{v}$$

with  $c$  any nonzero value.

This excludes the zero vector as eigenvector, since you would have

$$\mathbf{A} \cdot 0 = \lambda \cdot 0 = 0$$

In this case, every scalar would be an eigenvalue and thus would be undefined.

## 9.2 Change of Basis

You saw in Section 7.5 that vectors and matrices are defined with respect to a basis. If you take a matrix and change the basis, the numbers inside it change, but the associated linear transformation remains the same. This means that

multiple matrices can be associated with the same linear transformation. In this case, these matrices are called *similar*.

One way to understand eigendecomposition is to consider it as a change of basis. You'll see in this section that a change of basis is a kind of decomposition of the linear transformation: you go to another basis, you do a transformation, and you come back to the initial basis. As you'll see in Section 9.4, with eigendecomposition, you choose the basis such that the new matrix (the one that is similar to the original matrix) becomes diagonal.

You'll start by learning everything you'll need about change of basis and see how linear transformations can be done with respect to different bases.

### 9.2.1 Linear Combinations of the Basis Vectors

Vector spaces (the set of possible vectors) are characterized in reference to a basis. The expression of a geometrical vector as an array of numbers implies that you choose a basis. With a different basis, the same vector  $\mathbf{v}$  is associated with different numbers.

You learned in Section 7.5 that the basis is a set of linearly independent vectors that span the space. More precisely, a set of vectors is a basis if every vector from the space can be described as a finite linear combination of the components of the basis and if the set is linearly independent.

Consider the following two-dimensional vector:

$$\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

In the  $\mathbb{R}^2$  Cartesian plane, you can consider  $\mathbf{v}$  as a linear combination of the standard basis vectors  $\mathbf{i}$  and  $\mathbf{j}$ , as shown in Figure 9.3.

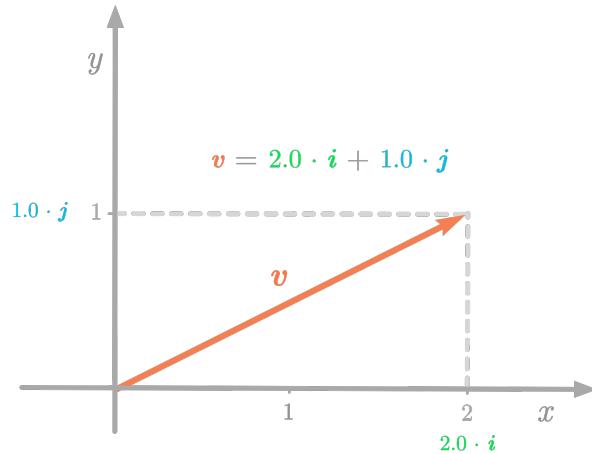


Figure 9.3: The vector  $v$  can be described as a linear combination of the basis vectors  $i$  and  $j$ .

But if you use another coordinate system,  $v$  is associated with new numbers. Figure 9.4 shows a representation of the vector  $v$  with a new coordinate system ( $i'$  and  $j'$ ).

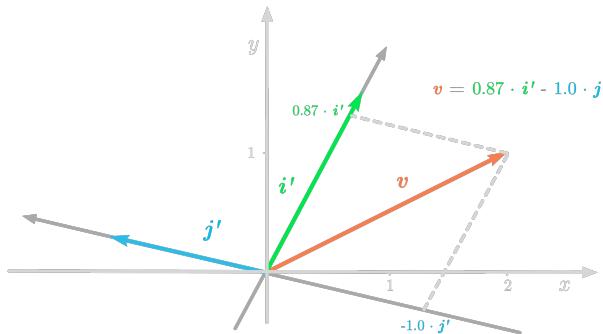


Figure 9.4: The vector  $v$  with respect to the coordinates of the new basis.

In the new basis,  $v$  is a new set of numbers:

$$\begin{bmatrix} 0.86757991 \\ -1.00456621 \end{bmatrix}$$

### 9.2.2 The Change of Basis Matrix

You can use a *change of basis matrix* to go from a basis to another. To find the matrix corresponding to new basis vectors, you can express these new basis vectors ( $\mathbf{i}'$  and  $\mathbf{j}'$ ) as coordinates in the old basis ( $\mathbf{i}$  and  $\mathbf{j}$ ).<sup>2</sup>

Let's take again the preceding example. You have:

$$\mathbf{i}' = \begin{bmatrix} 0.8 \\ 1.5 \end{bmatrix}$$

and

$$\mathbf{j}' = \begin{bmatrix} -1.3 \\ 0.3 \end{bmatrix}$$

This is illustrated in Figure 9.5.

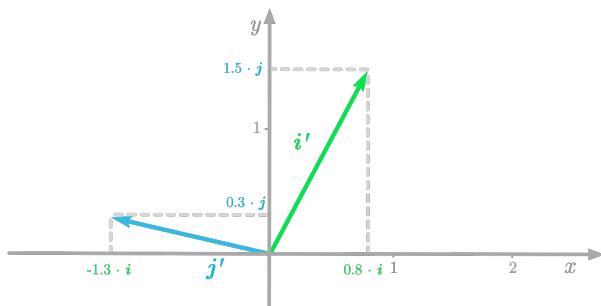


Figure 9.5: The coordinates of the new basis vectors with respect to the old basis.

---

<sup>2</sup>You can find a nice visual introduction to the change of basis from Grant Sanderson in Essence of Linear Algebra, video series by 3Blue1Brown: <https://www.youtube.com/watch?v=P2LTAU01TdA>.

Since they are basis vectors,  $\mathbf{i}'$  and  $\mathbf{j}'$  can be expressed as linear combinations of  $\mathbf{i}$  and  $\mathbf{j}$ :

$$\mathbf{i}' = 0.8 \cdot \mathbf{i} + 1.5\mathbf{j}$$

$$\mathbf{j}' = -1.3 \cdot \mathbf{i} + 0.3\mathbf{j}$$

Let's write these equations under the matrix form:

$$\begin{aligned} \begin{bmatrix} \mathbf{i}' \\ \mathbf{j}' \end{bmatrix} &= \mathbf{i} \begin{bmatrix} 0.8 \\ -1.3 \end{bmatrix} + \mathbf{j} \begin{bmatrix} 1.5 \\ 0.3 \end{bmatrix} \\ &= \begin{bmatrix} 0.8 & 1.5 \\ -1.3 & 0.3 \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ \mathbf{j} \end{bmatrix} \end{aligned}$$

To have the basis vectors as columns, you need to transpose the matrices. You get:

$$\begin{bmatrix} \mathbf{i}' \\ \mathbf{j}' \end{bmatrix}^T = \left( \begin{bmatrix} 0.8 & 1.5 \\ -1.3 & 0.3 \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ \mathbf{j} \end{bmatrix} \right)^T$$

$$\begin{bmatrix} \mathbf{i}' & \mathbf{j}' \end{bmatrix} = \begin{bmatrix} \mathbf{i} \\ \mathbf{j} \end{bmatrix}^T \begin{bmatrix} 0.8 & 1.5 \\ -1.3 & 0.3 \end{bmatrix}^T$$

$$\begin{bmatrix} \mathbf{i}' & \mathbf{j}' \end{bmatrix} = \begin{bmatrix} \mathbf{i} & \mathbf{j} \end{bmatrix} \begin{bmatrix} 0.8 & -1.3 \\ 1.5 & 0.3 \end{bmatrix}$$

This matrix is called the change of basis matrix. Let's call it  $\mathbf{C}$ :

$$\mathbf{C} = \begin{bmatrix} 0.8 & -1.3 \\ 1.5 & 0.3 \end{bmatrix}$$

As you can notice, each column of the change of basis matrix is a basis vector of the new basis. You'll see next that you can use the change of basis matrix  $\mathbf{C}$  to convert vectors from the output basis to the input basis.

### Change of basis vs linear transformation

The difference between change of basis and linear transformation is conceptual. Sometimes it is useful to consider the effect of a matrix as a change of basis; sometimes you get more insights when you think of it as a linear transformation.

Either you move the vector or you move its reference. This is why rotating the coordinate system has an inverse effect compared to rotating the vector itself.

For eigendecomposition and SVD, both of these views are usually taken together, which can be confusing at first<sup>a</sup>. Keeping this difference in mind will be useful throughout the end of the book.

The main technical difference between the two is that change of basis must be invertible, which is not required for linear transformations.

---

<sup>a</sup>You can find more details in the blog of Boris Belousov: <http://www.boris-belousov.net/2016/05/31/change-of-basis>.

#### 9.2.2.1 Finding the Change of Basis Matrix

A change of basis matrix maps an input basis to an output basis. Let's call the input basis  $\mathbf{B}_1$  with the basis vectors  $\mathbf{i}$  and  $\mathbf{j}$ , and the output basis  $\mathbf{B}_2$  with the basis vectors  $\mathbf{i}'$  and  $\mathbf{j}'$ . You have:

$$\mathbf{B}_1 = [\mathbf{i} \ \mathbf{j}]$$

and

$$\mathbf{B}_2 = [\mathbf{i}' \quad \mathbf{j}']$$

From the equation of the change of basis, you have:

$$[\mathbf{i}' \quad \mathbf{j}'] = [\mathbf{i} \quad \mathbf{j}] \mathbf{C}$$

$$\mathbf{B}_2 = \mathbf{B}_1 \mathbf{C}$$

If you want to find the change of basis matrix given  $\mathbf{B}_1$  and  $\mathbf{B}_2$ , you need to calculate the inverse of  $\mathbf{B}_1$  to isolate  $\mathbf{C}$ :

$$\mathbf{B}_2 = \mathbf{B}_1 \mathbf{C}$$

$$\mathbf{B}_1^{-1} \mathbf{B}_2 = \mathbf{B}_1^{-1} \mathbf{B}_1 \mathbf{C}$$

$$\mathbf{B}_1^{-1} \mathbf{B}_2 = \mathbf{C}$$

$$\mathbf{C} = \mathbf{B}_1^{-1} \mathbf{B}_2$$

In words, you can calculate the change of basis matrix by multiplying the inverse of the input basis matrix ( $\mathbf{B}_1^{-1}$ , which contains the input basis vectors as columns) by the output basis matrix ( $\mathbf{B}_2$ , which contains the output basis vectors as columns).

#### Converting vectors from the output to the input basis

Be careful, this change of basis matrix allows you to convert vectors from  $\mathbf{B}_2$  to  $\mathbf{B}_1$  and not the opposite. Intuitively, this is because moving an object is the opposite to moving the reference. Thus, to go from  $\mathbf{B}_1$  to  $\mathbf{B}_2$ , you must use the inverse of the change of basis matrix  $\mathbf{C}^{-1}$ .

Note that if the input basis is the standard basis ( $\mathbf{B}_1 = \mathbf{I}$ ), then the change of basis matrix is simply the output basis matrix:

$$\mathbf{C} = \mathbf{B}_1^{-1} \mathbf{B}_2 = \mathbf{I}^{-1} \mathbf{B}_2 = \mathbf{I} \mathbf{B}_2 = \mathbf{B}_2$$

### Invertible Change of Basis Matrix

Since the basis vectors are linearly independent, the columns of  $\mathbf{C}$  are linearly independent, and thus, as stated in Section 7.4  $\mathbf{C}$  is invertible.

### 9.2.3 Example: Changing the Basis of a Vector

Let's change the basis of a vector  $\mathbf{v}$ , using again the geometric vectors represented in Figure 9.4.

#### 9.2.3.1 Notation

You'll change the basis of  $\mathbf{v}$  from the standard basis to a new basis. Let's denote the standard basis as  $\mathbf{B}_1$  and the new basis as  $\mathbf{B}_2$ . Remember that the basis is a matrix containing the basis vectors as columns. You have:

$$\mathbf{B}_1 = [\mathbf{i} \ \mathbf{j}] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and

$$\mathbf{B}_2 = [\mathbf{i}' \ \mathbf{j}'] = \begin{bmatrix} 0.8 & -1.3 \\ 1.5 & 0.3 \end{bmatrix}$$

Let's denote the vector  $\mathbf{v}$  relative to the basis  $\mathbf{B}_1$  as  $[\mathbf{v}]_{\mathbf{B}_1}$ :

$$[\mathbf{v}]_{\mathcal{B}_1} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

The goal is to find the coordinates of  $\mathbf{v}$  relative to the basis  $\mathcal{B}_2$ , denoted as  $[\mathbf{v}]_{\mathcal{B}_2}$ .

### Square bracket notation

To distinguish the basis used to define a vector, you can put the basis name (like  $\mathcal{B}_1$ ) in subscript after the vector name enclosed in square brackets. For instance,  $[\mathbf{v}]_{\mathcal{B}_1}$  denotes the vector  $\mathbf{v}$  relative to the basis  $\mathcal{B}_1$ , also called the *representation* of  $\mathbf{v}$  with respect to  $\mathcal{B}_1$ .

#### 9.2.3.2 Using Linear Combinations

Let's express the vector  $\mathbf{v}$  as a linear combination of the input and output basis vectors:

$$\begin{cases} \mathbf{v} = c_1 \mathbf{i} + c_2 \mathbf{j} \\ \mathbf{v} = d_1 \mathbf{i}' + d_2 \mathbf{j}' \end{cases}$$

The scalars  $c_1$  and  $c_2$  are weighting the linear combination of the input basis vectors, and the scalars  $d_1$  and  $d_2$  are weighting the linear combination of the output basis vectors. You can merge the two equations:

$$c_1 \mathbf{i} + c_2 \mathbf{j} = d_1 \mathbf{i}' + d_2 \mathbf{j}'$$

Now, let's write this equation in matrix form:

$$[\mathbf{i} \quad \mathbf{j}] \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = [\mathbf{i}' \quad \mathbf{j}'] \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

$$\mathbf{B}_1 \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \mathbf{B}_2 \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

The vector containing the scalars  $c_1$  and  $c_2$  corresponds to  $[\mathbf{v}]_{\mathbf{B}_1}$  and the vector containing the scalars  $d_1$  and  $d_2$  corresponds to  $[\mathbf{v}]_{\mathbf{B}_2}$ . You have:

$$\mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_2[\mathbf{v}]_{\mathbf{B}_2}$$

That's good, this is an equation with the term you want to find:  $[\mathbf{v}]_{\mathbf{B}_2}$ . You can isolate it by multiplying each side by  $\mathbf{B}_2^{-1}$ :

$$\mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_2[\mathbf{v}]_{\mathbf{B}_2}$$

$$\mathbf{B}_2^{-1} \mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_2^{-1} \mathbf{B}_2[\mathbf{v}]_{\mathbf{B}_2} \tag{9.1}$$

$$[\mathbf{v}]_{\mathbf{B}_2} = \mathbf{B}_2^{-1} \mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1}$$

You have also:

$$\mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_2[\mathbf{v}]_{\mathbf{B}_2}$$

$$\mathbf{B}_1^{-1} \mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_1^{-1} \mathbf{B}_2[\mathbf{v}]_{\mathbf{B}_2} \tag{9.2}$$

$$[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_1^{-1} \mathbf{B}_2[\mathbf{v}]_{\mathbf{B}_2}$$

You'll see in the next section that the term  $\mathbf{B}_2^{-1} \mathbf{B}_1$  is the inverse of  $\mathbf{B}_1^{-1} \mathbf{B}_2$ , which is the change of basis matrix  $\mathbf{C}$  described before. This shows that  $\mathbf{C}^{-1}$  allows you to convert vectors from an input basis  $\mathbf{B}_1$  to an output basis  $\mathbf{B}_2$  and  $\mathbf{C}$  from  $\mathbf{B}_2$  to  $\mathbf{B}_1$ .

In the context of this example, since  $\mathbf{B}_1$  is the standard basis, it simplifies to:

$$[\mathbf{v}]_{\mathbf{B}_2} = \mathbf{B}_2^{-1} \mathbf{I} [\mathbf{v}]_{\mathbf{B}_1}$$

$$[\mathbf{v}]_{\mathbf{B}_2} = \mathbf{B}_2^{-1} [\mathbf{v}]_{\mathbf{B}_1}$$

This means that, applying the matrix  $\mathbf{B}_2^{-1}$  to  $[\mathbf{v}]_{\mathbf{B}_1}$  allows you to change its basis to  $\mathbf{B}_2$ .

Let's code this:

```
v_B1 = np.array([2, 1])
B_2 = np.array([
    [0.8, -1.3],
    [1.5, 0.3]
])
v_B2 = np.linalg.inv(B_2) @ v_B1
v_B2

array([ 0.86757991, -1.00456621])
```

These values are the coordinates of the vector  $\mathbf{v}$  relative to the basis  $\mathbf{B}_2$ . This means that if you go to  $0.86757991\mathbf{i}' - 1.00456621\mathbf{j}'$  you arrive to the position  $(2, 1)$  in the standard basis, as illustrated in Figure 9.4.

## 9.3 Linear Transformations in Different Bases

You just saw that vectors are defined by different arrays of numbers according to the basis you take. Similarly, linear transformations are expressed as different matrices, according to the basis. In this section, you'll see how to find the matrix corresponding to a transformation in another basis.

### 9.3.1 Transformations

You know that you can consider matrices as linear transformations. Consider for instance a transformation  $T$  which is a 90-degree clockwise rotation applied

to the vector  $\mathbf{v}$ , illustrated in Figure 9.6.

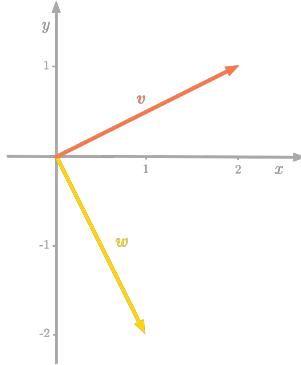


Figure 9.6: The initial vector  $\mathbf{v}$  (in red) and the transformed vector  $\mathbf{w}$  after the 90-degree clockwise rotation (yellow).

The transformation illustrated in Figure 9.6 is associated with the following matrix  $\mathbf{A}$  (its columns are the new basis vectors):

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

The transformed vector  $\mathbf{w}$  has the following coordinates:

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

### 9.3.1.1 Notation

Take an instant to review the elements and their notation.

In the input basis denoted as  $\mathbf{B}_1$ , a transformation  $T$  (for instance, a 90-degree rotation) is associated with the matrix  $\mathbf{A}$ . Let's denote  $\mathbf{D}$  the matrix similar to  $\mathbf{A}$  associated with the same transformation  $T$  relative to the output basis  $\mathbf{B}_2$ .

In addition, let's have the change of basis matrix  $\mathbf{C}$  allowing you to go from the basis  $\mathbf{B}_2$  to the basis  $\mathbf{B}_1$ . The inverse of the change of basis matrix,  $\mathbf{C}^{-1}$ , do the change of basis the other way around: from  $\mathbf{B}_1$  to  $\mathbf{B}_2$ .

You have also an initial vector  $\mathbf{v}$  (before being transformed by  $\mathbf{A}$ ) and a transformed vector  $\mathbf{w}$ .

Using the square bracket notation, you have:

- The vector  $[\mathbf{v}]_{\mathbf{B}_1}$  is the initial vector  $\mathbf{v}$  relative to the basis  $\mathbf{B}_1$ .
- The vector  $[\mathbf{v}]_{\mathbf{B}_2}$  is the same vector  $\mathbf{v}$  relative to the basis  $\mathbf{B}_2$ .
- The vector  $[\mathbf{w}]_{\mathbf{B}_1}$  is the transformed vector  $\mathbf{w}$  relative to the basis  $\mathbf{B}_1$ .
- The vector  $[\mathbf{w}]_{\mathbf{B}_2}$  is the same vector  $\mathbf{w}$  relative to the basis  $\mathbf{B}_2$ .

### 9.3.2 Transformation Matrix in Another Basis

You now have all you need to think about the transformation matrix associated with the transformation  $T$  with respect to different bases. The goal is to find  $\mathbf{D}$ , the matrix corresponding to the transformation  $T$  with respect to the basis  $\mathbf{B}_2$ .

#### 9.3.2.1 Finding the Matrix $\mathbf{D}$

To better understand the relationship between the different elements involved here (the transformation matrices, the change of basis matrices, the initial and transformed vectors etc.), look at the recap illustration in Figure 9.7.

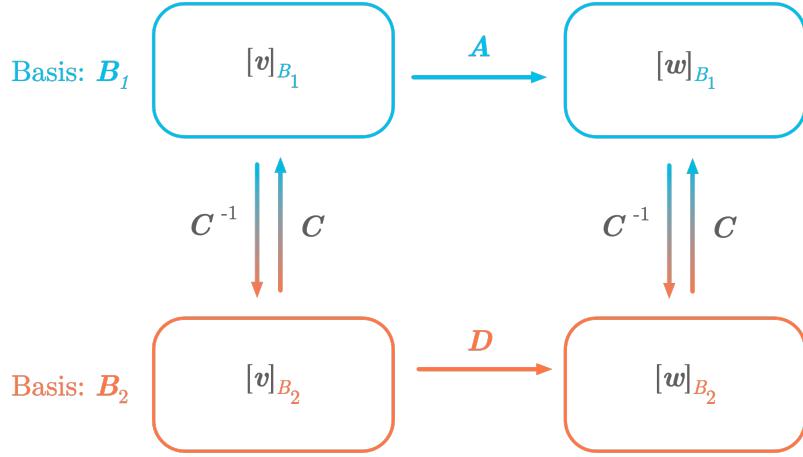


Figure 9.7: Illustration of the transformation converting a vector  $\mathbf{v}$  in a vector  $\mathbf{w}$  represented by the matrix  $\mathbf{A}$  in the basis  $\mathbf{B}_1$  (blue) and by the matrix  $\mathbf{D}$  in the basis  $\mathbf{B}_2$  (red).

You can see that the matrix  $\mathbf{D}$  transforms the initial vector  $\mathbf{v}$  relative to the basis  $\mathbf{B}_2$  ( $[\mathbf{v}]_{\mathbf{B}_2}$ , bottom left) to the transformed vector  $\mathbf{w}$  relative to the basis  $\mathbf{B}_2$  ( $[\mathbf{w}]_{\mathbf{B}_2}$ , bottom right). Mathematically, you have:

$$[\mathbf{w}]_{\mathbf{B}_2} = \mathbf{D}[\mathbf{v}]_{\mathbf{B}_2}$$

However, if you look again at Figure 9.7, you can see that there is another way to go from  $[\mathbf{v}]_{\mathbf{B}_2}$  to  $[\mathbf{w}]_{\mathbf{B}_2}$ : you can go through the basis  $\mathbf{B}_1$  (follow the arrows through the blue part in the figure).

The matrix  $\mathbf{C}$  changes the basis of  $\mathbf{v}$  (from bottom left to upper left):

$$[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{C}[\mathbf{v}]_{\mathbf{B}_2}$$

Then, you apply  $\mathbf{A}$  to transform  $[\mathbf{v}]_{\mathbf{B}_1}$  to  $[\mathbf{w}]_{\mathbf{B}_1}$  (from upper left to upper right):

$$[\mathbf{w}]_{\mathbf{B}_1} = \mathbf{A}[\mathbf{v}]_{\mathbf{B}_1}$$

$$\begin{aligned} [\mathbf{w}]_{B_1} &= \mathbf{A}[\mathbf{v}]_{B_1} \\ &= \mathbf{AC}[\mathbf{v}]_{B_2} \end{aligned}$$

The last step shows that you can replace  $[\mathbf{v}]_{B_1}$  with the expression found in the last equation ( $[\mathbf{v}]_{B_1} = \mathbf{C}[\mathbf{v}]_{B_2}$ ).

Finally, you can change the basis of  $[\mathbf{w}]_{B_1}$  to obtain  $[\mathbf{w}]_{B_2}$  (from upper right to bottom right):

$$\begin{aligned} [\mathbf{w}]_{B_2} &= \mathbf{C}^{-1}[\mathbf{w}]_{B_1} \\ &= \mathbf{C}^{-1}\mathbf{AC}[\mathbf{v}]_{B_2} \end{aligned}$$

You saw that:

$$[\mathbf{w}]_{B_2} = \mathbf{D}[\mathbf{v}]_{B_2}$$

So if you replace into the equation, you have:

$$\mathbf{D}[\mathbf{v}]_{B_2} = \mathbf{C}^{-1}\mathbf{AC}[\mathbf{v}]_{B_2}$$

$$\mathbf{D} = \mathbf{C}^{-1}\mathbf{AC}$$

That's nice: you have the matrix  $\mathbf{D}$ . This equation gives you the relationship between the two matrices  $\mathbf{A}$  and  $\mathbf{D}$  corresponding to the same transformation  $T$  relative with two different bases. You just need to know the change of basis matrix to convert a transformation matrix with respect to the new basis.

The important thing to note is that  $\mathbf{D} = \mathbf{C}^{-1}\mathbf{AC}$  can be interpreted as a decomposition of the matrix  $\mathbf{D}$  into the matrices  $\mathbf{C}$  and  $\mathbf{A}$ . You'll see more about that in the next section about eigendecomposition.

### 9.3.3 Interpretation

You can't use the transformation matrix  $\mathbf{A}$  to apply the transformation  $T$  to a vector in  $\mathbf{B}_2$ , because  $\mathbf{A}$  is with respect to  $\mathbf{B}_1$  and the vector you want to transform is with respect to  $\mathbf{B}_2$ .

The strategy is to follow these steps:

- Convert the vector to the basis  $\mathbf{B}_1$ .
- Apply the transformation corresponding to the matrix  $\mathbf{A}$ .
- Convert the transformed vector back to the basis  $\mathbf{B}_2$ .

These steps are summarized in Figure 9.8 (with the order being from right to left, as with matrix product).

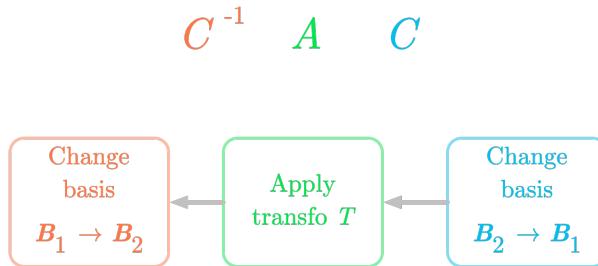


Figure 9.8: Change of basis for a transformation matrix  $\mathbf{A}$ .

Mathematically, it corresponds to applying the change of basis matrix  $\mathbf{C}$  (allowing you to go from  $\mathbf{B}_2$  to  $\mathbf{B}_1$ ), applying the transformation with  $\mathbf{A}$  (because at this time we're in the right basis), and then going back to  $\mathbf{B}_2$  with the inverse of the change of basis matrix.

## 9.4 Eigendecomposition

Eigendecomposition is the expression of a matrix in term of its eigenvectors and eigenvalues. It is a special case of a change of basis: you want to find a basis in which the transformation is associated with a diagonal matrix. As you'll see, this is equivalent to decompose the matrix into eigenvectors and eigenvalues.

### 9.4.1 First Step: Change of Basis

Considering a matrix  $\mathbf{A}$ , the goal of eigendecomposition is to find the basis in which the same transformation is described by a diagonal matrix.

This diagonal matrix is usually denoted as  $\Lambda$  (it is pronounced “capital lambda”, and it corresponds to  $\mathbf{D}$  in our previous example) and the change of basis matrix  $\mathbf{Q}$  (because, as you’ll see, this matrix is orthogonal, and  $\mathbf{Q}$  is a standard name for orthogonal matrices). From the previous section, you have:

$$\Lambda = \mathbf{Q}^{-1} \mathbf{A} \mathbf{Q}$$

You can rearrange this equation as follows to see how  $\mathbf{A}$  is expressed:

$$\mathbf{Q}^{-1} \mathbf{A} \mathbf{Q} = \Lambda$$

$$\mathbf{Q} \mathbf{Q}^{-1} \mathbf{A} \mathbf{Q} = \mathbf{Q} \Lambda$$

$$\mathbf{A} \mathbf{Q} = \mathbf{Q} \Lambda$$

$$\mathbf{A} \mathbf{Q} \mathbf{Q}^{-1} = \mathbf{Q} \Lambda \mathbf{Q}^{-1}$$

$$\mathbf{A} = \mathbf{Q} \Lambda \mathbf{Q}^{-1}$$

The matrix  $\mathbf{A}$  is decomposed into the change of basis matrix  $\mathbf{Q}$  and the diagonal matrix  $\Lambda$ .

### 9.4.2 Eigenvectors and Eigenvalues

But what are these matrices in the context of eigenvectors and eigenvalues? To answer this question, you need to do some customizations to the last equation.

First, you can write the relationship between  $\mathbf{A}$ ,  $\mathbf{Q}$  and  $\Lambda$  as:

$$\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^{-1}$$

$$\mathbf{A}\mathbf{Q} = \mathbf{Q}\Lambda\mathbf{Q}^{-1}\mathbf{Q}$$

$$\mathbf{A}\mathbf{Q} = \mathbf{Q}\Lambda$$

Remember that  $\mathbf{Q}$  is a change of basis matrix. This means that its columns are the basis vectors of the new basis. If  $\mathbf{A}$  is a  $n$  by  $n$  matrix, you have  $n$  basis vectors:

$$\mathbf{Q} = [\mathbf{q}_1 \ \cdots \ \mathbf{q}_n]$$

with  $\mathbf{q}_1$  to  $\mathbf{q}_n$  being the basis vectors of the new basis.

In addition, you're looking for  $\Lambda$  as a diagonal matrix. You'll have:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{bmatrix}$$

If you replace in the equation, you have:

$$\mathbf{A} [\mathbf{q}_1 \ \cdots \ \mathbf{q}_n] = [\mathbf{q}_1 \ \cdots \ \mathbf{q}_n] \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{bmatrix}$$

$$[\mathbf{A}\mathbf{q}_1 \ \cdots \ \mathbf{A}\mathbf{q}_n] = [\lambda_1\mathbf{q}_1 \ \cdots \ \lambda_n\mathbf{q}_n]$$

And this matrix form can be written as the following set of equations:

$$\left\{ \begin{array}{l} \mathbf{A}\mathbf{q}_1 = \lambda_1 \mathbf{q}_1 \\ \vdots \\ \mathbf{A}\mathbf{q}_n = \lambda_n \mathbf{q}_n \end{array} \right.$$

The first equation means that you are looking for a vector  $\mathbf{q}_1$ , which is just rescaled (it doesn't change direction) when you apply  $\mathbf{A}$  to it. This is the definition of an eigenvector that you saw in Section 9.1.

### 9.4.3 Diagonalization

You saw that finding a diagonal matrix similar to  $\mathbf{A}$  in another basis means that the vectors from the change of basis matrix are only rescaled by  $\Lambda$ . This shows the relationship between eigendecomposition and change of basis.

You can consider eigendecomposition as finding a basis where the matrix becomes diagonal. The vectors  $\mathbf{q}_0$  to  $\mathbf{q}_n$  are the eigenvectors of the matrix  $\mathbf{A}$  and  $\lambda_0$  to  $\lambda_n$  are its eigenvalues. The new basis is called the *eigenbasis*.

As you'll see in Section 9.5, this process, also called *diagonalization* is leveraged in PCA where the features of a dataset are transformed such that the components (the transformed features) can be sorted as a function of the amount of variance they explain.

### 9.4.4 Eigendecomposition of Symmetric Matrices

In addition, if the matrix  $\mathbf{A}$  is square and symmetric, its eigenvectors are orthogonal. You know from Section 6.4.5 that if a matrix  $\mathbf{B}$  is orthogonal, then  $\mathbf{B}^{-1} = \mathbf{B}^T$ . This means that  $\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^{-1} = \mathbf{Q}\Lambda\mathbf{Q}^T$ . Keep that in a corner of your mind for the next chapter.

The *spectral theorem* states that if a matrix  $\mathbf{A}$  is symmetric, then it is diagonalizable, that is, there exists an invertible matrix  $\mathbf{Q}$  such that  $\mathbf{Q}^{-1}\mathbf{A}\mathbf{Q}$  which is diagonal.

Finally, note that eigendecomposition can be used only with square matrices. To decompose nonsquare matrices, you'll need to use other decomposition

methods, like Singular Value Decomposition (SVD), as you'll see next in chapter 10.

## 9.5 Hands-On Project: Principal Component Analysis

*Principal Component Analysis*, or PCA, is an algorithm that you can use to reduce the dimensionality of a dataset. It is useful, for instance, to reduce computation time, to compress data, or to avoid what is called *the curse of dimensionality*<sup>3</sup>. It is also useful for visualization purposes: high dimensional data is hard to visualize and it can be useful to decrease the number of dimensions to plot your data.

In this hands-on project, you'll use various concepts that you learned along the book, as change of basis (Section 9.2), eigendecomposition (Section 9.4) or covariance matrices (Section 2.1.3) to understand how PCA is working.

In the first part, you'll learn about the relationship between projections, explained variance and error minimization, first with a bit of theory, and then by coding a PCA on the beer dataset (consumption of beer in relation of temperature). In the second part, you'll use Sklearn to use PCA on audio data to visualize audio samples according to their category, and then to compress these audio samples.

### 9.5.1 Under the Hood

#### 9.5.1.1 Theoretical context

The goal of PCA is to project data onto a lower dimensional space while keeping as much of the information contained in the data as possible. The problem can be seen as a *perpendicular least squares* problem also called *orthogonal regression*<sup>4</sup>.

---

<sup>3</sup>The curse of dimensionality refers to the issues arising in data analysis when the number of dimensions of the dataset increases. Some algorithms are working well with a low number of dimensions but fail when there is a high number of features.

<sup>4</sup>Note that this is different than least squares where the vertical distance is used (the distance according to the dependent variable).

You'll see here that the error of the orthogonal projections is minimized when the projection line corresponds to the direction where the variance of the data is maximal.

**Variance and Projections** It is first important to understand that, when the features of your dataset are not completely uncorrelated, some directions are associated with a larger variance than others.

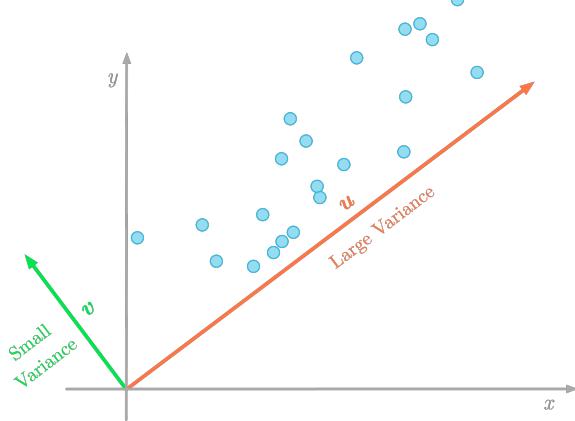
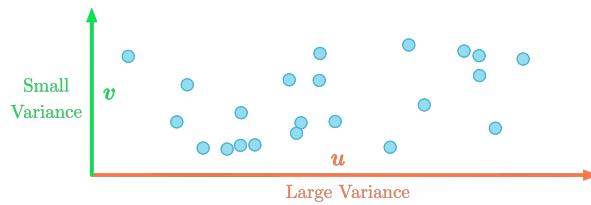


Figure 9.9: The variance of the data (data samples are represented in blue) according to the direction of the vector  $\mathbf{u}$  (red) is associated with a large variance, while the direction of the vector  $\mathbf{v}$  (green) is associated with a smaller variance.

Projecting data to a lower-dimensional space means that you might lose some information. In Figure 9.9, if you project two-dimensional data onto a line, the variance of the projected data tells you how much information you loose. For instance, if the variance of the projected data is near zero, it means that the data points will be projected to very close positions: you lose a lot of information.

For this reason, the goal of the PCA is to change the basis of the data matrix such that the direction with the maximum variance ( $\mathbf{u}$  in Figure 9.9) becomes the first *principal component*. The second component is the direction with the maximum variance which is orthogonal to the first one, and so on.

When you have found the components of the PCA, you change the basis of your data such that the components are the new basis vectors. This transformed dataset has new features, which are the components and which are linear combinations of the initial features. Reducing the dimensionality is done by selecting some of the components only.



*Figure 9.10: Change of basis such that the maximum variance is in the  $x$ -axis.*

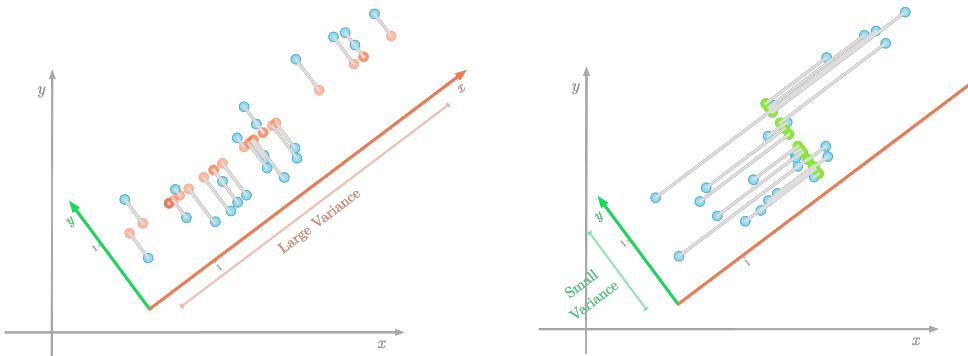
As an illustration, Figure 9.10 shows the data after a change of basis: the maximum variance is now associated with the  $x$ -axis. You can for instance keep only this first dimension.

In other words, expressing the PCA in terms of change of basis, its goal is to find a new basis (which is a linear combination of the initial basis) in which the variance of the data is maximized along the first dimensions.

**Minimizing the Error** Finding the directions that maximize the variance is similar as minimizing the error between the data and its projection<sup>5</sup>.

---

<sup>5</sup>You can read this Stackoverflow thread to have more details on the mathematical explanation of the relationship between maximizing the variance and minimizing the error: <https://stats.stackexchange.com/questions/32174/pca-objective-function-what-is-the-connection-between-maximizing-variance-and-m/136072#136072>



*Figure 9.11: The direction that maximizes the variance is also the one associated with the smallest error (represented in gray).*

You can see in Figure 9.11 that lower errors are shown in the left figure. Since projections are orthogonal, the variance associated to the direction of the line on which you project doesn't impact the error.

**Finding the Best Directions** After changing the basis of the dataset, you should have a covariance between features close to zero (as it is the case for instance in Figure 9.10). In other terms, you want that the transformed dataset has a diagonal covariance matrix: the covariance between each pair of principal components is equal to zero.

You saw in Section 9.4.3, that you can use eigendecomposition to diagonalize a matrix. Thus, you can calculate the eigenvectors of the covariance matrix of the dataset. They will give you the directions of the new basis in which the covariance matrix is diagonal.

To summarize, the principal components are calculated as the eigenvectors of the covariance matrix of the dataset. In addition, the eigenvalues gives you the explained variance of the corresponding eigenvector. Thus, by sorting the eigenvectors in the decreasing order according to their eigenvalues, you can sort the principal components by importance order, and eventually remove the ones associated with a small variance.

### 9.5.1.2 Calculating the PCA

**Dataset** Let's illustrate how PCA is working with the beer dataset showing the beer consumption and the temperature in São Paulo, Brazil for the year 2015<sup>6</sup>.

Let's load the data and plot the consumption as a function of the temperature:

```
data_beer = pd.read_csv("data/beer_dataset.csv")

plt.scatter(data_beer['Temperatura Maxima (C)'],
            data_beer['Consumo de cerveja (litros)'],
            alpha=0.3)
# [...] Add labels and custom axes
```

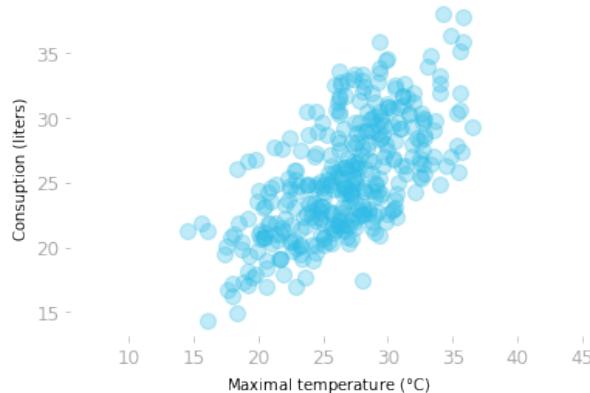


Figure 9.12: Consumption of beer as a function of temperature.

Now, let's create the data matrix  $\mathbf{X}$  with the two variables: temperatures and consumption.

```
X = np.array([data_beer['Temperatura Maxima (C)'],
              data_beer['Consumo de cerveja (litros)']]).T
X.shape
```

---

<sup>6</sup>More details about this dataset here: <https://www.kaggle.com/dongeorge/beer-consumption-sao-paulo>

(365, 2)

The matrix  $\mathbf{X}$  has 365 rows and two columns (the two variables).

**Eigendecomposition of the Covariance Matrix** As you saw, the first step is to compute the covariance matrix of the dataset<sup>7</sup>:

```
C = np.cov(X, rowvar=False)
C

array([[18.63964745, 12.20609082],
       [12.20609082, 19.35245652]])
```

Remember that you can read it as follows: the diagonal values are respectively the variances of the first and the second variable. The covariance between the two variables is around 12.2.

Now, you will calculate the eigenvectors and eigenvalues of this covariance matrix:

```
eigvals, eigvecs = np.linalg.eig(C)
eigvals, eigvecs

(array([ 6.78475896, 31.20734501]),
 array([[-0.71735154, -0.69671139],
       [ 0.69671139, -0.71735154]]))
```

You can store the eigenvectors as two vectors  $\mathbf{u}$  and  $\mathbf{v}$ .

```
u = eigvecs[:, 0].reshape(-1, 1)
v = eigvecs[:, 1].reshape(-1, 1)
```

Let's plot the eigenvectors with the data (note that you should use centered data because it is the data used to calculate the covariance matrix).

You can scale the eigenvectors by their corresponding eigenvalues, which is the explained variance. For visualization purpose, let's use a vector length

---

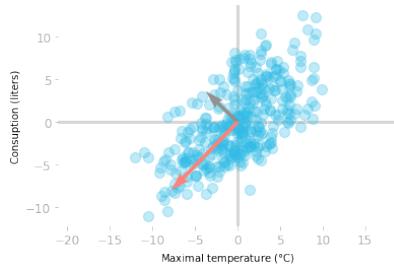
<sup>7</sup>If you calculate the covariance matrix using  $\mathbf{X}^T \mathbf{X}$ , be sure that the data is centered around zero.

of three standard deviations (equal to three times the square root of the explained variance):

```
X_centered = X - X.mean(axis=0)

plt.quiver(0, 0,
           2 * np.sqrt(eigvals[0]) * u[0], 2 * np.sqrt(eigvals[0]) * u[1],
           color="#919191", angles='xy', scale_units='xy', scale=1,
           zorder=2, width=0.011)
plt.quiver(0, 0,
           2 * np.sqrt(eigvals[1]) * v[0], 2 * np.sqrt(eigvals[1]) * v[1],
           color="#FF8177", angles='xy', scale_units='xy', scale=1,
           zorder=2, width=0.011)

plt.scatter(X_centered[:, 0], X_centered[:, 1], alpha=0.3)
# [...] Add axes
```



*Figure 9.13: The eigenvectors  $\mathbf{u}$  (in gray) and  $\mathbf{v}$  (in red) scaled according to the explained variance.*

You can see in Figure 9.13 that the eigenvectors of the covariance matrix give you the important directions of the data. The vector  $\mathbf{v}$  in red is associated with the largest eigenvalue and thus corresponds to the direction with the largest variance. The vector  $\mathbf{u}$  in gray is orthogonal to  $\mathbf{v}$  and is the second principal component.

Then, you just need to change the basis of the data using the eigenvectors as the new basis vectors. But first, you can sort the eigenvectors with respect to the eigenvalues in decreasing order:

```

sort_index = eigvals.argsort()[:, -1]
eigvals_sorted = eigvals[sort_index]
eigvecs_sorted = eigvecs[:, sort_index]
eigvecs_sorted

array([[-0.69671139, -0.71735154],
       [-0.71735154,  0.69671139]])
    
```

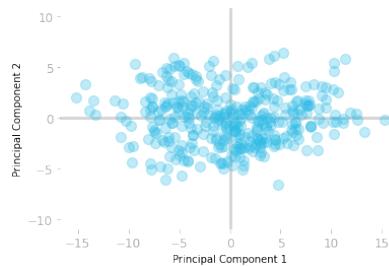
Now that your eigenvectors are sorted, let's change the basis of the data:

```
X_transformed = X_centered @ eigvecs_sorted
```

You can plot the transformed data to check that the principal components are now uncorrelated:

```

plt.scatter(X_transformed[:, 0], X_transformed[:, 1], alpha=0.3)
# [...] Add axes
    
```



*Figure 9.14: The dataset in the new basis.*

Figure 9.14 shows the data samples in the new basis. You can see that the first dimension (the  $x$ -axis) corresponds to the direction with the largest variance.

You can keep only the first component of the data in this new basis without losing too much information.

### Covariance matrix or Singular Value Decomposition?

One caveat of using the covariance matrix to calculate the PCA is that it can be hard to compute when there are many features (as with audio data, like in the second part of this hands-on). For this reason, it is usually preferred to use the Singular Value Decomposition (SVD) to calculate the PCA.

## 9.5.2 Making Sense of Audio

You saw how to use the PCA with two-dimensional data. However, PCA is particularly useful when you deal with high-dimensional data. In the second part of this hands-on project, you'll use it to visualize relationships between categories of audio samples, and then to compress these audio files.

### 9.5.2.1 Audio Data and Preprocessing

Audio data are time series with a value of amplitude for each time sample. The *sampling frequency* is the number of samples per seconds. For instance, many audio files have a sampling frequency of 44,100 Hz, meaning that there are 44,100 amplitude values per second.

**Dataset** You'll use an audio dataset, created for the *Making Sense of Sounds Data Challenge*, where the goal is to classify sounds.<sup>8</sup>

First, download the data here: <https://ndownloader.figshare.com/files/12610922> and unzip the file. You should see a folder called **Development**: put the audio files from the sub-folders (“Music”, “Urban” etc.) in the folder **Development**. Then, create a folder named **audio\_cat** in the folder **data** of the repository “Essential Math for Data Science” and put the folder **Development** and the file **Logsheet\_Development.csv** into it. You should have the following folder structure:

---

<sup>8</sup>You can find more details here: [https://cvssp.org/projects/making\\_sense\\_of\\_sounds/site/challenge/](https://cvssp.org/projects/making_sense_of_sounds/site/challenge/).

## Ch09. Eigenvectors, Eigenvalues, and Eigendecomposition

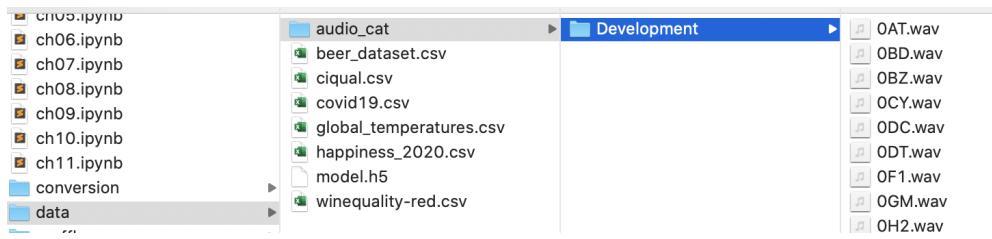


Figure 9.15: Folder structure.

You'll start by loading the `csv` file containing the categories using Pandas and look at random rows:

```
np.random.seed(1234)
categories = pd.read_csv("data/audio_cat/Logsheets_Development.csv")
categories.iloc[np.random.choice(categories.shape[0], 5), :]
```

	Category	Event	File
815	Music	Guitar	LCU.wav
723	Music	Drums	LPO.wav
1318	Urban	Engine	EV1.wav
1077	Nature	Rain	MGO.wav
1228	Urban	Can opening	3AS.wav

You can see that the audio files are categorized as “Music”, “Human” etc. and that the main event is also given, like “Brass”, “Thunderstorm”, etc. Let's encode these events in a way that a computer can understand, for instance with *label encoding*, that is, by associating each event with a number. Let's use Sklearn to do it.

```
from sklearn import preprocessing

le = preprocessing.LabelEncoder()
categories["event_enc"] = le.fit_transform(categories["Event"])
```

You now have a new column `event_enc` in the dataframe with the number

corresponding to the event:

```
categories.iloc[np.random.choice(categories.shape[0], 5), :]
```

	Category	Event	File	event_enc
1396	Urban	Helicopter	IF5.wav	47
664	Music	Brass	Y8L.wav	9
689	Music	Cello	L1C.wav	16
279	Effects	Whoosh	TBF.wav	92
1257	Urban	Chainsaw	VKM.wav	17

**Loading Audio Files** You can use the module `wavefile` from scipy to load the sounds. Let's load a single audio file as a start.

```
from scipy.io import wavfile
import librosa

fs, data = wavfile.read(f"data/audio_cat/Development/Y8L.wav")
data = data.astype(np.float32)
```

Note that if you run this hands-on on the Jupyter notebook, you can also play the audio with the following command:

```
import IPython.display as ipd

ipd.Audio(data, rate=fs)
```

Here, `fs` is the sampling frequency of the audio file. It is also needed to convert the data to float values, to be able to calculate the spectrograms for instance. Let's plot the waveform:

```
plt.plot(np.arange(data.shape[0]) / fs, data)
```

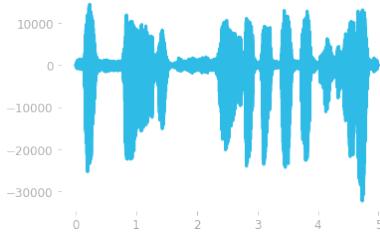


Figure 9.16: Waveform of our example audio file.

Figure 9.16 shows the waveform of our audio file. Since the file is 5-seconds length and the sampling frequency is 44,100, the number of values is  $5 \cdot 44100 = 220500$ .

```
data.shape
```

```
(220500,)
```

This is a lot of data points which leads to computational issues. This is one of the reasons why you might want to represent your audio files differently. One option is to calculate the *spectrogram* of the audio file. A spectrogram is a time-frequency representation of the audio. For each time sample, you have a certain number of values corresponding to the power for each frequency. It will be clearer when you'll soon look at the spectrograms.

A variant of the spectrogram, called *mel-spectrogram*, is widely used as a way to store audio data, mainly because the scale is closer to the sensitivity of the ear for each frequency band.

Let's now use the library Librosa to calculate the mel-spectrogram. The parameter `n_mels` is the number of frequency values at each time sample<sup>9</sup>:

```
n_mels = 300
# MEL spec from STFT 40 ms and 20 ms overlap
```

---

<sup>9</sup>The parameters used to calculate the spectrogram, like `n_fft` and `hop_len`, change the spectro-temporal resolution. Explaining these parameters is out of the scope of this hands-on project, but you can find complete explanations in this full signal processing course: <https://ccrma.stanford.edu/~jos/sasp/>

```
n_fft = int(0.04 * fs) + 1
hop_len = int(0.02 * fs) + 1
```

```
S = librosa.feature.melspectrogram(data, sr=fs, n_fft=n_fft,
                                    hop_length=hop_len, n_mels=n_mels)
S = S.astype(np.float32)
S = librosa.power_to_db(S)
```

The variable `S` contains the mel-spectrogram with the sound intensity in `db` (converted from power to db with `librosa.power_to_db()`).

Let's visualize it with `plt.imshow()`:

```
plt.imshow(S, origin="lower", extent=[0, 5, 0, 11000], aspect="auto")
```

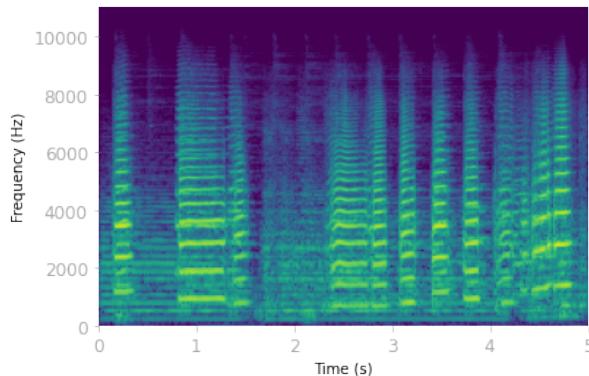


Figure 9.17: Mel-spectrogram of a brass melody.

Figure 9.17 shows the mel-spectrogram of a few notes played by a brass instrument. Feel free to listen to the sound and have a look at the spectrogram.

As said, the mel-spectrogram is a two-dimensional array and each value corresponds to the power for the time and frequency bin.

This mel-spectrogram contains a smaller number of values than the waveform:

```
S.shape
```

```
(300, 250)
```

There are only  $300 \cdot 250 = 75000$  values, but this is still a lot. Each value is considered as a feature so each audio file is described by these 75000 features. This would be impossible to visualize audio samples in such a high dimensionality, so let's use PCA.

Let's load all audio files, calculate the mel-spectrograms and store all of them in a data matrix `X`. Let's also store the corresponding labels in a variable `y`:

```
numFiles = categories.shape[0]

X = np.zeros((numFiles, n_mels, 250))
y = np.zeros((numFiles, 1))

for i, file in zip(np.arange(categories.shape[0]), categories["File"]):
    lab = categories["event_enc"][categories['File'] == file].values[0]
    y[i, :] = lab

    fs, data = wavfile.read(f"data/audio_cat/Development/{file}")
    data = data.astype(np.float32)

    S = librosa.feature.melspectrogram(
        data[: (44100 * 5)], sr=fs, n_fft=n_fft, hop_length=hop_len,
        n_mels=n_mels)
    S = S.astype(np.float32)
    S = librosa.power_to_db(S)

    X[i, :, :] = S

X_reshaped = X.reshape(numFiles, -1)
X_reshaped.shape
```

```
(1500, 75000)
```

We iterated on each row of the dataframe `categories` and loaded the corresponding file. At the end, we reshaped `X` in order to have all values of the mel-spectrogram as a single dimension.

### 9.5.2.2 Dimensionality Reduction

You'll use PCA to reduce the number of dimensions of the data. Let's use events that are discriminable (feel free to try with other events) because there is a lot of variability in five-seconds sounds of the same category (for instance, two audio samples with a brass melody will be quite different). However, some regularities are present and you'll visualize them by condensing the 75000 dimensions in two dimensions.

Let's select some kinds of events to have a subset of `x` and `y`:

```
sound_list = [3, 42, 44, 88, 54, 70, 18, 29, 75]

X_subset = X_reshaped[np.isin(y, sound_list).flatten(), :]
y_subset = y[np.isin(y, sound_list).flatten()]
```

You can use the class `PCA` from Sklearn to run the PCA (note that it implements what you saw in the first part of this hands-on project):

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2, svd_solver="full")
X_pca = pca.fit_transform(X_subset)
```

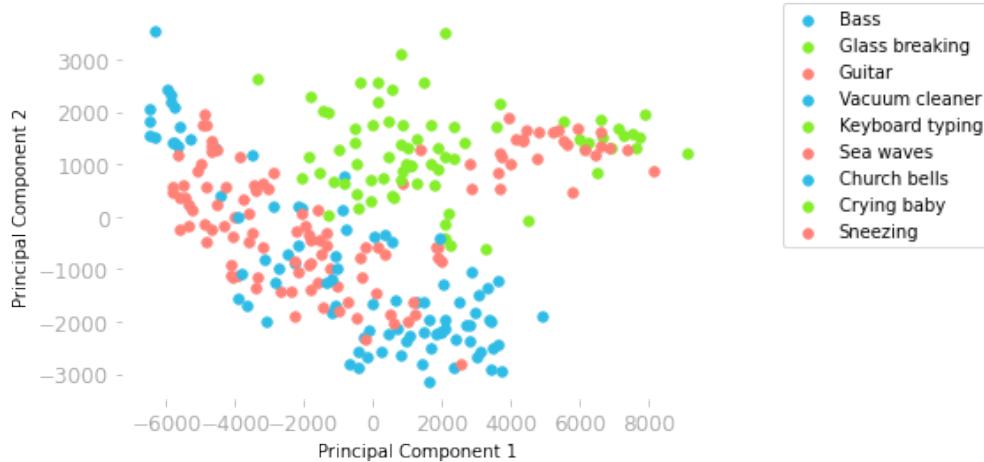
The parameter `n_components=2` allows you to select only two principal components.<sup>10</sup>

Let's plot the transformed data `X_pca` that contains only two dimensions:

```
for i in sound_list:
    plt.scatter(X_pca[(y_subset == i).flatten(), 0],
                X_pca[(y_subset == i).flatten(), 1],
                label=le.classes_[i],
                s=30)
# [...] Add legend, axes...
```

---

<sup>10</sup>The maximum number of components for the PCA corresponds to the maximum number of eigenvalues of the covariance matrix, which is the minimum between the number of rows and columns.



*Figure 9.18: Categories representation using the first and second components (x and y axes) of the PCA.*

You can see in Figure 9.18 that the sounds in each category are close together and also that some categories are closer than others (for instance, bass and guitar or vacuum cleaner and sea waves are close respectively).

You can look at the ratio of explained variance for each component:

```
pca.explained_variance_ratio_
```

```
array([0.52970331, 0.09414463])
```

This means that the first component alone explains almost 53% of the variance of the data. The second component drops and explains only less than 10% of the variance. The amount of variance explained by the components can give you an indication on how many components you want to keep.

### 9.5.2.3 Compression

Another reason to reduce the number of dimensions is to compress data. Let's see an example with one of the audio file of our dataset:

```
categories[categories["File"] == "HX9.wav"]
```

	Category	Event	File	event_enc
780	Music	Guitar	HX9.wav	44

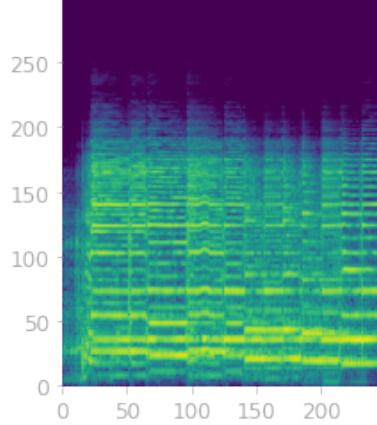
You can listen to this guitar melody (the file is HX9.wav). To be sure to get the right data file in the subset dataset, you need to subset the dataframe `categories` as well:

```
categories_subset = categories[categories["event_enc"].isin(sound_list)]
categories_subset = categories_subset.reset_index()
categories_subset.index[categories_subset["File"] == "HX9.wav"]

Int64Index([162], dtype='int64')
```

The new index of “HX9.wav” is 162. Let’s visualize its mel-spectrogram:

```
plt.imshow(X_subset[162].reshape(n_mels, 250), origin="lower")
plt.show()
```



*Figure 9.19: Mel-spectrogram of an example audio file: a guitar riff.*

Reshaping the data allows you to recover the time frequency shape (remember that you put time and frequency in a single dimension).

Let's do again the PCA using a larger number of components (only two would lead to poor reconstruction because a lot of information is lost). Let's choose 95% of variance explained:

```
pca_compression = PCA(n_components=0.95, svd_solver="full")
X_pca_compression = pca_compression.fit_transform(X_subset)
pca_compression.n_components_
```

121

This times, 121 components were used. To reconstruct the data from the PCA, you can use the method `inverse_transform()`:

```
reconstructed = pca_compression.inverse_transform(X_pca_compression)
```

Let's plot the reconstructed mel-spectrogram to compare it to the original:

```
f, axes = plt.subplots(1, 2)
axes[0].set_title("Original")
axes[0].imshow(X[780, :].reshape(n_mels, 250), origin="lower")
axes[1].imshow(reconstructed[162, :].reshape(n_mels, 250), origin="lower")
axes[1].set_title("Reconstructed")
plt.show()
```

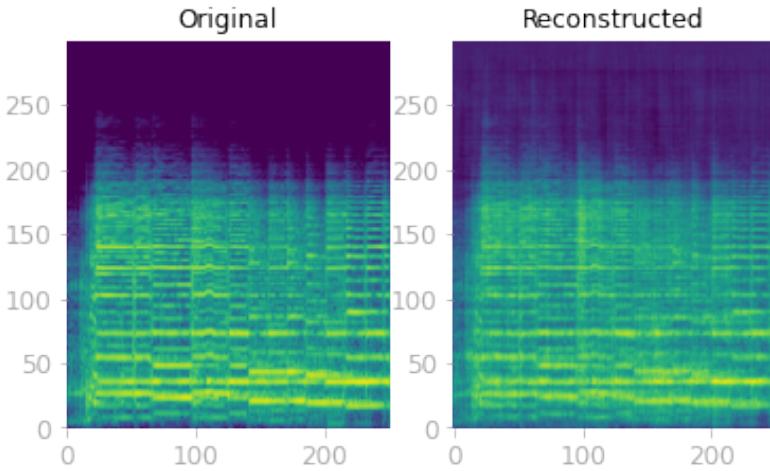


Figure 9.20: Mel-spectrogram of the original audio file (left) and the reconstruction from the PCA (right).

Figure 9.20 shows that the reconstructed data from the PCA is very close to the original data.

Let's do the same for different number of components. At each iteration, you'll calculate the PCA for this number of components, transform and reconstruct the data, going back to the audio from the mel-spectrogram (`librosa.feature.inverse.mel_to_audio`) and finally, save the corresponding `.wav` file.

```
all_reconstructed = []
n_comps = [5, 20, 50, 100, 200]
for n_comp in n_comps:
    pca_compression = PCA(n_components=n_comp, svd_solver="full")
    X_pca_compression = pca_compression.fit_transform(X_subset)
    reconstructed = pca_compression.inverse_transform(X_pca_compression)
    all_reconstructed.append(reconstructed)
    reconstructed_db = librosa.core.db_to_power(reconstructed[162,
    :].reshape(n_mels, 250))

    reconstructed_audio =
        librosa.feature.inverse.mel_to_audio(reconstructed_db,
        sr=fs, n_fft=n_fft, hop_length=hop_len)
    # convert to int16 data type
```

```

reconstructed_norm = 2 * ((reconstructed_audio -
reconstructed_audio.min()) / (reconstructed_audio.max() -
reconstructed_audio.min())) - 1
float32_data = reconstructed_norm * 32767
int16_data = float32_data.astype(np.int16)

# write to file
wavfile.write(f"audio/test_pca_{n_comp}.wav", fs, int16_data)

```

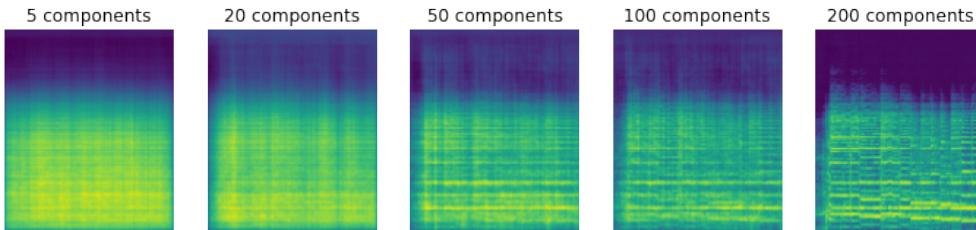
You can listen to the corresponding file here and see that the reconstruction is better and better when you add more components.

Visually, you can compare the mel-spectrograms:

```

f, axes = plt.subplots(1, len(all_reconstructed), figsize=(12, 4))
for i in range(len(all_reconstructed)):
    axes[i].imshow(all_reconstructed[i][162, :].reshape(n_mels, 250),
    origin="lower")
    axes[i].set_title(f"{n_comps[i]} components")
    axes[i].axis("off")
plt.show()

```



*Figure 9.21: Effect of the number of PCA components on the reconstructed Mel-spectrograms.*

Figure 9.21 shows that the reconstructed mel-spectrogram is closer and closer to the true data when you increase the number of components.

You saw in this hands-on project that you can use PCA to reduce the dimensionality of various types of data. In this last example, you reduced the very high dimensionality characterizing audio data. Concepts like eigendecomposi-

tion are useful in the every day life of a data scientist or machine learning scientist.

In the next chapter, you'll see another major concept of linear algebra: Singular Value Decomposition (SVD), that you can use for instance as an alternative way to calculate the PCA.

# Chapter 10

## Singular Value Decomposition

In this last chapter, you'll learn about Singular value decomposition (SVD), which is a major topic of linear algebra, data science and machine learning. You'll see that you'll need what you've learned so far to dive into the SVD and see how it's useful in data science.

I think that a key to understanding the SVD is to consider the input and output spaces of a linear transformation. For this reason, you'll dive deeper into the idea of change of basis, then reconsider eigendecomposition and build your understanding of the SVD on that.

The change of basis framework is interesting to conceive the effect of eigendecomposition and SVD: you go to a new basis (using the inverse of the change of basis matrix  $\mathbf{Q}^{-1}$ ) where the transformation is associated with another matrix ( $\Lambda$ ). Then you come back to the old basis with the change of basis matrix ( $\mathbf{Q}$ ).

When a matrix expresses a linear transformation, it converts input vectors into output vectors. These vectors can be expressed relative to different spaces (if their bases are not specified, they are the standard bases).

When you calculate the matrix  $\mathbf{Q}\Lambda\mathbf{Q}^{-1}$ , which corresponds to the same linear transformation than  $\mathbf{A}$  relative to another basis, you actually change the basis of the input vector and the basis of the output vector. However, you'll see that these input and output bases can't be the same with non-square matrices.

## 10.1 Non-square Matrices

You can only apply eigendecomposition to square matrices because it uses a single change of basis matrix, which implies that the initial vector and the transformed vector are relative to the same basis. You go to another basis with  $\mathbf{Q}$  to do the transformation, and you come back to the initial basis with  $\mathbf{Q}^{-1}$ .

As eigendecomposition, the goal of singular value decomposition (SVD) is to decompose a matrix into simpler components: orthogonal and diagonal matrices.

### 10.1.1 Different Input and Output Spaces

As you saw in Section 7.1.3.2, a non-square  $m$  by  $n$  matrix  $\mathbf{A}$  (with  $m$  different from  $n$ ) is associated with a transformation that takes  $n$ -dimensional vectors and gives  $m$ -dimensional vectors. The input and output spaces are necessarily different since they have not the same number of dimensions.

For instance, take a 3 by 2 matrix  $\mathbf{A}$  ( $m = 3$  and  $n = 2$ ) that transforms a vector  $\mathbf{v}$  into a vector  $\mathbf{w}$  such that:

$$\mathbf{A}\mathbf{v} = \mathbf{w}$$

The shapes of  $\mathbf{A}$ ,  $\mathbf{v}$  and  $\mathbf{w}$  are illustrated in Figure 10.1.

The diagram shows the multiplication of a 3x2 matrix  $\mathbf{A}$  by a 2x1 vector  $\mathbf{v}$  to produce a 3x1 vector  $\mathbf{w}$ . The matrix  $\mathbf{A}$  is represented by a bracket with height 3 and width 2, with a red double-headed arrow above it labeled '2'. The vector  $\mathbf{v}$  is a column vector with height 2 and width 1, with a red double-headed arrow to its right labeled '2'. The result  $\mathbf{w}$  is a column vector with height 3 and width 1, with a green double-headed arrow to its right labeled '1'.

$$\begin{matrix} 3 \\ \left[ \quad \right] \end{matrix} \cdot \begin{matrix} 2 \\ \left[ \quad \right] \end{matrix} = \begin{matrix} 3 \\ \left[ \quad \right] \end{matrix}$$

$(3, 2)$        $(2, 1)$        $(3, 1)$

Figure 10.1: Shape of the matrix  $\mathbf{A}$  and the vectors  $\mathbf{v}$  and  $\mathbf{w}$  corresponding to the equation  $\mathbf{A}\mathbf{v} = \mathbf{w}$ .

If you look at  $\mathbf{v}$  and  $\mathbf{w}$ , you can see that the input space ( $\mathbb{R}^2$ ) is different from the output space ( $\mathbb{R}^3$ ).

### 10.1.2 Specifying the Bases

The matrix associated with a linear transformation depends on the input and output bases you choose.

Let's take a linear transformation  $T$  associated with the matrix  $\mathbf{A}$  that converts a vector  $\mathbf{v}$  into the vector  $\mathbf{w}$ .

#### 10.1.2.1 Standard Basis

Let's start with the simpler case: a standard basis for the input and the output spaces. We'll denote  $[\mathbf{v}]_I$  and  $[\mathbf{w}]_I$  the vectors  $\mathbf{v}$  and  $\mathbf{w}$  relative to the standard basis. The matrix  $\mathbf{A}$  converts the vector  $\mathbf{v}$  in the vector  $\mathbf{w}$ . You can write:

$$\mathbf{A}[\mathbf{v}]_I = [\mathbf{w}]_I$$

The input basis (the basis of the initial vector) and the output basis (the basis of the transformed vector) are both the standard basis.

#### 10.1.2.2 Same Non Standard Input and Output Bases

The transformation  $T$  is associated with the matrix  $\mathbf{A}$  in the standard basis, but let's say you want to apply it to vectors in another basis  $\mathbf{B}_1$ , with  $\mathbf{B}_1$  being both the input and output basis. As you saw in Section 9.3.2, the values of the transformation matrix depend on the basis, and thus, applying the same matrix  $\mathbf{A}$  in the basis  $\mathbf{B}_1$  leads to a different transformation.

Let's call  $\mathbf{A}'$  the matrix associated with the transformation  $T$  in the basis  $\mathbf{B}_1$ . You have:

$$\mathbf{A}'[\mathbf{v}]_{\mathbf{B}_1} = [\mathbf{w}]_{\mathbf{B}_1} \tag{10.1}$$

If you apply the matrix  $\mathbf{A}'$  to the vector  $\mathbf{v}$  in the basis  $\mathbf{B}_1$ , you get the vector  $\mathbf{w}$  in the same basis  $\mathbf{B}_1$ .

In addition, as you saw in Section 9.2.3.2, the vectors relative to  $\mathbf{B}_1$  can be expressed as the vectors relative to the standard basis. You have:

$$\mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{I}[\mathbf{v}]_I$$

$$\mathbf{B}_1^{-1}\mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_1^{-1}\mathbf{I}[\mathbf{v}]_I$$

$$[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_1^{-1}[\mathbf{v}]_I$$

and similarly for the vector  $\mathbf{w}$ :

$$\mathbf{B}_1[\mathbf{w}]_{\mathbf{B}_1} = \mathbf{I}[\mathbf{w}]_I$$

$$\mathbf{B}_1^{-1}\mathbf{B}_1[\mathbf{w}]_{\mathbf{B}_1} = \mathbf{B}_1^{-1}\mathbf{I}[\mathbf{w}]_I$$

$$[\mathbf{w}]_{\mathbf{B}_1} = \mathbf{B}_1^{-1}[\mathbf{w}]_I$$

Replacing in equation 10.1, you obtain:

$$\mathbf{A}'[\mathbf{v}]_{\mathbf{B}_1} = [\mathbf{w}]_{\mathbf{B}_1}$$

$$\mathbf{A}'\mathbf{B}_1^{-1}[\mathbf{v}]_I = \mathbf{B}_1^{-1}[\mathbf{w}]_I$$

$$\mathbf{B}_1\mathbf{A}'\mathbf{B}_1^{-1}[\mathbf{v}]_I = \mathbf{B}_1\mathbf{B}_1^{-1}[\mathbf{w}]_I$$

$$\mathbf{B}_1\mathbf{A}'\mathbf{B}_1^{-1}[\mathbf{v}]_I = [\mathbf{w}]_I$$

This means that you can convert the vector  $\mathbf{v}$  into the vector  $\mathbf{w}$  using the matrix  $\mathbf{B}_1\mathbf{A}'\mathbf{B}_1^{-1}$ . And you know that, in the standard basis, you can also use the matrix  $\mathbf{A}$  to transform the vector  $\mathbf{v}$  into the vector  $\mathbf{w}$ :

$$\mathbf{A}[\mathbf{v}]_I = [\mathbf{w}]_I$$

This means that you have:

$$\mathbf{A} = \mathbf{B}_1 \mathbf{A}' \mathbf{B}_1^{-1}$$

and equivalently:

$$\mathbf{A} = \mathbf{B}_1 \mathbf{A}' \mathbf{B}_1^{-1}$$

$$\mathbf{B}_1^{-1} \mathbf{A} = \mathbf{B}_1^{-1} \mathbf{B}_1 \mathbf{A}' \mathbf{B}_1^{-1}$$

$$\mathbf{B}_1^{-1} \mathbf{A} = \mathbf{A}' \mathbf{B}_1^{-1}$$

$$\mathbf{B}_1^{-1} \mathbf{A} \mathbf{B}_1 = \mathbf{A}' \mathbf{B}_1^{-1} \mathbf{B}_1$$

$$\mathbf{B}_1^{-1} \mathbf{A} \mathbf{B}_1 = \mathbf{A}'$$

$$\mathbf{A}' = \mathbf{B}_1^{-1} \mathbf{A} \mathbf{B}_1$$

You have to apply  $\mathbf{A}'$  to do the transformation  $T$  in the basis  $\mathbf{B}_1$ . Using the last equivalence, you can also apply  $\mathbf{B}_1^{-1} \mathbf{A} \mathbf{B}_1$ . The matrix  $\mathbf{B}_1$  change the basis of the vector from  $\mathbf{B}_1$  to the standard basis, then you can apply the matrix  $\mathbf{A}$ , and finally, come back to the basis  $\mathbf{B}_1$  with  $\mathbf{B}_1^{-1}$ .

### 10.1.2.3 Input and Output Bases

The last step is to see how to do the same transformation when the input and the output bases are different.

Let's call  $\mathbf{A}''$  ( $\mathbf{A}$  “double prime”) the matrix associated with the transformation  $T$  with the input basis  $\mathbf{B}_1$  and the output basis  $\mathbf{B}_2$ . You have:

$$\mathbf{A}''[\mathbf{v}]_{\mathbf{B}_1} = [\mathbf{w}]_{\mathbf{B}_2} \tag{10.2}$$

Here is the trick: since the input vector  $\mathbf{v}$  is expressed relative to the input basis  $\mathbf{B}_1$  and the output vector  $\mathbf{w}$  to the output basis  $\mathbf{B}_2$ , you need to change the basis of  $\mathbf{v}$  and  $\mathbf{w}$  differently.

Let's see how you can go from the standard basis to  $\mathbf{B}_1$  and  $\mathbf{B}_2$ . You have the following mathematical relationship:

$$\mathbf{B}_1[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{I}[\mathbf{v}]_{\mathbf{I}}$$

$$[\mathbf{v}]_{\mathbf{B}_1} = \mathbf{B}_1^{-1}[\mathbf{v}]_{\mathbf{I}}$$

and

$$\mathbf{B}_2[\mathbf{w}]_{\mathbf{B}_2} = \mathbf{I}[\mathbf{w}]_{\mathbf{I}}$$

$$[\mathbf{w}]_{\mathbf{B}_2} = \mathbf{B}_2^{-1}[\mathbf{w}]_{\mathbf{I}}$$

Replacing in equation 10.2, you obtain:

$$\mathbf{A}''[\mathbf{v}]_{\mathbf{B}_1} = [\mathbf{w}]_{\mathbf{B}_2}$$

$$\mathbf{A}''\mathbf{B}_1^{-1}[\mathbf{v}]_{\mathbf{I}} = \mathbf{B}_2^{-1}[\mathbf{w}]_{\mathbf{I}}$$

$$\mathbf{B}_2\mathbf{A}''\mathbf{B}_1^{-1}[\mathbf{v}]_{\mathbf{I}} = [\mathbf{w}]_{\mathbf{I}}$$

As before, this leads to the following expression:

$$\mathbf{A} = \mathbf{B}_2\mathbf{A}''\mathbf{B}_1^{-1}$$

and equivalently:

$$\mathbf{A}'' = \mathbf{B}_2^{-1}\mathbf{A}\mathbf{B}_1$$

Let's see what this means. You can use the matrix  $\mathbf{B}_2^{-1}\mathbf{A}\mathbf{B}_1$  to apply the transformation  $T$  when the input basis is different than the output basis. It converts the vector from the basis  $\mathbf{B}_1$  to the standard basis with the matrix

$\mathbf{B}_1$ , then it applies the transformation  $\mathbf{A}$ , and finally it goes to the basis  $\mathbf{B}_2$  with the matrix  $\mathbf{B}_2^{-1}$ .

Once again, this is important to keep in mind that when we talk about linear transformations, it is in reference to an input and an output basis. The entries of the transformation matrix  $\mathbf{A}$  are with respect to these input and output bases.

Finding matrices similar to  $\mathbf{A}$  means finding a new basis *for both the input and the output space*. This was previously hidden by the fact that the input and output bases are the same in the eigendecomposition. Different input and output bases in the SVD makes this statement crucial for the following sections.

## 10.2 Expression of the SVD

With non-square matrices, the input space (before the transformation) is different than the output space (after the transformation). Considering different input and output bases allows you to decompose non square matrices. You'll see here that it leads to the formula of the SVD.

### 10.2.1 Notation

As you saw in the last section, while a single matrix  $\mathbf{Q}$  is used in eigendecomposition, with the SVD, you have a matrix to change the basis of  $\mathbf{v}$  and another one to change the basis of  $\mathbf{w}$ . You had:

$$\mathbf{A} = \mathbf{B}_2 \mathbf{A}'' \mathbf{B}_1^{-1}$$

To fit with the standard notation of the SVD, let's call the change of basis matrix of the input vector  $\mathbf{V}$  instead of  $\mathbf{B}_1$  and the change of basis matrix of the output vector  $\mathbf{U}$  instead of  $\mathbf{B}_2$ . In addition, let's call  $\Sigma$  (pronounced “capital Sigma”) the matrix corresponding to the transformation with the new input and output basis instead of  $\mathbf{A}''$ . You have:

$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^{-1}$$

Equivalently:

$$\Sigma = \mathbf{U}^{-1} \mathbf{A} \mathbf{V}$$

You're almost there: these last equations are almost expressing the SVD.

### 10.2.2 Singular Vectors and Singular Values

As you saw in Section 9.4, the eigendecomposition of symmetric matrices gives orthogonal eigenvectors<sup>1</sup>. With the SVD, you add the constraint that  $\mathbf{U}$  and  $\mathbf{V}$  are orthonormal. Since these matrices contains the basis vectors as columns, it implies that the input and output bases are orthonormal.

Orthonormal bases are nice. The advantage of using orthogonal bases is that you can decompose vectors into independent and separate components. Calculating the length or angles of vectors is also simpler with orthonormal bases.

#### 10.2.2.1 Orthonormal Requirements

In words, with the SVD, you want a set of orthogonal vectors  $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$  (the columns of the change of basis matrix  $\mathbf{V}$ ) relative to the *input space*, that gives a set of orthogonal vectors  $\{\mathbf{u}_1, \dots, \mathbf{u}_n\}$  (the columns of  $\mathbf{U}$ ) relative to the *output space*, scaled by  $\{\sigma_1, \dots, \sigma_n\}$  (pronounced “sigma”; they are the values in  $\Sigma$ ).

An orthogonal matrix has the property that its inverse is equal to its transpose. You can use this property to derive the equation of the SVD:

$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^T$$

This is the equation of the SVD: you decompose  $\mathbf{A}$  into three matrices  $\mathbf{U}$ ,  $\Sigma$  and  $\mathbf{V}^T$ . Since you constrained the vectors  $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$  and  $\{\mathbf{u}_1, \dots, \mathbf{u}_n\}$  to be orthonormal, the matrices  $\mathbf{V}$  and  $\mathbf{U}$  are orthonormal. Finally,  $\Sigma$  is a diagonal matrix.

---

<sup>1</sup>You can find more details about orthogonal vectors in Section 5.1.3.4 and orthogonal matrices in Section 6.4.5.

The vectors in  $U$  are called the *left singular vectors* and the vectors in  $V$ , the *right singular vectors*. The values in the diagonal of  $\Sigma$  (all the other entries are zero since it is a diagonal matrix) are called the *singular values*.

### Singular vectors and singular values

With the SVD, you don't decompose  $A$  into eigenvectors and eigenvalues. You need two sets of vectors that are the left and right singular vectors. These vectors are related by scaling values: the singular values.

#### 10.2.2.2 Set of Equations

As with eigendecomposition, you can write the SVD as a set of equations. Going from the equation of the SVD, and using the fact that  $V$  is orthonormal, and thus  $V^T V = V^{-1} V = I$ , you have:

$$A = U \Sigma V^T$$

$$AV = U \Sigma V^T V$$

$$AV = U \Sigma$$

Developing what are inside these matrices, you can write:

$$A [v_1 \cdots v_n] = [u_1 \cdots u_n] \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_n \end{bmatrix}$$

Leading to the following set of equations:

$$\left\{ \begin{array}{l} \mathbf{A}\mathbf{v}_1 = \sigma_1 \mathbf{u}_1 \\ \vdots \\ \mathbf{A}\mathbf{v}_n = \sigma_n \mathbf{u}_n \end{array} \right.$$

Keep these equations in mind: you'll use them in the next section.

### 10.2.3 Finding the Singular Vectors and the Singular Values

#### 10.2.3.1 The Special Matrices $\mathbf{AA}^T$ and $\mathbf{A}^T\mathbf{A}$

Even if  $\mathbf{A}$  is not square, the matrices  $\mathbf{AA}^T$  and  $\mathbf{A}^T\mathbf{A}$  are square. For instance, if the shape of  $\mathbf{A}$  is  $(3, 2)$ , you have the following shapes:

- $\mathbf{A}$  is  $(3, 2)$ .
- $\mathbf{A}^T$  is  $(2, 3)$ .
- $\mathbf{AA}^T$  is  $(3, 2)(2, 3) = (3, 3)$ .
- $\mathbf{A}^T\mathbf{A}$  is  $(2, 3)(3, 2) = (2, 2)$ .

More generally, if  $\mathbf{A}$  is a  $m$  by  $n$  matrix, then  $\mathbf{AA}^T$  is a  $m$  by  $m$  symmetric square matrix and  $\mathbf{A}^T\mathbf{A}$  is a  $n$  by  $n$  square matrix.

These matrices are also symmetric. For instance:

```
A = np.array([
    [1, 2],
    [4, 3],
    [7, 1]
])
A @ A.T

array([[ 5, 10,  9],
       [10, 25, 31],
       [ 9, 31, 50]])
```

For instance, the value 10 in the first row is calculated with  $4 \cdot 1 + 3 \cdot 2$ , and the value 10 in the first column with  $1 \cdot 4 + 2 \cdot 3$ , which gives the same result.

Since  $\mathbf{A}\mathbf{A}^T$  and  $\mathbf{A}^T\mathbf{A}$  are symmetric, they have orthogonal eigenvectors and real eigenvalues.

Another property of these matrices is that their eigenvalues are all non-negative. These matrices are said *positive semidefinite*: this is important because, as you'll soon see, you'll take the square root of their eigenvalues, so it would be defined only if they are non-negative.

You'll see here that the singular vectors and singular values can be found through the eigendecomposition of these special matrices  $\mathbf{A}\mathbf{A}^T$  and  $\mathbf{A}^T\mathbf{A}$ .

### 10.2.3.2 Eigenvectors of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$

Let's see how the eigenvectors of  $\mathbf{A}\mathbf{A}^T$  and  $\mathbf{A}^T\mathbf{A}$  correspond to the singular values.

In the context of SVD, you have  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ . So  $\mathbf{A}\mathbf{A}^T$  is:

$$\mathbf{A}\mathbf{A}^T = (\mathbf{U}\Sigma\mathbf{V}^T)(\mathbf{U}\Sigma\mathbf{V}^T)^T$$

Remember from Section 6.3.3 that the transposition of a matrix product is:

$$(\mathbf{ABC})^T = \mathbf{C}^T\mathbf{B}^T\mathbf{A}^T$$

so you have:

$$\begin{aligned} \mathbf{A}\mathbf{A}^T &= (\mathbf{U}\Sigma\mathbf{V}^T)(\mathbf{U}\Sigma\mathbf{V}^T)^T \\ &= \mathbf{U}\Sigma\mathbf{V}^T(\mathbf{V}^T)^T\Sigma^T\mathbf{U}^T \\ &= \mathbf{U}\Sigma\mathbf{V}^T\mathbf{V}\Sigma^T\mathbf{U}^T \end{aligned}$$

And since  $\mathbf{V}$  is orthogonal,  $\mathbf{V}^T\mathbf{V} = \mathbf{I}$  (see Section 6.4.5) and you can remove it from the equation:

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\Sigma\Sigma^T\mathbf{U}^T$$

Since  $\Sigma$  is diagonal,  $\Sigma = \Sigma^T$ , so you can simplify  $\Sigma\Sigma^T$  to  $\Sigma^2$ . In addition,  $\mathbf{U}$  is orthogonal and thus  $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ . You have:

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\Sigma^2\mathbf{U}^T$$

With a little rearrangement, you get:

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\Sigma^2\mathbf{U}^T$$

$$\mathbf{A}\mathbf{A}^T\mathbf{U} = \mathbf{U}\Sigma^2\mathbf{U}^T\mathbf{U}$$

$$\mathbf{A}\mathbf{A}^T\mathbf{U} = \mathbf{U}\Sigma^2$$

You saw in Section 9.4 that the eigendecomposition formula is  $\mathbf{A}\mathbf{Q} = \mathbf{Q}\Lambda$  with  $\mathbf{Q}$  being the eigenvectors and  $\Lambda$  the eigenvalues of  $\mathbf{A}$  (with the matrix  $\mathbf{A}$  being square). Similarly in the last equation,  $\mathbf{U}$  is the matrix containing the eigenvectors of  $\mathbf{A}\mathbf{A}^T$  and  $\Sigma^2$  contains the eigenvalues of  $\mathbf{A}\mathbf{A}^T$ .

If you apply the same logic to  $\mathbf{A}^T\mathbf{A}$ , you have:

$$\mathbf{A}^T\mathbf{A} = (\mathbf{U}\Sigma\mathbf{V}^T)^T(\mathbf{U}\Sigma\mathbf{V}^T)$$

$$\mathbf{A}^T\mathbf{A} = (\mathbf{V}^T)^T\Sigma^T\mathbf{U}^T\mathbf{U}\Sigma\mathbf{V}^T$$

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}\Sigma^T\Sigma\mathbf{V}^T$$

$$\mathbf{A}^T\mathbf{A}\mathbf{V} = \mathbf{V}\Sigma^2$$

The matrix  $\mathbf{V}$  contains the eigenvectors of  $\mathbf{A}^T\mathbf{A}$  and  $\Sigma^2$  the eigenvalues of  $\mathbf{A}^T\mathbf{A}$ .

This shows that, to construct the SVD ( $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ ), the matrix  $\mathbf{U}$  must be the eigenvectors of  $\mathbf{A}\mathbf{A}^T$  and  $\mathbf{V}$  the eigenvectors of  $\mathbf{A}^T\mathbf{A}$ .

### 10.2.3.3 Orthonormal Input Basis Gives Orthonormal Output Basis

In the context of the SVD, the vectors  $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$  are the eigenvectors of  $\mathbf{A}^T \mathbf{A}$ . In this special case, the vectors  $\{\mathbf{u}_1, \dots, \mathbf{u}_n\}$  are orthonormal because the vectors  $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$  are chosen to be orthonormal. Let's see why.

You want to check that the pair of vectors  $\mathbf{u}_i$  and  $\mathbf{u}_j$  are orthogonal if  $\mathbf{v}_i$  and  $\mathbf{v}_j$  are orthogonal. Mathematically, you want to know if  $\mathbf{u}_i \mathbf{u}_j^T = 0$  when  $\mathbf{v}_i \mathbf{v}_j^T = 0$ .

First, you saw that:

$$\mathbf{A} \mathbf{v}_i = \sigma_i \mathbf{u}_i$$

You can rearrange as:

$$\mathbf{u}_i = \frac{\mathbf{A} \mathbf{v}_i}{\sigma_i}$$

So you have:

$$\begin{aligned} \mathbf{u}_i^T \mathbf{u}_j &= \left( \frac{\mathbf{A} \mathbf{v}_i}{\sigma_i} \right)^T \left( \frac{\mathbf{A} \mathbf{v}_j}{\sigma_j} \right) \\ &= \frac{\mathbf{v}_i^T \mathbf{A}^T \mathbf{A} \mathbf{v}_j}{\sigma_i \sigma_j} \end{aligned} \tag{10.3}$$

Since the vectors  $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$  are the eigenvectors of  $\mathbf{A}^T \mathbf{A}$ , you have:

$$\mathbf{A}^T \mathbf{A} \mathbf{v}_j = \sigma_j^2 \mathbf{v}_j$$

So by replacing in equation 10.3, you have:

$$\mathbf{u}_i^T \mathbf{u}_j = \frac{\mathbf{v}_i^T \mathbf{A}^T \mathbf{A} \mathbf{v}_j}{\sigma_i \sigma_j}$$

$$= \frac{\mathbf{v}_i^T \sigma_j^2 \mathbf{v}_j}{\sigma_i \sigma_j}$$

$$= \frac{\mathbf{v}_i^T \sigma_j \mathbf{v}_j}{\sigma_i}$$

$$\mathbf{u}_i^T \mathbf{u}_j = \frac{\sigma_j}{\sigma_i} \mathbf{v}_i^T \mathbf{v}_j$$

This result shows that if the vectors  $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$  of  $\mathbf{A}^T \mathbf{A}$  are orthogonal, then  $\mathbf{v}_i^T \mathbf{v}_j = 0$  and thus the right-hand side is equal to zero, and  $\mathbf{u}_i^T \mathbf{u}_j = 0$ , meaning that the vectors  $\{\mathbf{u}_1, \dots, \mathbf{u}_n\}$  are also orthogonal.

#### 10.2.3.4 Example

Let's take the following matrix  $\mathbf{A}$ :

```
A = np.array([
    [1, 2, 2],
    [3, 7, 6],
    [6, 2, 12],
    [3, 1, 6]
])
```

You can start by calculating the SVD of  $\mathbf{A}$  with Numpy:

```
U, Sigma, V_transpose = np.linalg.svd(A)
```

Let's have a look at these matrices:

```
U.round(2)
```

```
array([[-0.16, -0.22,  0.96, -0.  ],
       [-0.49, -0.83, -0.27,  0.  ],
       [-0.76,  0.46, -0.02, -0.45],
       [-0.38,  0.23, -0.01,  0.89]])
```

```
Sigma.round(2)
```

```
array([17.44,  5.36,  0.  ])
```

```
V_transpose.round(2)
```

```
array([[-0.42, -0.33, -0.85],
       [ 0.15, -0.95,  0.29],
       [-0.89, -0.  ,  0.45]])
```

Now, you can calculate the eigenvectors and eigenvalues of  $\mathbf{A}\mathbf{A}^T$  and  $\mathbf{A}^T\mathbf{A}$ :

```
eigvals_AAT, eigvecs_AAT = np.linalg.eig(A @ A.T)
eigvecs_AAT.round(2)
```

```
array([[ 0.16,  0.22,  0.96, -0.06],
       [ 0.49,  0.83, -0.27,  0.02],
       [ 0.76, -0.46, -0.06,  0.45],
       [ 0.38, -0.23,  0.07, -0.89]])
```

You can note that the first two columns are identical with the first two columns of  $\mathbf{U}$  calculated with the `np.linalg.svd()` function. The other columns are not important because they don't correspond to nonzero singular values (`Sigma` contains only two nonzero values). In addition, the signs are not necessarily the same: they can be reversed and lead to the same result.

Let's look at the eigenvectors of  $\mathbf{A}^T\mathbf{A}$ :

```
eigvals_ATA, eigvecs_ATA = np.linalg.eig(A.T @ A)
eigvecs_ATA.round(2).T
```

```
array([[ 0.42,  0.33,  0.85],
       [ 0.89, -0.  , -0.45],
```

```
[ 0.15, -0.95,  0.29]])
```

The eigenvectors and eigenvalues are not necessarily sorted in the output of the function `np.linalg.eig()`.<sup>2</sup> This means that the eigenvectors corresponding to the largest eigenvalues are not necessarily in the first positions.

Look at the eigenvalues of  $\mathbf{A}\mathbf{A}^T$  and  $\mathbf{A}^T\mathbf{A}$ . The nonzero values are the same and corresponds to the squared singular values:

```
eigvals_ATA.round(2)
array([304.31, -0.   ,  28.69])

eigvals_AAT.round(2)
array([304.31,  28.69, -0.   , -0.   ])

(Sigma**2).round(2)
array([304.31,  28.69,    0.   ])
```

**Reconstruction** Finally, you can reconstruct the matrix  $\mathbf{A}$  from the decomposition. You'll use only the nonzero singular values.

Since `Sigma` is just the list of the singular values, you need to create a diagonal matrix with these values in the diagonal (using the function `np.diag()`).

You also need to add a row of zeros to `Sigma` and `V_transpose` to have matching shapes:

```
Sigma = np.diag(np.append(Sigma, 0))
V_transpose = np.vstack([V_transpose, np.array([0, 0, 0])])

U[:, :2] @ Sigma[:2, :] @ V_transpose[:, :]

array([[ 1.,  2.,  2.],
       [ 3.,  7.,  6.]])
```

---

<sup>2</sup>As it is explained here: <https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html>

```
[ 6.,  2., 12.],
[ 3.,  1.,  6.])
```

You can see that the matrix  $\mathbf{A}$  has been reconstructed.

### 10.2.4 Summary

As for eigendecomposition, let's summarize how the matrix  $\mathbf{A}$  that can be associated with different input and output bases, and corresponds to the product of the three matrices  $\mathbf{U}\Sigma\mathbf{V}^T$ .

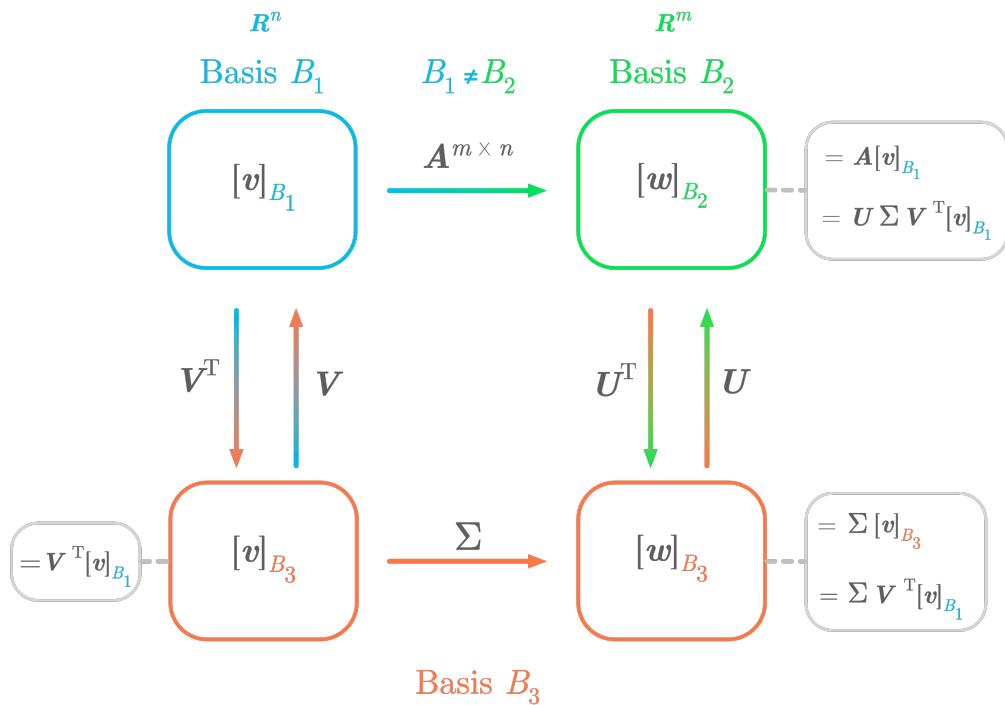


Figure 10.2: Change of basis when the output basis is different than the input basis.

Let's inspect the different components in Figure 10.2 and let's call  $T$  the transformation associated with the matrix  $\mathbf{A}$  in the standard basis.

The top left represents the input vector  $\mathbf{v}$  relative to the input basis  $\mathcal{B}_1$  ( $[\mathbf{v}]_{\mathcal{B}_1}$ ). When you apply  $\mathbf{A}$  to this vector, you get the output vector  $\mathbf{w}$

relative to the output basis  $\mathbf{B}_2$  ( $[\mathbf{w}]_{\mathbf{B}_2}$ , top right). Mathematically, it is expressed as  $[\mathbf{w}]_{\mathbf{B}_2} = \mathbf{A}[\mathbf{v}]_{\mathbf{B}_1}$ .

With the SVD, you want to find another basis (in red, named  $\mathbf{B}_3$  in the figure) where the transformation  $T$  is associated with a diagonal matrix  $\Sigma$ .

So, to reach  $[\mathbf{w}]_{\mathbf{B}_2}$ , you can also:

- Bottom left: change the basis of  $\mathbf{v}$  from  $\mathbf{B}_1$  to a new basis  $\mathbf{B}_3$  using the matrix  $\mathbf{V}^T$ . Mathematically, you have  $[\mathbf{v}]_{\mathbf{B}_3} = \mathbf{V}^T[\mathbf{v}]_{\mathbf{B}_1}$ .
- Bottom right: apply  $\Sigma$  so you get the output vector  $\mathbf{w}$  still in the basis  $\mathbf{B}_3$ . You have:  $[\mathbf{w}]_{\mathbf{B}_3} = \Sigma \mathbf{V}^T [\mathbf{v}]_{\mathbf{B}_1}$ .
- Top right: change the basis from  $\mathbf{B}_3$  to  $\mathbf{B}_2$  with the change of basis  $\mathbf{U}$ . You have:  $[\mathbf{w}]_{\mathbf{B}_2} = \mathbf{U} \Sigma \mathbf{V}^T [\mathbf{v}]_{\mathbf{B}_1}$ .

Going to the longest path, you do the same transformation than  $\mathbf{A}$ , but by applying three simpler transformations. Unlike eigendecomposition, the second change of basis is not the inverse of the first: the input basis of  $\mathbf{A}$  is different from its output basis.

## 10.3 Geometry of the SVD

Considering matrices as linear transformations, the decomposition of a matrix corresponds to the decomposition of the transformation into multiple sub-transformations. In the case of the SVD, the transformation is converted to three simpler transformations.

You'll see three examples: one in two dimensions, one to compare the transformations of the SVD and the eigendecomposition, and one in three dimensions.

### 10.3.1 Two-Dimensional Example

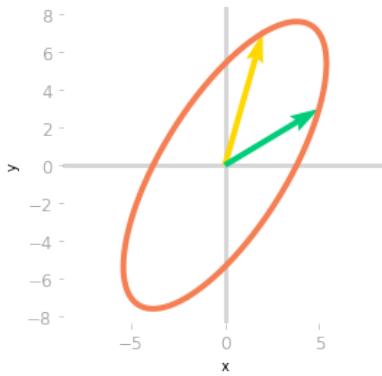
You'll see the action of these transformations using a custom function `matrix_2d_effect()`. This function plots the unit circle (you can find more details about the unit circle in Section 5.3.3), and the basis vectors transformed by a matrix.

Let's start by using the function to see the effect of the following matrix  $\mathbf{A}$ .

$$\mathbf{A} = \begin{bmatrix} 2 & 5 \\ 7 & 3 \end{bmatrix}$$

It should show the unit circle and the basis vectors transformed by the matrix:

```
A = np.array([
    [2, 5],
    [7, 3]
])
matrix_2d_effect(A)
# [...] Add labels
```



*Figure 10.3: Effect of the matrix  $\mathbf{A}$  on the unit circle and the basis vectors.*

Figure 10.3 illustrates the effect of  $A$  on your two-dimensional space. Let's compare this to the sub-transformations associated with the matrices of the SVD.

You can calculate the SVD of  $\mathbf{A}$  using Numpy:

```
U, Sigma, V_transpose = np.linalg.svd(A)
```

Remember that the matrices  $\mathbf{U}$ ,  $\Sigma$  and  $\mathbf{V}$  contain respectively the left singular vectors, the singular values and the right singular vectors. You can consider

$\mathbf{U}$  as a first change of basis matrix,  $\Sigma$  as the linear transformation in this new basis (this transformation should be a simple scaling since  $\Sigma$  is diagonal), and  $\mathbf{V}^T$  another change of basis matrix. You saw that the SVD constraints both change of basis matrices  $\mathbf{U}$  and  $\mathbf{V}^T$  to be orthogonal, meaning that the transformations will be simple rotations.

To summarize, the transformation corresponding to the matrix  $\mathbf{A}$  is decomposed into a rotation (or a reflexion, or a rotoreflection), a scaling, and another rotation (or a reflexion, or a rotoreflection).

Let's see the effect of each matrix successively:

```
matrix_2d_effect(V_transpose)
```

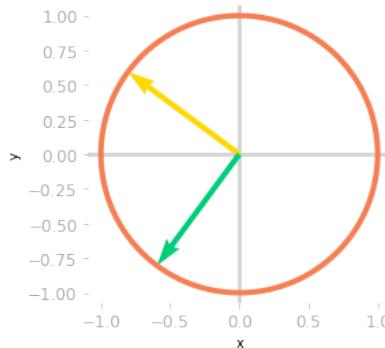


Figure 10.4: Effect of the matrix  $\mathbf{V}^T$  on the unit circle and the basis vectors.

You can see in Figure 10.4 that the basis vectors have been rotated by the matrix  $\mathbf{V}^T$ .

```
matrix_2d_effect(np.diag(Sigma) @ V_transpose)
```

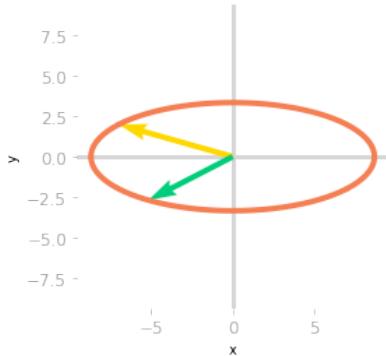


Figure 10.5: Effect of the matrices  $\mathbf{V}^T$  and  $\Sigma$ .

Then, Figure 10.5 shows that the effect of  $\Sigma$  is a scaling of the unit circle and the basis vectors.

```
matrix_2d_effect(U @ np.diag(Sigma) @ V_transpose)
```

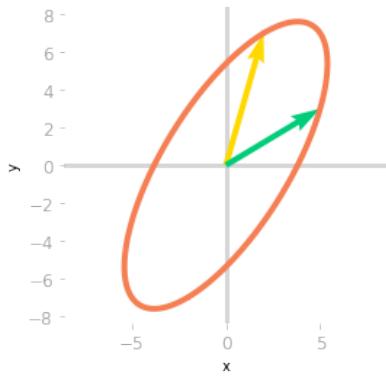


Figure 10.6: Effect of the matrices  $\mathbf{V}^T$ ,  $\Sigma$  and  $\mathbf{U}$ .

Finally, a third rotation is applied by  $\mathbf{U}$ . You can see in Figure 10.6 that the transformation is the same as the one associated with the matrix  $\mathbf{A}$ . You have decomposed the transformation into a rotation, a scaling and a rotoreflection (look at the basis vectors: a reflection has been done because the yellow vector is on the left of the green vector, which was not the case initially).

### 10.3.2 Comparison with Eigendecomposition

Since the matrix  $\mathbf{A}$  was square, you can compare this decomposition of the matrix  $\mathbf{A}$  with eigendecomposition and use the same type of visualization to get more insights about the difference between the two methods.

Let's calculate the eigendecomposition of  $\mathbf{A}$ :

```
lambd, Q = np.linalg.eig(A)
```

Note that, since the matrix  $\mathbf{A}$  is not symmetric, the eigenvectors are not orthogonal (their dot product is not equal to zero):

```
Q[:, 0] @ Q[:, 1]
```

```
-0.16609095970747995
```

Let's see the effect of  $\mathbf{Q}^{-1}$  on the basis vectors and the unit circle:

```
ax = matrix_2d_effect(np.linalg.inv(Q))
```

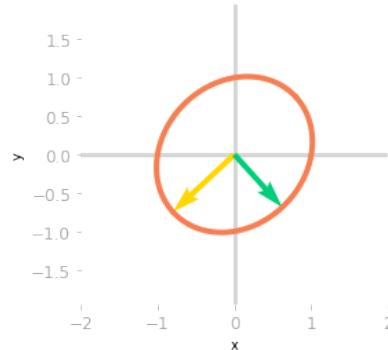


Figure 10.7: Effect of the matrix  $\mathbf{Q}^{-1}$ .

You can see in Figure 10.7 that  $\mathbf{Q}^{-1}$  rotates and scales the unit circle and the basis vectors. The transformation of a non-orthogonal matrix is not a simple rotation.

The next step is to apply  $\Lambda$ .

```
ax = matrix_2d_effect(np.diag(lambd) @ np.linalg.inv(Q))
```

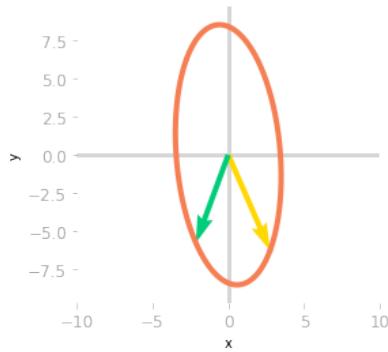


Figure 10.8: Effect of the matrix  $Q^{-1}$  and  $\Lambda$ .

The effect of  $\Lambda$ , shown in Figure 10.8 is a stretching and a reflection through the y-axis (the yellow vector is now on the right of the green vector).

```
ax = matrix_2d_effect(Q @ np.diag(lambd) @ np.linalg.inv(Q))
```

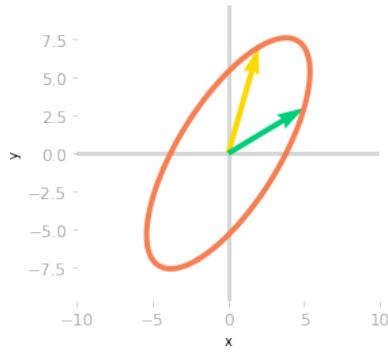


Figure 10.9: Effect of the matrix  $Q^{-1}$ ,  $\Lambda$  and  $Q$ .

The last transformation, shown in Figure 10.9 corresponds to the change of basis back to the initial one. You can see that it leads to the same result

than the transformation associated with  $\mathbf{A}$ : both matrices  $\mathbf{A}$  and  $\mathbf{Q}\Lambda\mathbf{Q}^{-1}$  are similar: they correspond to the same transformation in different bases.

It highlights the differences between eigendecomposition and SVD<sup>3</sup>. With SVD, you have three different transformations, but two of them are only rotation. With eigendecomposition, there are only two different matrices but the transformation associated with  $\mathbf{Q}$  is not necessarily a simple rotation (it is only the case when  $\mathbf{A}$  is symmetric).

### 10.3.3 Three-Dimensional Example

Since the SVD can be used with non square matrices, it is interesting to see how the transformations are decomposed in this case.

First, non square matrices map two spaces that have a different number of dimensions. Keep in mind that  $m$  by  $n$  matrices map a  $n$ -dimensional space with a  $m$ -dimensional space.

Let's take the example of a 3 by 2 matrix, mapping a two-dimensional space to a three-dimensional space. This means that input vectors are two-dimensional and output vectors three-dimensional. Take the matrix  $\mathbf{A}$ :

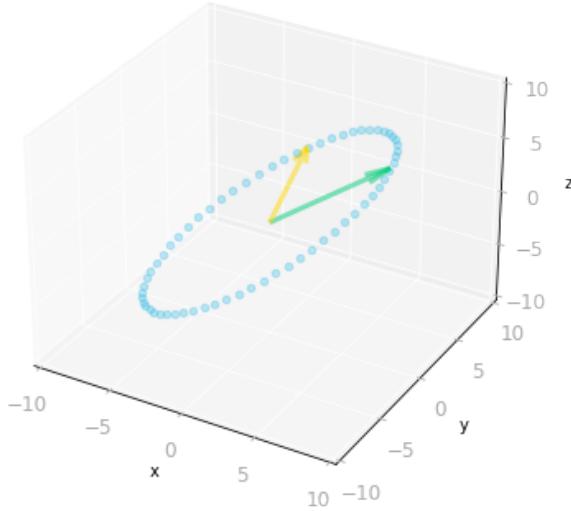
```
A = np.array([
    [2, 5],
    [1, 6],
    [7, 3]
])
```

To visualize the effect of  $\mathbf{A}$ , you'll still use the unit circle in two dimensions and you'll calculate the output for points on this circle. Each point is considered as an input vector and you can observe the effect of  $\mathbf{A}$  on each of these vectors.

```
ax = matrix_3_by_2_effect(A)
# [...] Add styles axes, limits etc.
```

---

<sup>3</sup>Note that eigendecomposition was possible here because we choose  $\mathbf{A}$  square to illustrate the difference between eigendecomposition and SVD.



*Figure 10.10: Effect of the matrix  $A$ : it transforms vectors on the unit circle and the basis vectors from a two-dimensional space to a three-dimensional space.*

As expected, and as represented in Figure 10.10, the two-dimensional unit circle is transformed into a three-dimensional ellipse. Feel free to use the notebook to be able to move the plot and have a better idea of the three-dimensional shape.

What you can note is that the output vectors land all on a two-dimensional plane. This is because the rank of  $A$  is two.<sup>4</sup>

Now that you know the output of the transformation by  $A$ , let's calculate the SVD of  $A$  and see the effects of the different matrices, as you did with the two-dimensional example.

```
U, Sigma, V_transpose = np.linalg.svd(A)
```

The shape of the left singular vectors ( $U$ ) is  $m$  by  $m$  and the shape of the

---

<sup>4</sup>Remind from Section 7.6.1 that the rank of a matrix is the number of linearly independent columns or rows. The maximum rank is the smaller number of rows or columns: in our example it is two.

right singular vectors ( $\mathbf{V}^T$ ) is  $n$  by  $n$ . There are two singular values in the matrix  $\Sigma$ .

The transformation associated with  $\mathbf{A}$  is decomposed into a first rotation in  $\mathbb{R}^n$  (associated with  $\mathbf{V}^T$ , in the example,  $\mathbb{R}^2$ ), a scaling going from  $\mathbb{R}^n$  to  $\mathbb{R}^m$  (in the example, from  $\mathbb{R}^2$  to  $\mathbb{R}^3$ ), and a rotation in the output space  $\mathbb{R}^m$  (in the example,  $\mathbb{R}^3$ ).

Let's start to inspect the effect of  $\mathbf{V}^T$  on the unit circle. You stay in two dimensions at this step:

```
matrix_2d_effect(V_transpose)
```

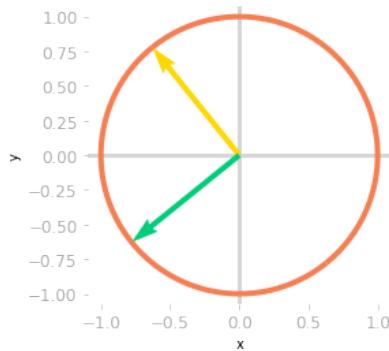


Figure 10.11: Effect of the matrix  $\mathbf{V}^T$ : at this step, you're still in a two-dimensional space.

You can see in Figure 10.11 that the basis vectors have been rotated.

Then, you need to reshape  $\Sigma$  because the function `np.linalg.svd()` gives a one-dimensional array containing the singular values. You want a matrix with the same shape of  $\mathbf{A}$ : a 3 by 2 matrix to go from 2D to 3D. This matrix contains the singular values as the diagonal, the other values are zero.

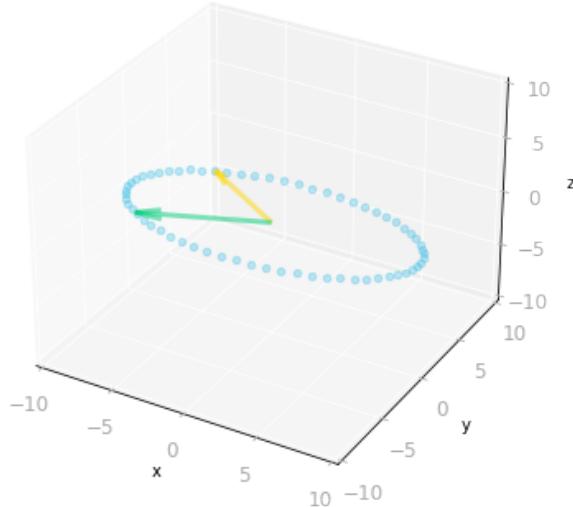
Let's create this matrix:

```
Sigma_full = np.zeros((A.shape[0], A.shape[1]))
Sigma_full[:A.shape[1], :A.shape[1]] = np.diag(Sigma)
Sigma_full
```

```
array([[9.99274669, 0.          ],
       [0.          , 4.91375758],
       [0.          , 0.        ]])
```

You can now add the transformation of  $\Sigma$  to see the result in 3D in Figure 10.12:

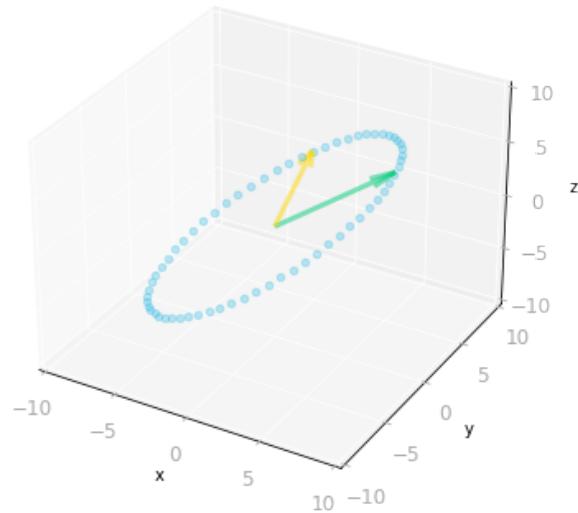
```
ax = matrix_3_by_2_effect(Sigma_full @ V_transpose)
# [...] Add styles axes, limits etc.
```



*Figure 10.12: Effect of the matrices  $V^T$  and  $\Sigma$ : since  $\Sigma$  is a three by two matrix, it transforms two-dimensional vectors into three-dimensional vectors.*

Finally, you need to operate the last change of basis. You stay in 3D because the matrix  $U$  is a 3 by 3 matrix.

```
ax = matrix_3_by_2_effect(U @ Sigma_full @ V_transpose)
# [...] Add styles axes, limits etc.
```



*Figure 10.13: Effect of the three matrices  $\mathbf{V}^T$ ,  $\Sigma$  and  $\mathbf{U}$ : the transformation is from a three-dimensional space to a three-dimensional space.*

You can see in Figure 10.13 that the result is identical to the transformation associated with the matrix  $\mathbf{A}$ .

### 10.3.4 Summary

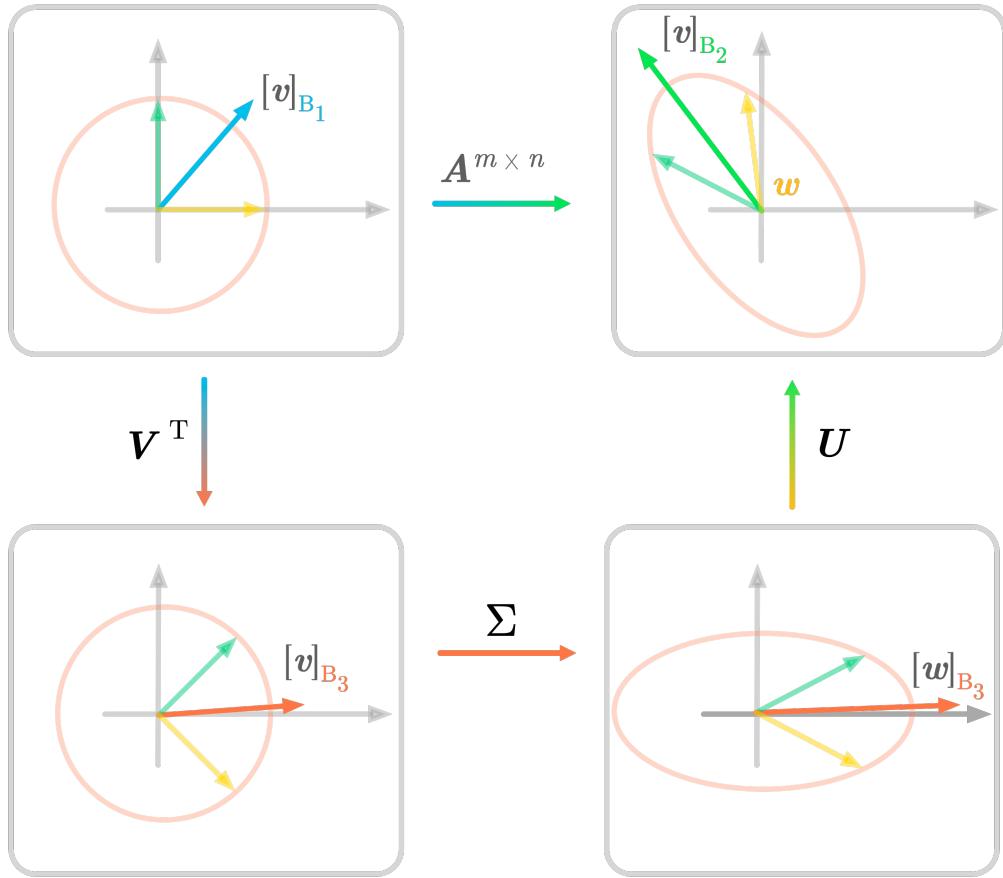


Figure 10.14: SVD in two dimensions.

Figure 10.14 summarizes the decomposition of a matrix  $A$  into three matrices. The transformation associated with  $A$  is done by the three sub-transformations. The notation is the same as in Figure 10.14 and illustrates the geometric perspective of the SVD.

## 10.4 Low-Rank Matrix Approximation

Sometimes, like with data compression, it is desirable to find a reduced version of a matrix that approximates it well. This is an application of the SVD: if

you keep the singular vectors corresponding to the largest singular values, you get the best approximation for this number of vectors.

#### 10.4.1 Full SVD, Thin SVD and Truncated SVD

You saw in Section 7.6.1 that the rank of a matrix is the number of linearly independent columns (that is equal to the number of linearly independent rows). The SVD tells you something important: the number of non-zero singular values found in the SVD corresponds to the rank of the matrix.

For instance, take the following matrix  $\mathbf{A}$ :

```
A = np.array([
    [1, 2],
    [2, 4],
    [3, 6]
])
np.linalg.matrix_rank(A)
```

1

The rank of the matrix is equal to 1. Let's calculate the SVD of  $\mathbf{A}$  and have a look at the number of nonzero singular values.

```
np.linalg.svd(A)[1].round(2)
array([8.37, 0.])
```

You can see that there is only one nonzero singular value.

The rule is that for a  $m$  by  $n$  matrix with  $m > n$ , the maximum number of nonzero singular values is  $n$ . In the preceding section, you filled the diagonal matrix  $\Sigma$  with zeros to match the shape  $m$  by  $n$ . This is called the *full svd* (in green in Figure 10.15).

It is possible to remove the rows of zeros in  $\Sigma$  (the rows and not the columns, because  $m > n$ : it would be the columns if  $m < n$ ) and the corresponding left or right singular vectors. This is called the *thin SVD* or *reduced SVD* (in red in Figure 10.15).

Finally, it is possible to keep only  $k$  singular vectors and singular values (for

some  $k < \min\{m, n\}$ ). In this case, the SVD matrix is an approximation of the initial matrix  $\mathbf{A}$  because you lose data. However, keeping only the singular vectors corresponding to large singular values ensure that you have the best approximation of  $\mathbf{A}$  for this lower rank ( $k$ ) matrix. This is called the *truncated SVD* (in blue in Figure 10.15).

The difference between the full, the thin and the truncated SVD is summarized in Figure 10.15 and shows that you can select only a part of the matrices involved in the SVD to compress data.

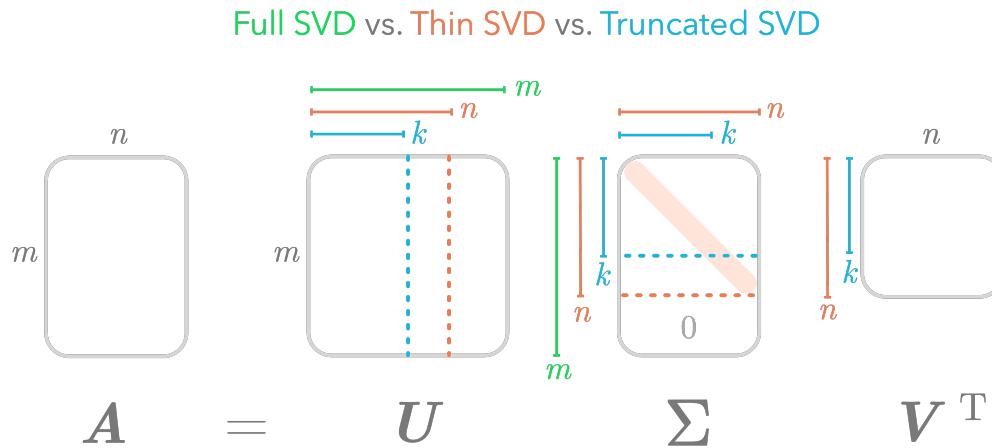


Figure 10.15: Comparison of the full SVD, the reduced SVD and the truncated SVD for a  $m$  by  $n$  matrix (with  $m > n$ ).

First, let's see the full SVD, illustrated in green. You can see that  $\mathbf{U}$  is  $m$  by  $m$ : you keep the whole matrix. Then,  $\Sigma$  is  $m$  by  $n$ , meaning that you must add rows ( $m - n$ ) filled with zeros. Finally,  $\mathbf{V}^T$  is  $n$  by  $n$ .

Second, the thin SVD is illustrated in red. In this case, you keep only  $n$  columns of  $\mathbf{U}$ , you don't add zeros in  $\Sigma$  and you take the whole matrix  $\mathbf{V}^T$ , as with the full SVD.

Third, the truncated SVD is illustrated in blue. You choose a value of  $k$  for some  $k < \min\{m, n\}$  and keep only  $k$  columns of  $\mathbf{U}$ ,  $k$  rows and  $k$  columns of  $\Sigma$  and  $k$  rows of  $\mathbf{V}^T$ .

The same idea applies to the SVD of a  $m$  by  $n$  where  $m < n$ .

You'll see the idea of low-rank matrix approximation in the context of image compression in the hands-on project in Section 10.5.

### 10.4.2 Decomposition into Rank One Matrices

The SVD decomposes a matrix into left and right singular vectors and singular values. You can also consider it as the sum of  $k$  rank one matrices, with  $k$  being the number of singular values (and their left and right associated singular vectors) that you keep.

Let's start with the full SVD and take the example of a 4 by 3 matrix  $\mathbf{A}$  which has a rank of 3.

The shapes of  $\mathbf{U}$ ,  $\Sigma$ , and  $\mathbf{V}^T$  must respectively be  $(4 \times 4)$ ,  $(4 \times 3)$  and  $(3 \times 3)$ . Since the rank of the matrix is 3, there are only three nonzero singular values in  $\Sigma$ . For this reason, the matrix  $\Sigma$  is filled with 0 to match the shape of  $\mathbf{A}$ .

You can write  $\mathbf{A}$  as the following factorization:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$$

$$\begin{aligned} &= [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \mathbf{u}_3 \quad \mathbf{u}_4] \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \mathbf{v}_3^T \end{bmatrix} \\ &= [\sigma_1 \mathbf{u}_1 \quad \sigma_2 \mathbf{u}_2 \quad \sigma_3 \mathbf{u}_3] \begin{bmatrix} \mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \mathbf{v}_3^T \end{bmatrix} \\ &= \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \sigma_3 \mathbf{u}_3 \mathbf{v}_3^T \end{aligned}$$

Keep in mind that  $\mathbf{u}_1$ ,  $\mathbf{u}_2$ ,  $\mathbf{u}_3$  and  $\mathbf{u}_4$  are column vectors (the columns of the matrix  $\mathbf{U}$ ),  $\mathbf{v}_1$ ,  $\mathbf{v}_2$ , and  $\mathbf{v}_3$  are also column vectors, so their transpose  $\mathbf{v}_1^T$ ,  $\mathbf{v}_2^T$ ,  $\mathbf{v}_3^T$  are row vectors (the rows of the matrix  $\mathbf{V}^T$ ). However,  $\sigma_1$ ,  $\sigma_2$  and  $\sigma_3$  are scalars (the diagonal of the matrix  $\Sigma$ ).

Each element in this sum is a rank one matrix. Let's consider that these matrices are sorted according to the value of  $\sigma_i$  in descending order. The first element  $\sigma_1 \mathbf{u}_1 \mathbf{v}_1^T$  is the rank one matrix that best approximates  $\mathbf{A}$ . If you take the first two elements, you get the rank two matrix that best approximates  $\mathbf{A}$  and so on.

## 10.5 Hands-On Project: Image Compression

The SVD is used in many data science and machine learning methods. Let's see an example where the SVD is applied to images.

To visually observe the effect of low rank approximation using the SVD, you'll use this method to compress an image.

Let's start by loading the image<sup>5</sup>:

```
import imageio
img = imageio.imread('https://github.com/hadrienj/' \
    'essential_math_for_data_science/raw/master/data/birds_SVD.jpg')
img.shape
```

(819, 1024, 3)

You can see that the image is 819 pixels height, 1024 pixels width and has 3 colors. Let's display the image using Matplotlib:

```
plt.imshow(img)
```

---

<sup>5</sup>If you're curious, this pattern has been designed by William Morris in 1878: <https://www.metmuseum.org/art/collection/search/221485>.



*Figure 10.16: The example image. You'll use the SVD on it.*

The goal is to calculate the SVD of the matrix corresponding to this picture, and then to reconstruct the image from the rank one matrices. You'll see the effect of selecting an increasing number of these matrices, which are selected in descending order of the singular values. To do so, and since this is a color image, you'll have to calculate the SVD for each of the 3 colors.

Let's have a look at this line of code. Starting from the right, you take the first  $k$  rows and all columns of  $V^T$  (`V_transpose[:k, :]`), the first  $k$  singular values as a diagonal matrix (`np.diag(Sigma[:k])`) and all rows and the first  $k$  columns of  $U$  (`U[:, :k]`).

This returns the initial matrix reconstructed from the  $k$  rank one matrices.

Let's write this as a function looping across the three colors:

```
def reconstruct(img, k):
    reconstructed = np.zeros(img.shape)
    for i in range(img.shape[2]):
        U, Sigma, V_transpose = np.linalg.svd(img[:, :, i])
        reconstructed[:, :, i] = U[:, :k] @ np.diag(Sigma[:k]) @
        V_transpose[:k, :]
    return reconstructed
```

Let's try to reconstruct the image using only the first rank one matrix ( $k = 1$ ):

```
reconstructed = reconstruct(img, k=1)
```

Finally, you'll need to rescale the reconstructed matrix to be able to display it through Matplotlib.

## Display images with Matplotlib

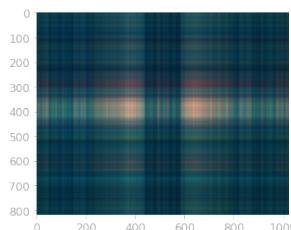
Matplotlib understands images as floats or integers. It expects floating values in the range 0 to 1, or integers in the range zero to 255.

To rescale the images between zero and one, you can use the following function:

```
def rescale_img(img):
    rescaled = (img - img.min()) / (img.max() - img.min())
    return rescaled
```

Let's plot the reconstructed image:

```
plt.imshow(rescale_img(reconstructed))
```



*Figure 10.17: The image reconstructed with the first component only.*

It is hard to see something in Figure 10.17: the image is way too compressed.

Let's try for larger values of  $k$ :

```
f, axes = plt.subplots(nrows=3, ncols=3, figsize=(16, 16))
f.subplots_adjust(hspace=0)

n_components = [1, 2, 3, 5, 10, 20, 50, 100, 700]

count = 0
```

```
for ax, n_component in zip(axes.flatten(), n_components):
    reconstructed = reconstruct(img, k=n_component)
    ax.axis('off')
    ax.imshow(rescale_img(reconstructed))
    ax.set_title(f'Reconstruction from\n{n_component} components')

    count += 1
```



*Figure 10.18: Reconstruction using more and more components up to 700.*

You can see in Figure 10.18 that even with 50 components, the reconstruction is not bad at all.

Let's compare the number of values in the compressed and the initial images. The initial image was 819 by 1024 for three colors, that is,  $819 * 1024 * 3 = 2,515,968$  values. Now, for 50 components you have the left singular vectors that are  $(m \times k)$ ,  $k$  singular values, and the right singular vectors that are  $(k \times k)$ . This gives you to

$$((819 \cdot 50) + 50 + (50 \cdot 50)) \cdot 3 = 130500$$

This is only 130,500 values, which is a reduction by a factor 20.

Using an image allows you to visualize nicely the effect of truncated SVD. However, you can use it in many different contexts, like to compress the weights matrices used in deep neural networks<sup>6</sup> to reduce the size of the parameters needed to make prediction without loosing too much accuracy.

---

<sup>6</sup>For instance, Xue, Jian, Jinyu Li, and Yifan Gong. “Restructuring of deep neural network acoustic models with singular value decomposition.” Interspeech. 2013.



# Conclusion

Congratulation! You reach the end of this book, designed to help you improve your math skills and increase your efficiency in data science and machine learning. I must admit that these last chapters were not easy, but they'll give you solid foundations for data related fields.

Learning the topics covered here will help you to understand what's under the hood of tools that you may already use. Let's take for instance the Support Vector Machines algorithm (SVM) for binary classification. If you read how it works, you'll see that the goal is to find hyperplanes separating the data with a maximum distance between them. To follow the description, you'll need for instance to understand vector equations, or the concept of distance expressed as norms. The purpose of *Essential Math for Data Science* is to give you all you need to dig more deeply about the algorithms you're interested in.

For this reason, the core math topics that you'll need are covered in this book: calculus, statistics and probability theory, and linear algebra. You can note that there is an emphasis on linear algebra: this content is more suited for people wanting to become data scientist or machine learning scientist than data analyst<sup>7</sup>.

As a side note, I think that even a short acculturation to the math concepts, the symbols, and the vocabulary encountered in data science and machine learning can stimulate your practice in the field and help you to dive into more specific resources if you need it.

---

<sup>7</sup>For instance, data analysts might need more details about inference, samples vs. population, etc.

**Next Steps** Now that you have learned about calculus, statistics, probability and linear algebra, you should be good to dive into more specialized content like seminal books in machine and deep learning<sup>8</sup>.

However, I recommend to keep a focus on practice while you continue to learn the theory. I like the vision of Rachel Thomas, co-founder and researcher at Fast.ai about the ‘top-down’ approach to learn deep learning (e.g.: <https://www.fast.ai/2016/10/08/teaching-philosophy/>): you start experimenting and building, and then you dive into the theory when you encounter limitations. This is opposed to the more traditional ‘bottom-up’ approach where you start with the foundations and then go to the applications.

This book is a bit between these two approaches: we start from the basics and move to more advanced concepts, but we try to use code and get our hands dirty from the beginning. The hands-on projects that you can find at the end of each chapter are also designed to show that you can use code to get more insights. The idea is that, when you study a theoretical concept, you can ask yourself: ‘is there a way to use code to verify or illustrate this concept?’

My opinion is that you should go forth and back between theory (and the math behind) and practical applications. Don’t wait to master all the math to start building data science pipelines or machine learning algorithms in your own projects. I think that it will help you to stay motivated while you learn and to see the big picture more easily.

---

<sup>8</sup>For instance, Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006, or Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016

# Appendix A

## Appendix A. Graphical Representation of Equations

In this section, you'll see a practical introduction to graphical representations of equations with Python.

### A.1 Plotting Equations

#### A.1.1 Two Variables

You'll use the function `plot()` from Matplotlib to plot the relationship between two variables. The `plot()` function draws lines between points, which are defined by coordinates, so you'll need at least two points to draw a linear equation.

Let's take an example. If you want to convert meters to feet, you'll need to know the relationship between these two units. In this case, 1 meter is approximately equal to 3.28 feet. If you call  $m$  the number of meters and  $f$  the number of feet, you can write:

$$f = 3.28m$$

For instance, the equation means that two meters corresponds to  $3.28 \cdot 2 = 6.56$  feet. It expresses the relationship between meters and feet. You can also

consider it as a *mapping* between the number of meters and the number of feet, that is to say, as a list of pairs of values.

meters	feet
1	3.28
2	6.56
3	9.84
:	:
10	32.8
:	:

You can take two points from this mapping and display  $m$  on the x-axis and  $f$  on the y-axis. For instance:

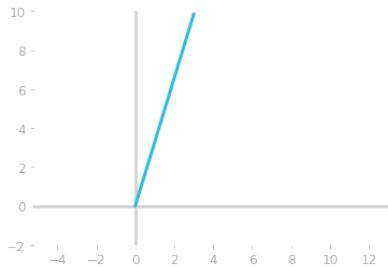
- The point at  $m = 0$ , that corresponds to  $f = 3.28 \times 0 = 0$ , giving coordinates zero on the  $x$ -axis and zero on the  $y$ -axis.
- The point at  $m = 3$ , that corresponds to  $f = 3.28 \times 3 = 9.84$ , giving coordinates  $(3, 9.84)$ .

Create a variable `x` that contains the  $x$  coordinates and a variable `y` that contains the  $y$  coordinates:

```
x = [0, 3]
y = [0, 9.84]
```

If you use `plt.plot()` to visualize this two points, Matplotlib will draw a line going from the coordinate  $(0, 0)$  to the coordinate  $(3, 9.84)$ .

```
plt.plot(x, y)
# [...] Add axes, styles etc.
```



*Figure A.1: Equation of the line corresponding to the relationship between feet and meters using two points.*

However, this procedure of connecting two points with a line will not work for nonlinear equations (those not represented by a line): you'll need more points for that.

To avoid manually filling in values from the equation mapping, you can create an array representing the  $x$  values with the resolution you need, for instance, all values in a range with a step of 0.1. Then, you calculate  $y$  for all values of  $x$ .

You can use `np.arange(start, stop, step)` that creates an array with values from `start` to `stop`. Each value is spaced by the value of `step`.

```
x = np.arange(-10, 11, 1)
x

array([-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2,
       3, 4, 5, 6, 7, 8, 9, 10])
```

The `x` array contains values from -10 to 10 (we stopped at 11 because the `stop` value is excluded) with a step of one.

Then, you can create a  $y$  that depends on  $x$ . We will take the example of the nonlinear sinusoid function  $\sin(x)$ :

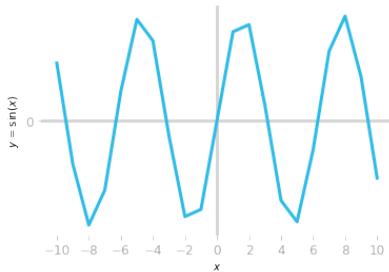
```
y = np.sin(x)
y

array([ 0.54402111, -0.41211849, -0.98935825, -0.6569866 ,  0.2794155 ,
       0.95892427,  0.7568025 , -0.14112001, -0.90929743, -0.84147098,
```

```
0.          ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
-0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849,
-0.54402111])
```

Now you have two arrays that you can plot.

```
plt.plot(x, y)
# [...] Add axes, styles etc.
```



*Figure A.2: Sinusoid function represented with a low resolution.*

Figure A.2 looks like a broken sine function! This is because you did not take enough points and thus the resolution is bad. Let's do it again with a different step.

```
x = np.arange(-10, 11, 0.1)
y = np.sin(x)
```

This time let's take a step of 0.1, leading to more values in both arrays (see Figure A.3).

```
plt.plot(x, y)
# [...] Add axes, styles etc.
```

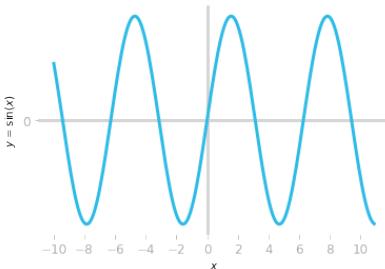


Figure A.3: Sinusoid function with a better resolution

Figure A.3 shows the sinusoid function with a better resolution.

### A.1.2 More Than Two Variables

The plots you did in the last section used two axes, meaning that only two variables can be used. How can you do to visualize the relationship between more than two variables?

Let's take an example of an equation with three variables.

$$z = 2x + 3y + 8$$

Each point of the mapping is this time defined by three values instead of two, and thus you'll need three dimensions to plot this equation. This also means that you'll not plot a curve but a plane.

Let's use a 3D plot to represent it, with the function `plot_surface()` from Matplotlib.

To do so, you'll need to draw a grid of values. First, the Numpy function `np.linspace(start, stop, num)` creates array of `num` values from `start` to `stop`:

```
np.linspace(-10, 10, 100)

array([-10.        , -9.7979798 , -9.5959596 , -9.39393939,
       -9.19191919, ...,  9.19191919,   9.39393939,   9.5959596 ,
       9.7979798 ,  10.        ])
```

You can use the function `np.meshgrid()` from Numpy to create all possible combinations of these 1D arrays. You give it two one-dimensional arrays and it returns two two-dimensional arrays:

```
x, y = np.meshgrid(np.linspace(-10, 10, 100), np.linspace(-10, 10, 100))
x.shape
```

```
(100, 100)
```

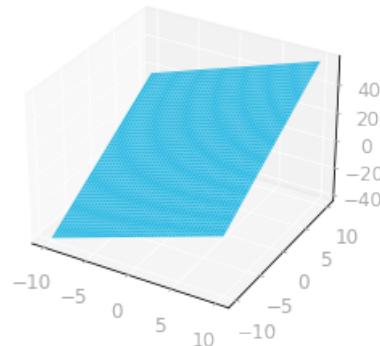
All pairs of values from `x` and `y` gives all combinations of values from -10 to 10. Each combination corresponds to one point of coordinates `x` and `y`, and you can calculate `z` from these two values according to the equation above.

```
z = 2 * x + 3 * y + 8
z.shape
```

```
(100, 100)
```

Let's try to plot the function:

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z)
```



*Figure A.4: Three-dimensional plot of a plane.*

Figure A.4 shows a plane corresponding to your equation.

Note that we represented it with arbitrary limitations (between -10 and 10 for  $x$  and  $y$ ) but it is infinite, just as with lines.

## A.2 Slope And Intercept

Lines in the Cartesian plane, and thus linear equations, are defined by two parameters. These two parameters correspond graphically to the *slope* and the *y-intercept* (the  $y$  value when  $x = 0$ ) of the line defined by the equation.

Let's take the example of the following equation,  $a$  is the slope and  $b$  the *y*-intercept:

$$y = ax + b$$

### A.2.1 Slope

The slope  $a$  is the parameter multiplied by  $x$ : it tells you that when you increase  $x$  of one unit,  $y$  increases of  $ax$  units. It tells how much  $y$  changes in comparison to  $x$ . This is the ratio of the change on the  $y$ -axis over the change on the  $x$ -axis.

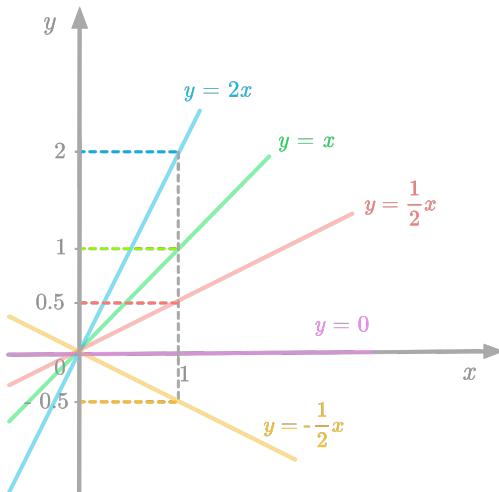


Figure A.5: Linear equations with different slopes.

Figure A.5 shows examples of linear equations with the  $y$ -intercept equal to zero ( $b = 0$ ) and different values of the slope. Note that when the slope equals zero, the function is constant (same  $y$  for all  $x$ ).

When you go from 0 to 1 on the  $x$ -axis,  $y$  changes accordingly to the slope. For instance, the blue line has a slope equal to 2, meaning that you add 2 on the vertical axis each time you add 1 on the horizontal axis. A negative slope means that you add negative values to  $y$  each time you add positive values to  $x$ .

### A.2.2 $y$ -Intercept

The  $y$ -intercept corresponds to the value on the vertical axis when  $x = 0$ . This value doesn't affect the slope of the line.

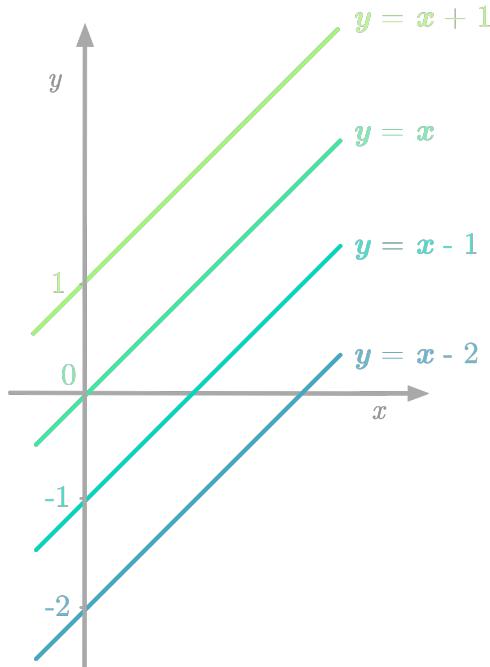


Figure A.6: Linear equations with different values of  $y$ -intercept and a fixed slope of 1.

Figure A.6 shows lines corresponding to linear equations with different  $y$ -intercepts. You can see that the lines are parallel (because the slope is the

same) and vertically shifted.



# Appendix B

## Appendix B. Mathematical Notation

An important step to increase your math skills is to understand what the symbols denote. It is important, to make the most of this book, that you understand the meaning of each math symbol you'll encounter. For this reason, you can find here all the mathematical notations used in this book.

Note that mathematical notation can be slightly different from one book to another. A large part of the notation conventions I use here comes from the Deep Learning book<sup>1</sup>

### B.1 Greek Letters

$\alpha$ : Alpha. Regularization parameter.

$\Delta$ : Capital delta. Distance between two points used to calculate the derivative for instance.

$\epsilon$ : Epsilon.

$\lambda$ : Lambda. Number of events in a Poisson distribution. It can also refer to the eigenvalues.

---

<sup>1</sup>Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016. You can find the detailed notation page corresponding to the book here: [https://github.com/goodfeli/dlbook\\_notation](https://github.com/goodfeli/dlbook_notation)

$\mu$ : Mu. The mean of a distribution.

$\nabla$ : Nabla. Applied to a function, it refers to its gradient.

$\prod$ : Capital Pi. The product notation, or Pi notation corresponds to a repeated product of the expression following it. It is similar to the Sigma notation, but for a product instead of a sum.

$\sigma$ : Sigma. Standard deviation. It can also refer to the singular values.

$\Sigma$ : Capital sigma. It is used to write sums (Sigma notation): it refers to a repeated sum of the expression following it. It can also refer to the matrix containing all the singular values.

$\theta$ : Theta. Models parameters.

## B.2 Calculus

$\lim_{a \rightarrow 0}$ : Limit when  $a$  approaches zero.

$\frac{df(x)}{dx}$ : Derivative of  $f(x)$  with respect to  $x$ .

$f'(x)$ : Derivative of  $f(x)$ .

$\partial$ : Partial derivative.

$\int f(x) dx$ : Integral of  $f(x)$  with respect to  $x$ .

## B.3 Dataset

$x^{(i)}$ : The  $i$ th observation in a dataset.

$\hat{y}$ : In the context of cost functions, “y hat” refers to the value of  $y$  estimated by the model.

$L$ : A loss function.

$J(\theta)$ : A cost function.

## B.4 Probability

$\bar{x}$ : x bar. Arithmetic mean of the variable  $x$ .

$\text{Var}(\mathbf{x})$ : Variance of the vector  $\mathbf{x}$ .

$\text{Cov}(\mathbf{x}, \mathbf{y})$ : Covariance of the vectors  $\mathbf{x}$  and  $\mathbf{y}$ .

$\text{Corr}(\mathbf{x}, \mathbf{y})$ : Correlation between the vectors  $\mathbf{x}$  and  $\mathbf{y}$ .

$\mathbb{E}_{X \sim P}[X]$ : Expected value of the random variable  $X$  that has a distribution  $P$ .

$S$ : Sample space

---

$X \sim \mathcal{P}$ : The random variable  $X$  has a distribution  $\mathcal{P}$ .

$\mathcal{N}$ : Normal distribution.

$\mathcal{N}(X = x; \mu, \sigma^2)$ : Normal distribution as a function of  $x$  and parameterized by the distribution parameters  $\mu$  and  $\sigma^2$ .<sup>2</sup>

Bern: Bernoulli distribution.

Bin: Binomial distribution.

Poi: Poisson distribution.

Exp: Exponential distribution.

---

$X$ : Random variable corresponding to a random experiment.

$x$ : An outcome (also called state, or realization) of the random variable  $X$  (more precisely, it is the outcome of the random experiment corresponding to the random variable).

$A$ : An event (a set of outcomes).

---

In the following definitions, the uppercase  $P$  are for discrete random variable and the lowercase  $p$  for continuous random variables.

---

<sup>2</sup>You can also find the use of a pipe symbol ( $|$ ) similar to conditional probabilities in some resources, as in Bishop, Christopher M. Pattern recognition and machine learning. Springer, 2006., Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong. Mathematics for machine learning. Cambridge University Press, 2020., or Murphy, Kevin P. Machine learning: a probabilistic perspective. MIT press, 2012.

$P(X)$ : Probability Mass Function of the discrete random variable  $X$ . This function takes a possible outcome of  $X$  and returns a probability.

$P(X = x)$  (or simply  $P(x)$ ): Probability that the discrete random variable  $X$  takes the value  $x$ . This is a number between 0 and 1.

$P(A)$ : Probability that the event  $A$  occurs. For instance, if the event corresponds to the the outcome  $x$ , you have  $P(A) = P(X = x)$ .

$P(X = x, Y = y)$  (or simply  $P(x, y)$ ): Joint probability. Probability that the discrete random variables  $X$  and  $Y$  respectively take the values  $x$  and  $y$ .

$P(X, Y)$ : Joint probability distribution function of the discrete random variables  $X$  and  $Y$ . This is a function that takes a possible outcome of  $X$  and  $Y$  and returns a probability.

$P(X = x|Y = y)$  (or simply  $P(x|y)$ ): Conditional probability. Probability that the discrete random variable  $X$  takes the value  $x$  given that the discrete random variable  $Y$  takes the value  $y$ .

---

$\mathcal{L}_x(\theta)$ : Likelihood of observing some data  $x$  drawn from a distribution with parameter  $\theta$ .

---

$\binom{N}{m}$ :  $N$  choose  $m$  is the binomial coefficient.

---

$N!$ : Factorial N.

## B.5 Information Theory

$I(x)$ : Shannon information of the event  $X = x$ . The input is a single outcome.

$H(P)$ : Shannon entropy of the probability distribution  $P$ .

$H(P, Q)$ : Cross-entropy between the probability distributions  $P$  and  $Q$ .

$D_{KL}$ : Kullback-Leibler divergence (or KL divergence).

## B.6 Sets

$\mathbb{R}$ : The set of the real number.

## B.7 Linear Algebra

$\mathbf{x}$ : A vector.

$x_i$ : Component at index  $i$  of the vector  $\mathbf{x}$  (first index is 1).

$\mathbf{i}$  and  $\mathbf{j}$ : The basis vectors corresponding to the  $x$  and  $y$  axis in the Cartesian plane.

$\mathbf{A}$ : A matrix.

$A_{i,j}$ : Component at row  $i$  and column  $j$  of the matrix  $\mathbf{A}$ .

$\mathbf{A}_{i,:}$ : Row  $i$  of the matrix  $\mathbf{A}$ .

$\mathbf{A}_{:,i}$ : Column  $i$  of the matrix  $\mathbf{A}$ .

$T$ : A linear transformation.

$\mathbf{A}^{-1}$ : The inverse of the matrix  $\mathbf{A}$ .

$\mathbf{A}^+$ : The Moore-Penrose inverse (or pseudo-inverse) of the matrix  $\mathbf{A}$ .

