A.Y. 2026/27 · LECTURE NOTES

Fundamentals of Data Analysis

Fondamenti di Analisi dei Dati

Antonino Furnari · University of Catania

https://antoninofurnari.github.io/fad-2627/



LECTURE NOTES

Contents

Lecture 1 — Course introduction and Python 4

1. Setting up the working environment 4

2. Python crash course 11

3. Python for data science crash course 49

Lecture 2 — Key concepts 132

4. Key concepts, loading and inspecting the data 132

Lecture 3 — Describing and visualizing the data 166

5. Describing and visualizing the data 166

Lecture 4 — Probability 220

6. Probability for data analysis 220

Lecture 5 — Association between variables 252

7. Association between variables 252

Lecture 6 — Data distributions 289

8. Data distributions 289

Lecture 7 — Statistical inference 327

9. Statistical inference 327

Lecture 9 — Linear regression 376

10. Linear regression 376

Lecture 10 — Logistic regression 420

11. Logistic regression 420

Lecture 11 — Extending regression models 445

12. Extending regression models: interactions, polynomial and multinomial terms 445

Lecture 12 — Causal analysis 461

13. Causal analysis 461

Lecture 13 — Storytelling with data 499

14. Storytelling with data 499

Lecture 15 — Introduction to predictive analysis 501

15. Introduction to predictive analysis 501

Lecture 16 — Regression for prediction 531

16. Regression for prediction: overfitting, regularization and the scikit-learn workflow 531

Lecture 17 — Classification 564

17. Classification, evaluation measures and KNN 564

Lecture 18 — Generative classifiers 635
18. Generative classifiers: QDA, LDA and naive Bayes 635
Lecture 19 — Data representation and clustering 655
19. Data representation 655
20. Clustering 691
Lecture 20 — Density estimation 734
21. Density estimation 734
Lecture 21 — Dimensionality reduction (PCA) 761
22. Dimensionality reduction: principal component analysis 761

Setting up the working environment

Chapter resources

Open in Colab | Practice quiz

This course includes lab hours where we will tackle practical data analysis manipulations. All labs will be conducted using the Python language and Jupyter notebooks (either through a local installation or Google Colab). We will start by seeing how to set up the work environment.

Installing Python

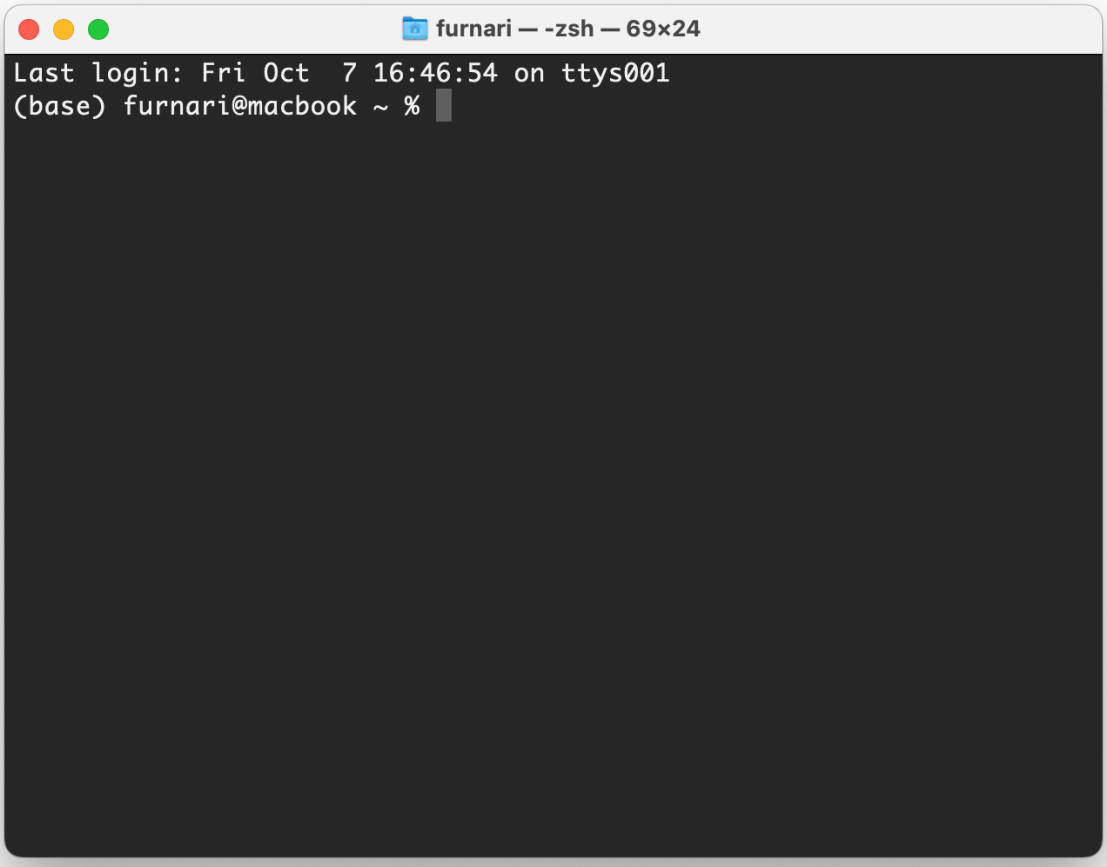
In this course, we will use the Python language. We choose Python as it is a modern, widespread, multi-platform, portable, easy-to-use, and elegant language that features an interactive mode. This last feature will be very useful for experimenting with code and learning how the language and its libraries work.

Python is a highly modular language: some basic functionalities are included in the language, while many others are provided by third-party packages, managed through a dedicated package manager called `pip`. To have a ready-to-use work environment for data analysis applications, we will use a distribution called Anaconda. It should be noted that in the past there were two branches of Python: 2.x and 3.x. Today, 2.x is no longer used. The Anaconda distribution used at the time of this writing will install Python 3.x.

Let's download the distribution for our platform from the following link: https://www.anaconda.com/products/distribution and install it. Once this is done, we should be able to access a terminal with access to the executables made available by Anaconda. Specifically:

- On Windows, the "Anaconda Prompt" program will be available;
- On MacOS and Linux, the executables should already be in the user's path, so it should be sufficient to open a standard terminal.

If Anaconda has been installed correctly, the indication `(base)` will appear before the command prompt, as shown below:



`(base)` indicates the `environment` (by default, it's the base environment) in which we are operating. Indeed, Anaconda allows you to define and use different environments that are independent of each other, in which you can install different packages (even different versions).

Programming in Python

Several tools will be installed with Anaconda. Among them:

- The Python interpreter;
- The ipython interactive shell;
- The Spyder IDE;
- Jupyter Notebook.

Python Interpreter

The Python interpreter allows you to run a Python program using a command like:

```
python program.py
```

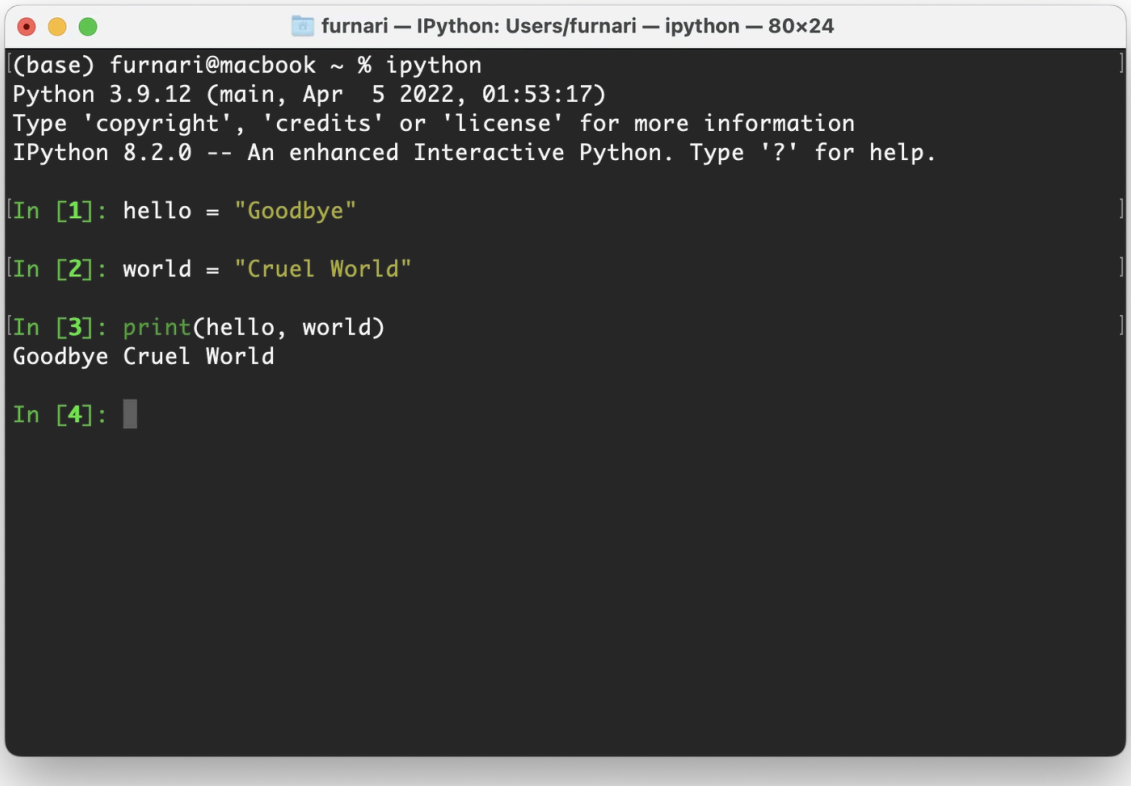
Where `program.py` is the source text file of the program. On the first run, Python will compile the program and generate a bytecode file `program.pyc`.

ipython Interactive Shell

The interactive shell is a more "advanced" version of the Python interpreter that allows you to:

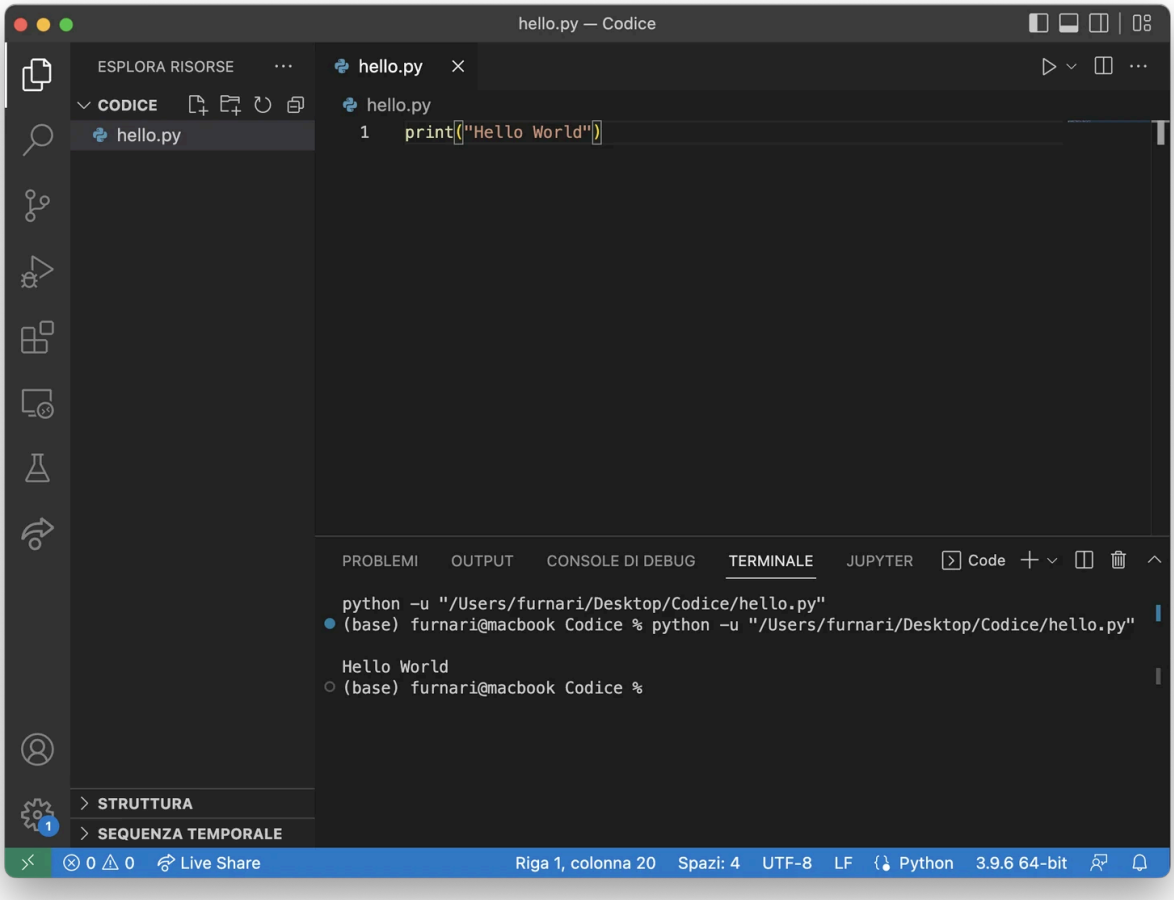
- Interpret commands;
- Run Python programs;
- Analyze the content of workspace variables;

We can launch the interactive shell with the `ipython` command:



IDE

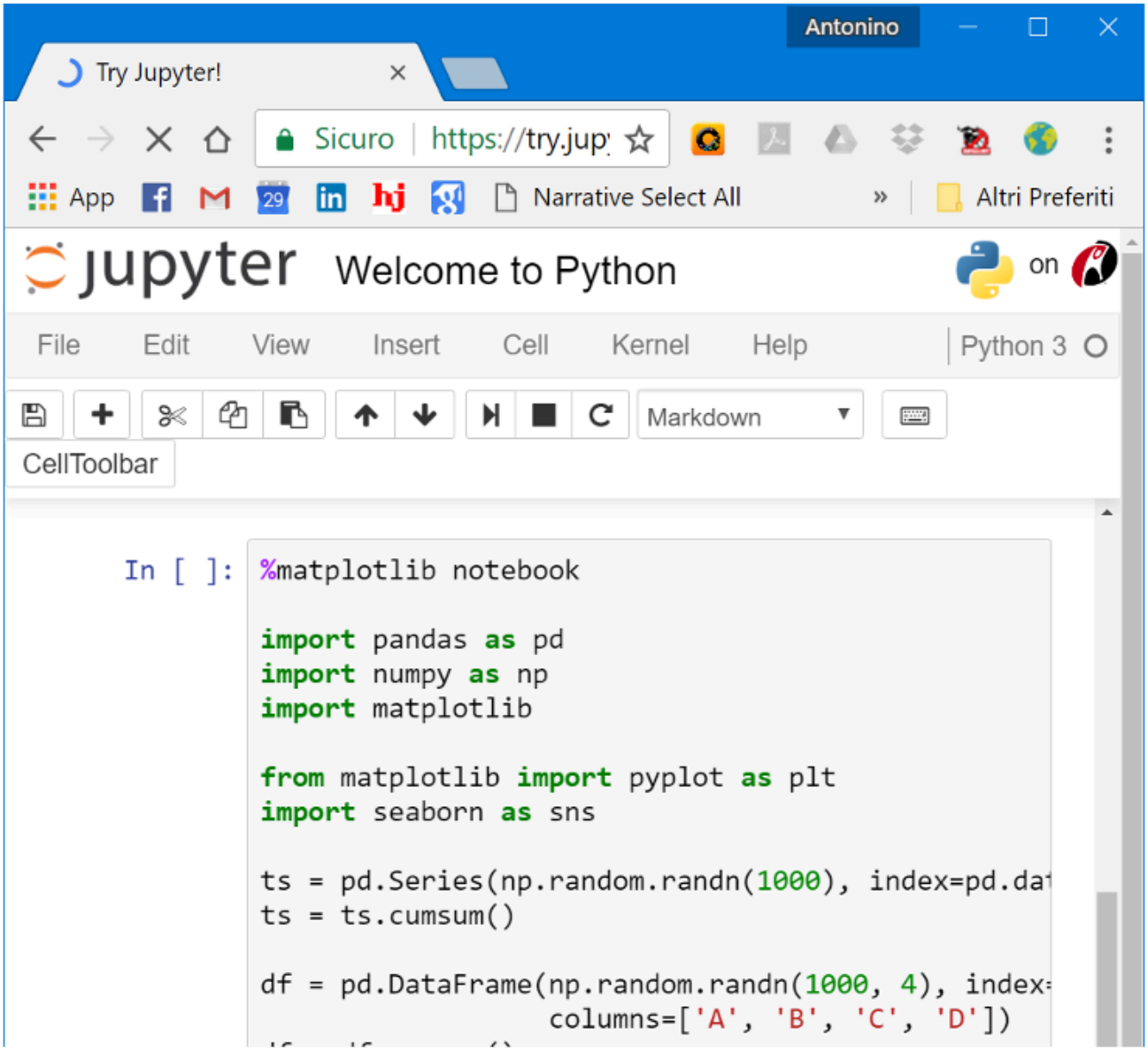
An IDE generally integrates an ipython shell and various debugging tools. It is an excellent tool for medium to large-sized projects. A widely used IDE for Python programming is Visual Studio Code.



Jupyter Notebook

You can start Jupyter with the command:

```
jupyter notebook
```



Jupyter allows you to create (via a web interface) “notebooks,” which are files containing:

- Formatted text;
- The code to be executed;
- The images obtained as results of the code’s execution.

It is a very powerful tool, as it allows for the generation of actual experimental reports.

IDE vs Notebook

One might wonder when it is useful to use one tool over the others.

IDEs are great for:

- medium or large projects, with multiple modules and classes;
- interactive computation (e.g., processing a video in real time).

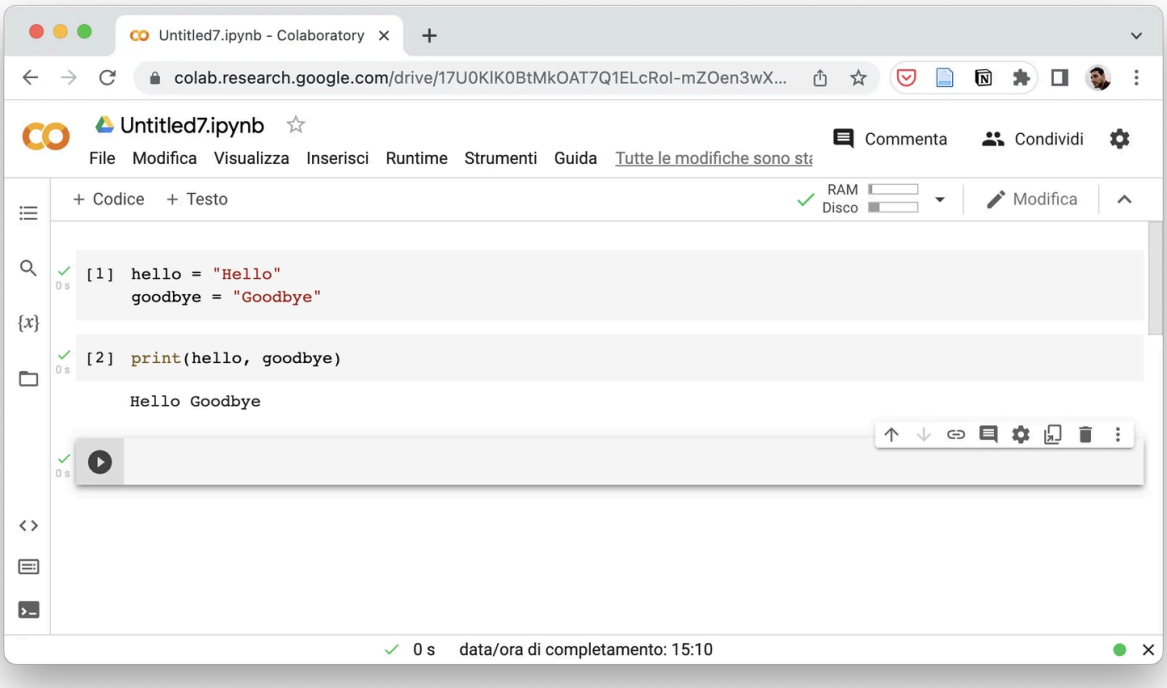
Jupyter Notebooks are great for:

- documenting data analysis processes;
- experimenting with new ideas;
- writing documentation and tutorials.

Google Colab

As an alternative to installing a work environment on your own computer, you can use a free service provided by Google called “Colab”: https://colab.research.google.com/

The service provides Python notebooks (similar to Jupyter) with GPU support.



In Google Colab, we will find the main packages already installed, but we can install additional packages using commands like:

```
pip install sklearn
```

Limited Resources

Google Colab is an excellent tool for experimenting or conducting small experiments. However, keep in mind that resources are limited, and GPU access may be revoked after a few hours of computation or if the machine is idle for a long time. Therefore, it might be difficult to use it for medium-to-large projects.

Data Persistence

With each notebook execution, Google Colab provides a temporary storage space where you can write and read data. Every time the notebook is closed and reopened, this space is cleared, and its content is

deleted. If you want to access persistent storage, you can mount a specific folder from your Google Drive by running a cell with the following code:

```
from google.colab import drive
drive.mount('/content/drive/path/to/folder')
```

Where `/path/to/folder` is the path to the folder within your Drive.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Python crash course

Chapter resources

Open in Colab | Practice quiz

Note for those who are already familiar with Python: Feel free to skim through the content and zoom into only those parts which you find new and interesting.

Python is a high-level, interpreted programming language designed for "general purpose" programming. Python supports a dynamic type system and various programming paradigms, including object-oriented programming, imperative programming, functional programming, and procedural programming. The language was created in 1991 by Guido van Rossum and its name is inspired by the satirical TV series Monty Python's Flying Circus (https://en.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus).

Although Python was not originally conceived as a programming language for scientific computing, its extreme versatility has contributed to the emergence of a series of libraries that make numerical computation in Python convenient and efficient. A good portion of these libraries are part of "SciPy" (https://www.scipy.org/), an open-source software ecosystem for scientific computing. In this laboratory, in addition to an introduction to Python, we will specifically cover the fundamentals of NumPy (scientific computing) and Matplotlib (2D Plotting).

Important references to consult during the course, only the following documentation:

- Python 3: https://docs.python.org/3/index.html;
- Numpy: http://www.numpy.org/;
- Matplotlib: https://matplotlib.org/.

Numbers

The numeric data types in Python are int, float, and complex. We will focus on int and float. Some examples of operations between numbers:

```
3 + 5 #sum
```

```
8
```

```
2 - 8 #difference
```

```
-6
```

```
3 * 5 #product
```

```
15
```

```
3 / 2 #division, note that in Python 3, integer division returns a float
```

```
1.5
```

```
3 // 2 #integer division
```

```
1
```

```
9 ** 2 #exponentiation
```

```
81
```

```
4 % 2 #modulus
```

```
0
```

```
(1 + 4) * (3 - 2) # Use of parentheses
```

```
5
```

Variables and Types

In general, numbers without a decimal point are interpreted as int, while those with a decimal point as float. Since Python is dynamically typed, we do not need to explicitly declare the type of a variable. The type will be associated with the variable as soon as we assign a value to it. We can check the type of a variable using the `type` function:

```
x = 3
y = 9
z = 1.5
h = x/y
l = x//y
type(x), type(y), type(z), type(h), type(l)
```

```
(int, int, float, float, int)
```

It is possible to cast from one type to another using the `int` and `float` functions:

```
int(2.5)
```

```
2
```

```
float(3)
```

```
3.0
```

As in C, the “in place” operations between variables are defined:

```
x = 8
y = 12
x+=y #equivalent to x=x+y
x
```

```
20
```

The notations “++” and “–” are not defined.

```
a++ #error!
```

To increment by one unit, the `+=1` notation must be used:

```
a=1
a+=1
a
```

```
2
```

Question 1

What is the type of the following variable?

```
x = (3//2*2)**2+(3*0.0)
```

Booleans

Booleans are represented by the keywords `True` and `False` (both start with a capital letter).

```
print(True)
print(False)
```

```
True
False
```

```
type(True)
```

```
bool
```

It is possible to generate booleans using comparison operators:

```
5==5
```

```
True
```

```
7==5
```

```
False
```

```
5>4
```

```
True
```

```
9>10
```

```
False
```

```
9<=10
```

```
True
```

```
11<=10
```

```
False
```

The logical operators are `and` and `or`:`

```
print(5==5 and 3<5)
print(3>5 or 3<5)
print(3>9 or 3<2)
```

```
True
True
False
```

It is possible to perform a check on types using `type` and `==`:

```
type(2.5)==float
```

```
True
```

```
type(2)==float
```

```
False
```

Alternatively, it is possible to use the `isinstance` function:

```
isinstance(2.5,float)
```

```
True
```

```
isinstance(2.5,int)
```

```
False
```

The `isinstance` function is particularly convenient when you want to check that a variable belongs to one of a set of types. For example, if we want to check that a variable contains a number:

```
isinstance(2.5,(float,int))
```

```
True
```

```
isinstance(5,(float,int))
```

```
True
```

Print

Printing is done using the `print` function:

```
var = 2.2
print(var)
```

```
2.2
```

We can print an empty line by omitting the print parameter:

```
print(2)
print()
print(3)
```

```
2

3
```

Alternatively, we can specify to insert two “newlines” at the end of the printout by specifying the parameter `end="\n\n"` (“” is a string - we will delve into strings later):

```
print(2, end="\n\n")
print(3)
```

```
2

3
```

The same method can be used to omit the insertion of spaces between two consecutive prints:

```
print(2, end="") #"" represents an empty string
print(3)
```

```
23
```

We can print multiple elements consecutively by separating the `print` arguments with commas. Furthermore, the `print` function allows printing numbers as well as strings:

```
print(1,8/2,7,14%6,True)
```

```
1 4.0 7 2 True
```

Lists

Lists are a sequential data structure that can be used to represent sequences of values of any type. Lists can also contain elements of mixed types. A list is defined using square brackets:

```
l = [1,2,3,4,5] #this is a list (square brackets)
print(l)
```

```
[1, 2, 3, 4, 5]
```

Lists can be indexed using square brackets. Indexing starts from 0 as in C:

```
print(l[0],l[2])
l[0]=8 #assignment of a new value to the first memory location
print(l)
```

```
1 3
[8, 2, 3, 4, 5]
```

It is possible to add new values to a list using the `append` function:

```
l = []
print(l)
l.append(1)
l.append(2.5)
l.append(8)
l.append(-12)
print(l)
```

```
[]
[1, 2.5, 8, -12]
```

Lists can be concatenated using the sum operator:

```
l1 = [1,5]
l2 = [4,6]
print(l1+l2)
```

```
[1, 5, 4, 6]
```

The multiplication operator can be used to repeat a list. For example:

```
l1 = [1,3]
print(l1*2) # Concatenates l1 to itself twice
```

```
[1, 3, 1, 3]
```

Using the multiplication operator, it is possible to quickly create lists with an arbitrary number of equal values. For example:

```
print([0]*5) #list of 5 zeros
print([0]*4+[1]*1) #4 zeros followed by 1 one
```

```
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 1]
```

The length of a list can be obtained using the `len` function:

```
print(l2)
print(len(l2))
```

```
[4, 6]
2
```

For lists, an ordering is defined that depends on their length: shorter lists are “less than” longer lists:

```
print([1,2,3]<[1,2,3,4])
print([1,2,3,5]⩾[1,2,3,4])
```

```
True
True
```

The `==` operator does not check if the lengths are equal, but verifies that the content of the two lists is effectively equal:

```
print([1,2,3]==[1,2,3])
print([1,2,3]==[1,3,2])
```

```
True
False
```

It is possible to check that an element belongs to the list using the keyword `in`:

```
print(7 in [1,3,4])
print(3 in [1,3,4])
```

```
False
True
```

The functions `max` and `min` can be used to calculate the maximum and minimum of a list:

```
l=[-5,2,10,6]
print(max(l))
print(min(l))
```

```
10
-5
```

It is possible to remove a value from a list using the `remove` method:

```
l=[1,2,3,4,2]
print(l)
l.remove(2)
print(l)
```

```
[1, 2, 3, 4, 2]
[1, 3, 4, 2]
```

However, this method only removes **the first occurrence** of the passed value. If we want to remove a value identified by a specific index, we can use the `del` construct:

```
l=[1,2,3,4,2]
print(l)
del l[4]
print(l)
```

```
[1, 2, 3, 4, 2]
[1, 2, 3, 4]
```

Additionally, to access the last element and remove the last element, we can use the `pop` method:

```
l=[1,2,3,4,5]
print(l)
print(l.pop())
print(l)
```

```
[1, 2, 3, 4, 5]
5
[1, 2, 3, 4]
```

Indexing and Slicing

It is possible to extract a sublist from a list by specifying the first index (inclusive) and the last index (exclusive) separated by the `:` symbol. This notation is somewhat reminiscent of C++ string's `substr` method.

```
l = [1,2,3,4,5,6,7,8]
print("List l     →", l)
print("l[0:3]      →", l[0:3]) #from index 0 (inclusive) to index 3 (exclusive)
print("l[1:2]      →", l[1:2]) #from index 1 (inclusive) to index 2 (exclusive)
```

```
List l     → [1, 2, 3, 4, 5, 6, 7, 8]
l[0:3]      → [1, 2, 3]
l[1:2]      → [2]
```

When the first index is omitted, it is automatically replaced with "0":

```
print("l[:2]       →", l[:2]) # from index 0 (inclusive) to index 2 (exclusive)
# equivalent to the following:
print("l[0:2]      →", l[0:2]) # from index 0 (inclusive) to index 2 (exclusive)
```

```
l[:2]       → [1, 2]
l[0:2]      → [1, 2]
```

Similarly, if we omit the second index, it is replaced with the last index of the list:

```
print("Last index of the list:",len(l))
print("l[3:]       →", l[3:]) #from index 3 (inclusive) to index 5 (exclusive)
#equivalent to the following:
print("l[3:8]      →", l[3:8]) #from index 3 (inclusive) to index 5 (exclusive)
```

```
Last index of the list: 8
l[3:]       → [4, 5, 6, 7, 8]
l[3:8]      → [4, 5, 6, 7, 8]
```

Omitting both indices:

```
print("l[:]         →", l[:]) #from index 0 (inclusive) to index 5 (exclusive)
#equivalent to:
print("l[0:8]       →", l[0:8]) #from index 0 (inclusive) to index 5 (exclusive)
```

```
l[:]         → [1, 2, 3, 4, 5, 6, 7, 8]
l[0:8]       → [1, 2, 3, 4, 5, 6, 7, 8]
```

It is also possible to specify the “step”, as a third number separated by another symbol `:`:

```
print("l[0:8:2]    →", l[0:8:2])
# from 0 (inclusive) to 8 (exclusive) with a step of 2 (every other element)
# equivalent to:
print("l[::2]      →", l[::2])
# from 0 (inclusive) to 8 (exclusive) with a step of 2 (every other element)
```

```
l[0:8:2]    → [1, 3, 5, 7]
l[::2]      → [1, 3, 5, 7]
```

To reverse the order of elements, it is also possible to specify a negative step. In this case, you must ensure that the first index is greater than the second:

```
print("l[5:2:-1]    →", l[5:2:-1])
# from 5 (inclusive) to 2 (exclusive) with a step of -1
print("l[2:5:-1]    →", l[2:5:-1])
# in this case, the first index is smaller than the second,
# so the result will be an empty list
```

```
l[5:2:-1]    → [6, 5, 4]
l[2:5:-1]    → []
```

In this case too, by omitting indices, these will be replaced with the most obvious choices. In the case of omission, however, the conditions for inclusion and exclusion of indices change. Let's see some examples:

```python
print("l[:2:-1]     →", l[:2:-1])
#from the last index (inclusive) to 2 (exclusive) with a step of -1
#equivalent to:
print("l[8:2:-1]    →", l[8:2:-1])

print()
print("l[3::-1]     →", l[3::-1])
#from the third index (inclusive) to 0 (inclusive, as omitted) with a step of -1
#similar, but not equivalent to:
print("l[3:0:-1]    →", l[3:0:-1])
#from the third index (inclusive) to 0 (exclusive) with a step of -1

print()
print("l[::-1]      →", l[::-1])
#from the last index (inclusive) to the first (inclusive, as omitted) with a step of
        -1
#similar, but not equivalent to:
print("l[8:0:-1]    →", l[8:0:-1])
#from the last index (inclusive) to the first (exclusive) with a step of -1
```

```
l[:2:-1]     → [8, 7, 6, 5, 4]
l[8:2:-1]    → [8, 7, 6, 5, 4]

l[3::-1]     → [4, 3, 2, 1]
l[3:0:-1]    → [4, 3, 2]

l[::-1]      → [8, 7, 6, 5, 4, 3, 2, 1]
l[8:0:-1]    → [8, 7, 6, 5, 4, 3, 2]
```

The notation `::-1`, in particular, is useful for reversing lists:

```python
print(l)
print(l[::-1])
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[8, 7, 6, 5, 4, 3, 2, 1]
```

Indexing and slicing can also be used to assign values to list elements. For example:

```
l = [5,7,9,-1,2,6,5,4,-6]
print(l)
l[3]=80
print(l)
```

```
[5, 7, 9, -1, 2, 6, 5, 4, -6]
[5, 7, 9, 80, 2, 6, 5, 4, -6]
```

It's also possible to assign more than one element at a time:

```
l[::2]=[0,0,0,0,0] # assign 0 to odd-positioned numbers
print(l)
```

```
[0, 7, 0, 80, 0, 6, 0, 4, 0]
```

The lists can also be nested:

```
a1 = [1,2,[4,8,[7,5]],[9],2]
print(a1)
```

```
[1, 2, [4, 8, [7, 5]], [9], 2]
```

The indexing of these nested structures occurs by concatenating the indices as follows:

```
print(a1[2][2][0]) #the first index selects the list [4,8,...]
#the second index selects the list [7,5]
#the third index selects the element 7
```

```
7
```

Question 2

Extract the list `[3, 1.2]` from the following list:

```
l = [1, 4, 5, [7, 9, -1, [0, 3, 2, 1.2], 8, []]]
```

Tuples

Tuples are similar to lists, but they are immutable. That is, they cannot be modified after their initialization. Unlike lists, tuples are defined using parentheses:

```
l = [1,2,3,4,5] # this is a list (square brackets)
t = (1,2,3,4,5) # this is a tuple (round brackets)
print(l)
print(t)
```

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

The indexing and slicing rules seen for lists also apply to tuples. However, as mentioned before, tuples cannot be modified:

```
t = (1,3,5)
t[0]=8 #returns an error because tuples are immutable
```

Initializing a tuple with a single element will produce a number. This is because parentheses are also used to group the different terms of an operation:

```
t=(1)
print(t)
```

```
1
```

To define a one-dimensional tuple, we must explicitly add a comma after the first element:

```
t=(1,)
print(t)
```

```
(1,)
```

It is also possible to omit the parentheses in the definition of tuples:

```
t1=1,3,5
t2=1,
print(t1,t2)
```

```
(1, 3, 5) (1,)
```

It is possible to convert tuples into lists and vice versa:

```
l=[1,2,3,4,5,6,7,8]
t=(4,5,6,7,4,8,2,4)
ttl = list(t)
ltt = tuple(l)

print(ttl)
print(ltt)
```

```
[4, 5, 6, 7, 4, 8, 2, 4]
(1, 2, 3, 4, 5, 6, 7, 8)
```

Tuples can also be created and “unpacked” on the fly:

```
t1=1,2,3

print(t1)

a,b,c=t1 # Unpacking the tuple
print(a,b,c)
```

```
(1, 2, 3)
1 2 3
```

This system allows to swap two variables in a single line of code:

```
var1 = "Var 1"
var2 = "Var 2"

print(var1,var2)

var1,var2=var2,var1
print(var1,var2)

#equivalent to:
var1 = "Var 1"
var2 = "Var 2"
t = (var2,var1)
var1=t[0]
var2=t[1]
print(var1,var2)
```

```
Var 1 Var 2
Var 2 Var 1
Var 2 Var 1
```

Nested tuples can be unpacked as follows:

```
t = (1,(2,3),(4,5,6))
x,(t11,t12),(t21,t22,t23) = t
print(t)
print(x, t11, t12, t21, t22, t23)
#The notation a,b,c,d,e,f = t would raise an error
```

```
(1, (2, 3), (4, 5, 6))
1 2 3 4 5 6
```

Question 3

What is the main difference between tuples and lists? Provide an example of a case where a tuple is a more appropriate type than a list.

Strings

In Python we can define strings in three ways:

```
s1 = 'Single quotes'
s2 = "Double quotes, can also contain single quotes '' "
s3 = """Triple
double quotes
can be defined on multiple lines"""
type(s1), type(s2), type(s3)
```

```
(str, str, str)
```

Printing takes place using the “print” function:

```
print(s1)
print(s2)
print(s3)
```

```
Single quotes
Double quotes, can also contain single quotes ''
Triple
double quotes
can be defined on multiple lines
```

Strings also have a set of predefined methods:

```
print("hello".upper()) #make everything uppercase
print("HELLO".lower()) #all lowercase
print("hello how are you".capitalize()) #first letter uppercase
print("hello how are you".split()) #splits a string and returns a list
print("hello, how are you".split(','))# splits when it finds the comma
print("-".join(["one","two","three"])) #builds a string by concatenating the elements
#of the list and separating them by the delimiter
```

```
HELLO
hello
Hello how are you
['hello', 'how', 'are', 'you']
['hello', ' how are you']
one-two-three
```

Strings can be indexed similarly to arrays to obtain substrings:

```
s = "Hello World"
print(s[:4]) #first 4 characters
print(s[4:]) #from the fourth character to the end
print(s[4:7]) #from the fourth to the sixth character
print(s[::-1]) #string reversal
```

```
Hell
o World
o W
dlroW olleH
```

The `split` method in particular can be used for tokenization or to easily extract substrings. For example, let's say we want to extract the number `2017` from the string `A-2017-B2`:

```
print("A-2017-B2".split('-')[1])
```

```
2017
```

The `==` operator checks that two strings are equal:

```
print("hello"=="hello")
print("hello"=="hello2")
```

```
True
False
```

The other operators reflect the lexicographical order between strings:

```
print("abc"<"def")
print("Abc">"def")
```

```
True
False
```

Question 4

Which code allows manipulating the string `azyp-kk9-382` to obtain the string `Kk9` ?

String Formatting

We can build formatted strings following a syntax similar to printf:

```
# To build the formatted string, I follow the string with the "%" symbol and then
        insert
# a tuple containing the arguments
s1 = "This %s is formatted. I can insert numbers, for example %0.2f" %
        ("string",3.00002)
print(s1)
```

```
This string is formatted. I can insert numbers, for example 3.00
```

An alternative and more recent way to format strings is to use the “format” method:

```
s2 = "This {} is formatted. I can insert numbers, for example {}"\
    .format("string",3.000002)# the "\" character allows to break the line
print(s2)
```

```
This string is formatted. I can insert numbers, for example 3.000002
```

It is possible to specify the type of each argument using the colon:

```
print("This {:s} is formatted. I can insert numbers, for example {:0.2f}"\
      .format("string",3.00002)) #Positional parameters, without specifying indices
```

```
This string is formatted. I can insert numbers, for example 3.00
```

It’s also possible to assign names to arguments so you can call them out of order:

```
print("This {str:s} is formatted. I can insert numbers, for example {num:0.2f}"\
    .format(num=3.00002, str="string"))
```

```
This string is formatted. I can insert numbers, for example 3.00
```

Question 5

Given the variables:

```
python a = "hello" b = "world" c = 2.0
```

Use string formatting to print the string `hello 2 times world`.

Dictionaries

Dictionaries are similar to lists, but they are indexed by "hashable" objects, for example strings:

```
d = {"val1":1, "val2":2}
print(d)

print(d["val1"])
```

```
{'val1': 1, 'val2': 2}
1
```

It is possible to obtain the list of keys and values as follows:

```
print(d.keys()) #keys and values are in random order
print(d.values())
```

```
dict_keys(['val1', 'val2'])
dict_values([1, 2])
```

It is possible to index dictionaries with tuples (which are "hashable")

```
d = {(2,3):5, (4,6):11}

print(d[(2,3)])
```

```
5
```

Dictionaries can also be extended dynamically:

```
d = dict() #empty dictionary
d["key"]="value"
print(d)
```

```
{'key': 'value'}
```

We can check that an element is among the keys of a dictionary as follows:

```
d = {1:'hello', '5': 5, 8: -1}
print(5 in d)
print('5' in d)
```

```
False
True
```

It is possible to check that an element is among the values of a dictionary as follows:

```
print(-1 in d.values())
```

```
True
```

Set

Sets are data structures that can contain only one instance of a given element:

```
s = {1,2,3,3}
print(s) # only one "3" can be contained
```

```
{1, 2, 3}
```

We can add an element to a set using the “add” method:

```
print(s)
s.add(5)
print(s)
s.add(1) # has no effect. 1 is already present
print(s)
```

```
{1, 2, 3}
{1, 2, 3, 5}
{1, 2, 3, 5}
```

Also in this case, we can check whether an element belongs to a set using the keyword `in` :

```
s={1,5,-1}
print(-1 in s)
print(8 in s)
```

```
True
False
```

It is also possible to create sets from lists:

```
set([1,3,3,2,5,1])
```

```
{1, 2, 3, 5}
```

If/elif/else constructs

Conditional constructs work similarly to C-based languages. Unlike those languages, however, Python replaces parentheses with mandatory indentation. The following C++ code:

```
int var1 = 5;
int var2 = 10;
if(var1<var2) {
    int var3 = var1+var2;
    cout << "Hello World "<<var3;
}
cout << "End";
```

is instead written as follows:

```
var1 = 5
var2 = 10
if var1<var2:
    var3 = var1+var2
    print("Hello World",var3)
print("End")
```

```
Hello World 15
End
```

In practice:

- the condition to be checked is not enclosed in parentheses;
- the colon indicates the beginning of the if body;
- indentation determines what belongs to the if body and what does not belong to the if body.

Since indentation has syntactic value, it becomes mandatory. Furthermore, it is not possible to indent parts of code where it is not significant. For example, the following code returns an error:

```
print("Hello")
    print("World") #indentation error
```

The indentation rules just seen also apply to loops and other constructs where it is necessary to delimit blocks of code. The if construct also allows specifying an `else` branch and an `elif` branch for cascaded controls. Let's look at some examples:

```
true_condition = True
false_condition = False

if true_condition: #the colon ":" is mandatory
    word="cool!"
    print(word) #indentation is mandatory

if false_condition:
    print("not cool :(")

if not false_condition: #we negate the condition with "not"
    word="cool"
    print(word,"again :)")

if false_condition:
    word="this"
    print(word+" is "+"false")
else: #colon + indentation
    print("true")

if false_condition:
    print("false")
elif 5>4: #implements an "else if"
    print("5>4")
else:
    print("4<5??")
```

```
cool!
cool again :)
true
5>4
```

It is also possible to check if a value belongs to a list. This is very useful for checking if a parameter is allowed.

```
allowed_parameters = ["single", "double", 5, -2]

parameter = "double"

if parameter in allowed_parameters:
    print(parameter,"is ok")
else:
    print(parameter,"not in",allowed_parameters)

x=8
if x in allowed_parameters:
    print(x,"is not ok")
else:
    print(x,"not in",allowed_parameters)
```

```
double is ok
8 not in ['single', 'double', 5, -2]
```

An “inline” variant of the if construct can be used for conditional assignments:

```
var1 = 5
var2 = 3
m = var1 if var1>var2 else var2 # Calculate the maximum
print(m)
```

```
5
```

In Python there is no switch construct; to implement it, a series of cascaded elif statements is used:

```
s = "hello"
if s=="help":
    print("help")
elif s=="world":
    print("hello",s)
elif s=="hello":
    print(s,"world")
else:
    print("Default")
```

```
hello world
```

Question 6

Consider the following code:

```
x=2
if x>0:
    y=12
    x=x+y
    print y
else:
        z=28
        h=12
        print z+h
```

Highlight any syntactic errors. Rewrite the code in C++ or Java and compare the two versions of the code.

While and For Loops

While loops are defined as follows:

```
i=0
while i<5: # Colon ":"
    print(i) # Indentation
    i+=1
```

```
0
1
2
3
4
```

The syntax of for loops is a bit different from the standard C syntax. For loops in Python are more like **foreach** and require an “iterable” (for example, a list or a tuple) to be executed:

```
l=[1,7,2,5]
for v in l:
    print(v)
```

```
1
7
2
5
```

To write something equivalent to the following C code:

```
for (int i=0; i<5; i++) {...}
```

we can use the **range** function which generates sequential numbers on the fly:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Range does not directly generate a list, but a “range-type” “generator”, i.e., an object capable of generating numbers. If we want to convert it into a list, we must do so explicitly:

```
print(range(5)) #range type object, generates numbers from 0 to 5 (exclusive)
print(list(range(5))) #range just generates consecutive numbers
#converting range to a list, we can verify which numbers are generated
```

```
range(0, 5)
[0, 1, 2, 3, 4]
```

If we wanted to simultaneously iterate through indices and values of an array, we could write:

```
array=[1,7,2,4,5]
for i in range(len(array)):
    print(i,"→",array[i])
```

```
0 → 1
1 → 7
2 → 2
3 → 4
4 → 5
```

In Python, however, it is possible to use the `enumerate` function to obtain the same result in a more compact way:

```
for index,value in enumerate(array): #get both index and value
    print(index,"→",value)
```

```
0 → 1
1 → 7
2 → 2
3 → 4
4 → 5
```

Now let's assume we want to simultaneously iterate through all the i-th elements of multiple lists. For example:

```
a1=[1,6,2,5]
a2=[1,8,2,7]
a3=[9,2,5,2]

for i in range(len(a1)):
    print(a1[i],a2[i],a3[i])
```

```
1 1 9
6 8 2
2 2 5
5 7 2
```

A very useful function when working with loops is `zip`, which allows to group the corresponding elements from different lists:

```
l1 = [1,6,5,2]
l2 = [3,8,9,2]
zipped=list(zip(l1,l2))
print(zipped)
```

```
[(1, 3), (6, 8), (5, 9), (2, 2)]
```

In practice, `zip` groups the i-th elements of the lists into tuples. The i-th tuple of `zipped` contains the i-th elements of the two lists. By combining `zip` with a for loop, we can obtain the following result:

```
for v1,v2,v3 in zip(a1,a2,a3):
    print(v1,v2,v3)
```

```
1 1 9
6 8 2
2 2 5
5 7 2
```

which is equivalent to the code seen previously. It is also possible to combine zip and enumerate as follows:

```
for i,(v1,v2,v3) in enumerate(zip(a1,a2,a3)):
    print(i,"→",v1,v2,v3)
```

```
0 → 1 1 9
1 → 6 8 2
2 → 2 2 5
3 → 5 7 2
```

Attention, also `zip`, like `range`, produces a generator. Therefore it is necessary to convert it explicitly into a list to print its values:

```
print(zip(l1,l2))
print(list(zip(l1,l2)))
```

```
<zip object at 0x105674cc0>
[(1, 3), (6, 8), (5, 9), (2, 2)]
```

Understanding Lists and Dictionaries

List comprehension is a syntactic tool that allows you to define lists on the fly from other lists, in an iterative manner. For example, it is possible to multiply all elements of a list by a value, as follows:

```
a = list(range(8))

b = [x*3 for x in a] # List "a" is iterated. Variable "x" will contain the values of
        "a" in each iteration.

print(a)
print(b)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 3, 6, 9, 12, 15, 18, 21]
```

It's also possible to include only some elements selectively:

```
print([x for x in a if x%2==0]) # Includes only the even numbers
```

```
[0, 2, 4, 6]
```

```
a = [1,3,8,2,9]
b = [4,9,2,1,4]

c = [x+y for x,y in zip(a,b)]

print(a)
print(b)
print(c) # its elements are the sums of the elements from a and b
```

```
[1, 3, 8, 2, 9]
[4, 9, 2, 1, 4]
[5, 12, 10, 3, 13]
```

The mechanisms of understanding can also be used in the case of dictionaries:

```
a = ["one","two","three","four","five","six","seven"]
b = range(1,8)
d = {i:s for i,s in zip(a,b)}
print(d)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'seven': 7}
```

Question 7

Can all operations that can be performed using list and dictionary comprehensions be performed using a for loop? What are the main advantages of these techniques compared to using for loops?

Definition of Functions

Given what has already been said about indentation, the definition of a function is very natural:

```
def fun(x, y):
    return x**y

print(fun(3,2)) # the default value "2" is used
```

```
9
```

Furthermore, similarly to what happens with the C language, it is possible to define default values for parameters:

```
def fun(x, y=2):
    return x**y

print(fun(3)) #the default value "2" is used
```

```
9
```

The parameters of a function can be specified in a different order than the one in which they were defined by calling them by name:

```
print(fun(y=3,x=2))
```

```
8
```

It is possible to define a function that returns more than one element using tuples:

```
def soMuchFun(x,y):
    return x**y, y**x

print(soMuchFun(2,3))

a,b=soMuchFun(2,3) # I can "unpack" the returned tuple
print(a,b)
```

```
(8, 9)
8 9
```

It is also possible to define anonymous functions as follows:

```
myfun = lambda x: x**2 #one input and one output
print(myfun(2))

myfun1 = lambda x,y: x+y #two inputs and one output
print(myfun1(2,3))

myfun2 = lambda x,y: (x**2,y**2) #two inputs and two outputs
print(myfun2(2,3))
```

```
4
5
(4, 9)
```

Question 8

What is the advantage of defining a function using **lambda**? What are its limitations?

Map and Filter

The functions **map** and **filter** allow operations to be performed on the elements of a list. In particular, **map** applies a function to all elements of a list:

```
def pow2(x):
    return(x**2)

l1 = list(range(6))
l2 = list(map(pow2,l1)) # Applies pow2 to all elements of the list
print(l1)
print(l2)
```

```
[0, 1, 2, 3, 4, 5]
[0, 1, 4, 9, 16, 25]
```

When using **map** and **filter**, anonymous functions become particularly useful, allowing us to write more compactly. For example, we can rewrite what was seen above as follows:

```
l2 = list(map(lambda x: x**2, l1))
print(l2)
```

```
[0, 1, 4, 9, 16, 25]
```

Filter allows selecting a subset of the elements of a list based on a condition. The condition is specified by passing a function that takes the list element as input and returns a boolean:

```
print(list(filter(lambda x: x%2==0,l1))) # Filter only even numbers
```

```
[0, 2, 4]
```

Object-Oriented Programming - Definition of Classes

Object-oriented programming in Python is intuitive. The main differences compared to the most common object-oriented programming languages are the following:

- visibility modifiers do not exist (**private**, **protected**, and **public**). By convention, all private symbols are preceded by "__";
- every method is a function with a default argument (**self**) that represents the object's state;
- the constructor is called "__init__".

```
class Class(object): # inherit from the standard "object" class
    def __init__(self, x): # constructor
        self.x=x # insert the value x into the object's state

    def print_value(self):
        print(self.x)

    def power(self,y=2):
        self.x =  self.x**y

c = Class(3)
c.print_value()
c.power(3)
c.print_value()
c.power()
c.print_value()
```

```
3
27
729
```

To extend a class, one does as follows:

```
class Class2(Class): # derive the class "Class" and inherit methods and properties
    def __init__(self,x):
        super(Class2, self).__init__(x) # call the constructor of the parent class
    def prt(self): # redefine the prt method
        print("yeah",self.x)

c2 = Class2(2)
c2.power()
c2.prt()
```

```
yeah 4
```

Duck Typing

To identify data types, Python follows the principle of duck typing. According to this principle, types are defined using the *duck test*:

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

This means that types are defined in relation to the operations that can be performed on them. Let's look at an example (taken from Wikipedia).

```
class Sparrow(object):
    def fly(self):
        print("Sparrow flying")

class Airplane(object):
    def fly(self):
        print("Airplane flying")

class Whale(object):
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

sparrow = Sparrow()
airplane = Airplane()
whale = Whale()

try:
    lift_off(sparrow)
    lift_off(airplane)
    lift_off(whale) #Error, the "type" of this object does not allow executing the
        "fly" method.
                    #According to the duck test, this type is incompatible
except AttributeError as e:
    print("Error:",e)
```

```
Sparrow flying
Airplane flying
Error: 'Whale' object has no attribute 'fly'
```

Exceptions

Similar to many modern languages, Python supports the use of exceptions. It is possible to catch an exception with the `try - except` construct:

```
try:
    5/0
except:
    print("Houston, we have a problem!")
```

```
Houston, we have a problem!
```

In Python, exceptions are typed. We can decide which types of exceptions to catch as follows:

```
try:
    5/0
except ZeroDivisionError:
    print("Houston, we have a problem!")
```

```
Houston, we have a problem!
```

We can access the triggered exception (e.g., to get more information) as follows:

```
try:
    5/0
except ZeroDivisionError as e:
    print("Houston, we have a problem!")
    print(e)
```

```
Houston, we have a problem!
division by zero
```

It is possible to raise an exception using `raise`:

```
def div(x,y):
    if y==0:
        raise ZeroDivisionError() #raises an exception
    else:
        return x/y
div(5,2)
div(5,0)
```

We can define new exceptions by extending the `Exception` class:

```
class MyException(Exception):
    def __init__(self, message):
        self.message = message

raise MyException("Exception!")
```

A fast and convenient way to raise an exception (an AssertionError specifically) when something goes wrong, is to use `assert`, which takes a boolean as input and, optionally, an error message. The boolean should be set to `False` if something went wrong. Let's see an example:

```
def div(x,y):
    assert y!=0, "Cannot divide by zero!"
    return x/y

div(5,2)
div(5,0)
```

Definition of Modules

When building complex programs, it can be useful to group function and class definitions into modules. The simplest way to define a module in Python is to place the definitions inside a dedicated file `modulo.py`. The definitions can then be imported using the `from modulo import funzione` syntax, provided that the file calling the functions and the one defining the module are in the same folder. Let's see an example:

```
#file modulo.py
def mysum(a,b):
    return a+b

def myprod(a,b):
    return a*b
```

```
#file main.py (same folder as modulo.py)
from modulo import mysum, myprod
print(mysum(2,3)) #5
print(myprod(2,3)) #6
```

Further information on more advanced uses of modules and packages can be found here: https://docs.python.org/3/tutorial/modules.html.

```
import numpy as np #The "as" notation allows us to reference the numpy namespace
        simply with np in the future
```

```
l = [[1,2,3],[4,5,2],[1,8,3]] #a list containing three lists
print("List of lists:",l) #it is displayed as defined
a = np.array(l) #Build a numpy array from the list of lists
print("Numpy array:\n",a) #each inner list is identified as a row of a two-
        dimensional matrix
print("Numpy array from tuple:\n",np.array(((1,2,3),(4,5,6)))) #Can also create numpy
        arrays from tuples
```

```
List of lists: [[1, 2, 3], [4, 5, 2], [1, 8, 3]]
Numpy array:
 [[1 2 3]
 [4 5 2]
 [1 8 3]]
Numpy array from tuple:
 [[1 2 3]
 [4 5 6]]
```

Every numpy array has a *shape* property that allows us to determine the number of dimensions of the structure:

```
print(a.shape) # This is a 3 x 3 matrix
```

```
(3, 3)
```

Let's look at a few more examples

```
array = np.array([1,2,3,4])
matrix = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
tensor = np.array([[[1,2,3,4],['a','b','c','d']],[[5,4,2,3],['a','b','c','d']],
        [[7,5,3,2],['a','b','c','d']],[[0,2,3,1],['a','b','c','d']]])
print('Array:',array, array.shape) #one-dimensional array, it will have only one
        dimension
print('Matrix:\n',matrix, matrix.shape)
print('matrix:\n',tensor, tensor.shape) #tensor, it will have two dimensions
```

```
Array: [1 2 3 4] (4,)
Matrix:
 [[1 2 3 4]
 [5 4 2 3]
 [7 5 3 2]
 [0 2 3 1]] (4, 4)
matrix:
 [['1' '2' '3' '4']
  ['a' 'b' 'c' 'd']]

 [['5' '4' '2' '3']
  ['a' 'b' 'c' 'd']]

 [['7' '5' '3' '2']
  ['a' 'b' 'c' 'd']]

 [['0' '2' '3' '1']
  ['a' 'b' 'c' 'd']]] (4, 2, 4)
```

Let's see some operations between NumPy arrays:

```
a1 = np.array([1,2,3,4])
a2 = np.array([4,3,8,1])
print("Sum:",a1+a2) #somma tra vettori
print("Elementwise multiplication:",a1*a2) #moltiplicazione tra elementi
        corrispondenti
print("Power of two:",a1**2) #quadrato degli elementi
print("Elementwise power:",a1**a2) #elevamento a potenza elemento per elemento
print("Scalar product:",a1.dot(a2)) #prodotto scalare tra vettori
print("Minimum:",a1.min()) #minimo dell'array
print("Maximum:",a1.max()) #massimo dell'array
print("Sum:",a2.sum()) #somma di tutti i valori dell'array
print("Product:",a2.prod()) #prodotto di tutti i valori dell'array
print("Mean:",a1.mean()) #media di tutti i valori dell'array
```

```
Sum: [ 5  5 11  5]
Elementwise multiplication: [ 4  6 24  4]
Power of two: [ 1  4  9 16]
Elementwise power: [   1    8 6561    4]
Scalar product: 38
Minimum: 1
Maximum: 4
Sum: 16
Product: 96
Mean: 2.5
```

Operations on matrices:

```
m1 = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
m2 = np.array([[8,2,1,4],[0,4,6,1],[4,4,2,0],[0,1,8,6]])

print("Sum:",m1+m2) # Matrix sum
print("Elementwise product:\n",m1*m2) # Element-wise product
print("Power of two:\n",m1**2) # Square of the elements
print("Elementwise power:\n",m1**m2) # Element-wise power
print("Matrix multiplication:\n",m1.dot(m2)) # Matrix multiplication
print("Minimum:",m1.min()) # Minimum
print("Maximum:",m1.max()) # Maximum
print("Minimum along columns:",m1.min(0)) # Minimum along columns
print("Minimum along rows:",m1.min(1)) # Minimum along rows
print("Sum:",m1.sum()) # Sum of values
print("Mean:",m1.mean()) # Mean value
print("Diagonal:",m1.diagonal()) # Main diagonal of the matrix
print("Transposed:\n",m1.T) # Transposed matrix
```

```
Sum: [[ 9  4  4  8]
 [ 5  8  8  4]
 [11  9  5  2]
 [ 0  3 11  7]]
Elementwise product:
 [[ 8  4  3 16]
 [ 0 16 12  3]
 [28 20  6  0]
 [ 0  2 24  6]]
Power of two:
 [[ 1  4  9 16]
 [25 16  4  9]
 [49 25  9  4]
 [ 0  4  9  1]]
Elementwise power:
 [[   1    4    3  256]
 [   1  256   64    3]
 [2401  625    9    1]
 [   1    2 6561    1]]
Matrix multiplication:
 [[20 26 51 30]
 [48 37 57 42]
 [68 48 59 45]
 [12 21 26  8]]
Minimum: 0
Maximum: 7
Minimum along columns: [0 2 2 1]
Minimum along rows: [1 2 2 0]
Sum: 47
Mean: 2.9375
Diagonal: [1 4 3 1]
Transposed:
 [[1 5 7 0]
 [2 4 5 2]
 [3 2 3 3]
 [4 3 2 1]]
```

Exercises

Exercise 1

Define the list `[1,8,2,6,15,21,76,22,0,111,23,12,24]`, then:

- Print the first number of the list;
- Print the last number of the list;
- Print the sum of numbers with odd indices (e.g., 1,3,5,...) in the list;

- Print the list sorted in reverse order;
- Print the average of the numbers contained in the list.

Exercise 2

Define a dictionary `mesi` that maps the names of the months to their numerical equivalents. For example, the result of:

```
print mesi['Gennaio']
```

should be

```
1
```

Exercise 3

Consider the following lists:

```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = [5,2,6]
```

Combine a for loop, **zip** and **enumerate** to obtain the following output:

```
0 → 10
1 → 9
2 → 15
```

Exercise 4

Repeat exercise **2** using dictionary comprehension. To do this, first define the list `['January','February','March','April','May','June', 'July','August','September','October','November','December']`.

Then build the desired dictionary using dictionary comprehension and the `enum` function

Exercise 5

Given the variables:

```
obj='triangle'
area=21.167822
```

print the strings:

- The area of the triangle is 21.16
- 21.1678 is the area of the triangle

Use string formatting to obtain the result.

Exercise 6

Write a function that extracts the domain from an email address. For example, if the address is “furnari@dmi.unict.it”, the function must extract “dmi.unict.it”.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Python for data science crash course

Chapter resources

Open in Colab | Practice quiz

Numpy

Numpy is the reference library of **scipy** for scientific computing. The core of the library consists of **numpy arrays** which allow for easy handling of operations between vectors and matrices. Python arrays are generally **tensors**, which are numerical structures with a variable number of dimensions, and can therefore be one-dimensional arrays, two-dimensional matrices, or multi-dimensional structures (e.g., 10 x 10 x 10 cuboids). To use numpy arrays, we must first import the **numpy** package:

```
import numpy as np #the "as" notation allows us to reference the numpy namespace
        simply with np in the future
```

Numpy Arrays

A multi-dimensional numpy array can be defined from a list of lists, as follows:

```
l = [[1,2,3],[4,5,2],[1,8,3]] #a list containing three lists
print("List of lists:",l) #it is displayed as we defined it
a = np.array(l) #I build a numpy array from the list of lists
print("Numpy array:\n",a) #each inner list is identified as a row of a two-
        dimensional matrix
print("Numpy array from tuple:\n",np.array(((1,2,3),(4,5,6)))) #I can also create
        numpy arrays from tuples
```

```
List of lists: [[1, 2, 3], [4, 5, 2], [1, 8, 3]]
Numpy array:
 [[1 2 3]
 [4 5 2]
 [1 8 3]]
Numpy array from tuple:
 [[1 2 3]
 [4 5 6]]
```

Question 1

What is an obvious advantage of NumPy arrays compared to lists, given the examples shown above?

Every numpy array has a *shape* property that allows us to determine the number of dimensions of the structure:

```
print(a.shape) #it is a 3 x 3 matrix
```

```
(3, 3)
```

Let's look at some other examples:

```
array = np.array([1,2,3,4])
matrix = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
tensor = np.array([[[1,2,3,4],['a','b','c','d']],[[5,4,2,3],['a','b','c','d']],
        [[7,5,3,2],['a','b','c','d']],[[0,2,3,1],['a','b','c','d']]])
print('Array:',array, array.shape) # one-dimensional array, will have only one
        dimension
print('Matrix:\n',matrix, matrix.shape)
print('matrix:\n',tensor, tensor.shape) # tensor, will have two dimensions
```

```
Array: [1 2 3 4] (4,)
Matrix:
 [[1 2 3 4]
 [5 4 2 3]
 [7 5 3 2]
 [0 2 3 1]] (4, 4)
matrix:
 [[['1' '2' '3' '4']
  ['a' 'b' 'c' 'd']]

 [['5' '4' '2' '3']
  ['a' 'b' 'c' 'd']]

 [['7' '5' '3' '2']
  ['a' 'b' 'c' 'd']]

 [['0' '2' '3' '1']
  ['a' 'b' 'c' 'd']]] (4, 2, 4)
```

Question 2

In Numpy, is there a formal difference between a one-dimensional array and a matrix containing a single row (or a single column)?

Numpy arrays support element-by-element operations. Let's look at the main operations of this type:

```
a1 = np.array([1,2,3,4])
a2 = np.array([4,3,8,1])
print("Sum:",a1+a2) #Vector sum
print("Elementwise multiplication:",a1*a2) #Element-wise multiplication
print("Power of two:",a1**2) #Square of elements
print("Elementwise power:",a1**a2) #Element-wise power
print("Scalar product:",a1.dot(a2)) #Vector scalar product
print("Minimum:",a1.min()) #Array minimum
print("Maximum:",a1.max()) #Array maximum
print("Sum:",a2.sum()) #Sum of all array values
print("Product:",a2.prod()) #Product of all array values
print("Mean:",a1.mean()) #Mean of all array values
```

```
Sum: [ 5  5 11  5]
Elementwise multiplication: [ 4  6 24  4]
Power of two: [ 1  4  9 16]
Elementwise power: [   1    8 6561    4]
Scalar product: 38
Minimum: 1
Maximum: 4
Sum: 16
Product: 96
Mean: 2.5
```

Operations on matrices:

```
m1 = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
m2 = np.array([[8,2,1,4],[0,4,6,1],[4,4,2,0],[0,1,8,6]])

print("Sum:",m1+m2) #matrix sum
print("Elementwise product:\n",m1*m2) #element-wise product
print("Power of two:\n",m1**2) #square of elements
print("Elementwise power:\n",m1**m2) #element-wise power
print("Matrix multiplication:\n",m1.dot(m2)) #matrix multiplication
print("Minimum:",m1.min()) #minimum
print("Maximum:",m1.max()) #maximum
print("Minimum along columns:",m1.min(0)) #minimum along columns
print("Minimum along rows:",m1.min(1)) #minimum along rows
print("Sum:",m1.sum()) #sum of values
print("Mean:",m1.mean()) #mean value
print("Diagonal:",m1.diagonal()) #main diagonal of the matrix
print("Transposed:\n",m1.T) #transposed matrix
```

```
Sum: [[ 9  4  4  8]
 [ 5  8  8  4]
 [11  9  5  2]
 [ 0  3 11  7]]
Elementwise product:
 [[ 8  4  3 16]
 [ 0 16 12  3]
 [28 20  6  0]
 [ 0  2 24  6]]
Power of two:
 [[ 1  4  9 16]
 [25 16  4  9]
 [49 25  9  4]
 [ 0  4  9  1]]
Elementwise power:
 [[   1    4    3  256]
 [   1  256   64    3]
 [2401  625    9    1]
 [   1    2 6561    1]]
Matrix multiplication:
 [[20 26 51 30]
 [48 37 57 42]
 [68 48 59 45]
 [12 21 26  8]]
Minimum: 0
Maximum: 7
Minimum along columns: [0 2 2 1]
Minimum along rows: [1 2 2 0]
Sum: 47
Mean: 2.9375
Diagonal: [1 4 3 1]
Transposed:
 [[1 5 7 0]
 [2 4 5 2]
 [3 2 3 3]
 [4 3 2 1]]
```

Question 3

What is the advantage of using the operations illustrated above? What code would be needed to perform the operation `a1**a2` if `a1` and `a2` were Python lists rather than numpy arrays?

Linspace, Arange, Zeros, Ones, Eye and Random

The functions **linspace**, **arange**, **zeros**, **ones**, **eye** and **random** of **numpy** are useful for generating numeric arrays on the fly. In particular, the function **linspace** allows generating a sequence of **n** equally spaced numbers ranging from a minimum value to a maximum value:

```
a=np.linspace(10,20,5) # generates 5 equally spaced values ranging from 10 to 20
print(a)
```

```
[10.  12.5 15.  17.5 20. ]
```

arange is very similar to **range**, but directly returns a **numpy** array:

```
print(np.arange(10)) #numbers from 0 to 9
print(np.arange(1,6)) #numbers from 1 to 5
print(np.arange(0,7,2)) #even numbers from 0 to 6
```

```
[0 1 2 3 4 5 6 7 8 9]
[1 2 3 4 5]
[0 2 4 6]
```

Question 4

Is it possible to obtain the same result as `arange` using `range` ? How?

We can create arrays containing zero or one of arbitrary shapes using **zeros** and **ones**:

```
print(np.zeros((3,4)))#zeros and ones take as a parameter a tuple containing the
        desired dimensions
print(np.ones((2,1)))
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1.]
 [1.]]
```

The function **eye** allows you to create an identity square matrix:

```
print(np.eye(3))
print(np.eye(5))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

To build an array of random values (uniform distribution) between 0 and 1 (zero included, one excluded), just write:

```
print(np.random.rand(5)) #an array with 5 random values between 0 and 1
print(np.random.rand(3,2)) #a 3x2 matrix of random values between 0 and 1
```

```
[0.4980907  0.4037207  0.83812821 0.91500638 0.35255439]
[[0.58713071 0.10768147]
 [0.76698998 0.5997833 ]
 [0.69390245 0.62038453]]
```

We can generate an array of random values normally (Gaussian) distributed with **randn**:

```
print(np.random.randn(5,2))
```

```
[[ 0.15396072  1.20625632]
 [-0.13404619 -1.03696306]
 [ 1.71033777  0.21354998]
 [ 0.60294632 -1.40177865]
 [ 1.02949267  0.01646333]]
```

We will talk in more detail about the Gaussian distribution later.

We can generate integers between a minimum (inclusive) and a maximum (exclusive) using **randint**:

```
print(np.random.randint(0,50,3))#three values between 0 and 50 (exclusive)
print(np.random.randint(0,50,(2,3)))#2x3 matrix of random integer values between 0
        and 50 (exclusive)
```

```
[45 44 21]
[[41 34 38]
 [17 13 30]]
```

To generate random values reproducibly, you can specify a seed:

```
np.random.seed(123)
print(np.random.rand(5))
```

```
[0.69646919 0.28613933 0.22685145 0.55131477 0.71946897]
```

The code above (including the seed definition) returns the same results if rerun:

```
np.random.seed(123)
print(np.random.rand(5))
```

```
[0.69646919 0.28613933 0.22685145 0.55131477 0.71946897]
```

Question 5

What is the effect of re-running the previous cell changing the seed?

max, min, sum, argmax, argmin

It is possible to calculate maxima and minima and sums of a matrix by rows and columns as follows:

```
mat = np.array([[1,-5,0],[4,3,6],[5,8,-7],[10,6,-12]])
print(mat)
print()
print( mat.max(0))# Column maximums
print()
print(mat.max(1))# Row maximums
print()
print (mat.min(0))# Column minimums
print()
print(mat.min(1))# Row minimums
print()
print(mat.sum(0))# Column sum
print()
print(mat.sum(1))# Row sum
print()
print(mat.max())# Global maximum
print(mat.min())# Global minimum
print(mat.sum())# Global sum
```

```
[[  1  -5   0]
 [  4   3   6]
 [  5   8  -7]
 [ 10   6 -12]]

[10  8  6]

[ 1  6  8 10]

[  1  -5 -12]

[ -5   3  -7 -12]

[ 20  12 -13]

[-4 13  6  4]

10
-12
19
```

It is also possible to obtain the indices at which there are maxima and minima using the argmax function:

```
print(mat.argmax(0))
```

```
[3 2 1]
```

We have a maximum in the fourth row in the first column, one in the third row in the second column, and one in the second row in the third column. Let's verify it:

```
print(mat[3,0])
print(mat[2,1])
print(mat[1,2])
print(mat.max(0))
```

```
10
8
6
[10  8  6]
```

Similarly:

```
print(mat.argmax(1))
print(mat.argmin(0))
print(mat.argmin(1))
```

```
[0 2 1 0]
[0 0 3]
[1 1 2 2]
```

Indexing and Slicing

Numpy arrays can be indexed in a similar way to Python lists:

```
arr = np.array([1,2,3,4,5])
print("arr[0]       →",arr[0]) #first element of the array
print("arr[:3]      →",arr[:3]) #first three elements
print("arr[1:5:2]   →",arr[1:4:2]) #from the second to the fourth (inclusive) with a
        step of 2
```

```
arr[0]       → 1
arr[:3]      → [1 2 3]
arr[1:5:2]   → [2 4]
```

When indexing an array with more than one dimension using a single index, the first dimension is automatically indexed. Let's look at some examples with two-dimensional matrices:

```
mat = np.array(([1,5,2,7],[2,7,3,2],[1,5,2,1]))
print("Matrix:\n",mat,mat.shape) #matrix 3 x 4
print("mat[0]     →",mat[0]) #a matrix is a collection of rows, so mat[0] returns
        the first row
print("mat[-1]     →",mat[-1]) #last row
print("mat[::2]     →",mat[::2]) #odd rows
```

```
Matrix:
 [[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]] (3, 4)
mat[0]     → [1 5 2 7]
mat[-1]     → [1 5 2 1]
mat[::2]     → [[1 5 2 7]
 [1 5 2 1]]
```

Let's see some examples with multi-dimensional tensors:

```
tens = np.array(([[1,5,2,7],
                  [2,7,3,2],
                  [1,5,2,1]],
                  [[1,5,2,7],
                  [2,7,3,2],
                  [1,5,2,1]]))
print("Matrix:\n",tens,tens.shape) #tensor 2x3x4
print("tens[0]    →",tens[0])#this is the first 3x4 matrix
print("tens[-1]    →",tens[-1])#last 3x4 matrix
```

```
Matrix:
 [[[1 5 2 7]
  [2 7 3 2]
  [1 5 2 1]]

 [[1 5 2 7]
  [2 7 3 2]
  [1 5 2 1]]] (2, 3, 4)
tens[0]    → [[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]]
tens[-1]    → [[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]]
```

Indexing can continue through the other dimensions by specifying an additional index in square brackets or by separating the various indices with a comma:

```
mat = np.array(([1,5,2,7],[2,7,3,2],[1,5,2,1]))
print("Matrix:\n",mat,mat.shape) #matrix 3 x 4
print("mat[2][1]  →",mat[2][1]) #third row, second column
print("mat[0,0]   →",mat[0,0]) #first row, first column (more compact notation)

print("mat[0]     → ",mat[0]) #returns the entire first row of the matrix
print("mat[:,0]   → ",mat[:,0]) #returns the first column of the matrix.
                                #The two dots ":" mean "leave everything unchanged
        along this dimension"
print("mat[0,:]   →",mat[0,:]) #alternative notation to get the first row of the
        matrix
print("mat[0:2,:] →\n",mat[0:2,:]) #first two rows
print("mat[:,0:2] →\n",mat[:,0:2]) #first two columns
print("mat[-1]    →",mat[-1]) #last row
```

```
Matrix:
 [[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]] (3, 4)
mat[2][1]  → 5
mat[0,0]   → 1
mat[0]     →  [1 5 2 7]
mat[:,0]   →  [1 2 1]
mat[0,:]   → [1 5 2 7]
mat[0:2,:] →
 [[1 5 2 7]
 [2 7 3 2]]
mat[:,0:2] →
 [[1 5]
 [2 7]
 [1 5]]
mat[-1]    → [1 5 2 1]
```

Case of multi-dimensional tensors:

```
mat=np.array([[[1,2,3,4],['a','b','c','d']],
              [[5,4,2,3],['a','b','c','d']],
              [[7,5,3,2],['a','b','c','d']],
              [[0,2,3,1],['a','b','c','d']]])
print("mat[:,:,0] →", mat[:,:,0]) #matrix contained in the "first channel" of the
        tensor
print("mat[:,:,1] →",mat[:,:,1]) #matrix contained in the "second channel" of the
        tensor
print("mat[...,0] →", mat[...,0]) #matrix contained in the "first channel" of the
        tensor (alternative notation)
print("mat[...,1] →",mat[...,1]) #matrix contained in the "second channel" of the
        tensor (alternative notation)
#the "..." notation means "leave everything unchanged along the omitted dimensions"
```

```
mat[:,:,0] → [['1' 'a']
 ['5' 'a']
 ['7' 'a']
 ['0' 'a']]
mat[:,:,1] → [['2' 'b']
 ['4' 'b']
 ['5' 'b']
 ['2' 'b']]
mat[...,0] → [['1' 'a']
 ['5' 'a']
 ['7' 'a']
 ['0' 'a']]
mat[...,1] → [['2' 'b']
 ['4' 'b']
 ['5' 'b']
 ['2' 'b']]
```

Generally, when a subset of data is extracted from an array, it is referred to as **slicing**.

Question 6

Python lists do not support slicing. How can one implement a statement of the type `a[2:8:2]` using Python lists?

Logical Indexing and Slicing

In numpy it is also possible to index arrays in a "logical" way, meaning by passing an array of boolean values as indices. For example, if we want to select the first and third value of an array, we must pass the array `[True, False, True]` as indices:

```
x = np.array([1,2,3])
print(x[np.array([True,False,True])]) # To select only 1 and 3
print(x[np.array([False,True,False])]) # To select only 2
```

```
[1 3]
[2]
```

Logical indexing is very useful when combined with the ability to build logical arrays "on the fly" by specifying a condition that array elements may or may not satisfy. For example:

```
x = np.arange(10)
print(x)
print(x>2) #generates an array of boolean values
#that will contain True for the values of x
#that satisfy the condition x>2
print(x==3) #True only for the value 3
```

```
[0 1 2 3 4 5 6 7 8 9]
[False False False  True  True  True  True  True  True  True]
[False False False  True False False False False False False]
```

By combining these two principles, it is simple to select only some values from an array, based on a condition:

```
x = np.arange(10)
print(x[x%2==0]) # selects even values
print(x[x%2≠0]) # selects odd values
print(x[x>2]) # selects values greater than 2
```

```
[0 2 4 6 8]
[1 3 5 7 9]
[3 4 5 6 7 8 9]
```

Question 7

Consider the two arrays `a=np.array([1,2,3])` and `b=np.array([5,2,4])`. Use logical indexing to extract the numbers in `a` that are located at positions containing even values in `b`.

Reshape

In some cases, it can be useful to change the "shape" of a matrix. For example, a 3x2 matrix can be modified by rearranging its elements to obtain a 2x3 matrix, a 1x6 matrix, or a 6x1 matrix. This can be done using the "reshape" method:

```
mat = np.array([[1,2],[3,4],[5,6]])
print(mat)
print(mat.reshape(2,3))
print(mat.reshape(1,6))
print(mat.reshape(6,1)) #matrix 6 x 1
print(mat.reshape(6)) #one-dimensional vector
print(mat.ravel())#equivalent to the previous one, but parameterless
```

```
[[1 2]
 [3 4]
 [5 6]]
[[1 2 3]
 [4 5 6]]
[[1 2 3 4 5 6]]
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
[1 2 3 4 5 6]
[1 2 3 4 5 6]
```

We note that, if we read by rows (from left to right, from top to bottom), the order of the elements remains unchanged. We can also let numpy calculate one of the dimensions by replacing it with -1:

```
print(mat.reshape(2,-1))
print(mat.reshape(-1,6))
```

```
[[1 2 3]
 [4 5 6]]
[[1 2 3 4 5 6]]
```

Reshape can take as input the individual dimensions or a tuple containing the shape. In the latter case, it is convenient to perform operations of this kind:

```
mat1 = np.random.rand(3,2)
mat2 = np.random.rand(2,3)
print(mat2.reshape(mat1.shape)) # Give mat2 the same shape as mat1
```

```
[[0.72904971 0.43857224]
 [0.0596779  0.39804426]
 [0.73799541 0.18249173]]
```

Composing Arrays Using `concatenate` and `stack`

Numpy allows combining different arrays using two main functions: `concatenate` and `stack`. The `concatenate` function takes as input a list (or tuple) of arrays and allows concatenating them along a specified existing dimension (`axis`), which by default is zero (row-wise concatenation):

```
a=np.arange(9).reshape(3,3)
print(a,a.shape,"\n")
cat=np.concatenate([a,a])
print(cat,cat.shape,"\n")
cat2=np.concatenate([a,a,a])
print(cat2,cat2.shape)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]] (3, 3)

[[0 1 2]
 [3 4 5]
 [6 7 8]
 [0 1 2]
 [3 4 5]
 [6 7 8]] (6, 3)

[[0 1 2]
 [3 4 5]
 [6 7 8]
 [0 1 2]
 [3 4 5]
 [6 7 8]
 [0 1 2]
 [3 4 5]
 [6 7 8]] (9, 3)
```

It is possible to concatenate arrays along a different dimension by specifying it using the `axis` parameter:

```
a=np.arange(9).reshape(3,3)
print(a,a.shape,"\n")
cat=np.concatenate([a,a], axis=1) #column-wise concatenation
print(cat,cat.shape,"\n")
cat2=np.concatenate([a,a,a], axis=1) #column-wise concatenation
print(cat2,cat2.shape)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]] (3, 3)

[[0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [6 7 8 6 7 8]] (3, 6)

[[0 1 2 0 1 2 0 1 2]
 [3 4 5 3 4 5 3 4 5]
 [6 7 8 6 7 8 6 7 8]] (3, 9)
```

For the concatenation to be compatible, the arrays in the list must have the same dimensions along those that **are not concatenated**:

```
print(cat.shape,a.shape) #concatenation along axis 0, dimensions along other axes
        must be equal
```

```
(3, 6) (3, 3)
```

```
np.concatenate([cat,a], axis=0) #concatenation by columns #error!
```

The `stack` function, unlike `concatenate`, allows concatenating arrays along a new dimension. Compare the outputs of the two functions:

```
cat=np.concatenate([a,a])
print(cat,cat.shape)
stack=np.stack([a,a])
print(stack,stack.shape)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]
 [0 1 2]
 [3 4 5]
 [6 7 8]] (6, 3)
[[[0 1 2]
  [3 4 5]
  [6 7 8]]

 [[0 1 2]
  [3 4 5]
  [6 7 8]]] (2, 3, 3)
```

In the case of `stack`, the arrays were concatenated along a new dimension. It is possible to specify alternative dimensions as in the case of `concatenate`:

```
stack=np.stack([a,a],axis=1)
print(stack,stack.shape)
```

```
[[[0 1 2]
  [0 1 2]]

 [[3 4 5]
  [3 4 5]]

 [[6 7 8]
  [6 7 8]]] (3, 2, 3)
```

In this case, the arrays have been concatenated along the second dimension.

```
stack=np.stack([a,a],axis=2)
print(stack,stack.shape)
```

```
[[[0 0]
  [1 1]
  [2 2]]

 [[3 3]
  [4 4]
  [5 5]]

 [[6 6]
  [7 7]
  [8 8]]] (3, 3, 2)
```

In this case, the arrays were concatenated along the last dimension.

Types

Every numpy array has its type (see https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html for the list of supported types). We can see the type of an array by inspecting the `dtype` property:

```
print(mat1.dtype)
```

```
float64
```

We can specify the type during array construction:

```
mat = np.array([[1,2,3],[4,5,6]],int)
print(mat.dtype)
```

```
int64
```

We can also change the type of an array on the fly using `astype`. This is useful, for example, if we want to perform a non-integer division:

```
print(mat/2)
print(mat.astype(float)/2)
```

```
[[0.5 1.  1.5]
 [2.  2.5 3. ]]
[[0.5 1.  1.5]
 [2.  2.5 3. ]]
```

Memory Management in Numpy

Numpy manages memory dynamically for efficiency reasons. Therefore, an assignment or a slicing operation generally **does not create a new copy of the data**. Consider for example this code:

```
a=np.array([[1,2,3],[4,5,6]])
print(a)
b=a[0,0:2]
print(b)
```

```
[[1 2 3]
 [4 5 6]]
[1 2]
```

The slicing operation `b=a[0,0:2]` only allowed obtaining a new "view" of a part of `a`, but the data was not replicated in memory. Therefore, if we modify an element of b, the modification will actually be applied to `a`:

```
b[0]=-1
print(b)
print(a)
```

```
[-1  2]
[[-1  2  3]
 [ 4  5  6]]
```

To avoid this kind of behavior, it is possible to use the `copy` method which forces numpy to create a new copy of the data:

```
a=np.array([[1,2,3],[4,5,6]])
print(a)
b=a[0,0:2].copy()

print(b)
b[0]=-1
print(b)
print(a)
```

```
[[1 2 3]
 [4 5 6]]
[1 2]
[-1  2]
[[1 2 3]
 [4 5 6]]
```

In this new version of the code, `a` is no longer modified upon modification of `b`.

Broadcasting

Numpy intelligently handles operations between arrays that have different shapes under certain conditions. Let's look at a practical example: suppose we have a 2×3 matrix and a 1×3 array:

```
mat=np.array([[1,2,3],[4,5,6]],dtype=float)
arr=np.array([2,3,8])
print(mat)
print(arr)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
[2 3 8]
```

Now, suppose we want to divide, element by element, all the values of each row of the matrix by the values of the array. We can perform the required operation using a for loop:

```
mat2=mat.copy() # Copy the content of the matrix to not overwrite it
for i in range(mat2.shape[0]):# Indexes the rows
    mat2[i]=mat2[i]/arr
print(mat2)
```

```
[[0.5        0.66666667 0.375     ]
 [2.         1.66666667 0.75      ]]
```

```
arr.shape
```

```
(3,)
```

If we didn't want to use for loops, we could replicate *arr* to obtain a 2×3 matrix and then perform a simple element-wise division:

```
arr2=np.stack([arr,arr])
print(arr2)

print(mat/arr2)
```

```
[[2 3 8]
 [2 3 8]]
[[0.5        0.66666667 0.375     ]
 [2.         1.66666667 0.75      ]]
```

The same result can be obtained simply by asking numpy to divide mat by arr:

```
print(mat/arr)
```

```
[[0.5        0.66666667 0.375     ]
 [2.         1.66666667 0.75      ]]
```

This happens because **numpy** compares the dimensions of the two operands (2×3 and 1×3) and adapts the operand with the smaller shape to the one with the larger shape, by replicating its elements along the unit dimension (the first one). In practice, broadcasting generalizes operations between scalars and vectors/matrices like:

```
print(2*mat)
print(2*arr)
```

```
[[ 2.  4.  6.]
 [ 8. 10. 12.]]
[ 4  6 16]
```

In general, when operations are performed between two arrays, numpy compares the shapes dimension by dimension, from the last to the first. Two dimensions are compatible if:

- They are equal;
- One of them is equal to one.

Furthermore, the two shapes do not necessarily have to have the same number of dimensions.

For example, the following shapes are compatible:

$$2 \times 3 \times 5$$
$$2 \times 3 \times 5$$

$$
\begin{array}{c}
2 \times 3 \times 5 \\
2 \times 1 \times 5
\end{array}
$$

$$
\begin{array}{c}
2 \times 3 \times 5 \\
\quad 3 \times 5
\end{array}
$$

$$
\begin{array}{c}
2 \times 3 \times 5 \\
\quad 3 \times 1
\end{array}
$$

Let's see more examples of broadcasting:

```
mat1=np.array([[[1,3,5],[7,6,2]],[[6,5,2],[8,9,9]]])
mat2=np.array([[2,1,3],[7,6,2]])
print("Mat1 shape",mat1.shape)
print("Mat2 shape",mat2.shape)
print()
print("Mat1\n",mat1)
print()
print("Mat2\n",mat2)
print()
print("Mat1*Mat2\n",mat1*mat2)
```

```
Mat1 shape (2, 2, 3)
Mat2 shape (2, 3)

Mat1
 [[[1 3 5]
  [7 6 2]]

 [[6 5 2]
  [8 9 9]]]

Mat2
 [[2 1 3]
 [7 6 2]]

Mat1*Mat2
 [[[ 2  3 15]
  [49 36  4]]

 [[12  5  6]
  [56 54 18]]]
```

The product between the two tensors was performed by multiplying the two-dimensional matrices `mat1[0,...]` and `mat2[0,...]` by `mat2`. This is equivalent to repeating the elements of `mat2` along the missing dimension and performing a point-by-point product between `mat1` and the adapted version of `mat2`.

```
mat1=np.array([[[1,3,5],[7,6,2]],[[6,5,2],[8,9,9]]])
mat2=np.array([[[1,3,5]],[[6,5,2]]])
print("Mat1 shape",mat1.shape)
print("Mat2 shape",mat2.shape)
print()
print("Mat1\n",mat1)
print()
print("Mat2\n",mat2)
print()
print("Mat1*Mat2\n",mat1*mat2)
```

```
Mat1 shape (2, 2, 3)
Mat2 shape (2, 1, 3)

Mat1
 [[[1 3 5]
  [7 6 2]]

 [[6 5 2]
  [8 9 9]]]

Mat2
 [[[1 3 5]]

 [[6 5 2]]]

Mat1*Mat2
 [[[ 1  9 25]
  [ 7 18 10]]

 [[36 25  4]
  [48 45 18]]]
```

In this case, the product between the two tensors was obtained by multiplying all rows of the two-dimensional matrices `mat1[0,...]` by `mat2[0]` (first row of mat2) and all rows of the two-dimensional matrices `mat1[1,...]` by `mat2[1]` (second row of mat2). This is equivalent to repeating all elements of `mat2` along the second dimension (the one containing 1) and performing an element-wise product between `mat1` and the adapted version of `mat2`.

Introduction to Matplotlib

Matplotlib is the reference library for creating graphs in Python. It is a very powerful and well-documented library (https://matplotlib.org/). Let's look at some classic examples of using the library.

2D plot

Let's see how to plot the function:

$$y = x^2.$$

To plot the function, we will need to provide matplotlib with a series of (x, y) value pairs that satisfy the equation shown above. The simplest way to do this is to:

- Define an arbitrary vector of x values extracted from the function's domain;
- Calculate the respective y points using the analytical form shown above;
- Plot the values using matplotlib.

The two vectors can be defined as follows:

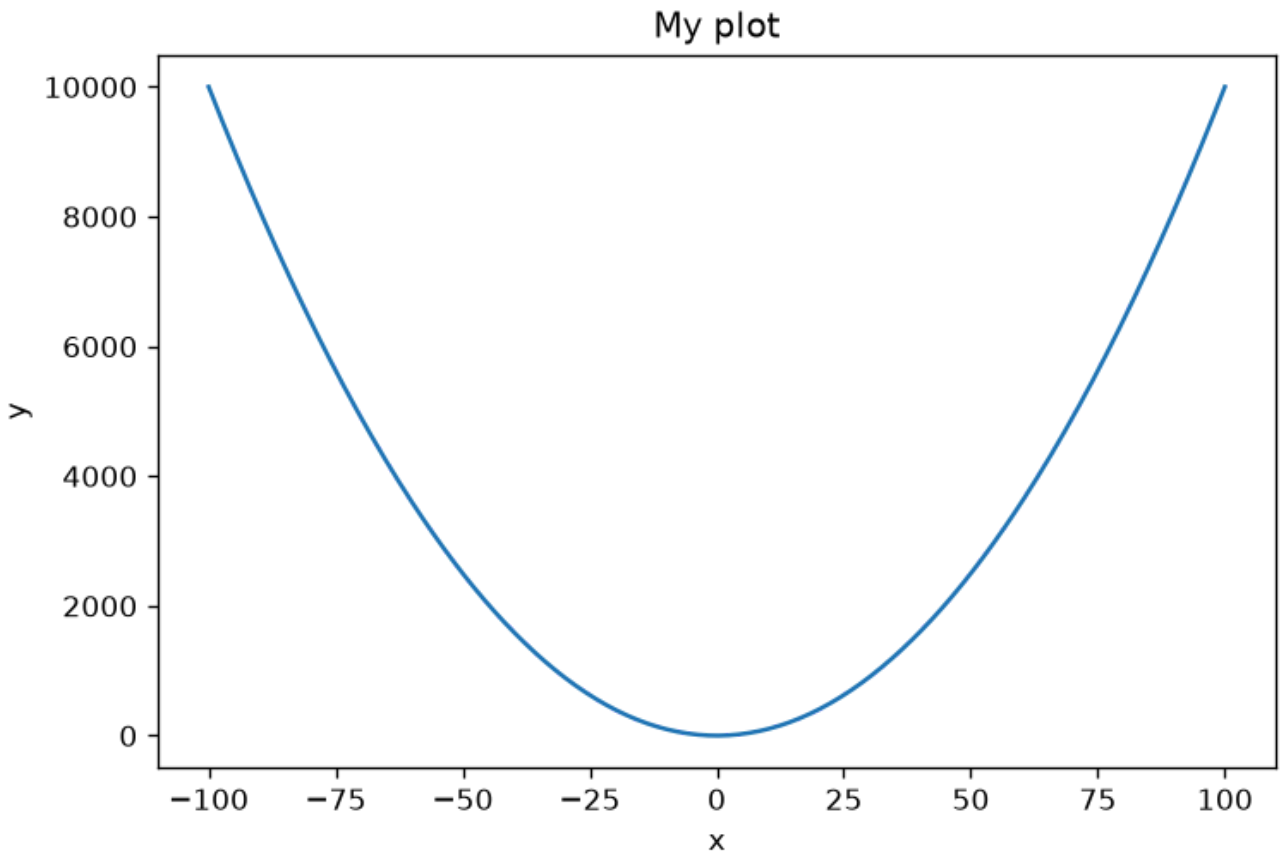
```
import numpy as np

x = np.linspace(-100,100,300) # linearly sample 300 points between -100 and 100
y = x**2 # calculate the square of each point in x
```

Proceed to print:

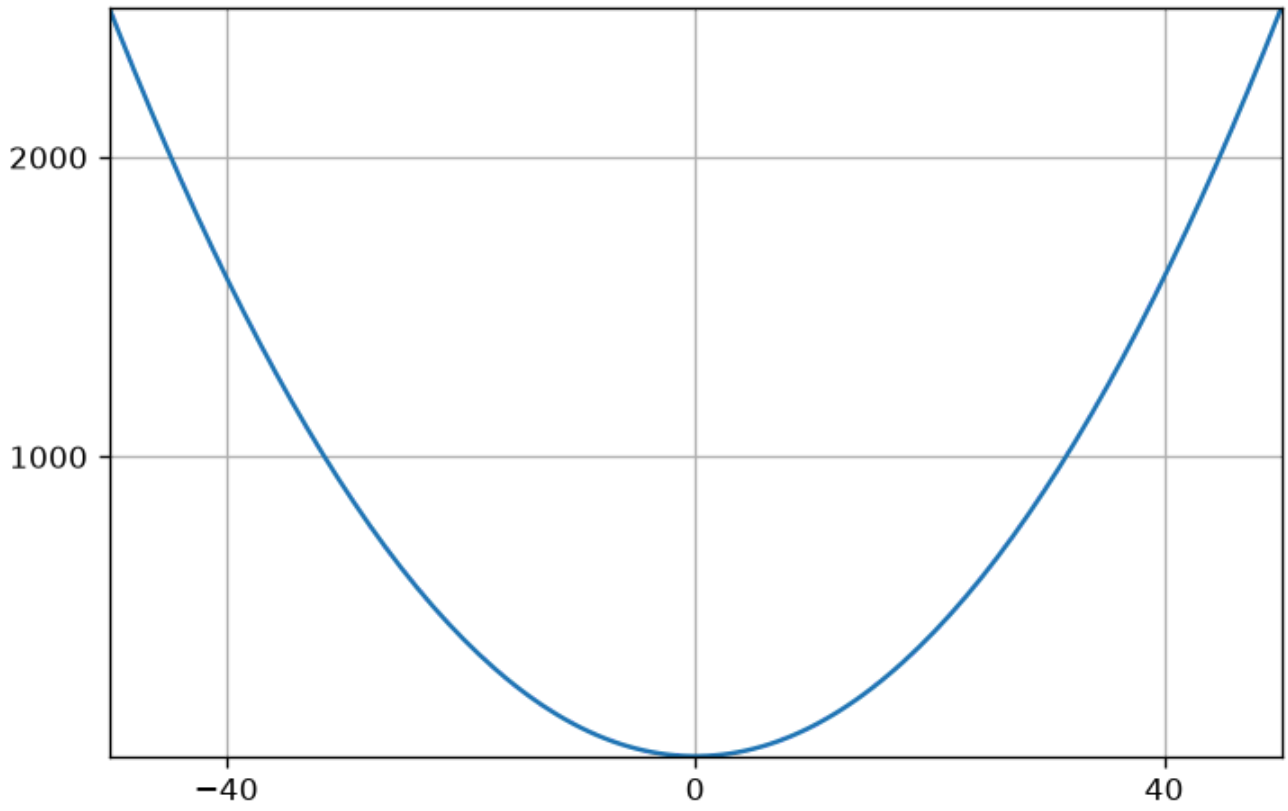
```
from matplotlib import pyplot as plt #imports the pyplot module from matplotlib and
        calls it plt

plt.plot(x,y) #plots the x,y pairs as points in the Cartesian space
plt.xlabel('x') #sets a label for the x-axis
plt.ylabel('y') #sets a label for the y-axis
plt.title('My plot') #sets the title of the plot
plt.show() #shows the plot
```



It is possible to control various aspects of the plot, including the limits of the x and y axes, the position of the “ticks” (the horizontal and vertical small lines on the axes) or adding a grid. Let’s see some examples:

```
plt.plot(x,y) #display the same plot
plt.xlim([-50,50]) #display x values only between -20 and 40
plt.ylim([0,2500]) #and y values only between 0 and 4000
plt.xticks([-40,0,40]) #insert only -40, 0 and 40 as "ticks" on the x-axis
plt.yticks([1000,2000]) #only 1000 and 2000 as y ticks
plt.grid() #add a grid
plt.show()
```



Subplot

It can often be useful to compare different plots. To do this, we can make use of the subplot function, which allows us to construct a “grid of plots”.

Let’s see, for example, how to plot the following functions in a 2 x 2 grid:

$$y = x^2, \quad y = log(x), \quad y = x^2 \cdot log(x), \quad y = log(x)/x$$

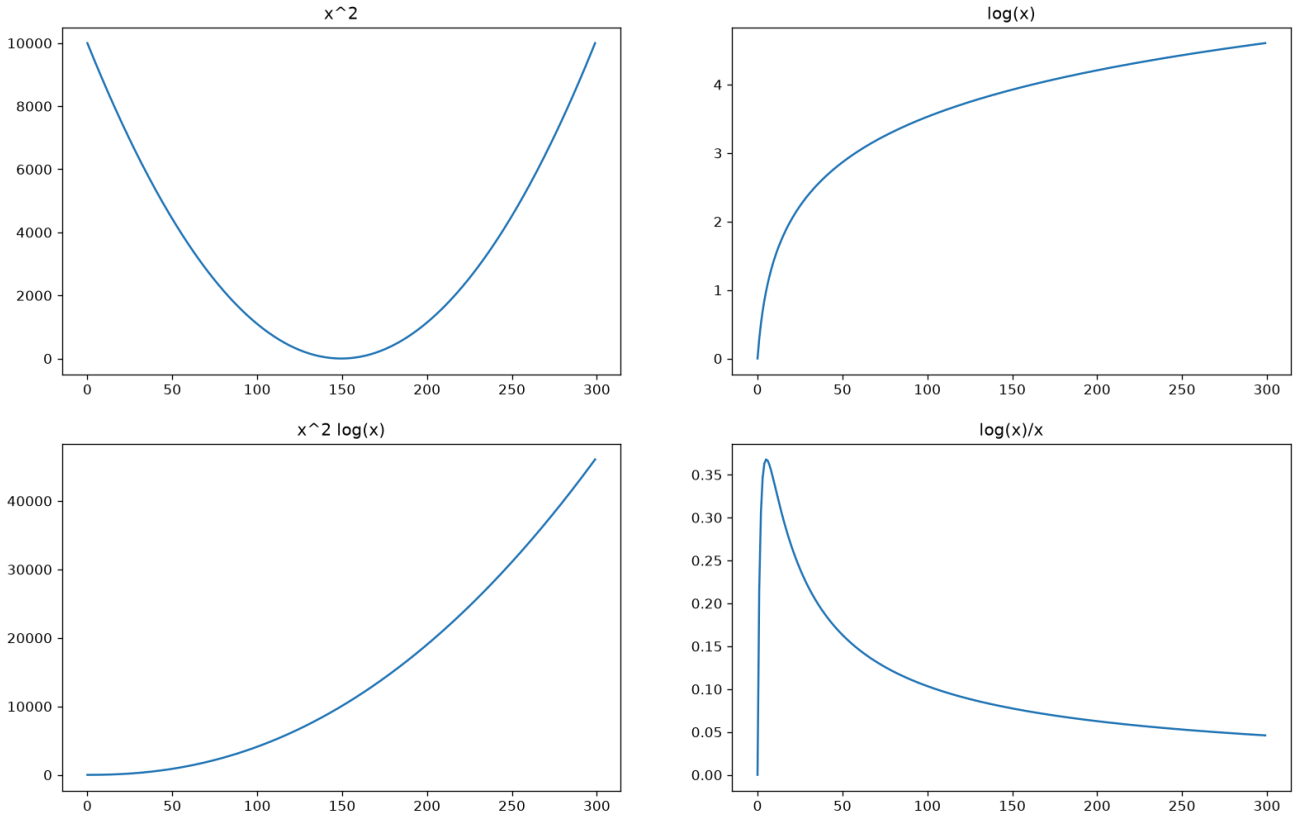
```
plt.figure(figsize=(16,10)) #defines a figure with a given size (in inches)
plt.subplot(2,2,1) #defines a 2x2 grid and selects the first cell as the current cell
x = np.linspace(-100,100,300)
plt.plot(x**2) #first graph
plt.title('x^2') #sets a title for the current cell

plt.subplot(2,2,2) #still in the same 2x2 grid, selects the second cell
x = np.linspace(1,100,300)
plt.plot(np.log(x)) #second graph
plt.title('log(x)') #sets a title for the current cell

plt.subplot(2,2,3) #selects the third cell
x = np.linspace(1,100,300)
plt.plot(x**2*np.log(x)) #third graph
plt.title('x^2 log(x)') #sets a title for the current cell

plt.subplot(2,2,4) #selects the fourth cell
x = np.linspace(1,100,300)
plt.plot(np.log(x)/x) #fourth graph
plt.title('log(x)/x') #sets a title for the current cell

plt.show() #shows the figure
```



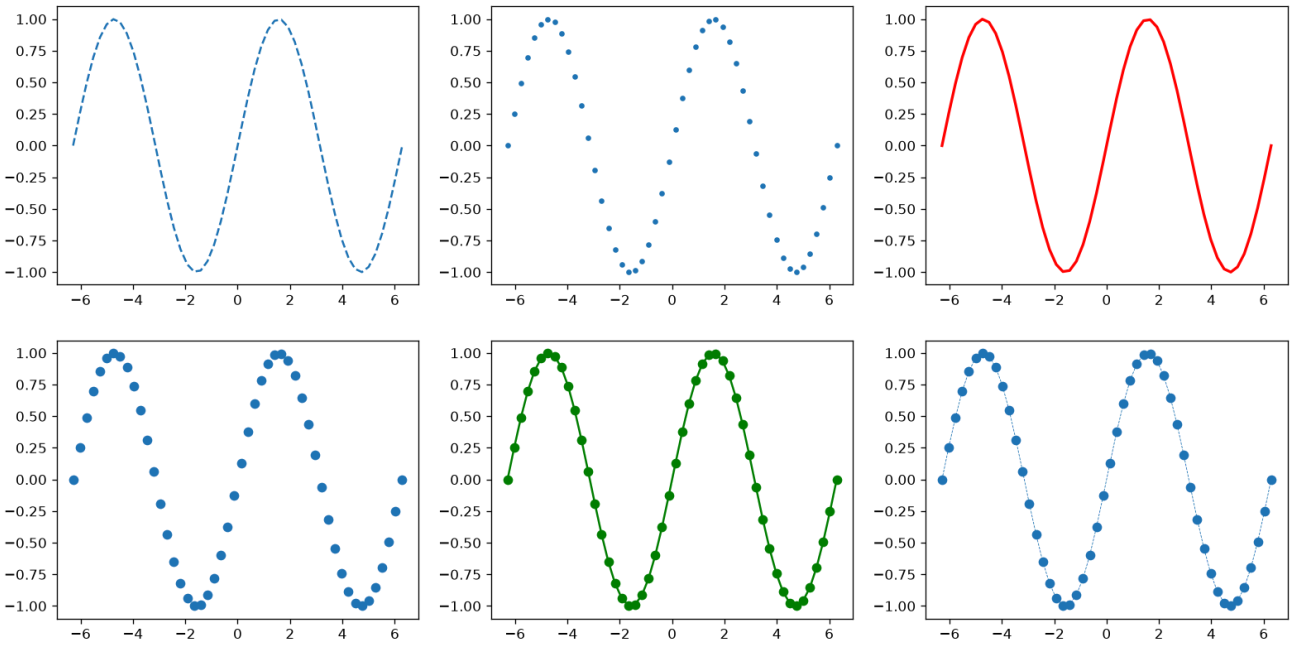
Colors and styles

It is possible to use different colors and styles for the charts. Below are some examples:

```
x = np.linspace(-2*np.pi,2*np.pi,50)
y = np.sin(x)

plt.figure(figsize=(16,8))
plt.subplot(231) #abbreviated version of plt.subplot(2,3,1)
plt.plot(x,y,'--') #dashed line
plt.subplot(232)
plt.plot(x,y,'.') #only points
plt.subplot(233)
plt.plot(x,y,'r', linewidth=2) #red line, linewidth 2
plt.subplot(234)
plt.plot(x,y,'o') #circles
plt.subplot(235)
plt.plot(x,y,'o-g') #circles connected by green lines
plt.subplot(236)
plt.plot(x,y,'o--', linewidth=0.5) #circles connected by dashed lines, linewidth 0.5

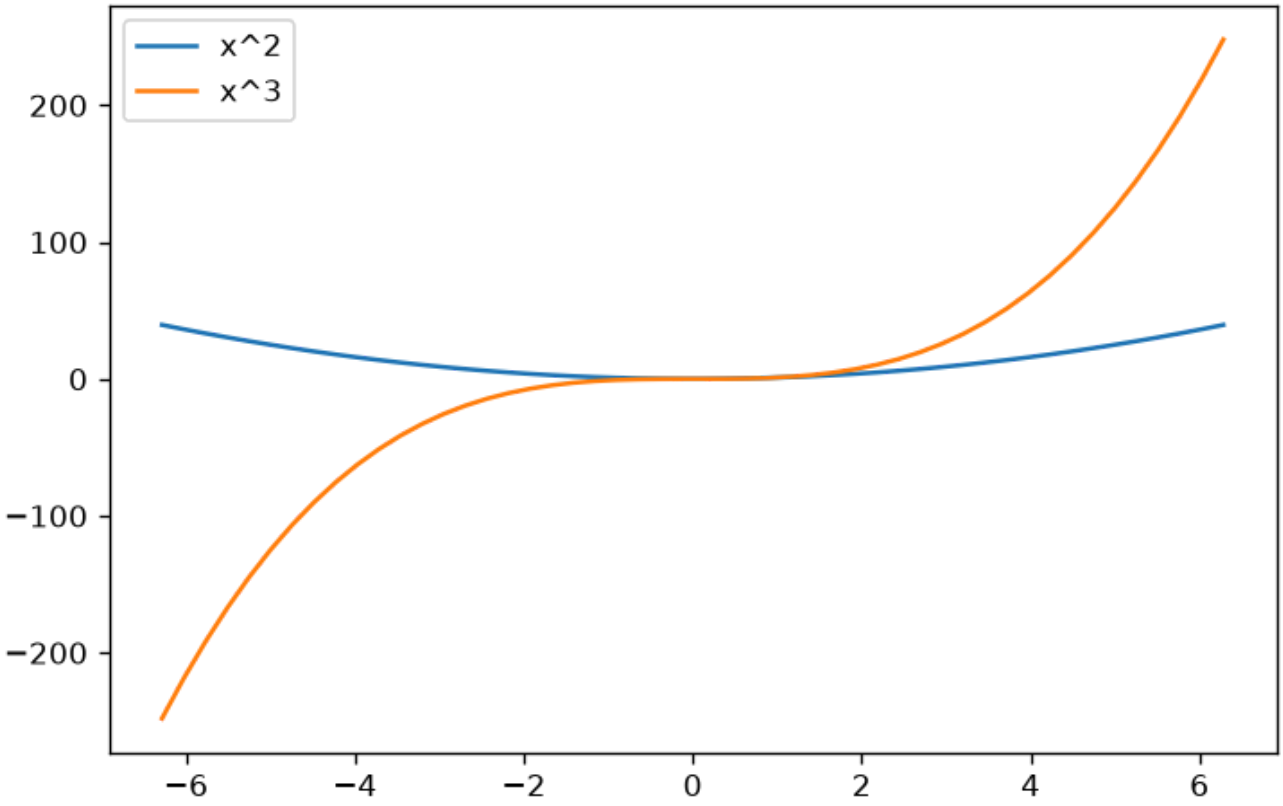
plt.show()
```



Line Overlay and Legend

It's simple to overlay multiple lines using **matplotlib**:

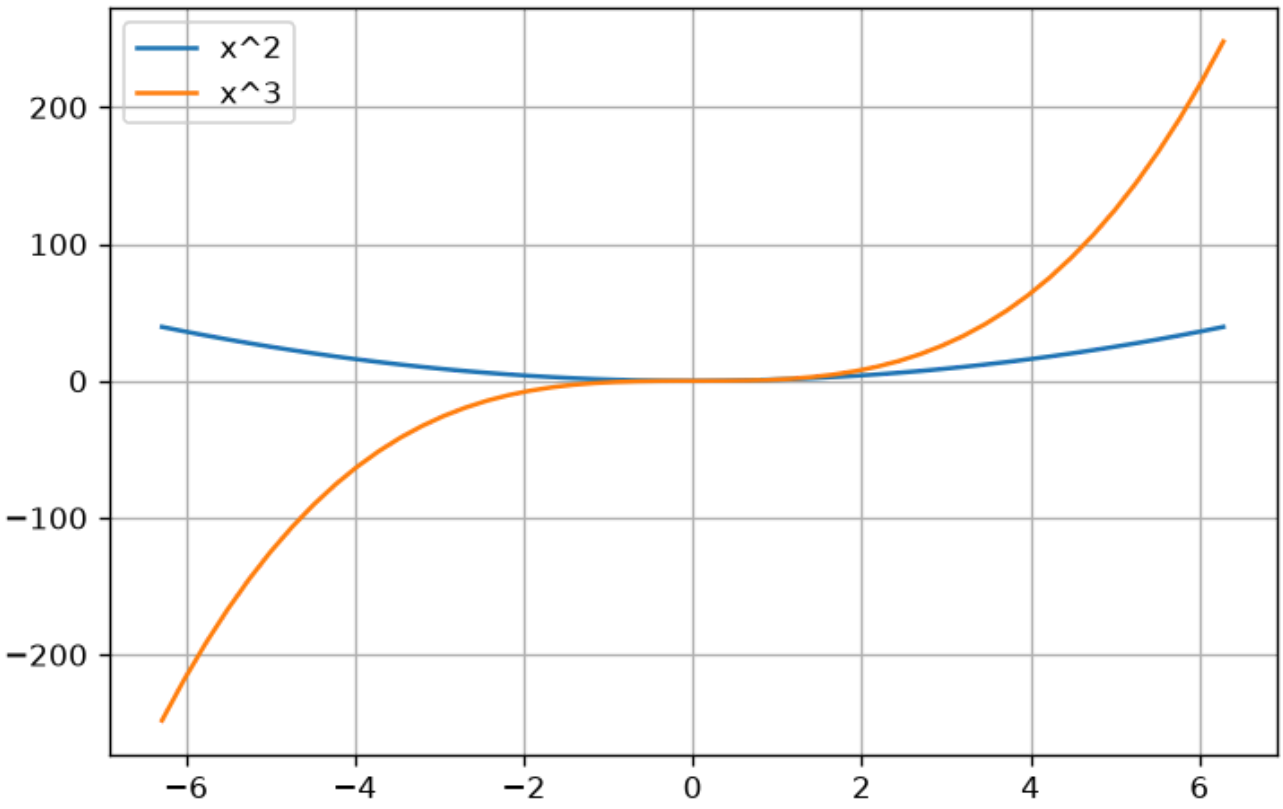
```
plt.figure()
plt.plot(x,x**2)
plt.plot(x,x**3) # For the second plot, matplotlib will automatically change color
plt.legend(['x^2','x^3']) # We can also add a legend.
# The legend elements will be associated with the lines in the order they were
        plotted
plt.show()
```



Grids

It is often useful to superimpose a grid on your plots. This can be done using `plt.grid()`:

```
plt.figure()
plt.plot(x,x**2)
plt.plot(x,x**3) # For the second plot, matplotlib will automatically change color
plt.legend(['x^2','x^3']) # We can also add a legend.
# The legend elements will be associated with the lines in the order they were
        plotted
plt.grid()
plt.show()
```



Matplotlib is a very versatile library. On Matplotlib's official website, there is a gallery of plot examples with their respective code (https://matplotlib.org/stable/gallery/index.html). We will see in more detail how to obtain more complex visualizations during the course.

Introduction to Pandas

We've seen how Python, combined with numpy and matplotlib, can be a powerful tool for scientific computing. However, there are other libraries that provide greater abstraction for easier data management and analysis. In this lab, we will cover the basics of the Pandas library, which is useful for data analysis. Specifically, we will see:

- How to create and manage data series (Pandas Series);
- How to create and manage Pandas DataFrames;
- Examples of how to manipulate data in a DataFrame.

Pandas is a high-level library that provides various tools and data structures for data analysis. In particular, Pandas is very useful for loading, manipulating, and visualizing data quickly and conveniently before moving on to the actual analysis. The two main data structures in Pandas are `Series` and `DataFrame`.

Series

A `Series` is a one-dimensional structure (a sequence of data) very similar to a NumPy array. Unlike it, its elements can be indexed using labels, as well as numbers. The values contained in a series can be

of any type.

Creation of Series

It is possible to define a `Series` starting from a list or a NumPy array:

```
import pandas as pd
import numpy as np
np.random.seed(123) # set a seed for reproducibility
s1 = pd.Series([7,5,2,8,9,6])# alternatively print(pd.Series(np.array([7,5,2])))
print(s1)
```

```
0    7
1    5
2    2
3    8
4    9
5    6
dtype: int64
```

The numbers displayed on the left represent the labels of the values contained in the series, which, by default, are numeric and sequential. When defining a series, you can specify appropriate labels (one for each value):

```
values = [7,5,2,8,9,6]
labels = ['a','b','c','d','e','f']
s2 = pd.Series(values, index=labels)
print(s2)
```

```
a    7
b    5
c    2
d    8
e    9
f    6
dtype: int64
```

It is possible to define a set also by means of a dictionary that specifies labels and values simultaneously:

```
s3=pd.Series({'a':14,'h':18,'m':72})
print(s3)
```

```
a    14
h    18
m    72
dtype: int64
```

Optionally, you can assign a name to a series:

```
pd.Series([1,2,3], name='My Series')
```

```
0    1
1    2
2    3
Name: My Series, dtype: int64
```

Question 1

What is the main difference between Pandas `Series` and Numpy `array`?

Indexing Series

When the indices are numeric, Series can be indexed like numpy arrays:

```
print(s1[0]) #indexing
print(s1[0:4:2]) #slicing
```

```
7
0    7
2    2
dtype: int64
```

When indices are the most generic labels, indexing occurs similarly, but slicing cannot be used:

```
print(s2['c'])
```

```
2
```

Clearly, it is possible to modify the values in a series using indexing:

```
s2['c']=4
s2
```

```
a    7
b    5
c    4
d    8
e    9
f    6
dtype: int64
```

If the specified index does not exist, a new element will be created:

```
s2['z']=-2
s2
```

```
a    7
b    5
c    4
d    8
e    9
f    6
z   -2
dtype: int64
```

If we wanted to specify more than one label at a time, we could pass a list of labels:

```
print(s2[['a','c','d']])
```

```
a    7
c    4
d    8
dtype: int64
```

Series with alphanumeric labels can also be indexed by following the order in which data is entered into the series, in a sense "discarding" the alphanumeric labels and indexing the elements positionally. This effect is achieved using the `iloc` method:

```
print(s2,'\n')
print("Element at index 'a':",s2['a'])
print("First element of the series:",s2.iloc[0])
```

```
a    7
b    5
c    4
d    8
e    9
f    6
z   -2
dtype: int64

Element at index 'a': 7
First element of the series: 7
```

In certain cases, it can be useful to reset the index numbering. This can be done using the `reset_index` method:

```
print(s3,'\n')
print(s3.reset_index(drop=True)) # drop=True indicates to discard old indices
```

```
a    14
h    18
m    72
dtype: int64

0    14
1    18
2    72
dtype: int64
```

Series also allow logical indexing:

```
print(s1,'\n') # series s1
print(s1>2,'\n') # logical indexing to select elements greater than 2
print(s1[s1>2],'\n') # application of logical indexing
```

```
0    7
1    5
2    2
3    8
4    9
5    6
dtype: int64

0     True
1     True
2    False
3     True
4     True
5     True
dtype: bool

0    7
1    5
3    8
4    9
5    6
dtype: int64
```

It is possible to specify the combination of two conditions using the logical operators “|” (or) and “&” (and), remembering to enclose the operands in parentheses:

```
(s1>2) & (s1<6)
```

```
0    False
1     True
2    False
3    False
4    False
5    False
dtype: bool
```

```
s1[(s1>2) & (s1<6)]
```

```
1    5
dtype: int64
```

Similarly for the or:

```
(s1<2) | (s1>6)
```

```
0     True
1    False
2    False
3     True
4     True
5    False
dtype: bool
```

```
print(s1[(s1<2) | (s1>6)])
```

```
0    7
3    8
4    9
dtype: int64
```

As with NumPy arrays, memory allocation is dynamically managed for Series. Therefore, if I assign a series to a new variable and modify the second variable, the first one will also be modified:

```
s11=pd.Series([1,2,3])
s12=s11
s12[0]=-1
s11
```

```
0   -1
1    2
2    3
dtype: int64
```

To obtain a new independent Series, we can use the `copy` method:

```
s11=pd.Series([1,2,3])
s12=s11.copy()
s12[0]=-1
s11
```

```
0    1
1    2
2    3
dtype: int64
```

Question 2

What does the following code print to the screen?

```
s=pd.Series([1,2,3,4,6])
print(s[s%2])
```

Data Types

`Series` can contain different data types:

```
pd.Series([2.5,3.4,5.2])
```

```
0    2.5
1    3.4
2    5.2
dtype: float64
```

A `Series` is associated with a single data type. If we specify heterogeneous data types, the `Series` will be of type "object":

```
s=pd.Series([2.5,'A',5.2])
s
```

```
0    2.5
1      A
2    5.2
dtype: object
```

You can change the data type of a Series on the fly with `astype` in a similar way to how it's done with Numpy arrays:

```
s=pd.Series([2,3,8,9,12,45])
print(s)
print(s.astype(float))
```

```
0     2
1     3
2     8
3     9
4    12
5    45
dtype: int64
0     2.0
1     3.0
2     8.0
3     9.0
4    12.0
5    45.0
dtype: float64
```

Question 3

Considering the Series defined as follows:

```
s=pd.Series([2.5,'A',5.2])
```

which of the following operations return an error? Why?

```
s.astype(str), s.astype(int)
```

Missing Data

In Pandas, it's possible to specify Series (and as we'll see, DataFrames as well) with missing data. To indicate such values, the value `np.nan` is used:

```
missing = pd.Series([2,5,np.nan,8])
missing
```

```
0    2.0
1    5.0
2    NaN
3    8.0
dtype: float64
```

The `NaN` (not a number) values indicate values that can be missing for various reasons (e.g., data resulting from invalid operations, data that were not collected correctly, etc.).

In many cases, it can be useful to discard `NaN` data. This can be done with the `dropna()` operation:

```
missing.dropna()
```

```
0    2.0
1    5.0
3    8.0
dtype: float64
```

Alternatively, we can replace `NaN` values with a specific value using `fillna`. We replace the `NaN`s with zero:

```
missing = pd.Series([2, 5, np.nan, 8])
print(missing)

missing.fillna(0)
```

```
0    2.0
1    5.0
2    NaN
3    8.0
dtype: float64
```

```
0    2.0
1    5.0
2    0.0
3    8.0
dtype: float64
```

Operations on and Between Series

The main operations available for NumPy arrays are defined on Series:

```
print("Min:",s1.min())
print("Max:",s1.max())
print("Mean:",s1.mean())
```

```
Min: 2
Max: 9
Mean: 6.166666666666667
```

Let's see how to replace missing values in a series with the average value:

```
data = pd.Series([1,5,2,np.nan,5,2,6,np.nan,9,2,3])
data

data.fillna(data.mean())
```

```
0     1.000000
1     5.000000
2     2.000000
3     3.888889
4     5.000000
5     2.000000
6     6.000000
7     3.888889
8     9.000000
9     2.000000
10    3.000000
dtype: float64
```

Note that the `mean` function simply ignored the `NaN` values.

It is possible to obtain the size of a series using the `len` function:

```
print(len(s1))
```

```
6
```

It is possible to obtain statistics on the unique values (i.e. the values of the series, once duplicates have been removed) contained in a series using the `unique` method:

```
print("Unique:",s1.unique()) #returns unique values
```

```
Unique: [7 5 2 8 9 6]
```

To know the number of unique values in a series, we can use the `nunique` method:

```
s1.nunique()
```

```
6
```

It is possible to obtain the unique values of a series along with the frequencies with which they appear in the series using the `value_counts` method:

```
tmp = pd.Series(np.random.randint(0,10,100))
print(tmp.unique()) # unique values
tmp.value_counts() # unique values with their frequencies
```

```
[2 6 1 3 9 0 4 7 8 5]
```

```
3    15
6    14
1    13
2    10
9    10
0    10
4    10
7     8
8     5
5     5
Name: count, dtype: int64
```

The result of `value_counts` is a `Series` where the indices represent the unique values, and the values are the frequencies with which they appear in the series. The series is sorted by values.

The `describe` method allows you to calculate various statistics for the values contained in the series:

```
tmp.describe()
```

```
count    100.000000
mean       4.130000
std        2.830765
min        0.000000
25%        2.000000
50%        4.000000
75%        6.000000
max        9.000000
dtype: float64
```

In operations between series, elements are associated based on their indices. In the case where there is a perfect correspondence between elements, we obtain the following result:

```
print(pd.Series([1,2,3])+pd.Series([4,4,4]),'\n')
print(pd.Series([1,2,3])*pd.Series([4,4,4]),'\n')
```

```
0    5
1    6
2    7
dtype: int64

0     4
1     8
2    12
dtype: int64
```

In the event that some indices are missing, the corresponding boxes will be filled with `NaN` (not a number) to indicate the absence of the value:

```
s1 = pd.Series([1,4,2], index = [1,2,3])
s2 = pd.Series([4,2,8], index = [0,1,2])
print(s1,'\n')
print(s2,'\n')
print(s1+s2)
```

```
1    1
2    4
3    2
dtype: int64

0    4
1    2
2    8
dtype: int64

0     NaN
1     3.0
2    12.0
3     NaN
dtype: float64
```

In this case, index `0` was present only in the second series (`s2`), while index `3` was present only in the first series (`s1`).

If we want to exclude `NaN` values (including their indices) we can use the `dropna` method:

```
s3=s1+s2
print(s3)
print(s3.dropna())
```

```
0     NaN
1     3.0
2    12.0
3     NaN
dtype: float64
1     3.0
2    12.0
dtype: float64
```

It is possible to apply a function to all elements of a Series using the `apply` method. For example, let's say we want to transform strings contained in a Series into uppercase. We can specify the `str.upper` function using the `apply` method:

```
s=pd.Series(['aba','cda','daf','acc'])
s.apply(str.upper)
```

```
0    ABA
1    CDA
2    DAF
3    ACC
dtype: object
```

Using apply we can also apply user-defined functions using lambda expressions or using the usual syntax:

```
def my_function(x):
    y="String: "
    return y+x
s.apply(my_function)
```

```
0    String: aba
1    String: cda
2    String: daf
3    String: acc
dtype: object
```

The same function can be written more compactly as a lambda expression:

```
s.apply(lambda x: "String: "+x)
```

```
0    String: aba
1    String: cda
2    String: daf
3    String: acc
dtype: object
```

It is possible to modify all occurrences of a given value in a Series using the `replace` method:

```
ser = pd.Series([1,2,15,-1,7,9,2,-1])
print(ser)
ser=ser.replace({-1:0}) # replace all occurrences of "-1" with zeros
print(ser)
```

```
0     1
1     2
2    15
3    -1
4     7
5     9
6     2
7    -1
dtype: int64
0     1
1     2
2    15
3     0
4     7
5     9
6     2
7     0
dtype: int64
```

Question 4

Try to apply the `apply` method to a `Series`, specifying a function that takes two arguments as input. Is it possible to do this? Why?

Conversion to Numpy Array

It is possible to access the values of the series in the form of a numpy array using the `values` property:

```
print(s3.values)
```

```
[nan  3. 12. nan]
```

You need to be careful about the fact that `values` does not create an independent copy of the series, so if we modify the numpy array accessible via `values`, we are actually modifying the series as well:

```
a = s3.values
print(s3)
a[0]=-1
print(s3)
```

```
0     NaN
1     3.0
2    12.0
3     NaN
dtype: float64
0    -1.0
1     3.0
2    12.0
3     NaN
dtype: float64
```

Question 5

How can you obtain a Numpy array from a Series such that the two entities are independent?

DataFrame

A **DataFrame** is essentially a table of numbers in which:

- Each row represents a different *observation*;
- Each column represents a variable.

Rows and columns can have names. In particular, it is very common to assign names to columns to indicate which variable each one corresponds to.

DataFrame Construction and Visualization

It is possible to construct a DataFrame from a two-dimensional NumPy array (a matrix):

```
data = np.random.rand(10,3) # matrix of random values 10 x 3
# it's a matrix of 10 observations, each characterized by 3 variables

df = pd.DataFrame(data,columns=['A','B','C'])

df # in jupyter or in an ipython shell we can print the dataframe
# by simply writing "df". In a script we should write "print df"
```

	A	B	C
0	0.309884	0.507204	0.280793
1	0.763837	0.108542	0.511655
2	0.909769	0.218376	0.363104
3	0.854973	0.711392	0.392944
4	0.231301	0.380175	0.549162
5	0.556719	0.004135	0.638023
6	0.057648	0.043027	0.875051
7	0.292588	0.762768	0.367865
8	0.873502	0.029424	0.552044
9	0.240248	0.884805	0.460238

Each line has been automatically assigned a numerical index. If we wish, we can also specify names for the lines:

```
np.random.seed(123)#set a seed for repeatability
df = pd.DataFrame(np.random.rand(4,3),columns=['A','B','C'],index=['X','Y','Z','W'])
df
```

	A	B	C
X	0.696469	0.286139	0.226851
Y	0.551315	0.719469	0.423106
Z	0.980764	0.684830	0.480932
W	0.392118	0.343178	0.729050

Analogous to what was seen in the case of series, it is possible to construct a DataFrame using a dictionary that specifies the name and values of each column:

```
pd.DataFrame({'A':np.random.rand(10), 'B':np.random.rand(10),
        'C':np.random.rand(10)})
```

	A	B	C
0	0.438572	0.724455	0.430863
1	0.059678	0.611024	0.493685
2	0.398044	0.722443	0.425830
3	0.737995	0.322959	0.312261
4	0.182492	0.361789	0.426351
5	0.175452	0.228263	0.893389
6	0.531551	0.293714	0.944160
7	0.531828	0.630976	0.501837
8	0.634401	0.092105	0.623953
9	0.849432	0.433701	0.115618

In the case of very large datasets, we can display only the first few rows using the **head** method:

```
df_big = pd.DataFrame(np.random.rand(100,3),columns=['A','B','C'])
df_big.head()
```

	A	B	C
0	0.317285	0.414826	0.866309
1	0.250455	0.483034	0.985560
2	0.519485	0.612895	0.120629
3	0.826341	0.603060	0.545068
4	0.342764	0.304121	0.417022

You can also specify the number of lines to display:

```
df_big.head(2)
```

	A	B	C
0	0.317285	0.414826	0.866309
1	0.250455	0.483034	0.985560

In a similar way, we can show the last lines with **tail**:

```
df_big.tail(3)
```

	A	B	C
97	0.811953	0.335544	0.349566
98	0.389874	0.754797	0.369291
99	0.242220	0.937668	0.908011

We can obtain the number of rows of the **DataFrame** using the **len** function:

```
print(len(df_big))
```

```
100
```

If we want to know both the number of rows and the number of columns, we can call the **shape** property:

```
print(df_big.shape)
```

```
(100, 3)
```

Analogous to what was seen for Series, we can view a DataFrame as a numpy array by calling the `values` property:

```
print(df,'\n')
print(df.values)
```

```
          A         B         C
X  0.696469  0.286139  0.226851
Y  0.551315  0.719469  0.423106
Z  0.980764  0.684830  0.480932
W  0.392118  0.343178  0.729050

[[0.69646919 0.28613933 0.22685145]
 [0.55131477 0.71946897 0.42310646]
 [0.9807642  0.68482974 0.4809319 ]
 [0.39211752 0.34317802 0.72904971]]
```

The same considerations about memory made for Series also apply to DataFrames. To obtain an independent copy of a DataFrame, you can use the `copy` method:

```
df2=df.copy()
```

Question 6

In what way does a DataFrame mainly differ from a two-dimensional Numpy array? What is the advantage of DataFrames?

Indexing

Pandas provides a range of tools for indexing `DataFrames` by selecting rows or columns. For example, we can select column B as follows:

```
s = df['B']
print(s, '\n')
print("Type of s:", type(s))
```

```
X    0.286139
Y    0.719469
Z    0.684830
W    0.343178
Name: B, dtype: float64

Type of s: <class 'pandas.core.series.Series'>
```

It should be noted that the result of this operation is a `Series` (a `DataFrame` is fundamentally a collection of `Series`, each representing a column) which has the name of the considered column as its name. We can select more than one row by specifying a list of rows:

```
dfAB = df[['A','C']]
dfAB
```

	A	C
X	0.696469	0.226851
Y	0.551315	0.423106
Z	0.980764	0.480932
W	0.392118	0.729050

The result of this operation is instead a `DataFrame`. Row selection is done using the `loc` property:

```
df5 = df.loc['X']
df5
```

```
A    0.696469
B    0.286139
C    0.226851
Name: X, dtype: float64
```

The result of this operation is also a `Series`, but in this case the indices represent the column names, while the name of the series corresponds to the index of the selected row. As with `Series`, we can use `iloc` to index rows positionally:

```
df.iloc[1] #equivalent to df.loc['Y']
```

```
A    0.551315
B    0.719469
C    0.423106
Name: Y, dtype: float64
```

It is also possible to chain indexing operations to select a specific value:

```
print(df.iloc[1]['A'])
print(df['A'].iloc[1])
```

```
0.5513147690828912
0.5513147690828912
```

Even in this case, we can use logical indexing in a similar way to what we saw for numpy:

```
df_big2=df_big[df_big['C']>0.5]
print(len(df_big), len(df_big2))
df_big2.head() # some indices are missing as the corresponding rows have been removed
```

```
100 53
```

	A	B	C
0	0.317285	0.414826	0.866309
1	0.250455	0.483034	0.985560
3	0.826341	0.603060	0.545068
5	0.681301	0.875457	0.510422
6	0.669314	0.585937	0.624904

It's possible to combine what we've seen so far to manipulate data simply and quickly. Let's assume we want to select rows for which the sum of the values in B and C is less than 0.7, and let's assume we are only interested in the A values of those rows (and not the entire row of A, B, C values). The result we expect is a one-dimensional array of values. We can obtain the desired result as follows:

```
average_result = df[(df['B'] + df['C']) > 0.7]['A']
print(average_result.head(), average_result.shape)
```

```
Y    0.551315
Z    0.980764
W    0.392118
Name: A, dtype: float64 (3,)
```

We can apply indexing also at the level of the entire table (in addition to the level of individual columns):

```
df>0.3
```

	A	B	C
X	True	False	False
Y	True	True	True
Z	True	True	True
W	True	True	True

If we apply this indexing to the `DataFrame`, we will get the appearance of some `NaN`s, which indicate the presence of elements that do not respect the condition considered:

```
df[df>0.3]
```

	A	B	C
X	0.696469	NaN	NaN
Y	0.551315	0.719469	0.423106
Z	0.980764	0.684830	0.480932
W	0.392118	0.343178	0.729050

We can remove `NaN` values using the `dropna` method, as seen in the case of `Series`. However, in this case, **all rows that have at least one `NaN` will be removed**:

```
df[df>0.3].dropna()
```

	A	B	C
Y	0.551315	0.719469	0.423106
Z	0.980764	0.684830	0.480932
W	0.392118	0.343178	0.729050

We can ask `dropna` to remove **columns with at least one NaN** by specifying `axis=1`:

```
df[df>0.3].dropna(axis=1)
```

	A
X	0.696469
Y	0.551315
Z	0.980764
W	0.392118

Alternatively, we can replace the `NaN` values on the fly using the `fillna` function:

```
df[df>0.3].fillna('VAL') #replaces NaNs with 'VAL'
```

	A	B	C
X	0.696469	VAL	VAL
Y	0.551315	0.719469	0.423106
Z	0.980764	0.68483	0.480932
W	0.392118	0.343178	0.72905

Even in this case, as seen with `Series`, we can restore the order of the indices using the `reset_index` method:

```
print(df)
print(df.reset_index(drop=True))
```

```
          A         B         C
X  0.696469  0.286139  0.226851
Y  0.551315  0.719469  0.423106
Z  0.980764  0.684830  0.480932
W  0.392118  0.343178  0.729050
          A         B         C
0  0.696469  0.286139  0.226851
1  0.551315  0.719469  0.423106
2  0.980764  0.684830  0.480932
3  0.392118  0.343178  0.729050
```

If we don't specify `drop=True`, the old indices will be maintained as a new column:

```
print(df.reset_index())
```

```
  index         A         B         C
0     X  0.696469  0.286139  0.226851
1     Y  0.551315  0.719469  0.423106
2     Z  0.980764  0.684830  0.480932
3     W  0.392118  0.343178  0.729050
```

We can set any column as the new index. For example:

```
df.set_index('A')
```

	B	C
A		
0.696469	0.286139	0.226851
0.551315	0.719469	0.423106
0.980764	0.684830	0.480932
0.392118	0.343178	0.729050

It should be noted that this operation does not actually modify the DataFrame, but creates a new "view" of the data with the requested modification.

To use this modified version of the dataframe, we can save it in another variable:

```
df2=df.set_index('A')
df2
```

	B	C
A		
0.696469	0.286139	0.226851
0.551315	0.719469	0.423106
0.980764	0.684830	0.480932
0.392118	0.343178	0.729050

Question 7

Consider the following DataFrame:

```
pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
```

Use logical indexing to select the rows where the sum of the values in 'C' and 'A' is greater than 1.

DataFrame Manipulation

The values contained in the rows and columns of the dataframe can be easily modified. The following example multiplies all values in column B by 2:

```
df['B'] *= 2
df.head()
```

	A	B	C
X	0.696469	0.572279	0.226851
Y	0.551315	1.438938	0.423106
Z	0.980764	1.369659	0.480932
W	0.392118	0.686356	0.729050

Similarly, we can divide all the values in row with index 2 by 3:

```
df.iloc[2]=df.iloc[2]/3
df.head()
```

	A	B	C
X	0.696469	0.572279	0.226851
Y	0.551315	1.438938	0.423106
Z	0.326921	0.456553	0.160311
W	0.392118	0.686356	0.729050

We can define a new column with a simple assignment operation:

```
df['D'] = df['A'] + df['C']
df['E'] = np.ones(len(df))*5
df.head()
```

	A	B	C	D	E
X	0.696469	0.572279	0.226851	0.923321	5.0
Y	0.551315	1.438938	0.423106	0.974421	5.0
Z	0.326921	0.456553	0.160311	0.487232	5.0
W	0.392118	0.686356	0.729050	1.121167	5.0

We can remove a column using the `drop` method and specifying `axis=1` to indicate that we want to remove a column:

```
df.drop('E',axis=1).head()
```

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167

The `drop` method does not modify the DataFrame but only generates a new "view" without the column to be removed:

```
df.head()
```

	A	B	C	D	E
X	0.696469	0.572279	0.226851	0.923321	5.0
Y	0.551315	1.438938	0.423106	0.974421	5.0
Z	0.326921	0.456553	0.160311	0.487232	5.0
W	0.392118	0.686356	0.729050	1.121167	5.0

We can actually remove the column by an assignment:

```
df=df.drop('E',axis=1)
df.head()
```

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167

The removal of rows happens in the same way, but you need to specify `axis = 0`:

```
df.drop('X', axis=0)
```

	A	B	C	D
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167

It is also possible to add a new row at the end of the DataFrame with `pd.concat`. Since the rows of a DataFrame are Series, we will need to construct a series with the correct indices (corresponding to the

DataFrame's columns) and the correct name (corresponding to the new index), and turn it into a one-row DataFrame:

```
new_row=pd.Series([1,2,3,4], index=['A','B','C','D'], name='H')
print(new_row)
pd.concat([df, new_row.to_frame().T])
```

```
A    1
B    2
C    3
D    4
Name: H, dtype: int64
```

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167
H	1.000000	2.000000	3.000000	4.000000

We can add more than one row at a time by specifying a DataFrame:

```
new_rows = pd.DataFrame({'A':[0,1],'B':[2,3],'C':[4,5],'D':[6,7]}, index=['H','K'])
new_rows
```

	A	B	C	D
H	0	2	4	6
K	1	3	5	7

```
pd.concat([df, new_rows])
```

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167
H	0.000000	2.000000	4.000000	6.000000
K	1.000000	3.000000	5.000000	7.000000

Question 8

Consider the following DataFrame:

```
pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
```

Insert a new column 'D' into the DataFrame that contains the value 1 in all rows where the value of B is greater than the value of C.

Hint: use "astype(int)" to transform booleans into integers.

Operations between and on DataFrames

The operations seen in the case of Series remain defined on DataFrames, with the appropriate differences. Generally, these are applied to all columns of the DataFrame independently:

```
df.mean() # mean of each column
```

```
A    0.491706
B    0.788531
C    0.384830
D    0.876535
dtype: float64
```

```
df.max() # maximum of each column
```

```
A    0.696469
B    1.438938
C    0.729050
D    1.121167
dtype: float64
```

```
df.describe() #statistics for each column
```

	A	B	C	D
count	4.000000	4.000000	4.000000	4.000000
mean	0.491706	0.788531	0.384830	0.876535
std	0.165884	0.443638	0.255159	0.272747
min	0.326921	0.456553	0.160311	0.487232
25%	0.375818	0.543347	0.210216	0.814298
50%	0.471716	0.629317	0.324979	0.948871
75%	0.587603	0.874502	0.499592	1.011108
max	0.696469	1.438938	0.729050	1.121167

Since the columns of a DataFrame are Series, the `apply` method can be applied to them:

```
df['A']=df['A'].apply(lambda x: "Number: "+str(x))
df
```

	A	B	C	D
X	Number: 0.6964691855978616	0.572279	0.226851	0.923321
Y	Number: 0.5513147690828912	1.438938	0.423106	0.974421
Z	Number: 0.3269213994615385	0.456553	0.160311	0.487232
W	Number: 0.3921175181941505	0.686356	0.729050	1.121167

It is possible to sort the rows of a DataFrame by the values of one of the columns using the `sort_values` method:

```
df.sort_values(by='D')
```

	A	B	C	D
Z	Number: 0.3269213994615385	0.456553	0.160311	0.487232
X	Number: 0.6964691855978616	0.572279	0.226851	0.923321
Y	Number: 0.5513147690828912	1.438938	0.423106	0.974421
W	Number: 0.3921175181941505	0.686356	0.729050	1.121167

To make the sorting permanent, we need to perform an assignment:

```
df=df.sort_values(by='D')
df
```

	A	B	C	D
Z	Number: 0.3269213994615385	0.456553	0.160311	0.487232
X	Number: 0.6964691855978616	0.572279	0.226851	0.923321
Y	Number: 0.5513147690828912	1.438938	0.423106	0.974421
W	Number: 0.3921175181941505	0.686356	0.729050	1.121167

Question 9

Consider the following DataFrame:

```
pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
```

Transform the “B” column so that the new values are:

- equal to zero if previously even;
- equal to -1 if previously odd.

Remove Rows and Columns

You can remove a row or a column from a DataFrame:

```
data = pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
print(data)

data.drop(1) # remove row 1
data.drop('B', axis=1) # remove column B
```

```
   A  B   C
0  1  3   0
1  2  2   2
2  3  6  -1
3  4  7  12
```

	A	C
0	1	0
1	2	2
2	3	-1
3	4	12

Please note that the operations shown above do not modify the original array, but return a modified copy of it:

```
data = pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
print(data)

data.drop(1) # remove row 1
data
```

```
   A  B   C
0  1  3   0
1  2  2   2
2  3  6  -1
3  4  7  12
```

	A	B	C
0	1	3	0
1	2	2	2
2	3	6	-1
3	4	7	12

We can perform a persistent modification using `inplace=True`:

```
data = pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
print(data)

data.drop(1, inplace=True) # remove row 1
data
```

```
   A  B   C
0  1  3   0
1  2  2   2
2  3  6  -1
3  4  7  12
```

	A	B	C
0	1	3	0
2	3	6	-1
3	4	7	12

Or more explicitly as follows:

```
data = pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
print(data)

data = data.drop(1) # remove row 1
data
```

```
   A  B   C
0  1  3   0
1  2  2   2
2  3  6  -1
3  4  7  12
```

	A	B	C
0	1	3	0
2	3	6	-1
3	4	7	12

Groupby

The `groupby` method allows you to group the rows of a `DataFrame` and call aggregate functions on them. Let's consider a more representative `DataFrame`:

```
df=pd.DataFrame({'income':
        [10000,11000,9000,3000,1000,5000,7000,2000,7000,12000,8000],\
                 'age':[32,32,45,35,28,18,27,45,39,33,32],\
                 'sex':['M','F','M','M','M','F','F','M','M','F','F'],\
                 'company':
        ['CDX','FLZ','PTX','CDX','PTX','CDX','FLZ','CDX','FLZ','PTX','FLZ']})
df
```

	income	age	sex	company
0	10000	32	M	CDX
1	11000	32	F	FLZ
2	9000	45	M	PTX
3	3000	35	M	CDX
4	1000	28	M	PTX
5	5000	18	F	CDX
6	7000	27	F	FLZ
7	2000	45	M	CDX
8	7000	39	M	FLZ
9	12000	33	F	PTX
10	8000	32	F	FLZ

The `groupby` method allows us to group the rows of a `DataFrame` by value, with respect to a specified column. Suppose we want to group all rows that have the same value for **sex**:

```
df.groupby('sex')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1657bbe60>
```

This operation returns an object of type `DataFrameGroupby` on which aggregate operations (e.g., sums and averages) can be performed. Let's now assume we want to calculate the average of all values that fall into the same group (i.e., we calculate the average of the rows that have the same value for sex):

```
df.groupby('sex').mean(numeric_only=True)
```

	income	age
sex		
F	8600.000000	28.400000
M	5333.333333	37.333333

If we are interested in only one of the variables, we can select it before or after the operation on the aggregated data:

```
df.groupby('sex')['age'].mean() #equivalently: df.groupby('sex').mean()['age']
```

```
sex
F    28.400000
M    37.333333
Name: age, dtype: float64
```

The table shows the average income and average age of male and female subjects. Since the mean operation can only be applied to numerical values, the Company column has been excluded. We can obtain a similar table showing the sum of incomes and the sum of ages by changing `mean` to `sum`:

```
df.groupby('sex').sum()
```

	income	age	company
sex			
F	43000	142	FLZCDXFLZPTXFLZ
M	32000	224	CDXPTXCDXPTXCDXFLZ

In general, it is possible to use various aggregate functions besides `mean` and `sum`. Some examples are `min`, `max`, `std`. Two particularly interesting functions to use in this context are `count` and `describe`. Specifically, `count` counts the number of relevant elements, while describe calculates various statistics of the relevant values. Let's see two examples:

```
df.groupby('sex').count()
```

	income	age	company
sex			
F	5	5	5
M	6	6	6

The number of elements is the same for the various columns as there are no `NaN` values in the DataFrame.

```
df.groupby('sex').describe()
```

	income								age		
	count	mean	std	min	25%	50%	75%	max	count	mean	std
sex											
F	5.0	8600.000000	2880.972058	5000.0	7000.0	8000.0	11000.0	12000.0	5.0	28.400000	6.268971
M	6.0	5333.333333	3829.708431	1000.0	2250.0	5000.0	8500.0	10000.0	6.0	37.333333	6.947422

For each numerical variable (age and income), several statistics were calculated. Sometimes it can be clearer to visualize the transposed dataframe:

```
df.groupby('sex').describe().transpose()
```

	sex	F	M
income	count	5.000000	6.000000
	mean	8600.000000	5333.333333
	std	2880.972058	3829.708431
	min	5000.000000	1000.000000
	25%	7000.000000	2250.000000
	50%	8000.000000	5000.000000
	75%	11000.000000	8500.000000
	max	12000.000000	10000.000000
age	count	5.000000	6.000000
	mean	28.400000	37.333333
	std	6.268971	6.947422
	min	18.000000	28.000000
	25%	27.000000	32.750000
	50%	32.000000	37.000000
	75%	32.000000	43.500000
	max	33.000000	45.000000

This view allows us to compare different statistics of the two variables `age` and `income` for `M` and `F`.

Question 10

Considering the previously created DataFrame, use `groupby` to obtain the sum of the incomes of employees of a given company.

Crosstab

`Crosstabs` allow for the description of relationships between two or more **categorical** variables. Once a pair of categorical variables is specified, the rows and columns of the crosstab (also known as a "contingency table") independently enumerate all unique values of the two categorical variables, so that each cell of the crosstab identifies a specific pair of values. Within the cells, the numbers of elements for which the two categorical variables take on a specific pair of values are then reported.

Suppose we want to study the relationships between `company` and `sex`:

```
pd.crosstab(df['sex'],df['company'])
```

company	CDX	FLZ	PTX
sex			
F	1	3	1
M	3	1	2

The table above tells us, for example, that in company CDX, 1 subject is female, while 3 subjects are male. Similarly, one subject from company FLZ is male, while three subjects are female. It is possible to obtain frequencies instead of counts, by using `normalize=True`:

```
pd.crosstab(df['sex'],df['company'], normalize=True)
```

company	CDX	FLZ	PTX
sex			
F	0.090909	0.272727	0.090909
M	0.272727	0.090909	0.181818

Alternatively, we can normalize the table only by rows or only by columns by specifying `normalize='index'` or `normalize='columns'`:

```
pd.crosstab(df['sex'],df['company'], normalize='index')
```

company	CDX	FLZ	PTX
sex			
F	0.2	0.600000	0.200000
M	0.5	0.166667	0.333333

This table shows the percentages of people working in the three different companies for each gender, e.g., “20% of women work at CDX”. Similarly, we can normalize by columns as follows:

```
pd.crosstab(df['sex'],df['company'], normalize='columns')
```

company	CDX	FLZ	PTX
sex			
F	0.25	0.75	0.333333
M	0.75	0.25	0.666667

This table shows the percentages of men and women working in each company, e.g., “25% of CDX workers are women”.

If we want to study the relationships between more than two categorical variables, we can specify a list of columns when building the crosstab:

```
pd.crosstab(df['sex'],[df['age'],df['company']])
```

age	18	27	28	32		33	35	39	45	
company	CDX	FLZ	PTX	CDX	FLZ	PTX	CDX	FLZ	CDX	PTX
sex										
F	1	1	0	0	2	1	0	0	0	0
M	0	0	1	1	0	0	1	1	1	1

Each cell of the crosstab above counts the number of observations reporting a specific triplet of values. For example, 1 subject is male, works for PTX, and is 28 years old, while 2 subjects are female, work for FLZ, and are 32 years old.

In addition to reporting counts and frequencies, a crosstab allows for the calculation of statistics for third variables considered non-categorical. Suppose we want to know the average age of people of a given sex working for a given company. We can build a crosstab by specifying a new variable (age) for `values`. Since some aggregate value needs to be calculated for this variable, we also need to specify `aggfunc`, for example, `mean` (to calculate the average of the relevant values):

```
pd.crosstab(df['sex'],df['company'], values=df['age'], aggfunc='mean')
```

company	CDX	FLZ	PTX
sex			
F	18.000000	30.333333	33.0
M	37.333333	39.000000	36.5

The table tells us that the average age of male employees for CDX is 37.33 years.

Question 11

Construct a crosstab that, for each company, reports the number of employees of a given age.

“Explicit” Manipulation of DataFrames

In some cases, it can be useful to treat DataFrames “explicitly” as matrices of values. Consider, for example, the following DataFrame:

```
df123 = pd.DataFrame({'Category':[1,2,3], 'NumberOfElements':[3,1,2]})
df123
```

	Category	NumberOfElements
0	1	3
1	2	1
2	3	2

Suppose we want to build a new DataFrame that, for each row of the `df123` DataFrame, contains exactly "NumberOfElements" rows with the value of "NumberOfElements" equal to one. We essentially want to "expand" the DataFrame above as follows:

```
df123 = pd.DataFrame({'Category':[1,1,1,2,3,3], 'NumberOfElements':[1,1,1,1,1,1]})
df123
```

	Category	NumberOfElements
0	1	1
1	1	1
2	1	1
3	2	1
4	3	1
5	3	1

To do this automatically, we can treat the DataFrame "more explicitly" as a matrix of values by iterating through its rows. The new DataFrame will first be built as a list of Series (the rows of the DataFrame) and then transformed into a DataFrame:

```
newdat = []
for i, row in df123.iterrows(): # iterrows allows iterating over the rows of a
        DataFrame
    for j in range(row['NumberOfElements']):
        newrow = row.copy()
        newrow['NumberOfElements']=1
        newdat.append(newrow)
pd.DataFrame(newdat)
```

	Category	NumberOfElements
0	1	1
1	1	1
2	1	1
3	2	1
4	3	1
5	3	1

```
pd.DataFrame({'Category':[1,2,3], 'NumberOfElements':[3,5,3], 'CheckedElements':
        [1,2,1]})
```

	Category	NumberOfElements	CheckedElements
0	1	3	1
1	2	5	2
2	3	3	1

DataFrame Concatenation

DataFrames can be concatenated by rows as shown below:

```
d1 = pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
d2 = pd.DataFrame({'A':[92,8],'B':[44,-2],'C':[0,2]})
print(d1)
print(d2)
```

```
   A  B   C
0  1  3   0
1  2  2   2
2  3  6  -1
3  4  7  12
    A   B  C
0  92  44  0
1   8  -2  2
```

```
pd.concat([d1,d2])
```

	A	B	C
0	1	3	0
1	2	2	2
2	3	6	-1
3	4	7	12
0	92	44	0
1	8	-2	2

If the columns are not compatible, `NaN` will appear:

```
d1 = pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
d2 = pd.DataFrame({'A':[92,8],'D':[0,2]})

pd.concat([d1,d2])
```

	A	B	C	D
0	1	3.0	0.0	NaN
1	2	2.0	2.0	NaN
2	3	6.0	-1.0	NaN
3	4	7.0	12.0	NaN
0	92	NaN	NaN	0.0
1	8	NaN	NaN	2.0

Concatenation can also occur by columns:

```
d1 = pd.DataFrame({'A':[1,2],'B':[3,7],'C':[-1,12]})
d2 = pd.DataFrame({'A':[92,8],'D':[0,2]})

pd.concat([d1,d2], axis=1)
```

	A	B	C	A	D
0	1	3	-1	92	0
1	2	7	12	8	2

DataFrame Merge

Two DataFrames can be merged using a `merge` operation similar to what happens in databases. Let's consider two example dataframes:

```
a = pd.DataFrame({'key': ['k0', 'k1', 'k2'],
                  'a': ['v1', 'v2', 'v3'],
                  'b': ['v4', 'v5', 'v6'],
                  'c': ['v7', 'v8', 'v9']})

a
```

	key	a	b	c
0	k0	v1	v4	v7
1	k1	v2	v5	v8
2	k2	v3	v6	v9

```
b = pd.DataFrame({'key': ['k0', 'k1', 'k2'],
                  'd': ['v9', 'v11', 'v12'],
                  'e': ['v13', 'v4', 'v5']})
b
```

	key	d	e
0	k0	v9	v13
1	k1	v11	v4
2	k2	v12	v5

The two dataframes have one column in common that acts as a key and other columns that act as values. The merge operation allows to combine the dataframes based on the keys:

```
pd.merge(a,b,on='key')
```

	key	a	b	c	d	e
0	k0	v1	v4	v7	v9	v13
1	k1	v2	v5	v8	v11	v4
2	k2	v3	v6	v9	v12	v5

DataFrame Joins

`join` operations can be applied to combine dataframes with compatible indices:

```
a = pd.DataFrame({'a': ['v1', 'v2', 'v3'],
                  'b': ['v4', 'v5', 'v6'],
                  'c': ['v7', 'v8', 'v9']},
                  index = ['k0', 'k1', 'k2'])

a
```

	a	b	c
k0	v1	v4	v7
k1	v2	v5	v8
k2	v3	v6	v9

```
b = pd.DataFrame({'d': ['v9', 'v11', 'v12'],
                  'e': ['v13', 'v4', 'v5']},
                  index = ['k0', 'k3', 'k2'],)
b
```

	d	e
k0	v9	v13
k3	v11	v4
k2	v12	v5

```
a.join(b)
```

	a	b	c	d	e
k0	v1	v4	v7	v9	v13
k1	v2	v5	v8	NaN	NaN
k2	v3	v6	v9	v12	v5

As can be seen, the join operation only occurs by using the common indexes.

Input/Output

Pandas provides several functions for reading and writing data. We will specifically look at the functions for reading and writing `csv` files. It is possible to read csv files (https://it.wikipedia.org/wiki/Comma-separated_values) using the `pd.read_csv` function. Reading can be done from local files by passing the relative path or from URLs:

```
data=pd.read_csv(DATA + "students.csv")
data.head()
```

	admit	gre	gpa	prestige
0	0	380	3.61	3
1	1	660	3.67	3
2	1	800	4.00	1
3	1	640	3.19	4
4	0	520	2.93	4

Likewise, you can write a DataFrame to a CSV file as follows:

```
data.to_csv('file.csv')
```

Since indices may not be sequential in a DataFrame, Pandas inserts them into the csv as an “unnamed column”. For example, the first few lines of the file we just wrote (file.csv) are as follows:

```
,admit,gre,gpa,prestige
0,0,380,3.61,3
1,1,660,3.67,3
2,1,800,4.0,1
3,1,640,3.19,4
4,0,520,2.93,4
```

As you can see, the first column contains the index values, but it has no name. This can cause problems when we reload the DataFrame:

```
data=pd.read_csv('file.csv')
data.head()
```

	Unnamed: 0	admit	gre	gpa	prestige
0	0	0	380	3.61	3
1	1	1	660	3.67	3
2	2	1	800	4.00	1
3	3	1	640	3.19	4
4	4	0	520	2.93	4

We can solve the problem in several ways:

- Eliminate the “Unnamed: 0” column from the newly loaded DataFrame (only if the indices are sequential);
- Specify to use the “Unnamed: 0” column as the index column during loading;
- Save the DataFrame without indices (only if the indices are sequential).

Let’s look at the three cases:

```
data=pd.read_csv('file.csv')
data.drop('Unnamed: 0', axis=1).head()
```

	admit	gre	gpa	prestige
0	0	380	3.61	3
1	1	660	3.67	3
2	1	800	4.00	1
3	1	640	3.19	4
4	0	520	2.93	4

```
data=pd.read_csv('file.csv', index_col='Unnamed: 0')
data.head()
```

	admit	gre	gpa	prestige
0	0	380	3.61	3
1	1	660	3.67	3
2	1	800	4.00	1
3	1	640	3.19	4
4	0	520	2.93	4

```
data.to_csv('file.csv', index=False)
pd.read_csv('file.csv').head()
```

	admit	gre	gpa	prestige
0	0	380	3.61	3
1	1	660	3.67	3
2	1	800	4.00	1
3	1	640	3.19	4
4	0	520	2.93	4

Data Manipulation Example

Often, data needs to be properly processed to be analyzed. Let's look at an example of how to combine the tools discussed above to manipulate a real dataset.

We will consider the dataset available at this URL: https://www.kaggle.com/lava18/google-play-store-apps. A copy of the dataset is served by this site for educational purposes, at `https://antoninofurnari.github.io/fad-2627/data/googleplaystore.csv`, and that is the copy the code below reads. Let's load the dataset and display the first few rows to ensure it has been loaded correctly:

```
data = pd.read_csv(DATA + "googleplaystore.csv")
data.head()
```

	App	Category	Rating	Reviews	Size	Installs	Type	Price	Content Rating	Genres	L
0	Photo Editor & Candy Camera & Grid & ScrapBook	ART_AND_DESIGN	4.1	159	19M	10,000+	Free	0	Everyone	Art & Design	J 7
1	Coloring book moana	ART_AND_DESIGN	3.9	967	14M	500,000+	Free	0	Everyone	Art & Design;Pretend Play	J 1 2
2	U Launcher Lite – FREE Live Cool Themes, Hide ...	ART_AND_DESIGN	4.7	87510	8.7M	5,000,000+	Free	0	Everyone	Art & Design	A 1
3	Sketch - Draw & Paint	ART_AND_DESIGN	4.5	215644	25M	50,000,000+	Free	0	Teen	Art & Design	J 2
4	Pixel Draw - Number Art Coloring Book	ART_AND_DESIGN	4.3	967	2.8M	100,000+	Free	0	Everyone	Art & Design;Creativity	J 2

Display the dataset properties:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   App             10841 non-null  object 
 1   Category        10841 non-null  object 
 2   Rating          9367 non-null   float64
 3   Reviews         10841 non-null  object 
 4   Size            10841 non-null  object 
 5   Installs        10841 non-null  object 
 6   Type            10840 non-null  object 
 7   Price           10841 non-null  object 
 8   Content Rating  10840 non-null  object 
 9   Genres          10841 non-null  object 
 10  Last Updated    10841 non-null  object 
 11  Current Ver     10833 non-null  object 
 12  Android Ver     10838 non-null  object 
dtypes: float64(1), object(12)
memory usage: 1.1+ MB
```

The dataset contains 10841 observations and 13 variables. We observe that many of the variables (except "Rating") are of type "object", even though they represent numerical values (e.g., Reviews, Size, Installs, and Price). Observing the first rows of the DataFrame displayed above, we can deduce that:

- Size is not represented as a numerical value as it contains the unit of measurement "M";
- Installs is actually a categorical variable (it has a "+" at the end, thus indicating the "more than xxxx installations" class);

There are no apparent reasons why Reviews and Rating were not interpreted as numbers. Let's build a filter function that identifies whether a value is convertible to a given type or not:

```python
def cannot_convert(x, t=float):
    try:
        t(x)
        return False
    except:
        return True
print(cannot_convert('12'))
print(cannot_convert('12f'))
```

```
False
True
```

We apply the filter to the Review column to display values that cannot be converted:

```python
list(filter(cannot_convert,data['Reviews']))
```

```
['3.0M']
```

We can replace this value with its full version using the `replace` method:

```
data['Reviews']=data['Reviews'].replace({'3.0M':3000000})
```

At this point, we can convert the column's value types to integers:

```
data['Reviews']=data['Reviews'].astype(int)
```

Let's visualize the DataFrame information again:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   App             10841 non-null  object
 1   Category        10841 non-null  object
 2   Rating          9367 non-null   float64
 3   Reviews         10841 non-null  int64
 4   Size            10841 non-null  object
 5   Installs        10841 non-null  object
 6   Type            10840 non-null  object
 7   Price           10841 non-null  object
 8   Content Rating  10840 non-null  object
 9   Genres          10841 non-null  object
 10  Last Updated    10841 non-null  object
 11  Current Ver     10833 non-null  object
 12  Android Ver     10838 non-null  object
dtypes: float64(1), int64(1), object(11)
memory usage: 1.1+ MB
```

Reviews is now a whole. We perform a similar analysis on the Price variable:

```
list(filter(cannot_convert, data['Price']))[:10] #display only the first 10 elements
```

```
['$4.99',
 '$4.99',
 '$4.99',
 '$4.99',
 '$3.99',
 '$3.99',
 '$6.99',
 '$1.49',
 '$2.99',
 '$3.99']
```

We can convert strings to float by removing the initial dollar using `apply`:

```
def strip_dollar(x):
    if x[0]=='$':
        return x[1:]
    else:
        return x
data['Price']=data['Price'].apply(strip_dollar)
```

Let's see if there are any values that cannot be converted:

```
list(filter(cannot_convert, data['Price']))
```

```
['Everyone']
```

Since we don't know how to interpret the value "Everyone", let's replace it with a NaN:

```
data['Price']=data['Price'].replace({'Everyone':np.nan})
```

Now we can proceed with the conversion:

```
data['Price'] = data['Price'].astype(float)
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   App             10841 non-null  object 
 1   Category        10841 non-null  object 
 2   Rating          9367 non-null   float64
 3   Reviews         10841 non-null  int64  
 4   Size            10841 non-null  object 
 5   Installs        10841 non-null  object 
 6   Type            10840 non-null  object 
 7   Price           10840 non-null  float64
 8   Content Rating  10840 non-null  object 
 9   Genres          10841 non-null  object 
 10  Last Updated    10841 non-null  object 
 11  Current Ver     10833 non-null  object 
 12  Android Ver     10838 non-null  object 
dtypes: float64(2), int64(1), object(10)
memory usage: 1.1+ MB
```

We also modify “Size” by removing the final “M”:

```
data['Size']=data['Size'].apply(lambda x : x[:-1])
```

We verify the existence of non-convertible values:

```
list(filter(cannot_convert,data['Size']))[:10]
```

```
['Varies with devic',
 'Varies with devic',
 'Varies with devic',
 'Varies with devic',
 'Varies with devic',
 'Varies with devic',
 'Varies with devic',
 'Varies with devic',
 'Varies with devic',
 'Varies with devic']
```

We replace these values with NaN:

```
data['Size']=data['Size'].replace({'Varies with devic':np.nan})
```

Let’s re-check for values that cannot be converted:

```
list(filter(cannot_convert, data['Size']))
```

```
['1,000']
```

We can remove the comma used to indicate thousands and convert to float:

```
data['Size']=data['Size'].apply(lambda x: float(str(x).replace(',','')))
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   App             10841 non-null  object
 1   Category        10841 non-null  object
 2   Rating          9367 non-null   float64
 3   Reviews         10841 non-null  int64
 4   Size            9146 non-null   float64
 5   Installs        10841 non-null  object
 6   Type            10840 non-null  object
 7   Price           10840 non-null  float64
 8   Content Rating  10840 non-null  object
 9   Genres          10841 non-null  object
 10  Last Updated    10841 non-null  object
 11  Current Ver     10833 non-null  object
 12  Android Ver     10838 non-null  object
dtypes: float64(3), int64(1), object(9)
memory usage: 1.1+ MB
```

Let's evaluate whether it's appropriate to also transform "Installs" into a numerical value. Let's see how many unique values of "Installs" there are in the dataset:

```
data['Installs'].nunique()
```

```
22
```

We only have 22 values, which compared to the 10841 observations, indicate that "Installs" is a highly quantized variable, so it probably makes sense to consider it as categorical, rather than a numerical value.

The DataFrame contains `NaN` values. Depending on the case, it can make sense to keep them or remove them. Since there are many data points, we can remove them with `dropna`:

```
data=data.dropna()
```

We can therefore start exploring the data with the tools we've seen. Let's visualize, for example, the average values of the numerical variables by category:

```
data.groupby('Category').mean(numeric_only=True)
```

	Rating	Reviews	Size	Price
Category				
ART_AND_DESIGN	4.381034	1.874517e+04	12.939655	0.102931
AUTO_AND_VEHICLES	4.147619	1.575057e+04	24.728571	0.000000
BEAUTY	4.291892	5.020243e+03	15.513514	0.000000
BOOKS_AND_REFERENCE	4.320139	2.815291e+04	23.543750	0.145069
BUSINESS	4.119919	2.497765e+04	26.217480	0.249593
COMICS	4.130612	1.254822e+04	34.626531	0.000000
COMMUNICATION	4.102844	5.549962e+05	60.765403	0.197773
DATING	3.957803	2.254489e+04	18.312717	0.086590
EDUCATION	4.387273	6.435159e+04	30.588182	0.163273
ENTERTAINMENT	4.146667	1.621530e+05	21.853333	0.033222
EVENTS	4.478947	3.321605e+03	23.213158	0.000000
FAMILY	4.190347	1.801297e+05	37.459963	1.390074
FINANCE	4.112030	3.903023e+04	31.249624	9.172444
FOOD_AND_DRINK	4.097619	5.103793e+04	24.163095	0.059405
GAME	4.269507	1.386276e+06	46.827823	0.281704
HEALTH_AND_FITNESS	4.223767	4.519072e+04	29.658296	0.154350
HOUSE_AND_HOME	4.162500	2.935998e+04	17.505357	0.000000
LIBRARIES_AND_DEMO	4.204918	1.632359e+04	225.267213	0.000000
LIFESTYLE	4.093571	3.109027e+04	28.650714	6.976429
MAPS_AND_NAVIGATION	4.013684	3.858909e+04	26.282105	0.157474
MEDICAL	4.184259	4.392454e+03	49.643827	3.077901
NEWS_AND_MAGAZINES	4.143787	5.826802e+04	14.948521	0.023550
PARENTING	4.347727	1.999950e+04	21.579545	0.113409
PERSONALIZATION	4.323381	1.257211e+05	50.115827	0.427338
PHOTOGRAPHY	4.147034	3.260313e+05	18.970763	0.331992
PRODUCTIVITY	4.143830	1.854692e+05	38.899149	0.225362
SHOPPING	4.227933	2.624461e+05	33.397207	0.030615
SOCIAL	4.257062	1.830882e+05	24.317514	0.011186
SPORTS	4.204858	2.128411e+05	30.159109	0.324818
TOOLS	4.010742	1.663130e+05	47.929068	0.290047
TRAVEL_AND_LOCAL	4.043125	4.824554e+04	25.357500	0.165688
VIDEO_PLAYERS	4.025862	2.074172e+05	22.881897	0.008534

	Rating	Reviews	Size	Price
Category				
WEATHER	4.241176	7.639225e+04	24.805882	0.459608

Question 12

Display the average number of reviews per type (variable Type).

Exercises

Exercise 1

Define a 4x4 matrix, then:

- Print the first row of the matrix;
- Print the second column of the matrix;
- Sum the first and last columns of the matrix;
- Sum the elements along the main diagonal;
- Print the number of elements in the matrix.

Exercise 2

Generate an array of 100 numbers between 2 and 4 and calculate the sum of the elements whose squares have a value greater than 8.

Exercise 3

Generate the following matrix writing the smallest amount of code possible:

```
0 1 2
3 4 5
6 7 8
```

Repeat the exercise generating a 25×13 matrix of the same type.

Exercise 4

Write the code to obtain this diagram:

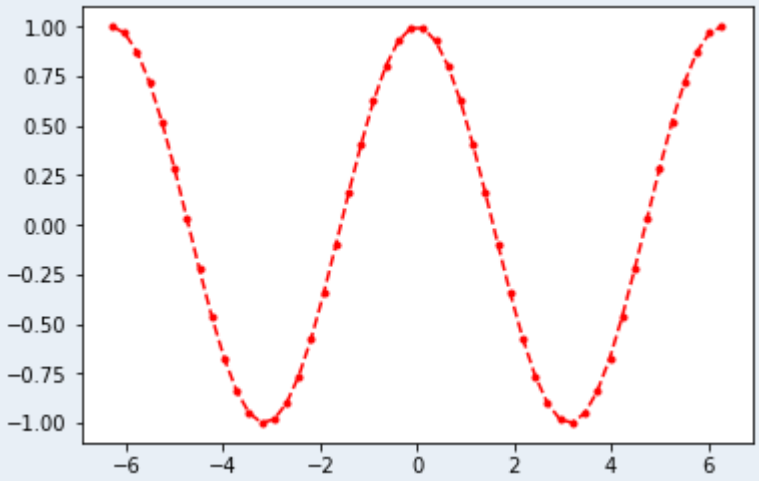


image.png

Exercise 5

Load the dataset available at the following URL: https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv so as to use the values of the "PassengerId" column as indices and apply the following transformations:

- Remove rows containing NaN values;
- Modify the DataFrame by replacing every occurrence of "male" with "M" and every occurrence of "female" with "F" in the "Sex" column;
- Insert a new column "Old" with a value of "1" if Age is greater than the average age of passengers and "0" otherwise;
- Convert the values in the "Age" column to integers.

Exercise 6

Consider the dataset loaded and modified in the previous exercise. Find the name of the passenger with ticket code "345779".

Exercise 7

What are the names of the 5 youngest passengers?

Exercise 8

Consider the dataset loaded and modified in the previous exercise. Select all observations relating to passengers between 20 and 30 years of age.

Exercise 9

Consider the dataset loaded and modified in the previous exercises. Calculate the average fare paid by female passengers in second class.

Exercise 10

Consider the dataset loaded and modified in the previous exercises. Construct a crosstab to answer the following questions:

- How many male passengers survived?
- How many female passengers did not survive?

Exercise 11

Consider the dataset loaded and modified in the previous exercises. Construct a crosstab that allows you to answer the following questions:

- How many male passengers survived in first class?
- How many female passengers survived in third class?
- How many male passengers survived in third class?

Exercise 12

Consider the following DataFrame:

```
pd.DataFrame({'Category':[1,2,3], 'NumberOfElements':[3,2,3],
        'CheckedElements':[1,2,1]})
```

Construct a new DataFrame that contains the same columns as the considered DataFrame and that for each row of it contains `NumberOfElements` new rows with category equal to `Category` of which `CheckedElements` with value of `CheckedElements` equal to one and the rest with value of `CheckedElements` equal to zero.

The result of the manipulation should be equal to the following dataset:

```
pd.DataFrame({'Category':[1,1,1,2,2,3,3,3],
              'NumberOfElements':[1,1,1,1,1,1,1,1],
              'CheckedElements':[1,0,0,1,1,1,0,0]})
```

References

- Pandas Documentation: https://pandas.pydata.org/pandas-docs/stable/
- Kaggle Website (where to find data to analyze): http://kaggle.com/
- Page from which to download various CSV datasets from the R library: https://vincentarelbundock.github.io/Rdatasets/datasets.html

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Key concepts, loading and inspecting the data

Chapter resources

Slides (PDF) Open in Colab Practice quiz

In this lecture, we will begin to see the main definition and foundational concepts in data analysis.

What is Data? An informal Definition

We will start by giving an informal definition of data. It is not by any means a complete definition, but it will suffice as a starting point to get an intuitive understanding of what data is:

> Data is a **set of values collected** with respect to some **variables** which describe a given **phenomenon**.

In the definition above, we mentioned some key concepts, which are discussed in the following sections.

Observations, Populations, Samples

We will see three fundamental concepts: observations, populations, samples.

Observations



When we deal with data, we actually deal with *multiple instances* of values associated to the same event or phenomenon. Examples:

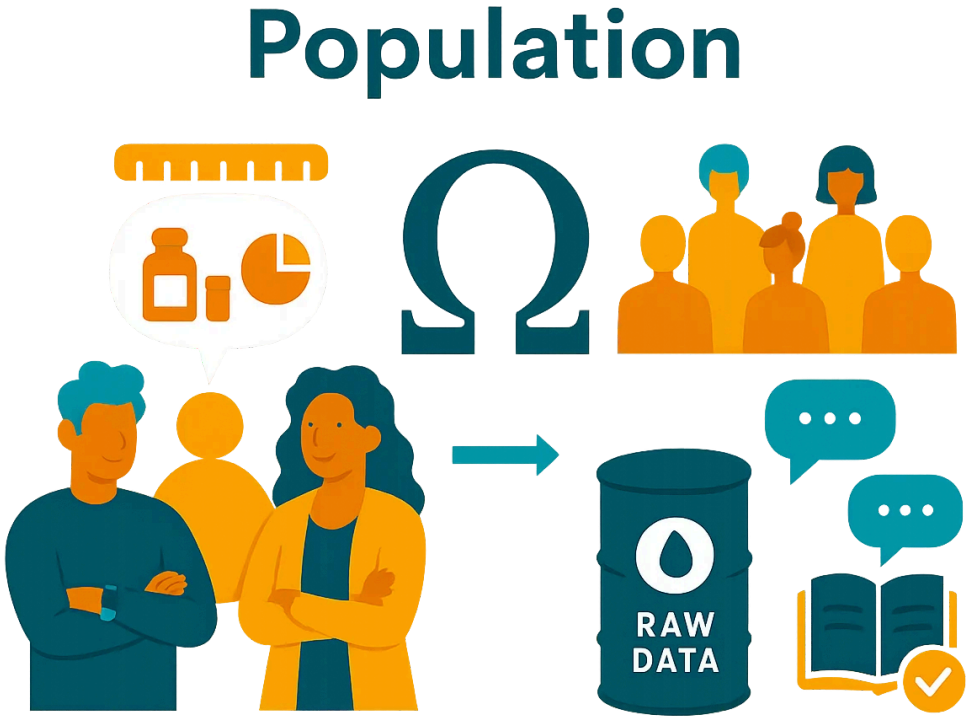
Example	Comment
We want to study how the heights of students change with the years	We need sets of recorded heights, not just one height
We want to study how a given drug affects the recovering from a disease	We need to record sets of values about drug assumption and recover, just one case will not be enough
We need to create a system which can generate text from a prompt	We need to record several examples of prompt-text generation, just one would not be enough to study how such generation should be made

We will call **observations, the units by which we measure data**. These could be persons, cars, animals, plants, etc. We often indicate an observation as

$$\omega$$

We should consider observation as **"abstract" entities**, not necessarily numerical observations. E.g., "let's consider a person ω".

Population



When we study a given phenomenon, we will be interested in a set of observations, which is called a "population". For instance:

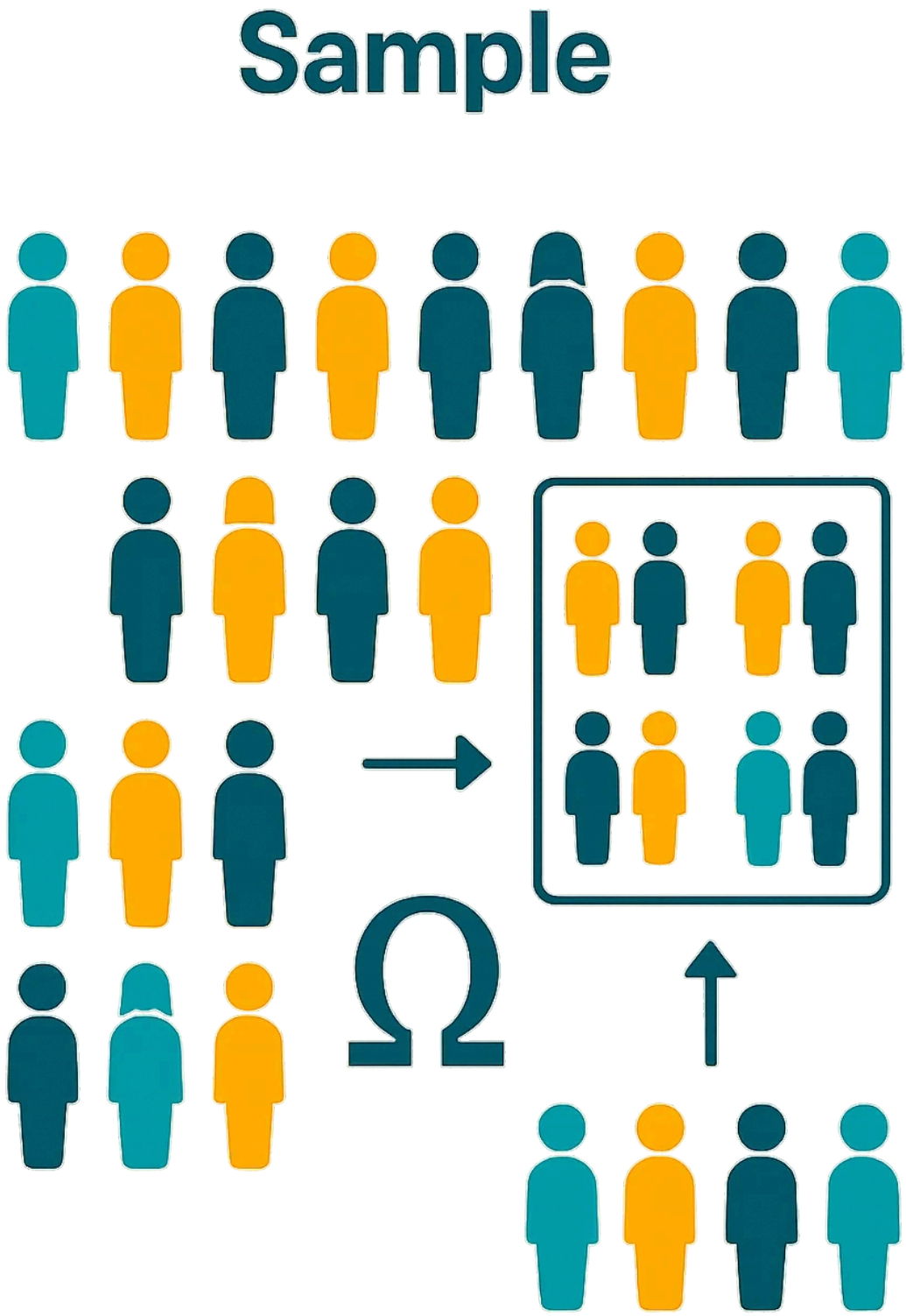
- if we want to study the distribution of heights of people in the world, we will need to look at the population of all people in the world.
- if we want to study the age of people attending a computer science course in Italy, then we need to look at the population of all students of computer science courses in Italy.

Note that **a population can sometimes be a theoretical concept** and identify sets of elements which are not even finite. E.g., *"all movies which will ever be filmed"*.

We can denote a population with the symbol Ω. Al our observations will be

$$\omega \in \Omega$$

Sample



In practice, working with population can be very hard, as it is not always possible to obtain

observations from those large sets. Intuitively, in practice, working on a large enough set of observations from a population could be good enough. We refer to a subset of a population as "a sample":

$$\{\omega^{(1)}, \omega^{(2)}, \ldots, \omega^{(n)}\} \subseteq \Omega$$

Example

We want to study how the height of people in the world changed in the years. Here:

- the **population** Ω is the set of people ever existed on earth,
- a **sample** $\{\omega^{(1)}, \ldots, \omega^{(n)}\} \subseteq \Omega$ is a subset of people for which we have some data (e.g., say we measured the heights of many people in most countries since the 16th century),
- an observation is a person $\omega^{(i)}$.

Variables (or Features)



We have identified our problem, defined a suitable population and identified a sample of observations. While observations are abstract concepts such as "a person" or "a student", we **usually want to capture specific features of such observations**, such as "the person's age" or "the student's height". We collect these features by means of **statistical variables**.

Statistical variables are concept similar to that of mathematical of programming variables, in that they can be seen as sorts of **"containers for the data"**.

We may also be interested in different features of an observation. For instance, for each person in a population, **we may want to record their age, gender, and height**. We can introduce a variable to capture each of these features. For instance, given observation ω, we may obtain "height = 180cm", "weight=80Kg", "gender=male".

Formally, we'll define a variable X as:

$$X : \Omega \rightarrow S$$

$$\omega \mapsto x$$

Where S is the set of possible values for variable X. The definition above specifies that a variable maps an abstract observation ω to some (possibly more concrete) value $x \in S$.

> Note: we use a capital letter X to denote the variable, while a lowercase letter x to represent the value assumed by the variable. For instance, we could write $X = x \in S$.

The concept of variable is often used interchangeably with the term **feature**, which is more used in the Machine Learning community and jargon. This is because a variable capture a specific "feature" or "characteristic" of each observation. For instance, the variable "height" capture the height feature of a person.

Example

Given the population of all people currently living in the world Ω, we define a variable H to collect the heights of the observed people ω:

$$H : \Omega \rightarrow \mathbb{R}$$

$$\omega \mapsto h$$

Given an observation $\omega^{(1)}$, we may obtain $H(\omega) = 180$. We often say that H assumes the value 180 and write: $H = 180$.

Quantitative and Qualitative Variables

Variables can be of two main kinds:

- **Qualitative**: represent properties that can be observed and cannot generally be measured with a numerical value (e.g., 'color', 'sex');
- **Quantitative**: represent properties that can be quantified with a number (e.g., 'height', 'weight', 'age').

Discrete and Continuous Variables

Variables can also be discrete or continues:

- **Discrete variables** can assume a finite number (or a countable infinite number) of possible values. Binary variables are a special case of discrete variables which can take only two possible values (0 and 1).
- **Continuous variable** assume a continuous, infinite number of values, which can be generally denoted with real numbers

Variables which take numerical values are generally called also “numeric” variables.

Scalar and Multi-Dimensional Variables

Variables can be:

- **scalar**, **uni-dimensional** or **uni-variate**: they assume real numbers (e.g., $X = 1$)
- **multi-dimensional** or **multi-variate**: they assume vector or matrix values, (e.g., $X = \binom{1}{3}$, or $X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$).

Examples

Let’s see some examples:

Discrete Scalar Variables

We want to assess if a coin is fair or not.

- We consider as population all possible tosses of that coin.
- An observation will be a specific tossing.
- A discrete scalar variable X may record the outcome of a given tossing. The set of possible values will be $S = \{head, \ tail\}$ (discrete values). The variable is scalar as it will contain a single value.
- If we toss a coin, we may get $X = tail$.

Continuous Scalar Variables

We want to study the heights in centimeters of students in this class.

- Our population is the set of all students in this class.
- We can use a continuous scalar variable X to record the heights of the students. In this case, we can choose $S = \mathbb{R}_+$.
- If we pick a student, we may get $X = 175$.

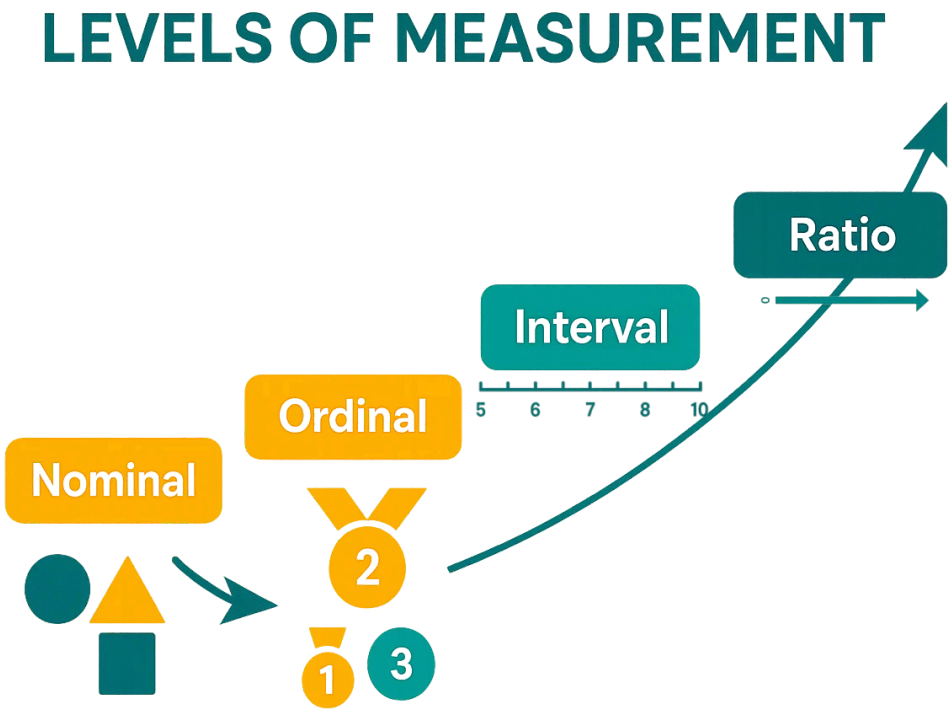
Continuous Multi-Dimensional Variables

We want to study the positions of all cars in the world.

- Our population is the set of all cars in the world.

- We could use the variable X to denote the latitude and longitude coordinates of a car in the world. The set of possible values may be $S = \mathbb{R}^2$.
- Once we pick a car, we may have $X = \begin{pmatrix} 37 \\ 15 \end{pmatrix}$.

Scales



When working with variables, it's important to understand whether their values can be meaningfully ordered. This is captured by the concept of **measurement scales**:

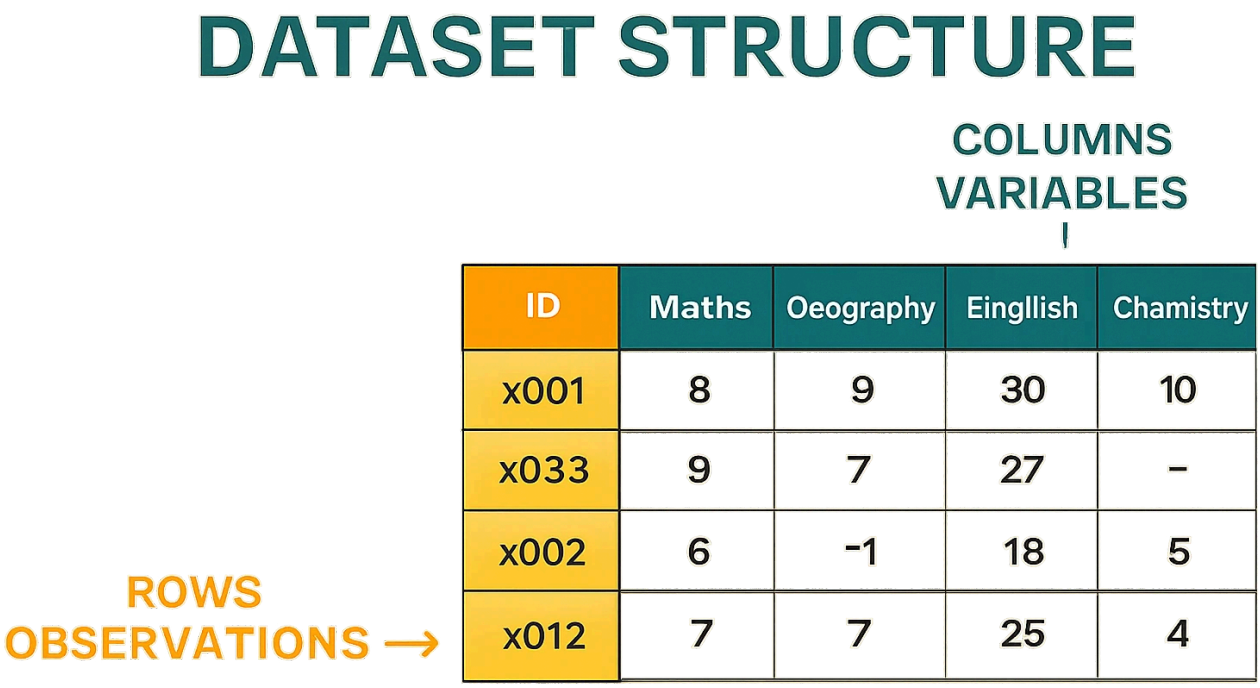
- **Nominal scale**: values **cannot be ordered**.
  Examples: gender, eye color, car manufacturer. We cannot say that "green > brown".

- **Ordinal scale**: values **can be ordered**, but **differences are not meaningful**.
  Example: levels of expertise (novice, amateur, intermediate, expert). We can say "novice < amateur", but "amateur - novice" is usually different than "intermediate - amateur".

- **Interval scale**: values **can be ordered**, and **differences are meaningful**,
  but there is **no true zero denoting the absence of the quantity**.
  Example: temperature in Celsius or Fahrenheit. We can say 25° - 15° = 32°-22°, but 0° does not denote that "there is no temperature".

- **Ratio scale**: values **can be ordered**, **differences are meaningful**, and there **is a true zero**.
  Examples: height, weight, age, income. 0cm or 0Kg denotes the absence of the quantity, plus differences are meaningful and ordering is possible.

Data Sets, Design Matrix, and Missing Values



A set of data related to a specific phenomenon is called a "dataset". Datasets are usually stored in tables in which **columns represent the different variables** and **rows represent the different observations**. If you have ever had a look at a spreadsheet, you probably already saw an example of a dataset!

Let's consider the following example of a dataset of marks of 5 subjects obtained by three students:

ID	Maths	Geography	English	Physics	Chemistry
x001	8	9	30	8	10
x038	9	7	27	6	
x002	6	-1	18	5	6
x012	7	7	25	4	10
x042	10	10	30	10	10

We can find the different elements that we discussed about in this table:

- **Observations**: each row of the table is an observation;
- **Variables**: each column of a table is a different table, mapping an observation to a numerical value. In the example, above the variable ID is qualitative, while all others are quantitative, discrete, and ratio scale (they have a true zero);
- **Features**: same as variables, each column is a "feature" or a characteristic of each observation;

- **Samples and Populations**: the dataset can be seen as a sample (a subset of students) of a wider population (e.g., all students in the school);

This kind of matrix is also called **design matrix**.

Note that, due to the way the data was collected, one value is not there. We call these values **missing data**. These may be due to errors in the way we collected and stored our data or due to specific phenomena. For instance, in this case, the student may have not taken yet the test for that specific subject. We will see later how to deal with **missing data**.

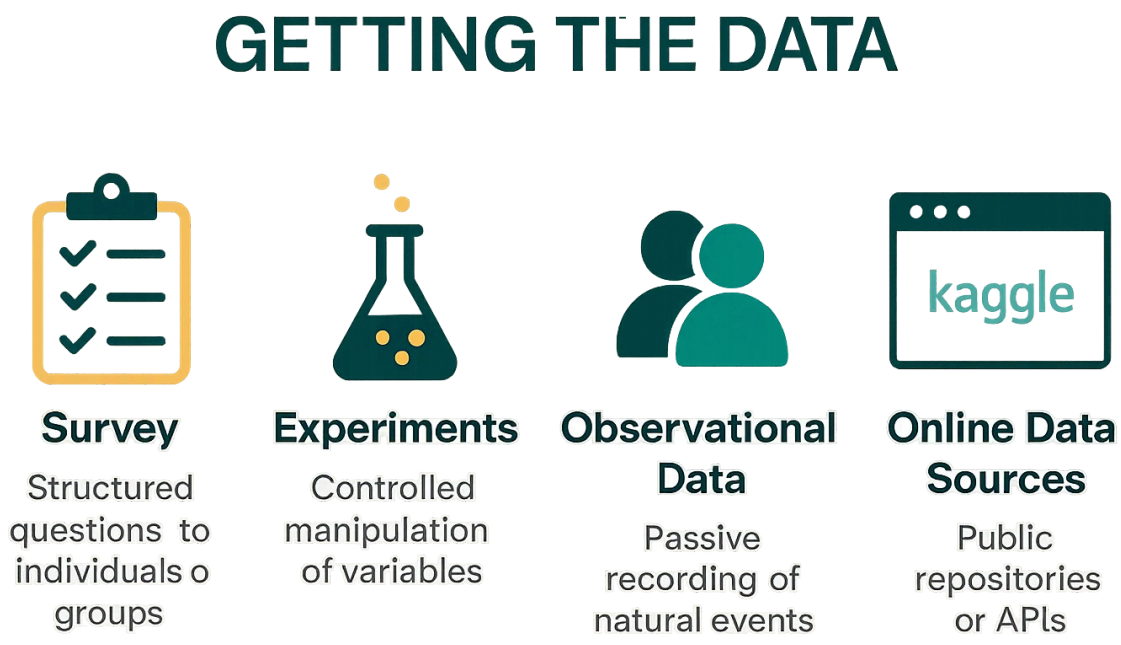
Before moving on, think for a moment what you could do with a dataset like this (maybe imagine a larger one):

- You could take the average of all marks obtained by a student (average by rows) to get a ranking of the students. This could be useful to understand which students may need help.
- You could compute the average of the votes obtained in each course (average by column) to identify the subjects which are "more difficult" for the students than others.
- You could group the courses into humanity-based and science-based to identify which students excel in each field.

The examples above are all (very simple) examples of data analysis. As you can see, even with a simple dataset like this and no knowledge of complex notions of data analysis, we can already do a lot of analysis.

The "table" structure containing a dataset is often called a **data matrix** or a **design matrix**. In this format, each column of the matrix represents a variable, which is often also referred to as a **feature**, while each row is a different observation.

Getting the Data



Data collection is the essential first step in any data analysis workflow. It involves gathering observations that will serve as the foundation for modeling, visualization, and interpretation. The method you choose depends on your goals, the type of data needed, and the resources available.

Surveys

Surveys are structured tools used to collect information from individuals or groups. They typically involve asking predefined questions and are ideal for capturing opinions, preferences, demographics, or behavioral patterns.

Common formats:

- Online forms
- Paper questionnaires
- Phone or in-person interviews

Example:
A university wants to understand students' satisfaction with online courses. It distributes a digital survey asking students to rate their experience across different dimensions (e.g., content clarity, platform usability, instructor engagement).

Key features:

- Standardized questions
- Scalable across large populations

- Quantifiable and analyzable responses

Experiments

Experiments allow researchers to collect data in a **controlled** environment, often to test causal relationships between variables. By manipulating one or more independent variables and observing the outcome, experiments can provide strong evidence for cause-and-effect.

Most common type:

- **Randomized Controlled Trials (RCTs)**

Example:
To evaluate the effectiveness of a new drug, researchers randomly assign participants to two groups: one receives the drug, the other a placebo. Recovery rates are compared to assess the drug's impact.

Key features:

- Controlled manipulation of variables
- Random assignment to reduce bias
- Replicable design for hypothesis testing

Observational Data

Observational methods involve recording data as it naturally occurs, without intervention. This approach is useful when experiments are impractical, unethical, or too costly.

Example:
A public health researcher studies the impact of air pollution on respiratory health by analyzing hospital admission records and pollution levels across different cities.

Key features:

- No manipulation of variables
- Captures real-world behavior
- Suitable for exploratory and correlational analysis

Online Data Sources

In modern data science, many projects begin by retrieving data from online platforms and public repositories. These sources offer ready-to-use datasets for exploration, benchmarking, and model development.

Popular platforms:

- Kaggle: competitions, datasets, notebooks
- UCI Machine Learning Repository
- Google Dataset Search

- APIs (e.g., Twitter, OpenWeather, Spotify)
- Web scraping tools

Example:
A student downloads a Kaggle dataset of movie reviews to train a sentiment analysis model using natural language processing.

Key features:

- Fast access to diverse datasets
- Often pre-cleaned and well-documented
- Ideal for prototyping and learning

Data Loading in Python

We will now put our hands on a real dataset. We will consider the `Titanic` dataset, which is a popular dataset used in data science education and competitions. It contains information about passengers aboard the Titanic and whether they survived the disaster.

This dataset is particularly useful for practicing data cleaning, exploratory data analysis, and predictive modeling. It includes both **qualitative** and **quantitative** variables, and offers a rich context for discussing missing data, bias, and causal inference.

In the `Titanic` dataset, each row represents a passenger, and each column represents a variable describing that passenger. Common variables include:

- `PassengerId`: Unique identifier for each passenger
- `Survived`: Whether the passenger survived (0 = No, 1 = Yes)
- `Pclass`: Ticket class (1st, 2nd, or 3rd)
- `Name`: Full name of the passenger
- `Sex`: Gender
- `Age`: Age in years
- `SibSp`: Number of siblings/spouses aboard
- `Parch`: Number of parents/children aboard
- `Ticket`: Ticket number
- `Fare`: Price paid for the ticket
- `Cabin`: Cabin number (often missing)
- `Embarked`: Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

The dataset is a good didactic example. Indeed:

- It mixes **categorical** and **numerical** data
- It contains **missing values**, which are common in real-world datasets
- It allows us to explore **correlations** and build **classification models**
- It's historically grounded, making it easier to interpret and explain

In the next steps, we will:

1. **Load the dataset** and inspect its structure
2. **Clean the data** by handling missing values and inconsistencies
3. **Explore the variables** using summary statistics and visualizations
4. **Build a simple model** to predict survival based on selected features

We will start by loading the dataset using the Pandas library in Python. We will use a csv version of the dataset, originally published at `https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv` and served by this site at `https://antoninofurnari.github.io/fad-2627/data/titanic.csv` — which is the copy the code below actually reads, so the same code runs here, on a local clone and on Colab.

Before loading the data with Python, let's briefly point our browser to https://antoninofurnari.github.io/fad-2627/data/titanic.csv to see the content.

The data is in CSV format, meaning **Comma-Separated Values**. This is a plain text format where each line represents a row in the dataset, and each value (or cell) is separated by a comma.

CSV files are widely used because they are:

- Simple to read and write
- Compatible with spreadsheet software like Excel or Google Sheets
- Easily imported into programming environments like Python (e.g., using `pandas.read_csv()`)

Here's a small example of how a CSV file might look (first line of the dataset):

```
PassengerId,Survived,Pclass,Name,Sex,Age,SibSp,Parch,Ticket,Fare,Cabin,Embarked
1,0,3,"Braund, Mr. Owen Harris",male,22,1,0,A/5 21171,7.25,,S
2,1,1,"Cumings, Mrs. John Bradley (Florence Briggs Thayer)",female,38,1,0,PC
17599,71.2833,C85,C
3,1,3,"Heikkinen, Miss. Laina",female,26,0,0,STON/O2. 3101282,7.925,,S
4,1,1,"Futrelle, Mrs. Jacques Heath (Lily May Peel)",female,35,1,0,113803,53.1,C123,S
5,0,3,"Allen, Mr. William Henry",male,35,0,0,373450,8.05,,S
```

Note that Pandas can also read other formats, such as tab separated values file and Excel files.

Let's load and visualize the data:

```
import pandas as pd
titanic = pd.read_csv(DATA + "titanic.csv",
                      index_col='PassengerId')
titanic
```

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId											
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...
887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

891 rows × 11 columns

As you can see, the visualization has been truncated, removing some rows, as the dataset is big. Another way to visualize such big dataset is to use the `.head()` method, optionally specifying a number of rows to be shown (5 by default):

```
titanic.head()
```

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId											
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

The dataset shows different observations related to the passengers of the Titanic. We can see each row of this matrix as a multi-dimensional variable (a vector) summarizing the main properties of the observations. Indeed, in the example above, each passenger is a different observation, while the different column are the different **features**.

The `NaN` values denote missing data.

With this data in front of us, let's try to answer these questions:

- How many observations are there?
- How many variables are there?
- Which variables are qualitative?
- Which variables are quantitative?
- Which variables are discrete?
- Which variables are continuous?
- What are the scales of each variable?

To make our life easier answering those questions, we can use the `.info()` method of the Pandas dataframe:

```
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 891 entries, 1 to 891
Data columns (total 11 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Survived  891 non-null    int64
 1   Pclass    891 non-null    int64
 2   Name      891 non-null    object
 3   Sex       891 non-null    object
 4   Age       714 non-null    float64
 5   SibSp     891 non-null    int64
 6   Parch     891 non-null    int64
 7   Ticket    891 non-null    object
 8   Fare      891 non-null    float64
 9   Cabin     204 non-null    object
 10  Embarked  889 non-null    object
dtypes: float64(2), int64(4), object(5)
memory usage: 83.5+ KB
```

This table shows some features of each column, including the name, the number of non-null (non missing) data and the type of the column.

In this dataset, each observation is identified by the `PassengerId` variable, which is a qualitative discrete variable. This act as an "index" of the dataset, which is also called a "DataFrame" in Pandas. We can verify this as follows:

```
titanic.index
```

```
Index([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,
       ...
       882, 883, 884, 885, 886, 887, 888, 889, 890, 891],
      dtype='int64', name='PassengerId', length=891)
```

We can select a single observation using this notation:

```
titanic.loc[101]
```

```
Survived                          0
Pclass                            3
Name        Petranec, Miss. Matilda
Sex                          female
Age                            28.0
SibSp                             0
Parch                             0
Ticket                       349245
Fare                         7.8958
Cabin                           NaN
Embarked                          S
Name: 101, dtype: object
```

A row of a DataFrame is a Pandas Series, which is a one-dimensional array-like object. Each element of the Series corresponds to a column in the DataFrame, and the index of the Series corresponds to the column names.

As we can see, the observation contains different "features" or variable values, such as the Name, Sex, Age etc. of the passenger. The property "Name" of the Series denotes the value of the index. We can access a given feature as follows:

```
titanic.loc[101]['Name']
```

```
'Petranec, Miss. Matilda'
```

Note that we can discard the index and just take the i-th element in order with the following notation:

```
titanic.iloc[101]
```

```
Survived                                    0
Pclass                                      3
Name        Petroff, Mr. Pastcho ("Pentcho")
Sex                                     male
Age                                      NaN
SibSp                                      0
Parch                                      0
Ticket                                349215
Fare                                  7.8958
Cabin                                    NaN
Embarked                                   S
Name: 102, dtype: object
```

As we can see, this returned a different observation (the one with id/Name equal to 102).

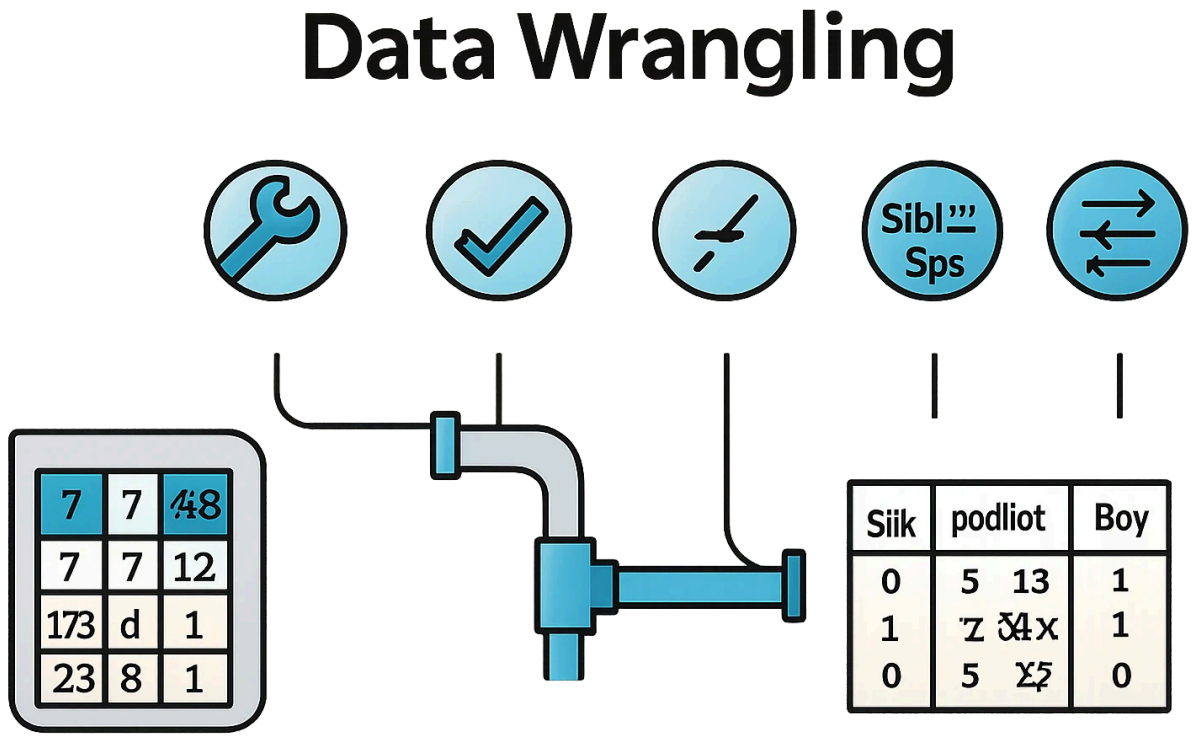
We can extract a given column from the dataset using this notation:

```
titanic['Age']
```

```
PassengerId
1      22.0
2      38.0
3      26.0
4      35.0
5      35.0
       ...
887    27.0
888    19.0
889     NaN
890    26.0
891    32.0
Name: Age, Length: 891, dtype: float64
```

This is still a Pandas series, but this time it represents an **uni-variate** sample.

Data Wrangling (or Data Munging)



Data never comes perfectly clean or ready to use. Real-world datasets often contain missing values, inconsistent formats, outliers, or irrelevant information. **Data wrangling** is the process of transforming raw data into a structured and usable format—preparing it for analysis or modeling. This may involve tasks such as renaming columns, converting data types, handling null values, filtering rows, or creating new features. Good wrangling is essential: it ensures that the insights we extract are based on reliable, well-organized information.

Data wrangling is an unstructured and varied process. We will get back on it multiple times, but here are the main concepts which is good to know:

Handling Missing Values

Real-world datasets often contain missing entries. These can appear as empty cells, `NaN`, or placeholders like `?`.

Titanic example:
The `Age` and `Cabin` columns contain missing values.
We can choose to:

- Remove rows with missing data (`dropna`)
- Fill them with a default or statistical value (`fillna`)
- Flag them as missing and treat them separately

For instance, we can obtain a dataset free of `NaN` values as follows:

```
titanic_clean = titanic.dropna()
titanic_clean.info()
titanic_clean.head()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 183 entries, 2 to 890
Data columns (total 11 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Survived  183 non-null    int64
 1   Pclass    183 non-null    int64
 2   Name      183 non-null    object
 3   Sex       183 non-null    object
 4   Age       183 non-null    float64
 5   SibSp     183 non-null    int64
 6   Parch     183 non-null    int64
 7   Ticket    183 non-null    object
 8   Fare      183 non-null    float64
 9   Cabin     183 non-null    object
 10  Embarked  183 non-null    object
dtypes: float64(2), int64(4), object(5)
memory usage: 17.2+ KB
```

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId											
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
11	1	3	Sandstrom, Miss. Marguerite Rut	female	4.0	1	1	PP 9549	16.7000	G6	S
12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113783	26.5500	C103	S

The dataset is now super-clean, but also very smaller (183 rows vs 891!). This is due to the fact that we removed all rows containing at least one missing value. This is often not a good idea, as we may loose a lot of data.

If we are working with a specific variable, we may want to **first select it, then remove missing values**. For example:

```
age_clean = titanic['Age'].dropna()
age_clean.info()
age_clean.head()
```

```
<class 'pandas.core.series.Series'>
Index: 714 entries, 1 to 891
Series name: Age
Non-Null Count  Dtype
--------------  -----
714 non-null    float64
dtypes: float64(1)
memory usage: 11.2 KB
```

```
PassengerId
1    22.0
2    38.0
3    26.0
4    35.0
5    35.0
Name: Age, dtype: float64
```

We have a much larger sample with 714 elements.

A third way to handle missing values is to **fill them in**. For example, we can replace missing ages with the average age:

```
age_clean2 = titanic['Age'].fillna(titanic['Age'].mean())
age_clean2.info()
age_clean2.head()
```

```
<class 'pandas.core.series.Series'>
Index: 891 entries, 1 to 891
Series name: Age
Non-Null Count  Dtype
--------------  -----
891 non-null    float64
dtypes: float64(1)
memory usage: 46.2 KB
```

```
PassengerId
1    22.0
2    38.0
3    26.0
4    35.0
5    35.0
Name: Age, dtype: float64
```

We now have 891 entries and no missing data. The process of filling in missing data is also called **data imputation**.

We can also mix the different strategies depending on the variables. For instance, replacing ages with the mean age may make sense, but we cannot do the same with `Cabin`. Let's see an example to get rid of missing value with a mix of techniques:

```
titanic2 = titanic.copy() # to avoid modifying the original dataset
titanic2['Age'] = titanic2['Age'].fillna(titanic2['Age'].mean())
titanic2['Cabin'] = titanic2['Cabin'].fillna('Unknown')
titanic2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 891 entries, 1 to 891
Data columns (total 11 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Survived  891 non-null    int64
 1   Pclass    891 non-null    int64
 2   Name      891 non-null    object
 3   Sex       891 non-null    object
 4   Age       891 non-null    float64
 5   SibSp     891 non-null    int64
 6   Parch     891 non-null    int64
 7   Ticket    891 non-null    object
 8   Fare      891 non-null    float64
 9   Cabin     891 non-null    object
 10  Embarked  889 non-null    object
dtypes: float64(2), int64(4), object(5)
memory usage: 115.8+ KB
```

In practice, we filled age with the mean value, but marked all missing Cabin with “Unknown”.

Converting Data Types

Sometimes we need to convert columns to the correct type—especially when reading from CSV files.

Titanic example:
Convert `Pclass` from integer to categorical using `astype('category')`.

Let’s first inspect the type of the `Pclass` variable. This can be done with info, but also with dtype as follows:

```
titanic2['Pclass'].dtype
```

```
dtype('int64')
```

While we have an integer value, this represents the class of passengers, so it probably contains only few unique values. Let’s check this by seeing the unique values of the column (without duplicates):

```
titanic2['Pclass'].unique()
```

```
array([3, 1, 2])
```

It does make sense to convert this to a categorical variable (a nominal variable). Let’s do it as follows:

```
titanic2['Pclass'] = titanic2['Pclass'].astype('category')
titanic2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 891 entries, 1 to 891
Data columns (total 11 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Survived  891 non-null    int64
 1   Pclass    891 non-null    category
 2   Name      891 non-null    object
 3   Sex       891 non-null    object
 4   Age       891 non-null    float64
 5   SibSp     891 non-null    int64
 6   Parch     891 non-null    int64
 7   Ticket    891 non-null    object
 8   Fare      891 non-null    float64
 9   Cabin     891 non-null    object
 10  Embarked  889 non-null    object
dtypes: category(1), float64(2), int64(3), object(5)
memory usage: 109.9+ KB
```

Renaming and Reformatting

Clean, consistent column names make analysis easier.

You might want to rename columns, standardize formats, or strip whitespace.

Titanic example:

Rename `SibSp` to `SiblingsSpouses` for clarity.

```
titanic2.rename(columns={'SibSp': 'SiblingsSpouses'}, inplace=True)
titanic2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 891 entries, 1 to 891
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Survived         891 non-null    int64
 1   Pclass           891 non-null    category
 2   Name             891 non-null    object
 3   Sex              891 non-null    object
 4   Age              891 non-null    float64
 5   SiblingsSpouses  891 non-null    int64
 6   Parch            891 non-null    int64
 7   Ticket           891 non-null    object
 8   Fare             891 non-null    float64
 9   Cabin            891 non-null    object
 10  Embarked         889 non-null    object
dtypes: category(1), float64(2), int64(3), object(5)
memory usage: 109.9+ KB
```

Creating New Features

Sometimes, we need to make some characteristics of the data more explicit. We can derive new columns from existing ones to enrich your analysis.

Titanic example:
Create a `FamilySize` column by summing `SibSp` and `Parch`.

```
titanic2['FamilySize'] = titanic2['SiblingsSpouses'] + titanic2['Parch']
titanic2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 891 entries, 1 to 891
Data columns (total 12 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Survived         891 non-null    int64
 1   Pclass           891 non-null    category
 2   Name             891 non-null    object
 3   Sex              891 non-null    object
 4   Age              891 non-null    float64
 5   SiblingsSpouses  891 non-null    int64
 6   Parch            891 non-null    int64
 7   Ticket           891 non-null    object
 8   Fare             891 non-null    float64
 9   Cabin            891 non-null    object
 10  Embarked         889 non-null    object
 11  FamilySize       891 non-null    int64
dtypes: category(1), float64(2), int64(4), object(5)
memory usage: 116.8+ KB
```

Filtering and Subsetting

You may want to focus on a subset of the data—e.g., only adult passengers, or only those who embarked from Southampton.

In the case of Titanic, we may select only adult passengers as follows:

```
titanic2 = titanic2[titanic2['Age'] ⩾ 18]
titanic2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 778 entries, 1 to 891
Data columns (total 12 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Survived         778 non-null    int64
 1   Pclass           778 non-null    category
 2   Name             778 non-null    object
 3   Sex              778 non-null    object
 4   Age              778 non-null    float64
 5   SiblingsSpouses  778 non-null    int64
 6   Parch            778 non-null    int64
 7   Ticket           778 non-null    object
 8   Fare             778 non-null    float64
 9   Cabin            778 non-null    object
 10  Embarked         776 non-null    object
 11  FamilySize       778 non-null    int64
dtypes: category(1), float64(2), int64(4), object(5)
memory usage: 73.8+ KB
```

We can check that our subsetting worked as follows:

```
titanic2['Age'].min()
```

```
np.float64(18.0)
```

We can also drop a column as follows:

```
titanic2.drop(columns=['Ticket'], inplace=True)
titanic2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 778 entries, 1 to 891
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Survived         778 non-null    int64
 1   Pclass           778 non-null    category
 2   Name             778 non-null    object
 3   Sex              778 non-null    object
 4   Age              778 non-null    float64
 5   SiblingsSpouses  778 non-null    int64
 6   Parch            778 non-null    int64
 7   Fare             778 non-null    float64
 8   Cabin            778 non-null    object
 9   Embarked         776 non-null    object
 10  FamilySize       778 non-null    int64
dtypes: category(1), float64(2), int64(4), object(4)
memory usage: 67.7+ KB
```

Besides this, we can also select a subset of variables as follows:

```
titanic2[['Age', 'Cabin', 'Fare']].head()
```

	Age	Cabin	Fare
PassengerId			
1	22.0	Unknown	7.2500
2	38.0	C85	71.2833
3	26.0	Unknown	7.9250
4	35.0	C123	53.1000
5	35.0	Unknown	8.0500

Long vs Wide Formats

Data can be organized in different formats depending on the goals of the analysis. Understanding these formats helps us reshape and prepare data effectively for modeling, visualization, or statistical testing.

Wide Format

- Each **variable** has its own column
- Each **observation** is a single row
- **Advantages**: Easy to read, intuitive structure, compatible with most analysis tools
- **Use cases**: Statistical modeling, machine learning, spreadsheet-style inspection

Example:
A dataset where each row is a student and each column is a subject score.

```
Wide Format:
   Student  Math  Science  English
0    Alice    85       88       92
1      Bob    92       85       88
2  Charlie    78       92       85
```

Long Format

- Variables are **stacked** into fewer columns
- Each observation may span **multiple rows**
- **Advantages**: More compact, flexible for grouping and plotting
- **Use cases**: Time series, repeated measures, tidy data workflows, faceted plots

Example:
Instead of one row per student, we have one row per student–subject pair, with columns like `StudentID`, `Subject`, and `Score`.

Most of the time, especially in introductory analysis and machine learning, we'll encounter datasets in **wide format**. However, certain tasks—like time-based analysis or grouped visualizations—may require us to **reshape the data into long format**. We'll revisit these concepts and apply them later when working with real datasets.

```
Long Format:
   Student  Subject  Score
0    Alice     Math     85
1      Bob     Math     92
2  Charlie     Math     78
3    Alice  Science     88
4      Bob  Science     85
5  Charlie  Science     92
6    Alice  English     92
7      Bob  English     88
8  Charlie  English     85
```

Conversion across formats can be done as follows:

```
# Example: Wide vs Long format

# Create sample data in wide format
wide_data = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Charlie'],
    'Math': [85, 92, 78],
    'Science': [88, 85, 92],
    'English': [92, 88, 85]
})

print("Wide Format:")
print(wide_data)

# Convert to long format
long_data = pd.melt(wide_data,
                    id_vars=['Student'],
                    var_name='Subject',
                    value_name='Score')

print("\nLong Format:")
print(long_data)

# Convert back to wide format
wide_again = long_data.pivot(index='Student', columns='Subject', values='Score')
print("\nBack to Wide Format:")
print(wide_again)
```

```
Wide Format:
   Student  Math  Science  English
0    Alice    85       88       92
1      Bob    92       85       88
2  Charlie    78       92       85

Long Format:
   Student  Subject  Score
0    Alice     Math     85
1      Bob     Math     92
2  Charlie     Math     78
3    Alice  Science     88
4      Bob  Science     85
5  Charlie  Science     92
6    Alice  English     92
7      Bob  English     88
8  Charlie  English     85

Back to Wide Format:
Subject  English  Math  Science
Student
Alice         92    85       88
Bob           88    92       85
Charlie       85    78       92
```

These are some of the techniques used in data wrangling. In practice, depending on the kind of analysis, we may perform different operations, such as:

- **Converting units of measurement** (e.g., from inches to centimeters, or Fahrenheit to Celsius)
- **Merging different tables** (e.g., using a join operation like in SQL to combine passenger data with ticket prices)
- **Parsing and extracting information** from strings (e.g., splitting full names into first and last names)
- **Normalizing or scaling values** (e.g., rescaling age or fare to a 0–1 range for modeling)
- **Binning continuous variables** into categories (e.g., grouping ages into child, adult, senior)
- **Removing duplicates or irrelevant columns**
- **Filtering outliers** or suspicious values (e.g., negative ages or fares)
- **Reshaping data** between wide and long formats depending on the analysis or visualization needs

We'll revisit many of these techniques and apply them to real datasets as we move forward. For now, it's important to understand that data wrangling is not a fixed recipe—it's a flexible toolbox that adapts to the structure and goals of each project.

The Data Analysis Workflow

Now that we've seen the main concepts, terminology, and tools involved in working with data, we can introduce a broader perspective: the **data analysis workflow**. This is the structured process that guides how we move from raw data to insights, decisions, and predictions.

Let's begin with a working definition, adapted from Wikipedia:

> Data analysis is the **process** of inspecting, **cleaning**, **transforming**, and **modeling** data with the goal of **discovering useful information**, **informing conclusions**, and **supporting decision-making**.

While this definition is informal, it provides a useful starting point. In the sections below, we'll unpack its key components and introduce the main types and goals of data analysis.

Data Analysis as a Process

The definition above emphasizes that data analysis is a **process**—not a single algorithm or technique, but a sequence of steps that combine statistical reasoning, domain knowledge, and computational tools. These steps often include:

- **Inspecting**: Assessing the structure and content of the dataset—number of rows and columns, types of variables, typical values, and potential anomalies. For example, in a dataset of student marks, we might expect scores to range from 0 to 10.
- **Cleaning**: Correcting or removing problematic data entries. This could involve handling missing values, removing duplicates, or filtering out invalid entries (e.g., negative marks or scores above the maximum).
- **Transforming**: Modifying the data to make it more suitable for analysis. This might include creating new columns (e.g., a mean score per student), rescaling variables, or converting formats (e.g., dividing English scores out of 30 by 3 to match other subjects scored out of 10).
- **Modeling**: Applying statistical or machine learning models to summarize, explain, or predict aspects of the data. A model can help us understand relationships between variables—for instance, predicting English scores based on performance in other subjects.

The Typical Workflow

A standard data analysis workflow includes the following stages:

1. **Define** your data analysis question
2. **Collect** the data needed to answer that question
3. **Clean and format** the data
4. **Explore and describe** the data

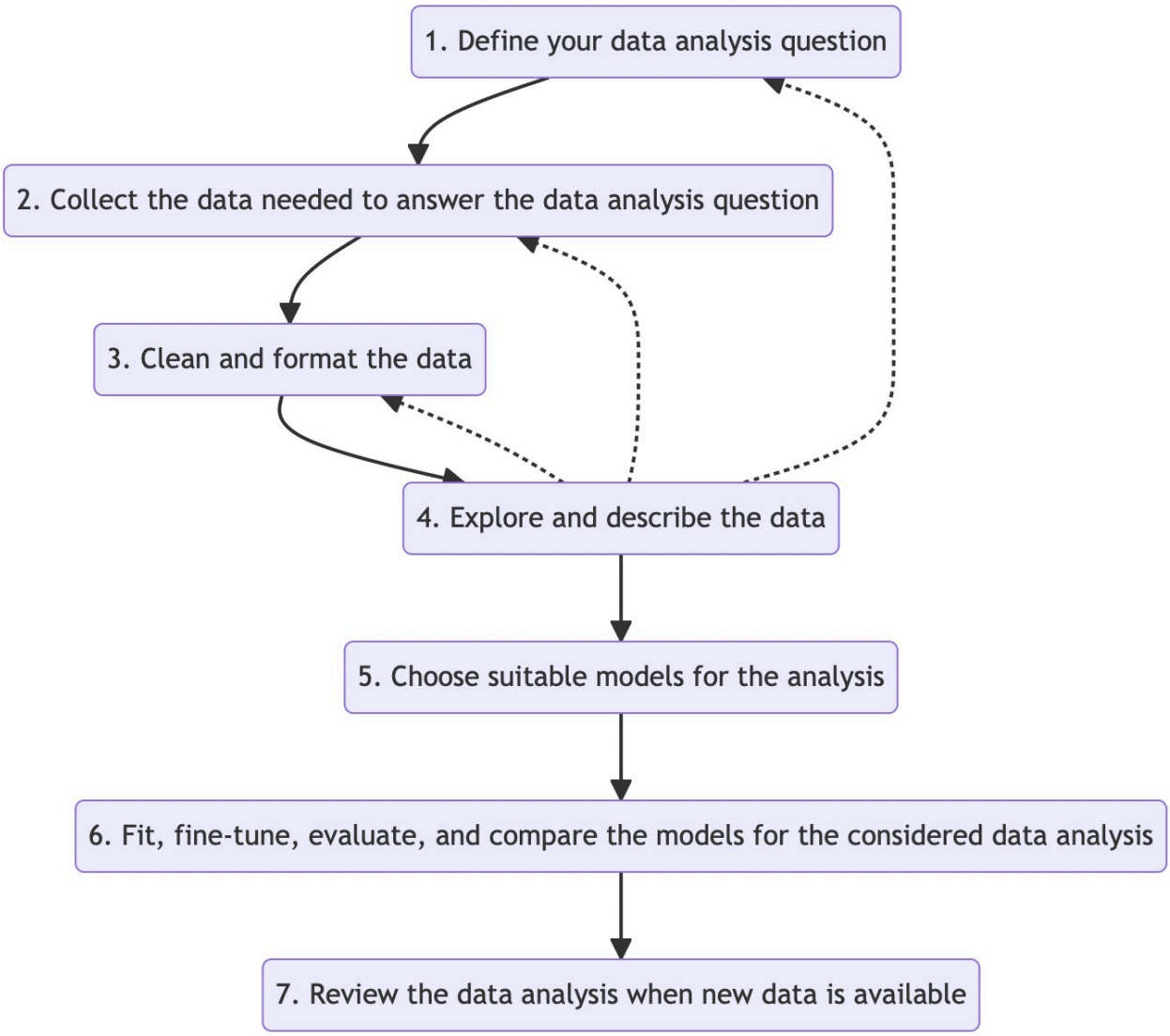
5. **Choose suitable models** for the analysis
6. **Fit, fine-tune, evaluate, and compare** the models
7. **Review and update** the analysis as new data becomes available

A Non-Linear Process

Although the workflow is presented as a sequence, real-world data analysis is rarely linear. Analysts often revisit earlier steps based on what they learn along the way. For example:

- After exploring the data (step 4), you might discover outliers and return to cleaning (step 3)
- You might realize that your initial question (step 1) needs refinement
- You might need additional data (step 2) to improve your model

The diagram below illustrates this flexible structure. Solid arrows show the main flow, while dashed arrows indicate possible backward steps—especially after exploration.



Example: Analyzing Customer Reviews

Let's walk through a practical example. A data analyst at an e-commerce company is tasked with improving product quality and customer satisfaction by analyzing customer reviews.

- **Step 1**: Define the question: “What are the common themes and issues in customer reviews?”
- **Step 2**: Collect reviews from the company website, social media, and third-party platforms
- **Step 3**: Clean the data—remove duplicates, correct spelling, standardize formatting
- **Step 4**: Explore the data—use word clouds, sentiment analysis, and frequency counts
- **Step 5**: Choose a model—apply topic modeling (e.g., Latent Dirichlet Allocation)
- **Step 6**: Evaluate the model—realize that topics are overlapping and unclear
- **Step 4 (again)**: Re-explore the data—identify that product and service issues are mixed
- **Step 5 (again)**: Separate reviews into categories and reapply topic modeling
- **Step 6**: Obtain clearer topics like “defective products” and “responsive support”
- **Step 7**: Commit to ongoing review as new reviews come in

This iterative approach allows the analyst to refine insights and adapt strategies over time.

This workflow will guide our journey through the course. As we move forward, we’ll apply each step to real datasets, starting with the Titanic dataset and expanding to more complex scenarios.

Exercises

1. **Population and sample.** For each, name the population, the sample and the unit of observation: (a) an exit poll on election day; (b) the server logs of a web shop for one month; (c) a clinical trial of a new drug. Which of the three lets you generalise, and to what?
2. **Variable types.** Classify every column of the Titanic dataset as nominal, ordinal, discrete or continuous. Two of them are genuinely debatable — say which, and why the choice matters for the analysis.
3. **Loading and inspecting.** Load the Titanic data and answer, with code: how many rows and columns? Which columns have missing values, and how many? What is the data type of each column, and is it the type you would expect?
4. **Missing is information.** `Cabin` is missing for most passengers. Is it missing at random? Compare the survival rate of passengers with and without a recorded cabin, and say what the comparison suggests about *why* the value is missing.

References

- Chapter 1 of *Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.*

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Describing and visualizing the data

Chapter resources

Slides (PDF) Open in Colab Practice quiz

We have seen how to access the data and perform some basic manipulations. In practice, datasets are often large enough to make it difficult to grasp their structure or patterns by simply printing rows or scanning tables. This is where **descriptive statistics** and **visualization techniques** become essential tools.

Describing and visualizing the data helps us:

- Gather insights on the dataset
- Understand the main characteristics of each variable
- Detect missing or anomalous values
- Identify relationships between variables
- Spot trends or clusters of data which are similar
- Guide the choice of pre-processing and further modeling

To make some example, we will consider a simple dataset where weights (in pounds), heights (in inches), and sex of various subjects. The dataset will look like this:

```
import pandas as pd
hw=pd.read_csv(DATA + "height_weight_pounds.csv")
hw.info()
hw.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4231 entries, 0 to 4230
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   sex     4231 non-null   object
 1   height  4231 non-null   int64
 2   weight  4231 non-null   float64
dtypes: float64(1), int64(1), object(1)
memory usage: 99.3+ KB
```

	sex	height	weight
0	M	74	53.484771
1	M	70	38.056472
2	F	61	34.970812
3	M	68	35.999365
4	F	66	34.559390

Since we are inspecting the dataset, it is worth to answer these questions:

- How many observations do we have?
- How many variables?
- What are the measuring scales of each variable?
- Which variables are discrete? Which continuous? Which numerical?
- Are there missing values?

We will begin by focusing on the univariate sample of heights:

```
heights = hw['height']
heights
```

```
0       74
1       70
2       61
3       68
4       66
        ..
4226    69
4227    69
4228    64
4229    64
4230    61
Name: height, Length: 4231, dtype: int64
```

Absolute and Relative Frequencies

Absolute Frequencies

A first way to describe data is to calculate the number of times each value appears. These are called “absolute frequencies”. Absolute frequencies are generally calculated for discrete variables where observations take a finite number of values.

Let

$$a_1, a_2, \ldots, a_n$$

be the values that the variable under consideration can take.

The absolute frequencies n_i are defined as the number of times that a_i appears in the sample. Note that:

$$\sum_i n_i = N$$

Where N is the total number of elements in the sample.

We can compute unique values a_j as follows:

```
heights.unique()
```

```
array([74, 70, 61, 68, 66, 65, 64, 67, 72, 71, 76, 69, 63, 75, 60, 59, 73])
```

The absolute frequencies can be computed as follows:

```
heights.value_counts()
```

```
height
64    435
61    411
66    391
67    377
68    355
65    351
69    302
63    291
70    272
72    260
71    235
73    146
60    130
74    104
75     72
59     52
76     47
Name: count, dtype: int64
```

Note that this will sort the values by frequency. We can sort them by value as follows:

```
heights.value_counts().sort_index()
```

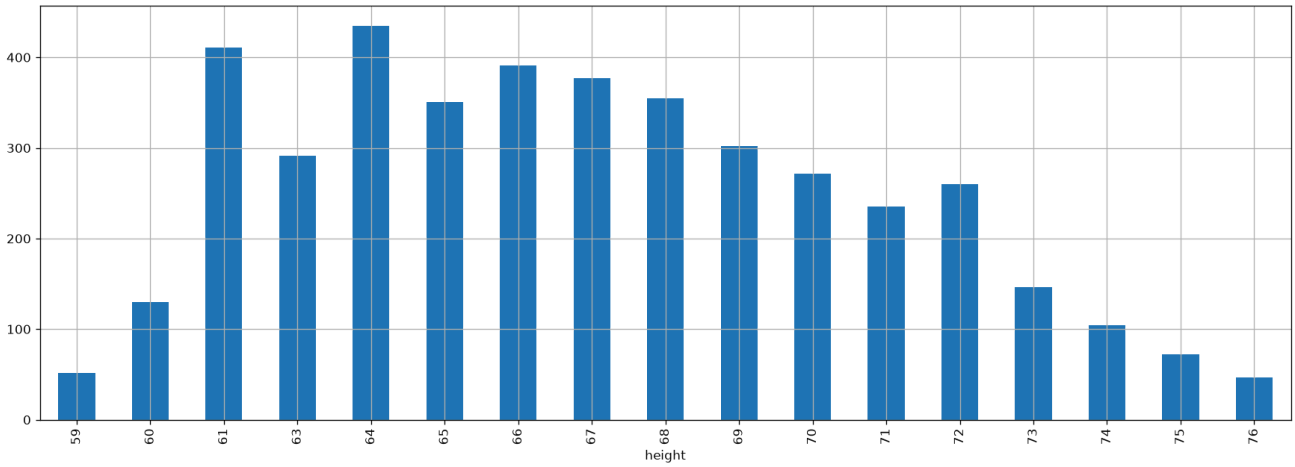
```
height
59     52
60    130
61    411
63    291
64    435
65    351
66    391
67    377
68    355
69    302
70    272
71    235
72    260
73    146
74    104
75     72
76     47
Name: count, dtype: int64
```

Bar Chart of Absolute Frequencies

The data frequencies can be graphically represented using a bar chart that places the unique value (a_i) on the x-axis and represents the absolute frequency n_i as the height of the bar.

In our example:

```
from matplotlib import pyplot as plt
hw['height'].value_counts().sort_index().plot.bar(figsize=(18,6))
plt.grid()
plt.show()
```



The graph above tells us something about the number of occurrences of a given value and also gives us an idea of which values are more or less frequent. We see that the data follows a “bell” shape. We

will often find this type of shape and will discuss it further later.

Relative Frequencies

Absolute frequencies allow us to get a more precise idea of how data are distributed, regardless of the size of our sample. For example, we know that the sample contains more individuals with a height of $167.64\ cm$ than individuals with a height of $193.04\ cm$. However, this representation is linked to the total number of elements contained in the sample. For example, a similarly distributed sample, but with more observations, will result in larger absolute frequencies. We can obtain a representation independent of the sample size by analyzing relative frequencies, defined as follows:

$$f_j = f(a_j) = \frac{n_j}{N}, j = 1, 2, \ldots, k$$

Note that, given the definition, we will have:

$$n_j \leq n \Rightarrow f_j \leq 1 \; \forall j$$

$$\sum_j f_j = \sum_j \frac{n_j}{N} = \frac{1}{N} \sum_j n_j = \frac{N}{N} = 1$$

We can obtain relative frequencies still with value_counts, but this time we will pass the argument `normalize=True`:

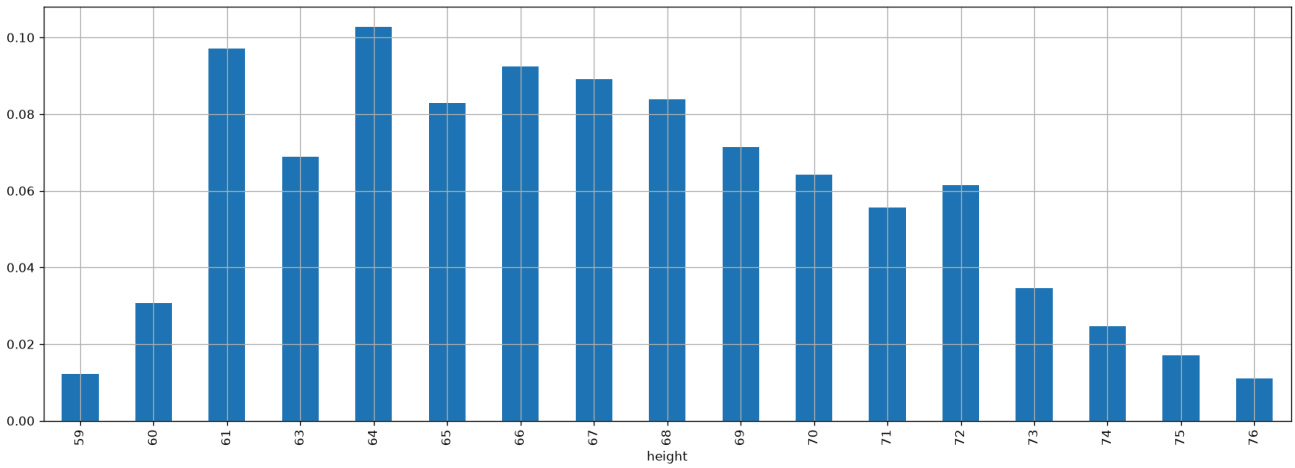
```
heights.value_counts(normalize=True).sort_index()
```

```
height
59    0.012290
60    0.030726
61    0.097140
63    0.068778
64    0.102813
65    0.082959
66    0.092413
67    0.089104
68    0.083905
69    0.071378
70    0.064287
71    0.055542
72    0.061451
73    0.034507
74    0.024580
75    0.017017
76    0.011108
Name: proportion, dtype: float64
```

Bar Chart of Relative Frequencies

By plotting the relative frequencies using a bar chart, we obtain a graph very similar to the one showing absolute frequencies, but with a different scale on the y-axis.

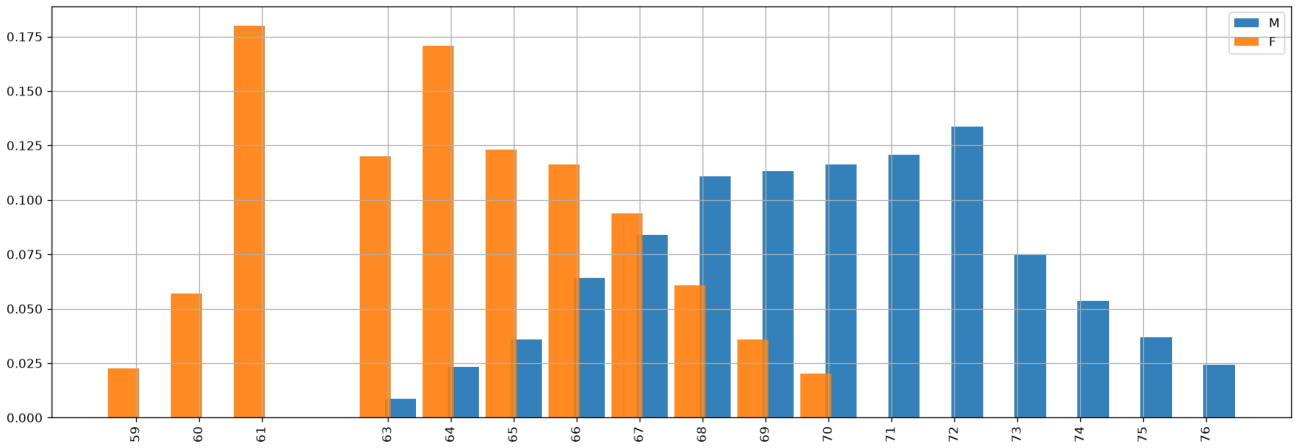
```
(hw['height'].value_counts(normalize=True).sort_index()).plot.bar(figsize=(18,6))
plt.grid()
plt.show()
```



Relative frequency bar charts are useful for comparing different samples. For example, we can consider the heights of men and women in the weights-and-heights dataset:

```
pmf_height_m = hw[hw['sex']=='M']['height'].value_counts(normalize=True).sort_index()
pmf_height_f = hw[hw['sex']=='F']['height'].value_counts(normalize=True).sort_index()

plt.figure(figsize=(18,6))
# We add and subtract 0.2 from the indices to "shift" the bars and make them
# visible when overlapping. We also set alpha=0.9 to make the bars
# partially transparent.
plt.bar(pmf_height_m.index+0.2, pmf_height_m.values, width=0.5, alpha=0.9)
plt.bar(pmf_height_f.index-0.2, pmf_height_f.values, width=0.5, alpha=0.9)
plt.xticks(hw['height'].unique(), rotation='vertical')
plt.legend(['M','F']) # Display a legend
plt.grid()
plt.show()
```



From the comparison above, we can already make some qualitative observations about the two samples. In particular, we notice (unsurprisingly) that men are generally taller than women. This doesn't mean that there aren't men who are shorter than some women, or vice versa, but such cases are reasonably less frequent.

Barplots and Stacked Barplots

We can also used barplots and stacked barplots to count the number of elements in a given class. For instance, a barplot counting the number of subjects in the "male" or "female" class looks like this:

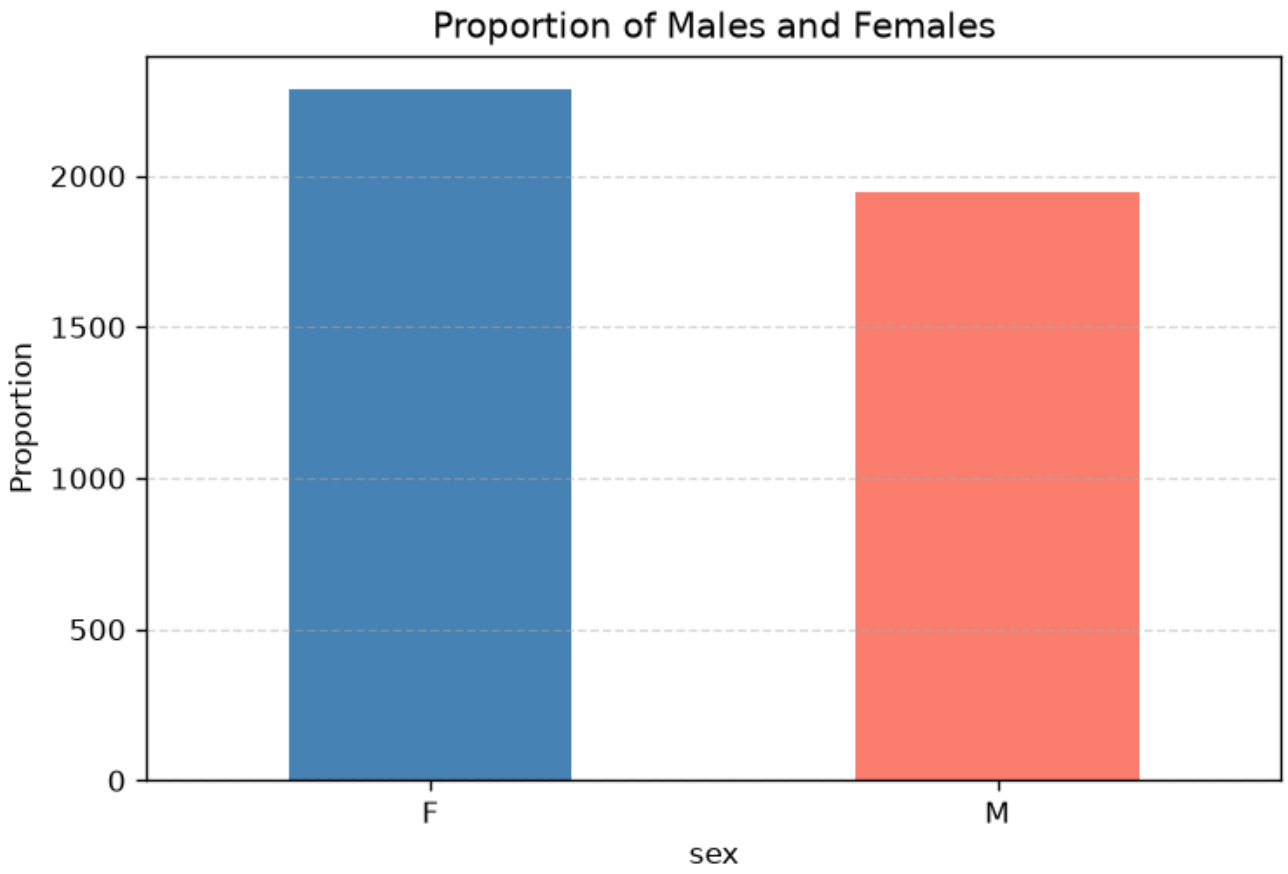
```
import matplotlib.pyplot as plt

# Calculate proportions
sex_counts = hw['sex'].value_counts()

# Create bar plot
sex_counts.plot.bar(color=['steelblue', 'salmon'])

# Customize plot
plt.title('Proportion of Males and Females')
plt.ylabel('Proportion')
plt.xticks(rotation=0)
plt.grid(axis='y', linestyle='--', alpha=0.5)

plt.show()
```



We can obtain the normalized version by passing normalize=True to value_counts:

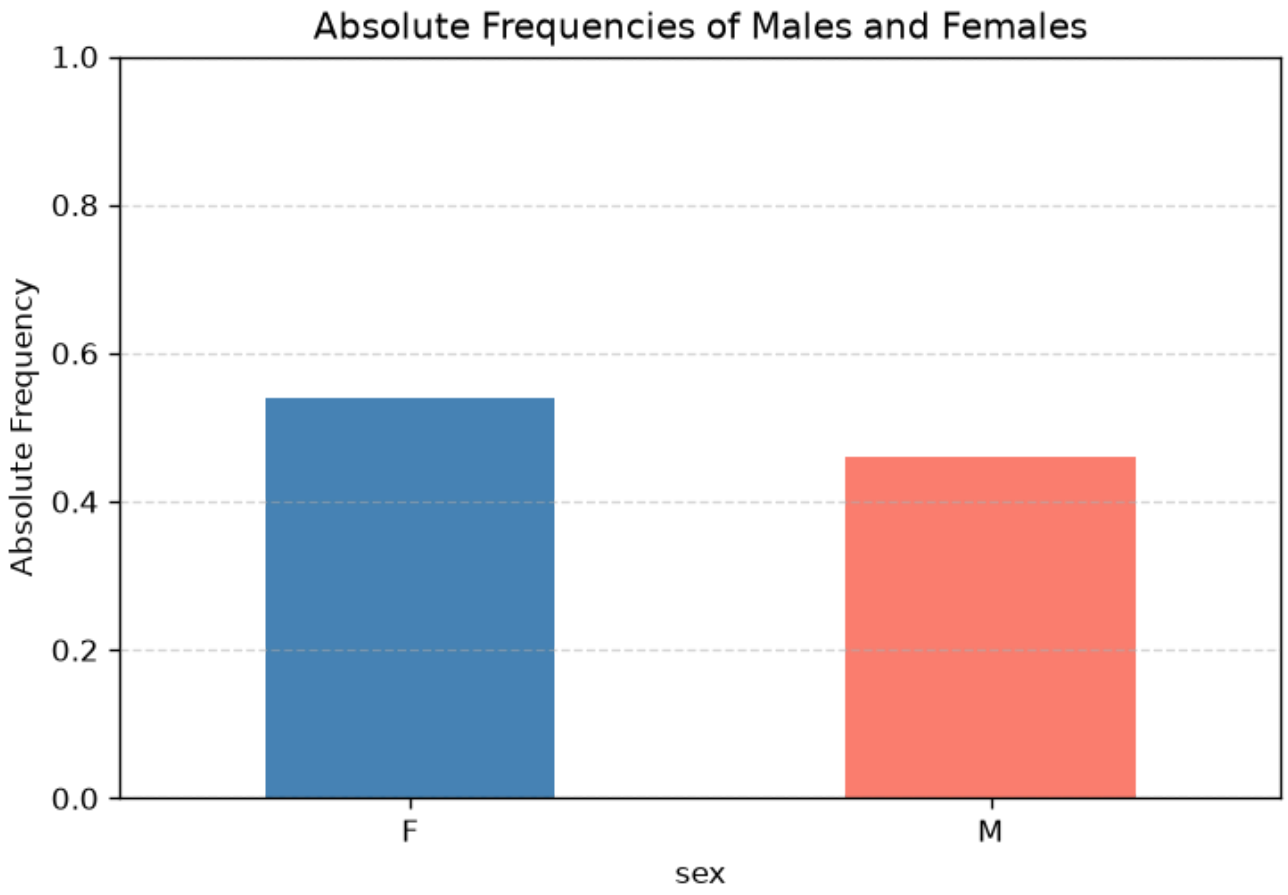
```
import matplotlib.pyplot as plt

# Calculate proportions
sex_counts = hw['sex'].value_counts(normalize=True)

# Create bar plot
sex_counts.plot.bar(color=['steelblue', 'salmon'])

# Customize plot
plt.title('Absolute Frequencies of Males and Females')
plt.ylabel('Absolute Frequency')
plt.xticks(rotation=0)
plt.ylim(0, 1)
plt.grid(axis='y', linestyle='--', alpha=0.5)

plt.show()
```



When numbers sum to one, another popular visualization is that of the stacked plot:

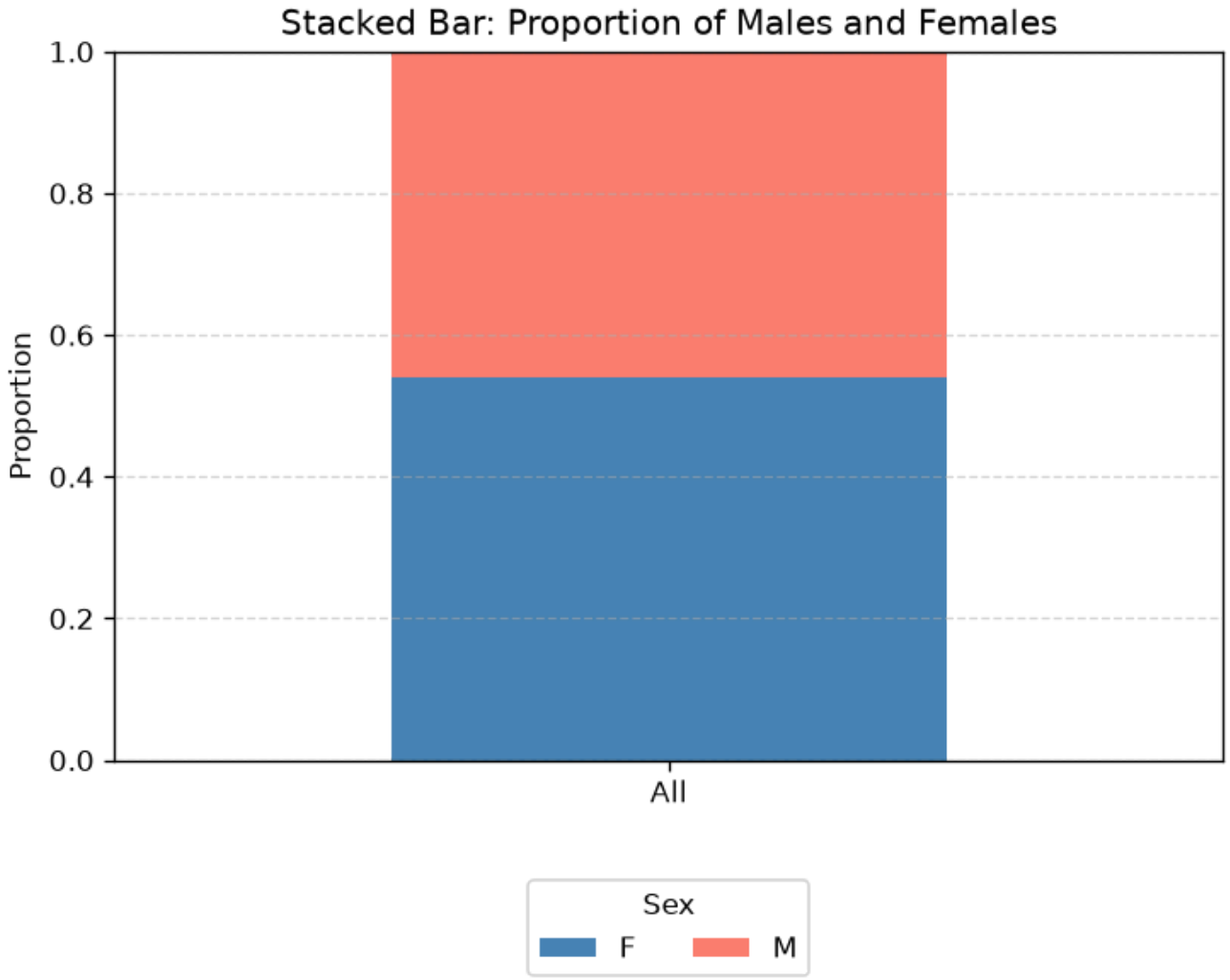
```
import matplotlib.pyplot as plt

# Compute proportions grouped by a single dummy category
sex_props = hw.groupby(lambda _: 'All')['sex'].value_counts(normalize=True).unstack()

# Plot as stacked bar
sex_props.plot(kind='bar', stacked=True, color=['steelblue', 'salmon'])

# Customize plot
plt.title('Stacked Bar: Proportion of Males and Females')
plt.ylabel('Proportion')
plt.xticks(rotation=0)
plt.ylim(0, 1)
plt.legend(title='Sex', loc='upper center', bbox_to_anchor=(0.5, -0.15), ncol=2)
plt.grid(axis='y', linestyle='--', alpha=0.5)

plt.show()
```

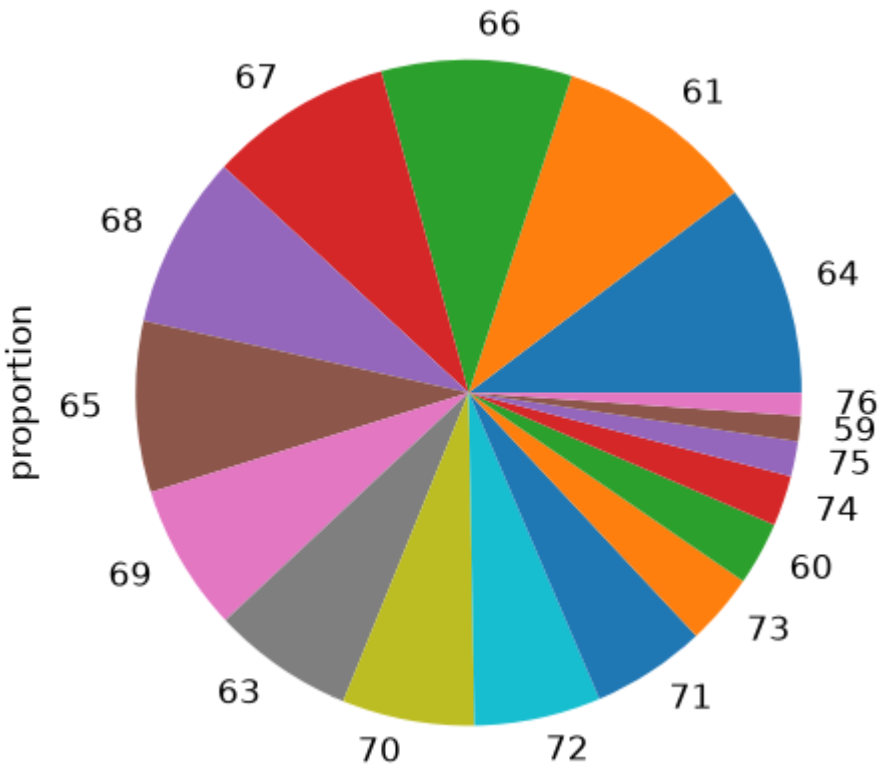


Pie Charts

As an alternative to bar charts, relative frequencies can also be displayed using pie charts.

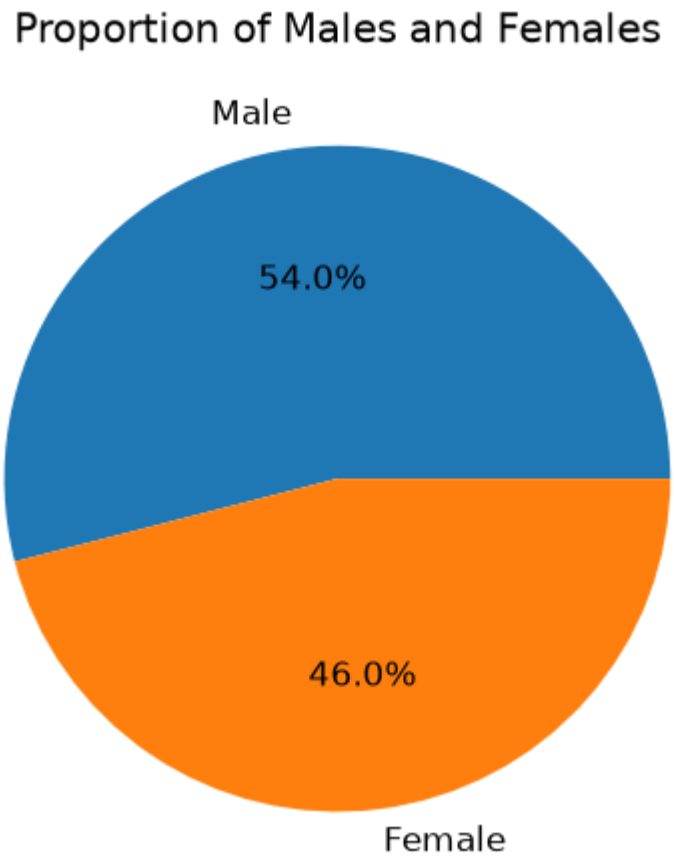
In the case of heights:

```
heights.value_counts(normalize=True).plot.pie()
plt.show()
```



Pie charts works better when we have fewer elements. For instance, let's say we want to show the percentage of males and females in our dataset:

```
hw['sex'].value_counts(normalize=True).plot.pie(autopct='%1.1f%%', labels=['Male',
        'Female'])
plt.title('Proportion of Males and Females')
plt.ylabel('')
plt.show()
```



While pie charts can be intuitive when there is no specific ordering of the elements we want to show, they often introduce more problems than other chart types. In particular, it's difficult for the human eye to accurately compare angles or assess relative sizes—especially when slices are similar in proportion or not arranged consistently. This can lead to misinterpretation or visual clutter, making bar charts or other alternatives generally more effective for comparing categories.

An illustrative example is below:

```python
import matplotlib.pyplot as plt

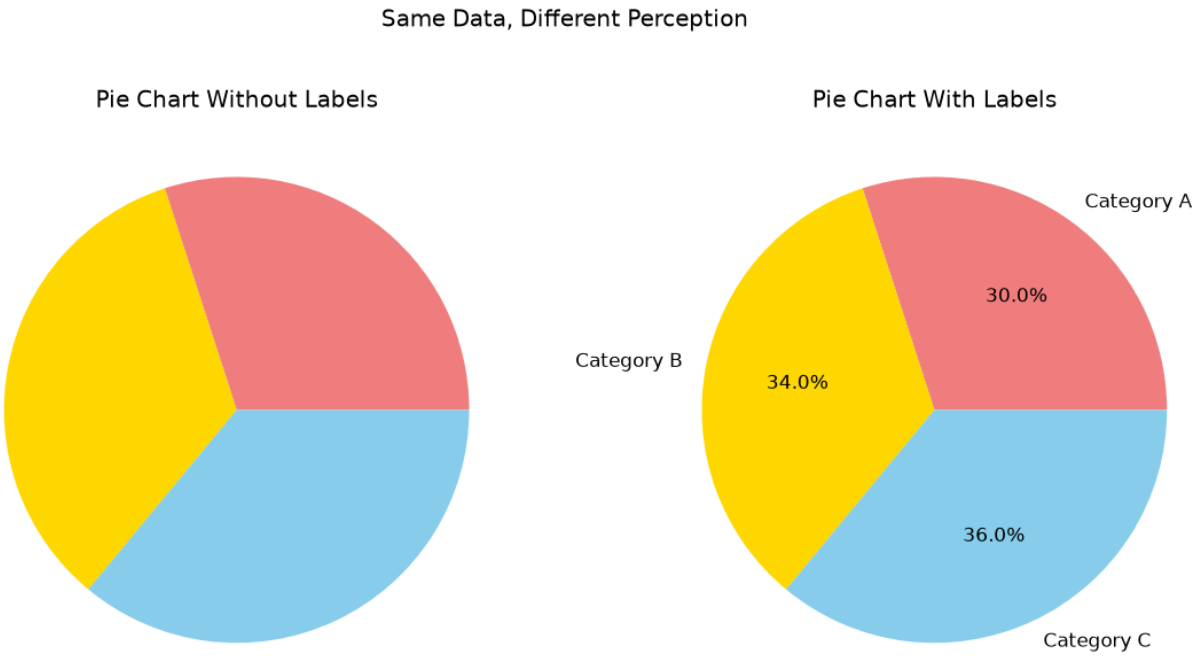
# Sample data: three categories with similar proportions
sizes = [30, 34, 36]
labels = ['Category A', 'Category B', 'Category C']
colors = ['lightcoral', 'gold', 'skyblue']

fig, axs = plt.subplots(1, 2, figsize=(12,6))

# Pie chart without labels
axs[0].pie(sizes, colors=colors)
axs[0].set_title('Pie Chart Without Labels')

# Pie chart with labels
axs[1].pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%')
axs[1].set_title('Pie Chart With Labels')

plt.suptitle('Same Data, Different Perception')
plt.show()
```

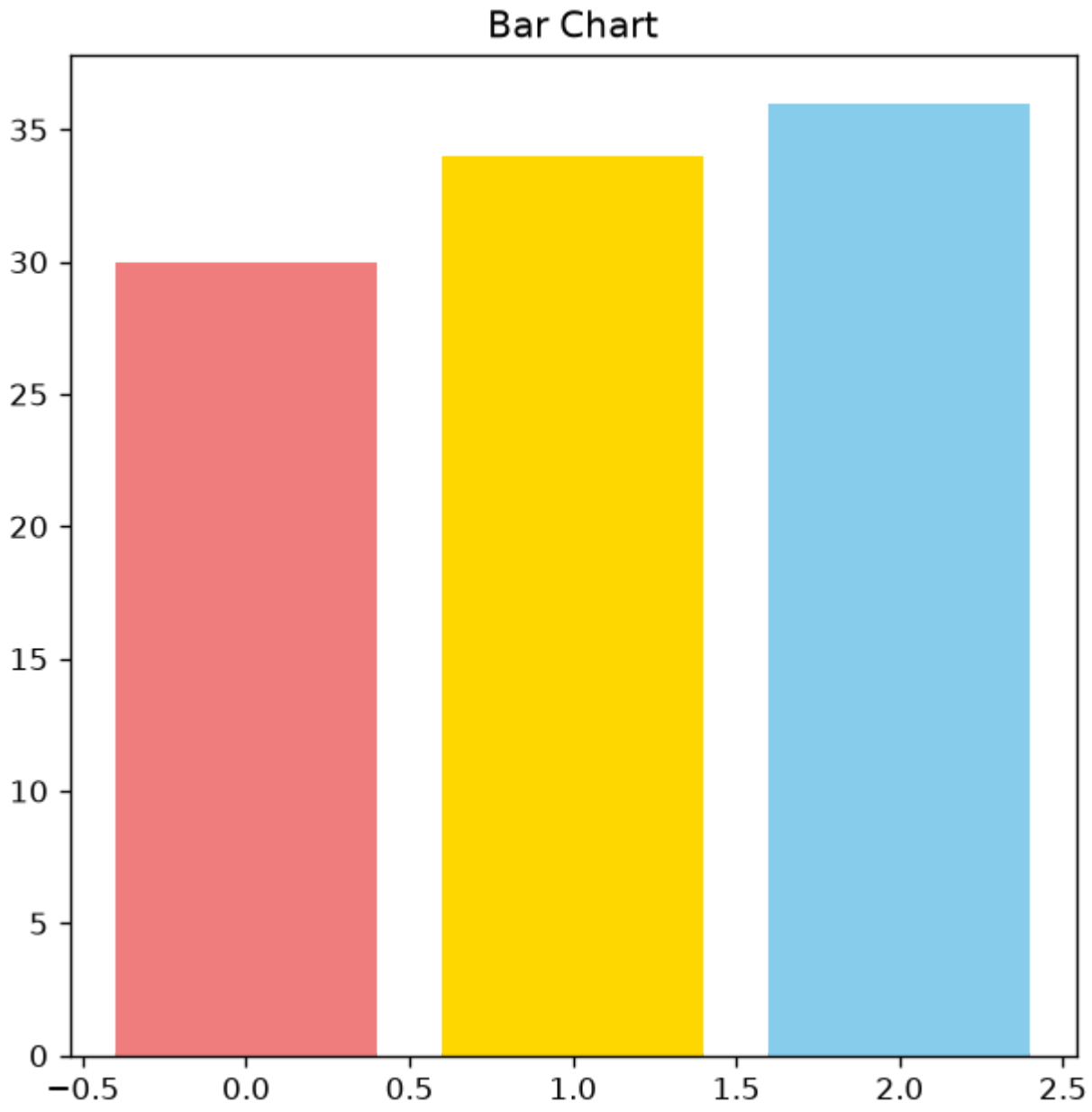


The figure shows the same plot with and without labels. Without labels, even significant differences as 4 are not easy to spot.

Compare the pie chart above with the bar chart below. Where is it easier to spot differences, even without labels?

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6,6))
ax.bar(range(len(sizes)), sizes, color=colors)
ax.set_title('Bar Chart')
plt.show()
```



In short: pie charts should be used with extra care!

Empirical Cumulative Distribution Function (ECDF)

Relative frequencies work particularly well when there are few unique values. However, when the number of unique values grows, the discrete frequencies calculated for the values become very small and thus subject to noise (e.g., due to measurement errors).

Let's now consider the sample of weights and its relative frequencies:

```
weights = hw['weight']
weights.value_counts(normalize=True).sort_index()
```

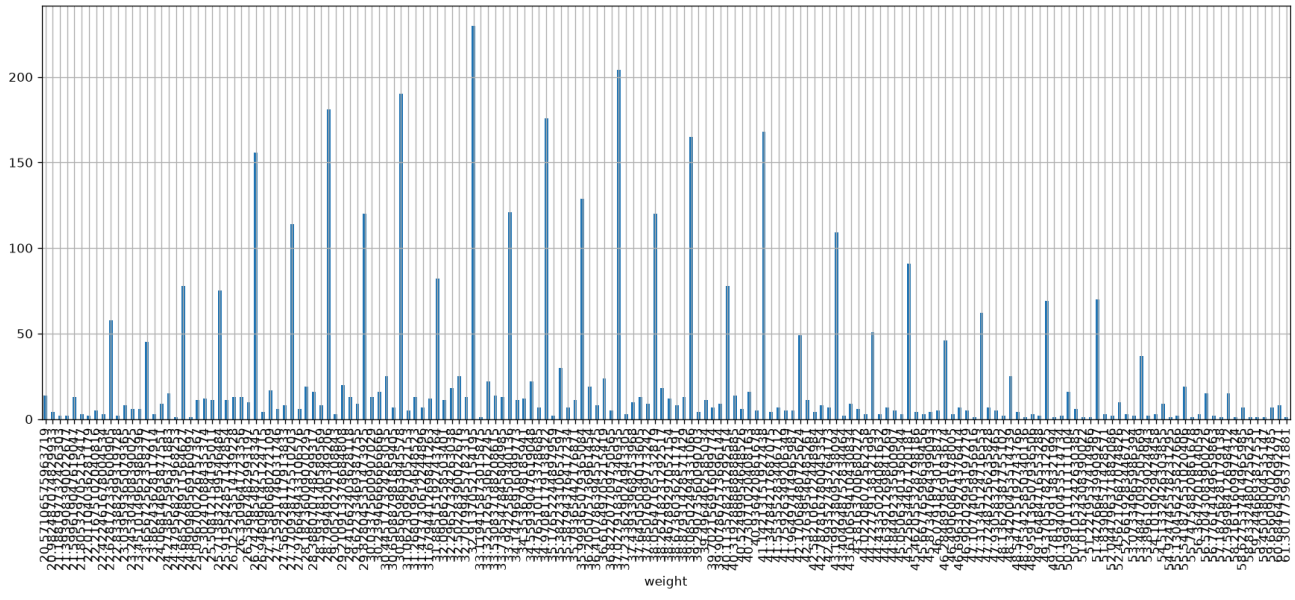
```
weight
20.571066    0.003309
20.982487    0.000945
21.188198    0.000473
21.393908    0.000473
21.599619    0.003073
               ...
59.244669    0.000236
59.450380    0.000236
59.656091    0.001654
60.684644    0.001891
61.301776    0.000236
Name: proportion, Length: 172, dtype: float64
```

We note that we now have many more unique values (172) and that weights now have decimal values as well.

This suggests that, while heights were discretized—probably rounded when collected—weights were recorded with greater precision. As a result, the weight variable behaves more like a continuous numerical feature, whereas height appears more discretized or categorical in nature. This distinction influences how we visualize and model these variables.

Let's see the plot of absolute frequencies:

```
weights.value_counts().sort_index().plot.bar(figsize=(18,6))
plt.grid()
```



Not very easy to read, right?

The representation is affected by noise, whereby two very close values are treated as two distinct cases in probability calculations. We will see some ways to overcome this problem. One of them consists of calculating an **Empirical Cumulative Distribution Function (ECDF)**. An **ECDF** calculates for a value x the sum of the relative frequencies of all values y less than or equal to x:

$$ECDF(x) = \sum_{a_j : a_j \leq x} f(a_j)$$

Where a_j are the unique values within the sample, while in general $x \in \mathbb{R}$.

Consider a simple dataset of numerical values:

```
a = pd.Series([1, 5, 2, 6, 5, 4, 3, 5, 4, 2, 4, 5, 6, 4, 4, 3])
a
```

```
0     1
1     5
2     2
3     6
4     5
5     4
6     3
7     5
8     4
9     2
10    4
11    5
12    6
13    4
14    4
15    3
dtype: int64
```

The relative frequencies will be as follows:

```
a.value_counts(normalize=True).sort_index()
```

```
1    0.0625
2    0.1250
3    0.1250
4    0.3125
5    0.2500
6    0.1250
Name: proportion, dtype: float64
```

The ECDF calculated on the unique values will be the following:

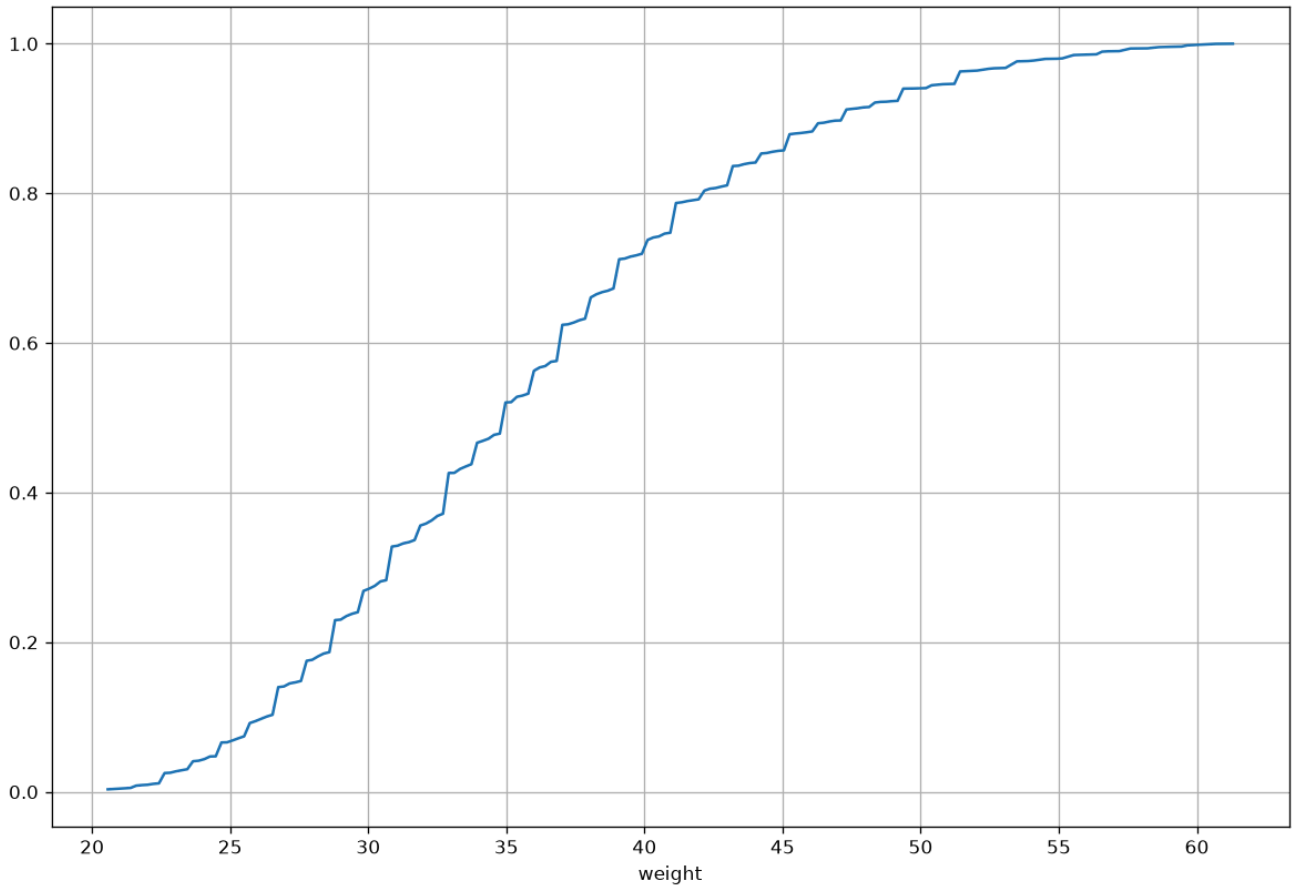
```
a.value_counts(normalize=True).sort_index().cumsum()
```

```
1    0.0625
2    0.1875
3    0.3125
4    0.6250
5    0.8750
6    1.0000
Name: proportion, dtype: float64
```

It should be noted that the values of the ECDF are always increasing and the last value is equal to 1 (the sum of all relative frequencies).

An ECDF can be represented graphically by placing the x values on the x-axis and the $ECDF(x)$ values on the y-axis. For example, the ECDF of the weights in our weights-heights dataset will be the following:

```
hw['weight'].value_counts(normalize=True).sort_index().cumsum().plot(figsize=(12,8))
plt.grid()
```



The definition of the ECDF tells us that, given a point with coordinates (x, y), the sum of the frequencies of elements less than or equal to x is equal to y. Alternatively, we can say that $y\%$ of the

elements have a value less than x.

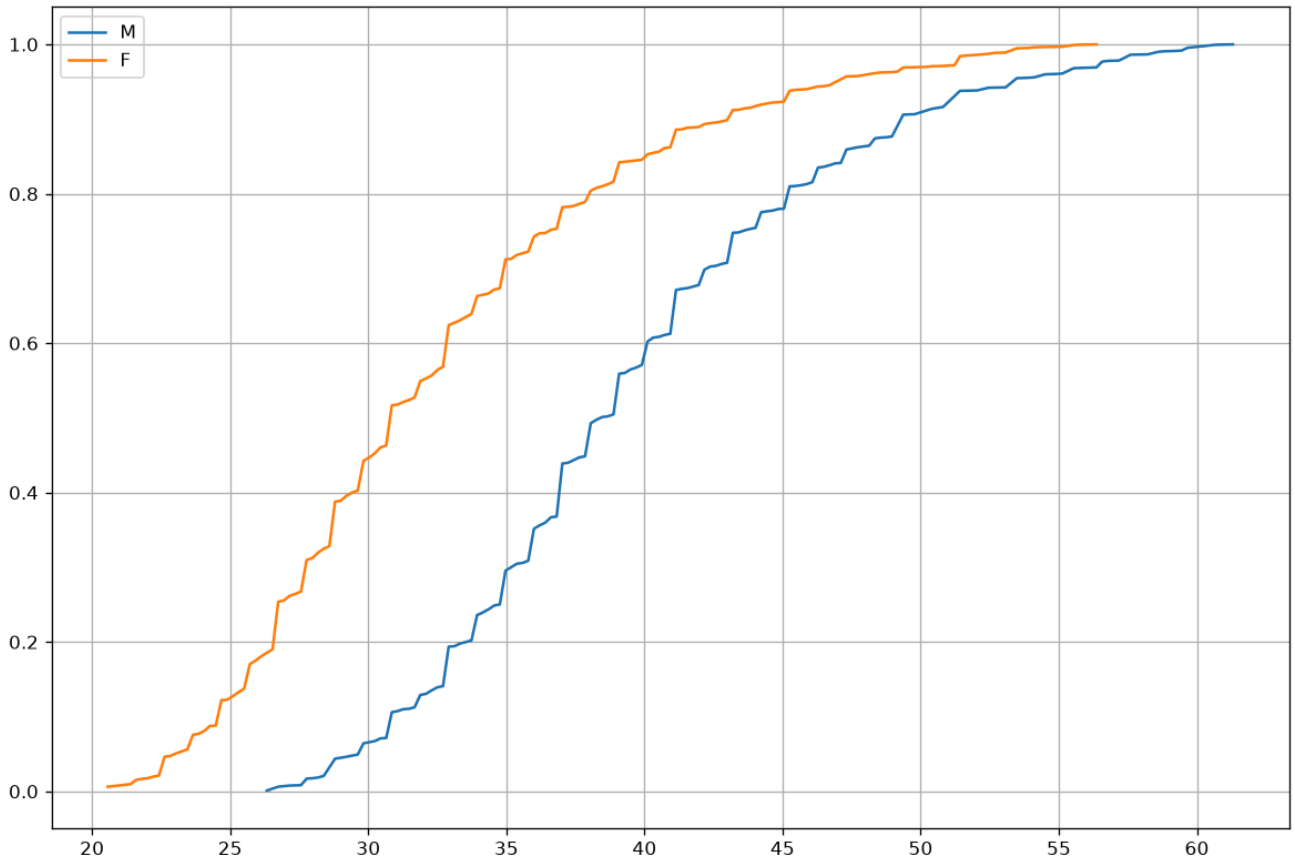
Observing the graph above, we can say that:

- Approximately 40% of subjects have a weight less than ≈ 33 pounds (about $66kg$);
- Approximately 80% of subjects have a weight less than ≈ 42 pounds (about $84Kg$);

ECDFs are useful for graphically checking if two phenomena have similar distributions. We can, for example, use them to compare the weight distributions of men and women:

```
ecdf_weight_m = hw[hw['sex']=='M']
        ['weight'].value_counts(normalize=True).sort_index().cumsum()
ecdf_weight_f = hw[hw['sex']=='F']
        ['weight'].value_counts(normalize=True).sort_index().cumsum()

plt.figure(figsize=(12,8))
plt.plot(ecdf_weight_m.index, ecdf_weight_m.values)
plt.plot(ecdf_weight_f.index, ecdf_weight_f.values)
plt.legend(['M','F'])
plt.grid()
plt.show()
```



Observing the graph above, we can say that:

- Approximately 40% of men weigh less than $80Kg$;
- Approximately 40% of women weigh less than $60Kg$;
- 100% of men weigh less than $140Kg$;

- 100% of women weigh less than $130Kg$.

In general, the graph above tells us that men tend to be heavier than women.

Histograms

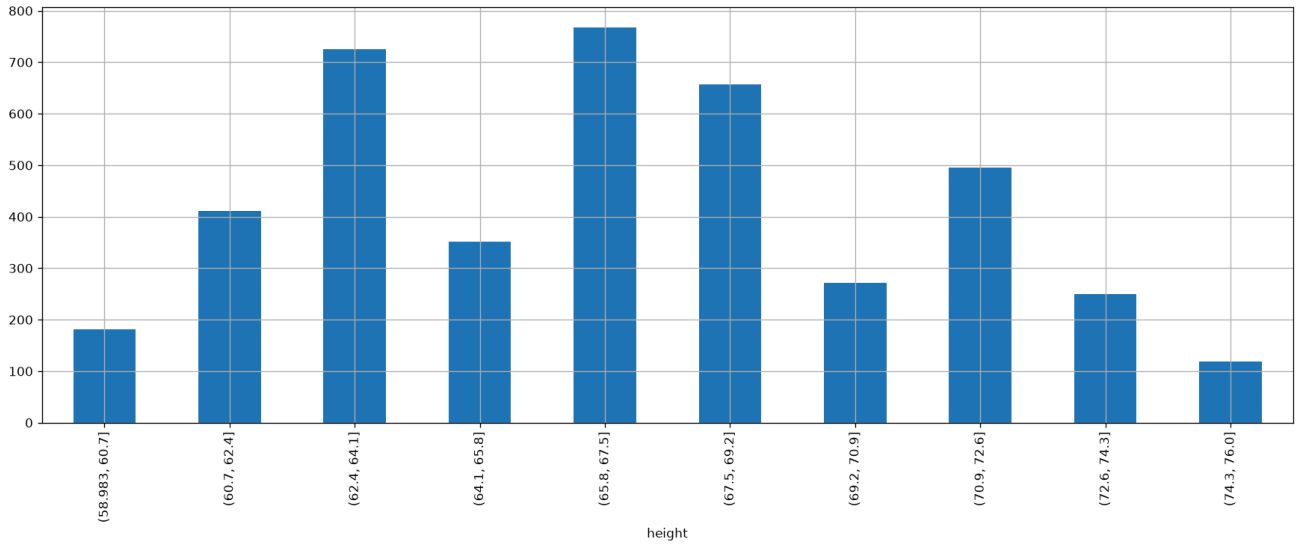
Another way to fix the noise problem is to group similar observations in the same "class". In practice, we will define intervals, which we call "bins" and assign each element to its corresponding bin. While this can be done in a variety of ways, bins are often chosen to be equals. This can be done using the cut function of Pandas:

```
heights_quantized = pd.cut(heights, bins=10)
heights_quantized
```

```
0       (72.6, 74.3]
1       (69.2, 70.9]
2       (60.7, 62.4]
3       (67.5, 69.2]
4       (65.8, 67.5]
           ...
4226    (67.5, 69.2]
4227    (67.5, 69.2]
4228    (62.4, 64.1]
4229    (62.4, 64.1]
4230    (60.7, 62.4]
Name: height, Length: 4231, dtype: category
Categories (10, interval[float64, right]): [(58.983, 60.7] < (60.7, 62.4] < (62.4,
64.1] < (64.1, 65.8] ... (69.2, 70.9] < (70.9, 72.6] < (72.6, 74.3] < (74.3, 76.0]]
```

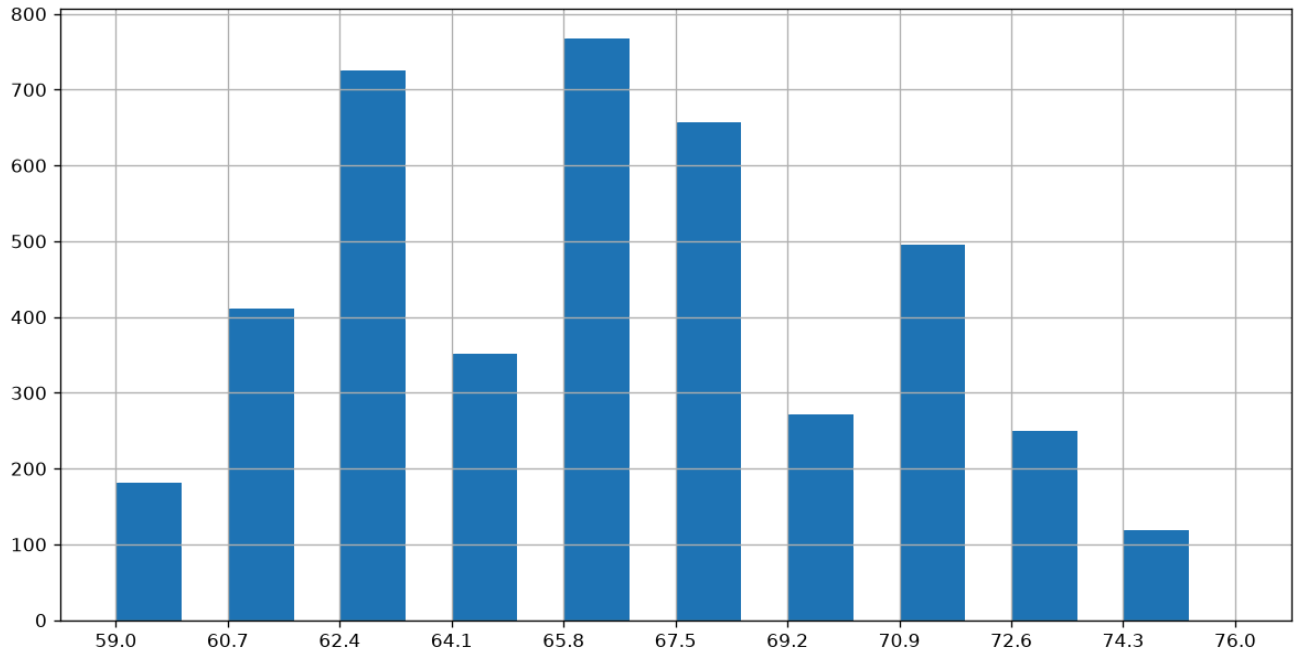
Let's visualize absolute frequencies of this binned version of the data:

```
heights_quantized.value_counts().sort_index().plot.bar(figsize=(18,6))
plt.grid()
plt.show()
```



The visualization we obtain is an histogram, and can be obtained also as follows:

```
plt.figure(figsize=(12,6))
_,edges,_=plt.hist(heights, width=1, bins=10) # We capture the "edges" of the bins
plt.xticks(edges) # Set x-ticks to the edges of the bins for better readability
plt.grid()
plt.show()
```



Each histogram “bin” covers a specific range. Common practice is to divide the data range, from minimum to maximum, into a certain number of bins of equal width.

Choosing the Number of Bins: Sturges and Rice

The number of bins can be specified arbitrarily or determined based on the graphical result you want to achieve. There are two heuristic criteria for finding starting values:

- Sturges: $\#bins = 3.3 \log(n)$

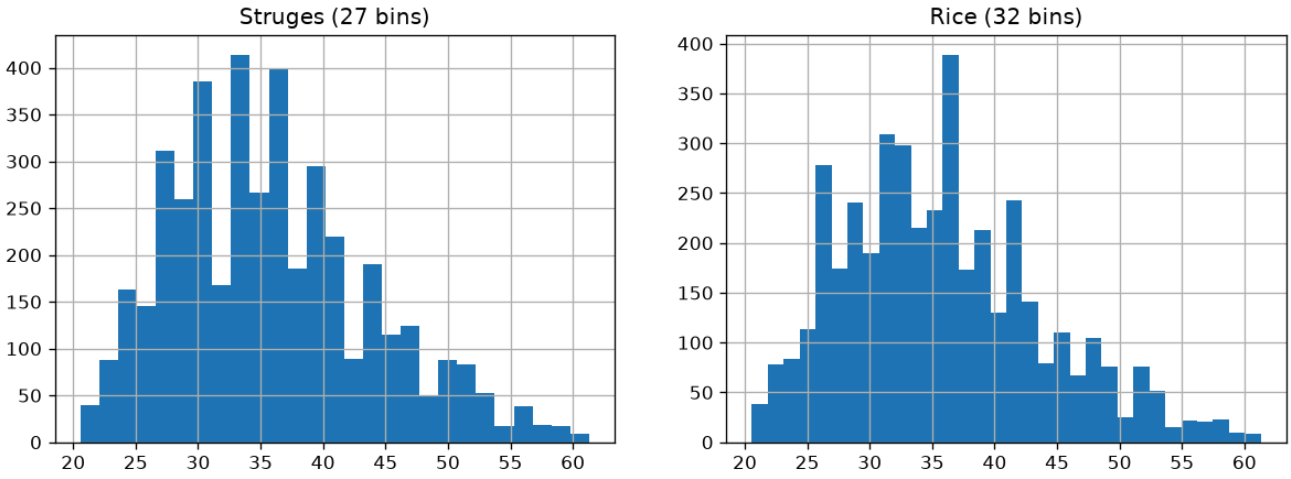
- Rice: $\#bins = 2 \cdot n^{1/3}$

Where n is the number of elements.

It should be noted that the number of bins can change the graphical result. Below are two examples obtained by calculating the number of bins with the two criteria considered:

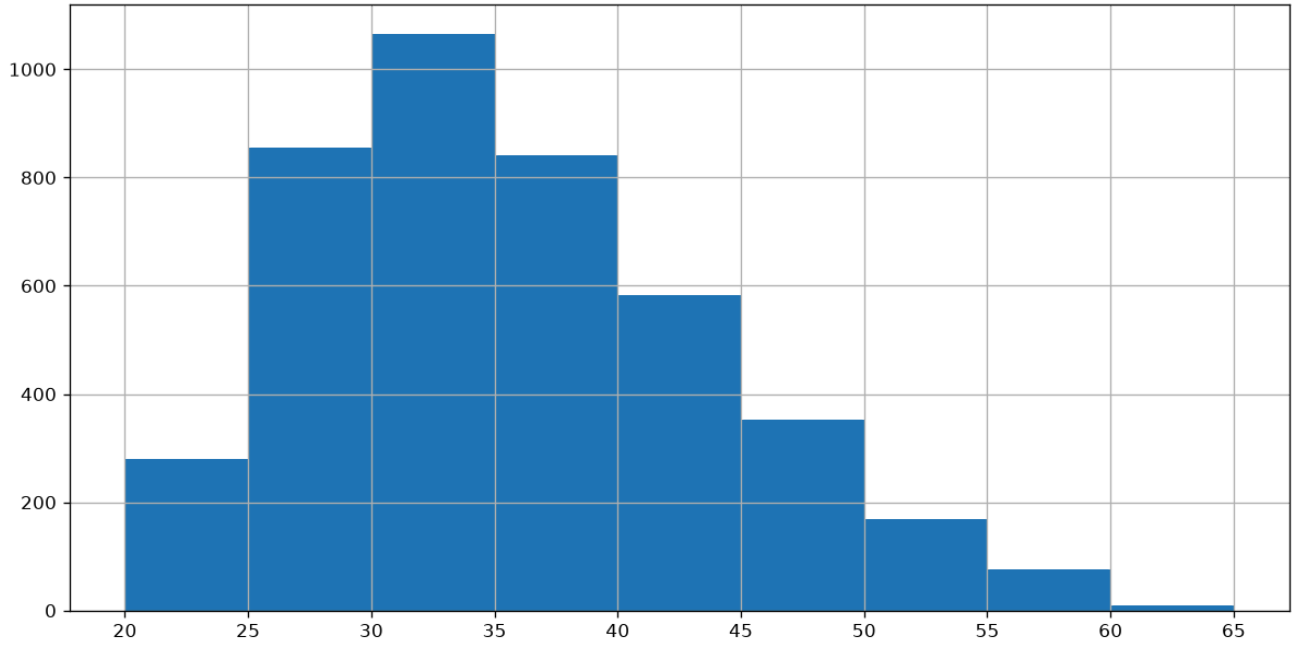
```
import numpy as np
bins_struges=int(3.3*np.log(len(weights)))
bins_rice=int(2*len(weights)**(1/3))

plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.title('Struges ({} bins)'.format(bins_struges))
plt.hist(weights, bins=bins_struges)
plt.grid()
plt.subplot(1,2,2)
plt.title('Rice ({} bins)'.format(bins_rice))
plt.grid()
plt.hist(weights, bins=bins_rice)
plt.show()
```



Alternatively, we can define arbitrary ranges for our bins. For example, considering the following limits for the bins `[20,25,30,35,40,45,50,55,60,65]`, we would obtain the following histogram:

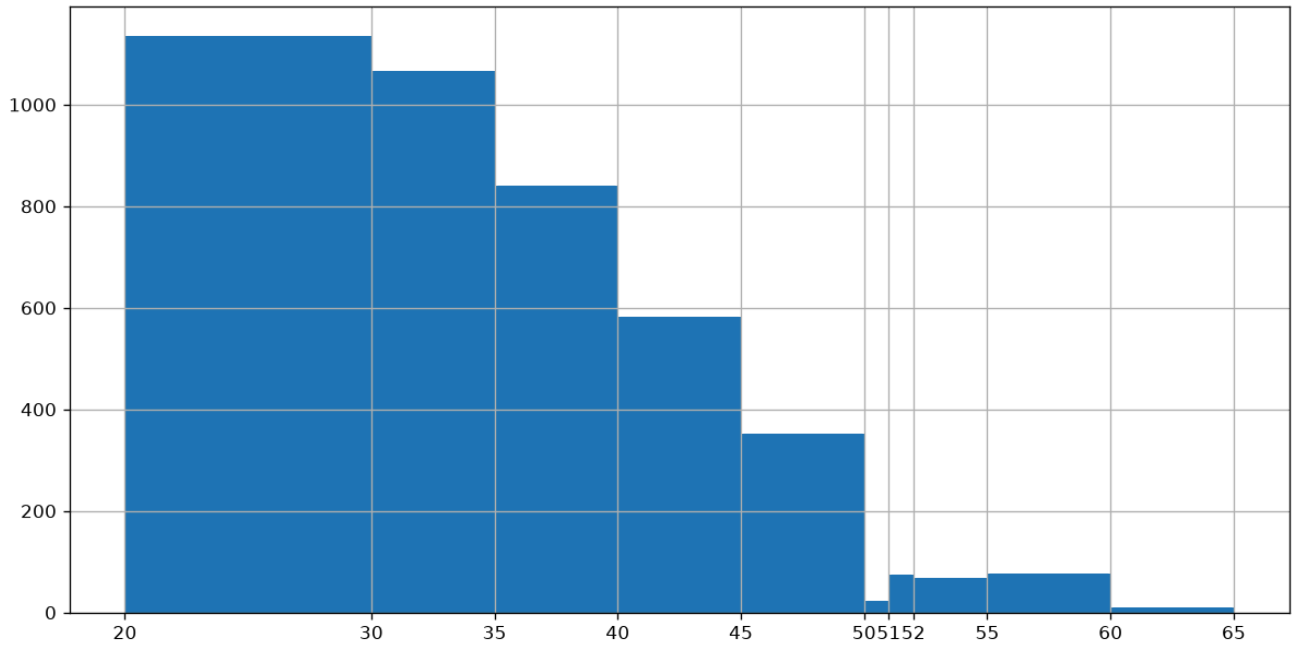
```
plt.figure(figsize=(12,6))
_,edges,_=plt.hist(weights, bins=[20,25,30,35,40,45,50,55,60,65]) # construct a
        histogram with defined bins
plt.xticks(edges)
plt.grid()
plt.show()
```



The histogram shown above reports the absolute frequencies for each bin and allows us to answer questions such as "how many subjects have a weight between 30 and 35 pounds"?

In particular cases, it is also possible to define histograms with bins of variable size. For example:

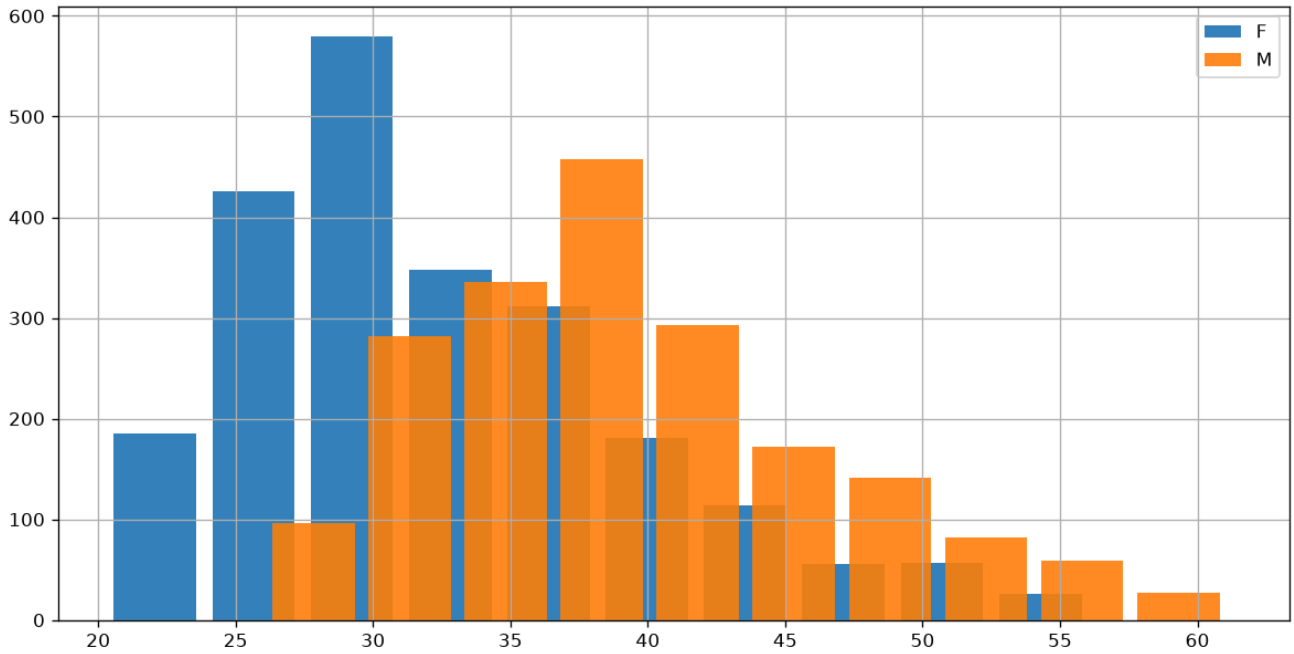
```
plt.figure(figsize=(12,6))
_,edges,_=plt.hist(weights, bins=[20,30,35,40,45,50,51,52,55,60,65]) # construct a
        histogram with the defined bins
plt.xticks(edges)
plt.grid()
plt.show()
```



Comparing Samples Using Histograms

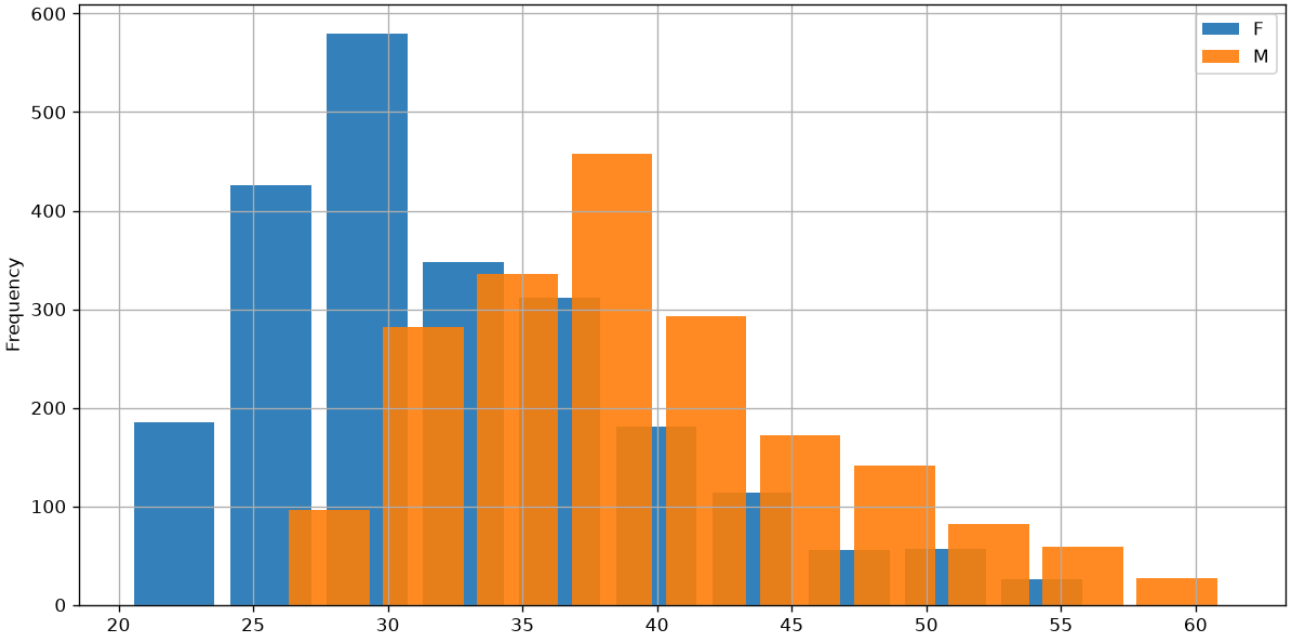
Histograms can be useful for comparing samples. The following graph compares the weights of men and women in our weights-heights sample:

```
plt.figure(figsize=(12,6))
plt.hist(hw[hw['sex']=='F']['weight'], alpha=0.9, width=3)
plt.hist(hw[hw['sex']=='M']['weight'], alpha=0.9, width=3)
plt.legend(['F','M'])
plt.grid()
plt.show()
```



Note that we can obtain a similar results directly from Pandas using the “groupby” function:

```
hw.groupby('sex')['weight'].plot.hist(width=3, alpha=0.9, figsize=(12,6))
plt.legend()
plt.grid()
plt.show()
```



Density Estimation

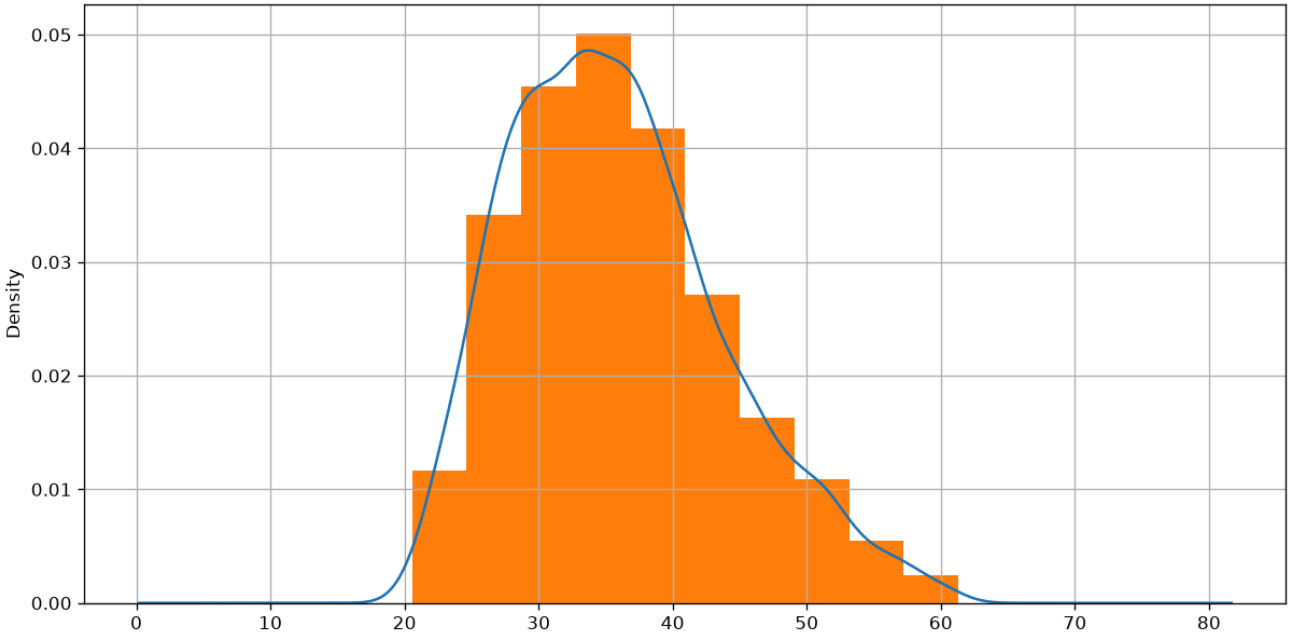
One disadvantage of histograms is that they arbitrarily categorize continuous data using bins. The choice of bin intervals changes the final appearance of the histogram.

Density estimation attempts to solve this problem by obtaining a "continuous" version of the histogram. Instead of dividing the x-axis into bins, density estimation calculates a value for each point on the x-axis, thus obtaining a continuous representation.

We will look more closely at how density estimation is performed later in the course. In the one-dimensional case, it can be calculated as shown in the optional deep dive below. For now, we know that density estimation can be calculated with a dedicated function "density" that depends on a single **bandwidth** parameter (with a good default value being computed by the libraries), determining the "sensitivity to detail" of the density estimation.

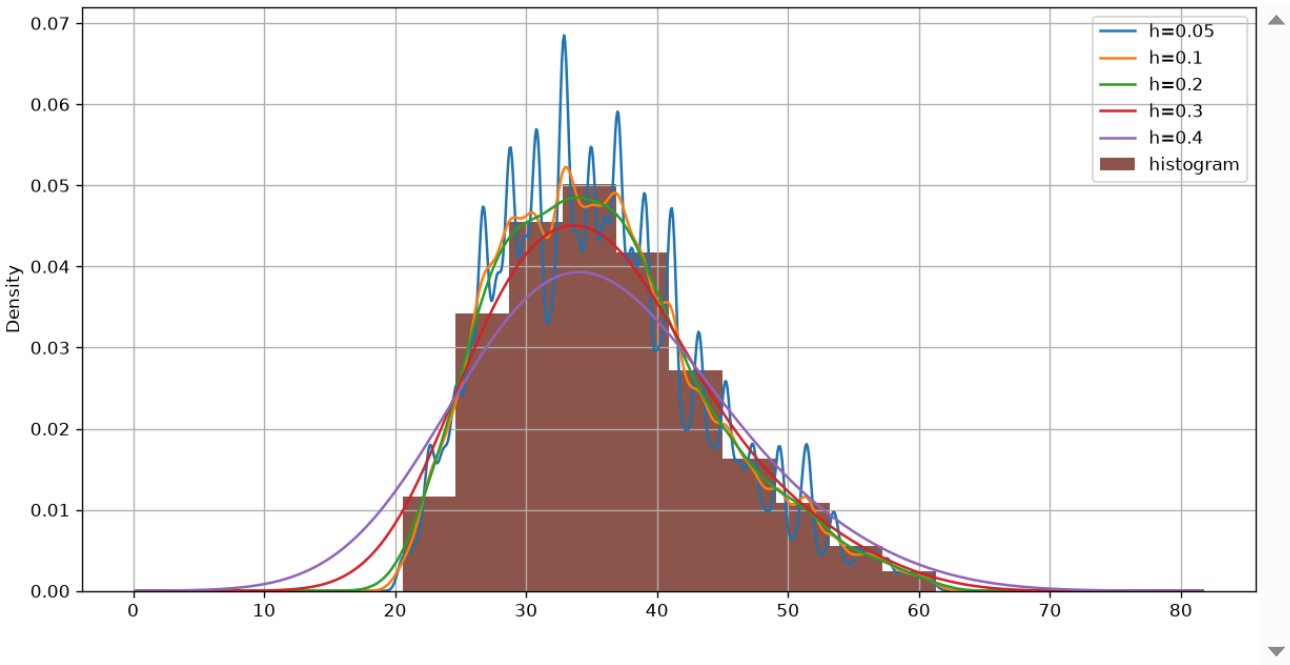
Let's compare the weight density estimation in our weight-height dataset with its corresponding histogram:

```
hw['weight'].plot.density(figsize=(12,6))
plt.hist(hw['weight'], density=True)
plt.grid()
plt.show()
```



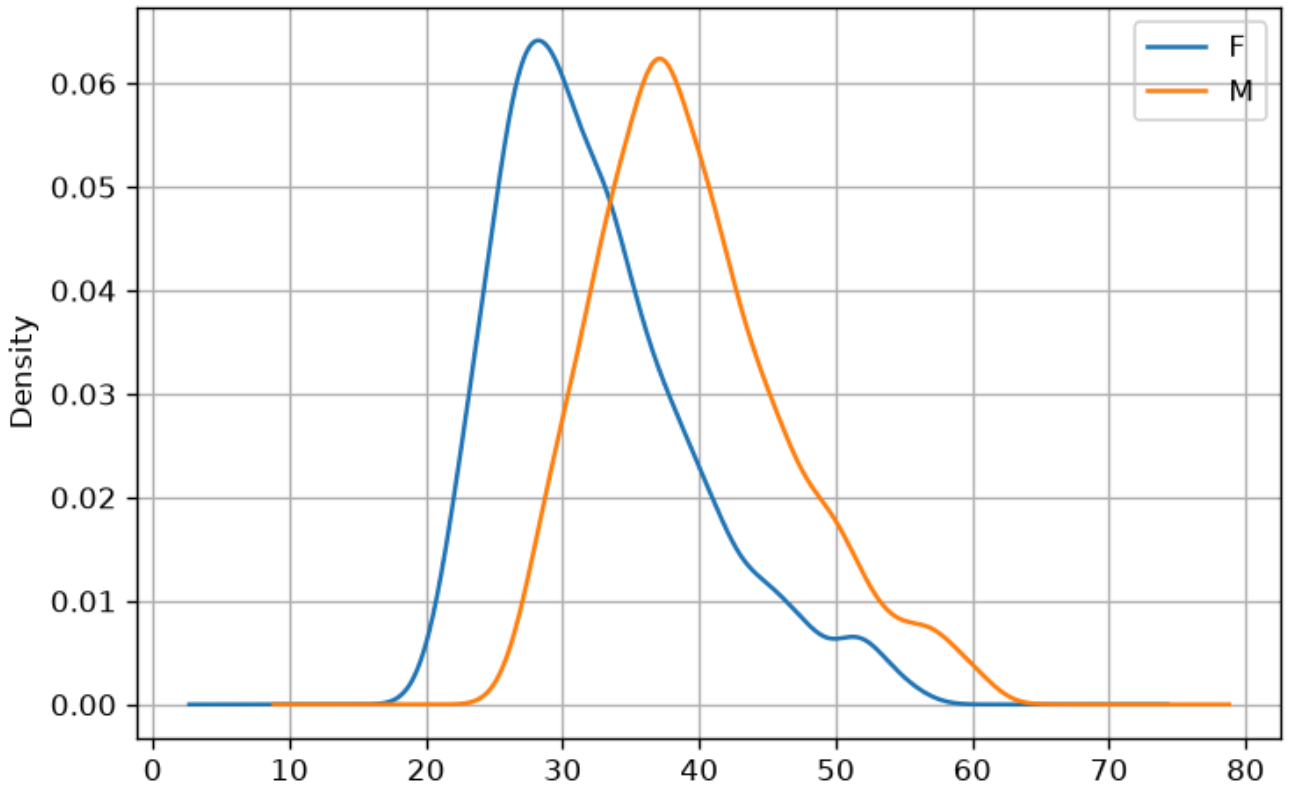
If in the case of histograms changing the number of bins changed the graphical result, here it is changing the bandwidth that changes the graphical result. The following graph shows several examples of density estimation with different bandwidth values:

```
fig, ax = plt.subplots()
hw['weight'].plot.density(figsize=(12,6), bw_method=0.05)
hw['weight'].plot.density(figsize=(12,6), bw_method=0.1)
hw['weight'].plot.density(figsize=(12,6), bw_method=0.2)
hw['weight'].plot.density(figsize=(12,6), bw_method=0.5)
hw['weight'].plot.density(figsize=(12,6), bw_method=0.8)
plt.hist(hw['weight'], density=True)
ax.legend(['h=0.05','h=0.1','h=0.2','h=0.3','h=0.4','histogram'])
plt.grid()
plt.show()
```



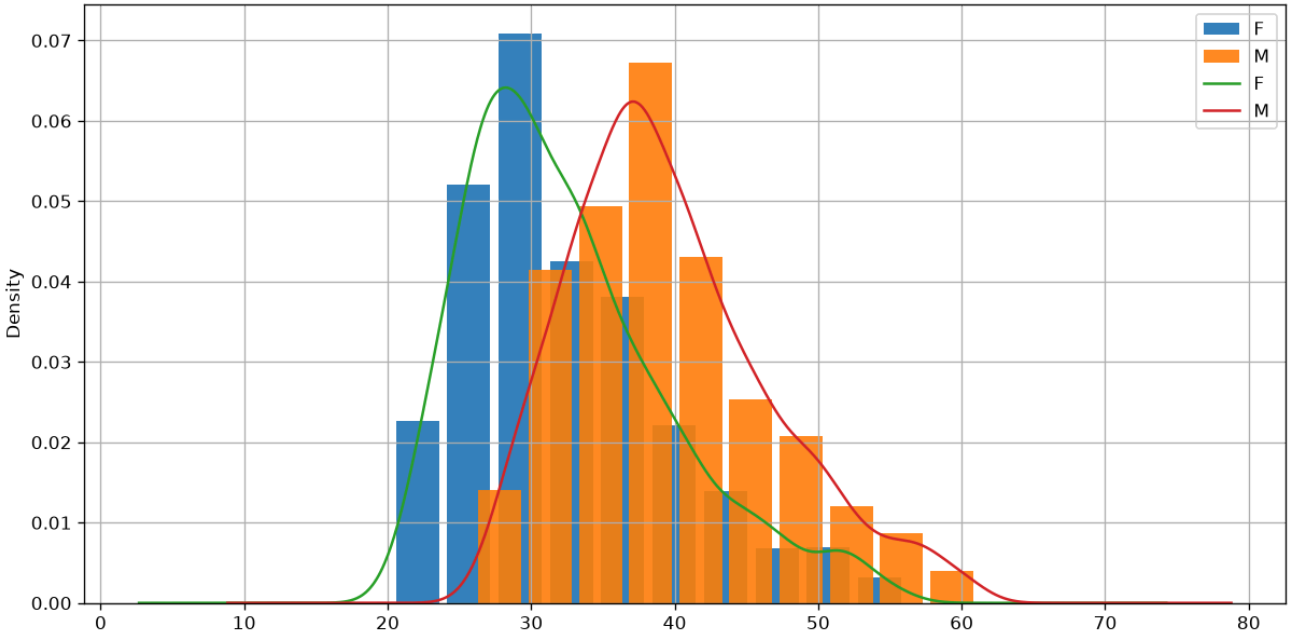
We can compare two samples using density estimations in Pandas very easily as follows:

```
hw.groupby('sex')['weight'].plot.density()
plt.legend()
plt.grid()
plt.show()
```



We can also mix histograms and density plots as follows:

```
hw.groupby('sex')['weight'].plot.hist(width=3, alpha=0.9, figsize=(12,6),
        density=True) # density=True to normalize the histograms
hw.groupby('sex')['weight'].plot.density()
plt.legend()
plt.grid()
plt.show()
```



In this case, we had to specify “density=True” when computing the histogram to make sure that the ranges on the y axes of density estimation and histogram were the same. This leads us to a **density histogram**. We will see the theoretical details of this in a later lecture.

Summary Statistics

For now, we have seen graphical ways to represent the data. In practice, it can be useful to attach some numerical values to the data in order to summarize it.

Descriptive statistics deals with describing, representing, and summarizing a data sample pertaining to a population. The tools of descriptive statistics can be both numerical and graphical. The analyzed data can be described according to various aspects. Therefore, there are various objective “indicators”:

- **sample size**;
- **central indicators**: mean, median, mode;
- **Dispersion indicators**: extremes, range, quantiles, percentiles, quartiles, interquartile range, variance;

We will investigate these tools on the `Titanic` dataset. Let’s load it:

```
import pandas as pd
titanic = pd.read_csv(DATA + "titanic.csv",
                      index_col='PassengerId')
titanic.head()
```

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId											
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

We will start by focusing on the **univariate** sample of ages:

```
age = titanic['Age']
age
```

```
PassengerId
1      22.0
2      38.0
3      26.0
4      35.0
5      35.0
       ...
887    27.0
888    19.0
889     NaN
890    26.0
891    32.0
Name: Age, Length: 891, dtype: float64
```

The sample contains `NaN` values, which will have to be handled appropriately.

Size

The size of a univariate sample $\{x^{(i)}\}_i^N$ is given by the number of values it contains: $|\{x^{(i)}\}_i^N| = N$. The size of each column can also be different as there may be missing values,

generally indicated by `NA` or `NaN`. In our case:

```
age.info()
```

```
<class 'pandas.core.series.Series'>
Index: 891 entries, 1 to 891
Series name: Age
Non-Null Count  Dtype
--------------  -----
714 non-null    float64
dtypes: float64(1)
memory usage: 13.9 KB
```

The summary above indicates that, although we have 891 values, only 714 of these are non-null. These are the "real" size of the sample.

Measures of central tendency

Central measures give an approximate idea of the order of magnitude of the sample values.

Mean

The mean of a sample is defined as the sum of its values divided by its size:

$$\overline{X} = \frac{1}{N} \sum_{i}^{N} x^{(i)}$$

The mean of our sample will be:

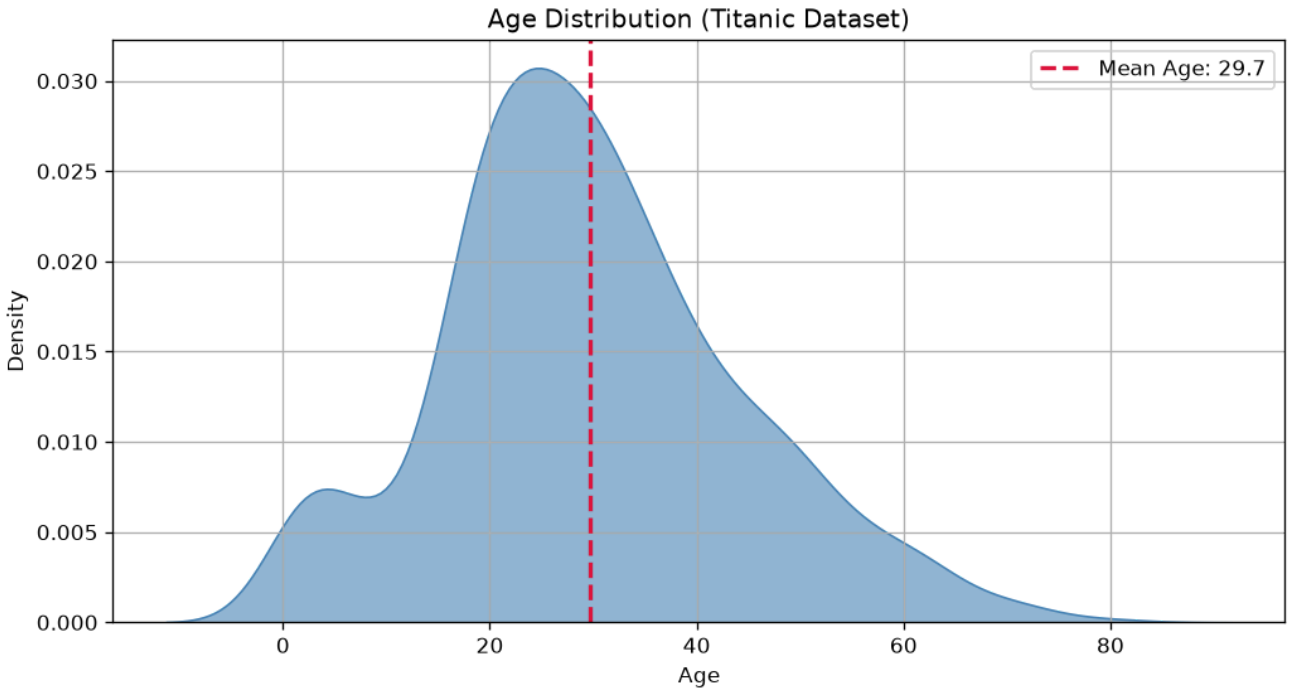
```
age.mean()
```

```
np.float64(29.69911764705882)
```

Note that the mean function will automatically ignore the missing data.

This statistic is very easy to interpret: the average age is about 30 years.

Let's see where this descriptive statistic stands in a graphical representation of the density of the sample:



Median

When the elements of a sample can be ordered (for example, if they are numerical values), the median of a sample (or the median element) is the element that divides the ordered set of sample values into two equal parts.

The median element can be obtained by ordering the sample values and proceeding as follows:

- If the number of elements is odd, take the central element. For example $[1, 2, \mathbf{2}, 3, 5] \rightarrow 2$.
- If the number of elements is even, take the average of the two central ones. For example $[1, 2, \mathbf{2}, \mathbf{3}, 3, 5] \rightarrow \frac{2+3}{2} = 2.5$.

In the case of our sample:

```
print("Median value:",age.median())
```

```
Median value: 28.0
```

From a formal point of view, if we have n observations $x^{(1)}, \ldots, x^{(n)}$, which can be ordered as $x^{(i_i)}, \ldots, x^{(i_n)}$, the calculation of the median can be expressed as follows:

$$\tilde{x}_{0.5} = \begin{cases} x^{(n+1)/2} & \text{if } n \textit{ is odd} \\ \frac{1}{2}(x^{(n/2)} + x^{(n/2+1)}) & \text{otherwise.} \end{cases}$$

To be convinced of the above formulation, we can see two toy examples:

```
odd_sample = pd.Series([1,2,3,4,100])
print(odd_sample.sort_values())
odd_sample.median()
```

```
0      1
1      2
2      3
3      4
4    100
dtype: int64
```

```
np.float64(3.0)
```

The median is the central element 3.

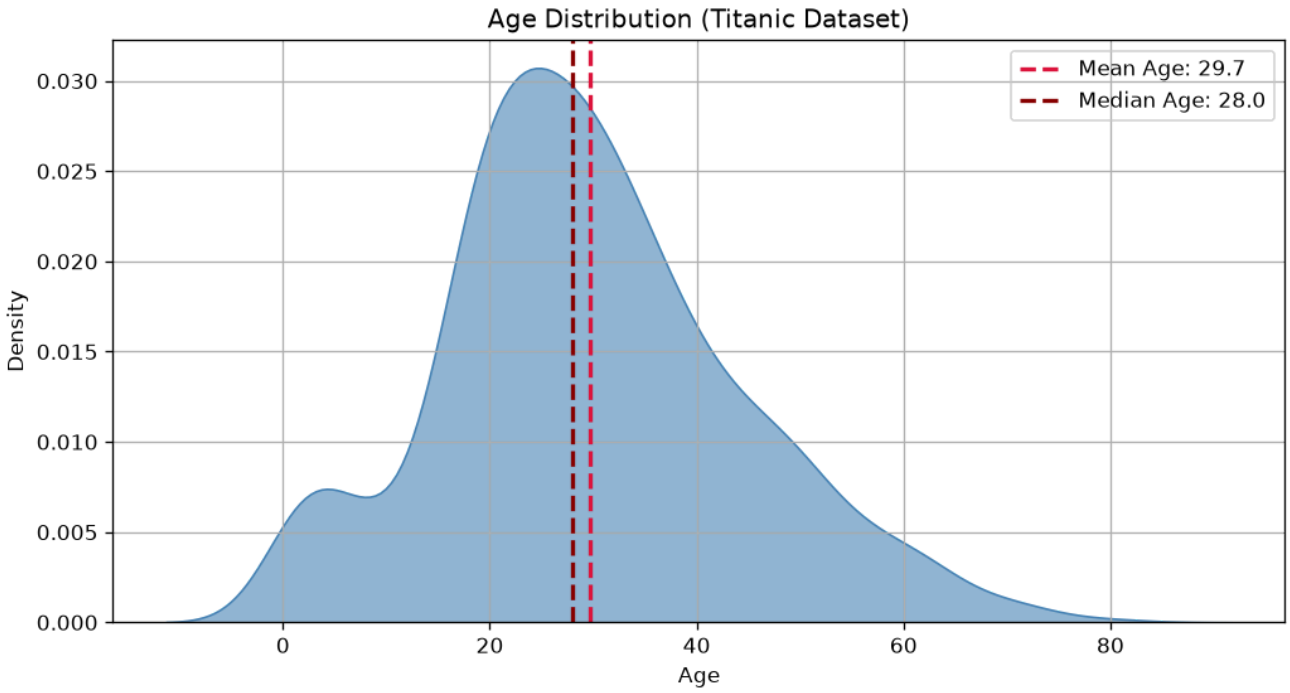
```
even_sample = pd.Series([1,2,3,4,5,100])
print(even_sample.sort_values())
even_sample.median()
```

```
0      1
1      2
2      3
3      4
4      5
5    100
dtype: int64
```

```
np.float64(3.5)
```

The median is the average between the two central elements 3 and 4.

Let's see where this descriptive statistic stands in a graphical representation of the density of the sample:



As we can see, depending on the density of the data, the two values are not always identical.

Quantiles, Percentiles, and Quartiles

Quantiles, percentiles, and quartiles generalize the concept of the median.

Quantiles

A quantile of order α is a value q_α that divides a sample into two parts with sizes proportional to α and $1-\alpha$. Values less than or equal to q_α belong to the first part of the division, while values greater than q_α belong to the second part.

For example, given the already ordered sample `[1,2,3,3,4,5,6,6,7,8,8,9]`, a quantile $q_{0.25}$ will divide the sample into two parts with sizes proportional to 0.25 and $1-0.25=0.75$. In this case $q_{0.25}=3$ and the two parts will be `[1,2,3,3]` and `[4,5,6,6,7,8,8,9]`.

In this case too, as with the median, averages of adjacent values are taken where appropriate.

Quantiles should be interpreted as follows:

> If a quantile of order α is equal to the number x, then it means that $\alpha \times n$ elements have a value less than or equal to x, where n is the number of elements in the sample.

It should be noted that:

- The minimum is a quantile of order 0;

- The maximum is a quantile of order 1;
- The median is a quantile of order 0.5.

Let's see some examples on our sample:

```
print("Order 0 Quantile (minimum):", age.quantile(0))
print("Order 0.5 Quantile (median):", age.quantile(0.5))
print("Order 1 Quantile (maximum):", age.quantile(1))
print("Order 0.15 Quantile:", age.quantile(0.15))
```

```
Order 0 Quantile (minimum): 0.42
Order 0.5 Quantile (median): 28.0
Order 1 Quantile (maximum): 80.0
Order 0.15 Quantile: 17.0
```

From the above data we deduce that:

- 50% of the passengers are younger than 28.
- 15% of the passengers are younger than 17.

Percentiles

Percentiles are simply quantiles expressed as a percentage. A quantile of order 0.25 is a percentile of order 25%. In practice, we still use the quantile function to compute them:

```
age.quantile(25/100)
```

```
np.float64(20.125)
```

Quartiles

Quartiles are specific quantiles that divide the sample into four parts. In particular:

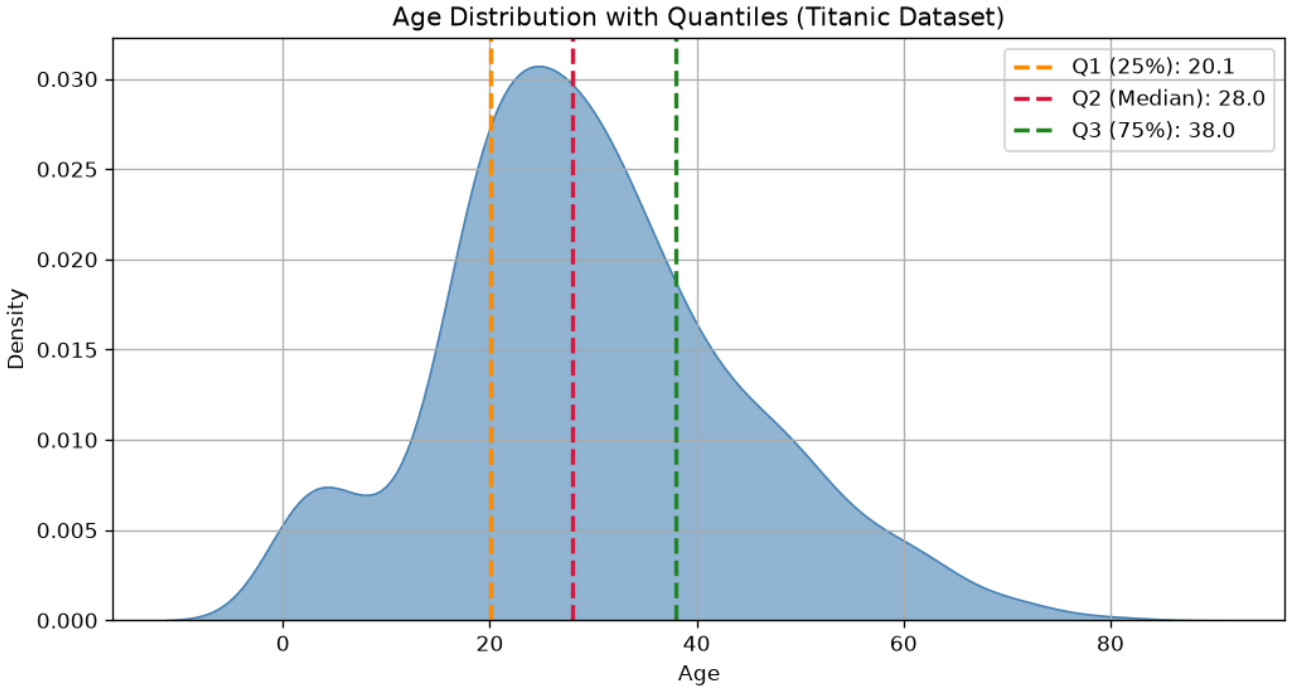
- The quartile of order 0 is a quantile of order 0;
- The quartile of order 1 is a quantile of order $1/4 = 0.25$;
- The quartile of order 2 is a quantile of order $2/4 = 0.5$;
- The quartile of order 3 is a quantile of order $3/4 = 0.75$;
- The quartile of order 4 is a quantile of order $4/4 = 1$.

Also in this case, we use the quantile function:

```
print("Quartile of order 0 (minimum):", age.quantile(0/4))
print("Quartile of order 1:", age.quantile(1/4))
print("Quartile of order 2 (median):", age.quantile(2/4))
print("Quartile of order 3:", age.quantile(3/4))
print("Quartile of order 4 (maximum):", age.quantile(4/4))
```

```
Quartile of order 0 (minimum): 0.42
Quartile of order 1: 20.125
Quartile of order 2 (median): 28.0
Quartile of order 3: 38.0
Quartile of order 4 (maximum): 80.0
```

Let's see this in graphical terms:



Mode

The **mode** of a sample is the value that appears **most frequently** within the data. It represents the most common observation. It can se computed as follows:

```
age.mode()
```

```
0    24.0
Name: Age, dtype: float64
```

In formal terms, the mode $\overline{x}_M$ of a sample is given by:

$$\overline{x}_M = a_j \Leftrightarrow n_j = \max\{n_1, \ldots, n_k\}$$

Where:

- a_j are the unique values of the sample (values without repetition)
- n_j are their respective absolute frequencies (how many times a value repeats)

Let's see this with a toy example:

Let's see how to compute it manually, with a toy example:

```
data=pd.Series([1,2,3,4,2,5,4,2,6,5,8,4,3,2,3])
data
```

```
0     1
1     2
2     3
3     4
4     2
5     5
6     4
7     2
8     6
9     5
10    8
11    4
12    3
13    2
14    3
dtype: int64
```

The unique values a_j with the respective absolute frequencies n_j (counts) are computed as:

```
print(data.value_counts())
```

```
2    4
3    3
4    3
5    2
1    1
6    1
8    1
Name: count, dtype: int64
```

In the output above, the first column represents the unique values a_j, while the second column represents the absolute frequencies n_j. Since the list is already sorted by counts, we can extract the maximum as follows:

```
print("Mode:\n",data.value_counts().index[0])
```

```
Mode:
 2
```

The mode is 2 as this number repeats most times (it repeats 4 times).

In graphical terms:

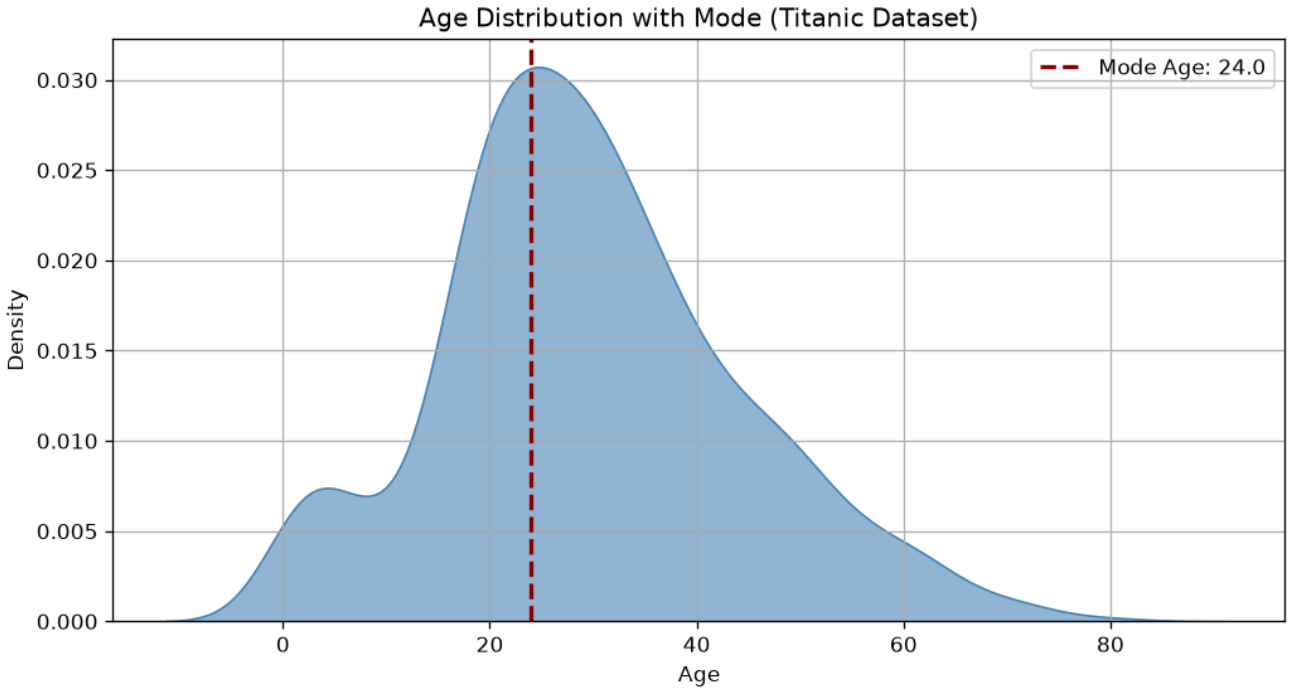
```
import seaborn as sns
import matplotlib.pyplot as plt

# Drop missing ages
ages = titanic['Age'].dropna()

# Compute mode (may return multiple values, so we take the first)
mode_age = ages.mode().iloc[0]

# Plot density
plt.figure(figsize=(10,5))
sns.kdeplot(ages, fill=True, color='steelblue', alpha=0.6)
plt.axvline(mode_age, color='darkred', linestyle='--', linewidth=2, label=f'Mode Age:
        {mode_age:.1f}')

# Customize plot
plt.title('Age Distribution with Mode (Titanic Dataset)')
plt.xlabel('Age')
plt.ylabel('Density')
plt.legend()
plt.grid()
plt.show()
```



The mode is the “peak” of the density of the data as it represents the most common value.

Measures of Dispersion

Measures of dispersion serve to quantify to what extent the values of a distribution are “dispersed”, or “far from each other”.

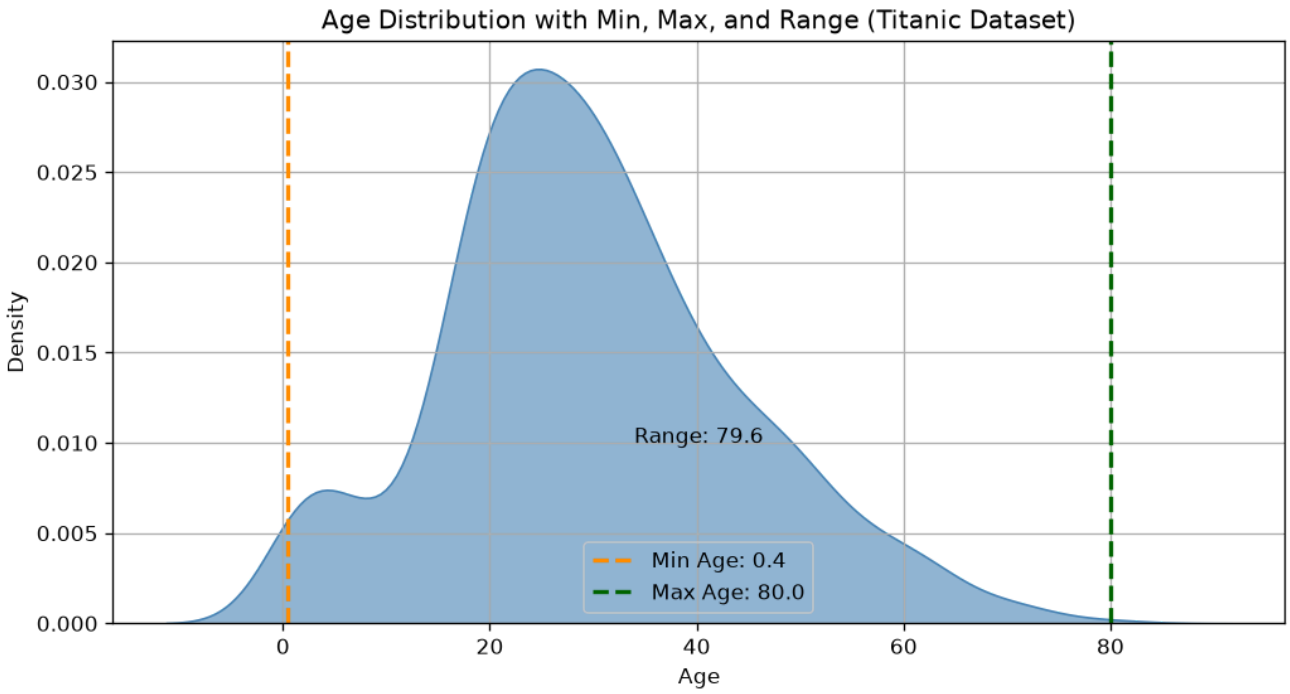
Minimum, Maximum, and Range

Simple indices of dispersion are the minimum ($min\{x^{(i)}\}_i^N$), the maximum ($max\{x^{(i)}\}_i^N$), and the range ($max\{x^{(i)}\}_i^N$ - $min\{x^{(i)}\}_i^N$). These are easily computed as follows:

```
print("Min value:",age.min())
print("Max value:",age.max())
print("Range:",age.max()-age.min())
```

```
Min value: 0.42
Max value: 80.0
Range: 79.58
```

Let’s see this graphically:



We can see that there is some density before 0 because the density estimation function “interpolates” what may happen before the zero if we had data there. This is an artifacts of the continuous nature of this visualization.

Interquartile Range (IQR)

The range is not a very robust measure of dispersion, as it does not take into account the presence of any outliers. Consider, for example, the following “artificial” samples:

```
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
sample1 = np.linspace(3,7,10)
sample2 = sample1.copy()
sample2[0]=-5 # we replace two values with the outliers -5
sample2[-1]=15 # and 15

df = pd.DataFrame({'s1':sample1,'s2':sample2})
print(df)

# df.plot.box()
# plt.grid()
# plt.show()

print("Range sample 1:",df['s1'].max()-df['s1'].min())
print("Range sample 2:",df['s2'].max()-df['s2'].min())
```

```
         s1         s2
0  3.000000  -5.000000
1  3.444444   3.444444
2  3.888889   3.888889
3  4.333333   4.333333
4  4.777778   4.777778
5  5.222222   5.222222
6  5.666667   5.666667
7  6.111111   6.111111
8  6.555556   6.555556
9  7.000000  15.000000
Range sample 1: 4.0
Range sample 2: 20.0
```

The samples are similar, but the presence of two outliers (-5 and 15) in the second sample makes the ranges very different (4 and 20).

A **more expressive measure of dispersion** is therefore the **interquartile range** (or **interquartile distance**), which is measured as the difference between the third and first quartiles:

```
q11,q13 = df['s1'].quantile([1/4,3/4])
q21,q23 = df['s2'].quantile([1/4,3/4])
print("The interquartile range of sample 1 is:",q13-q11)
print("The interquartile range of sample 2 is:",q23-q21)
```

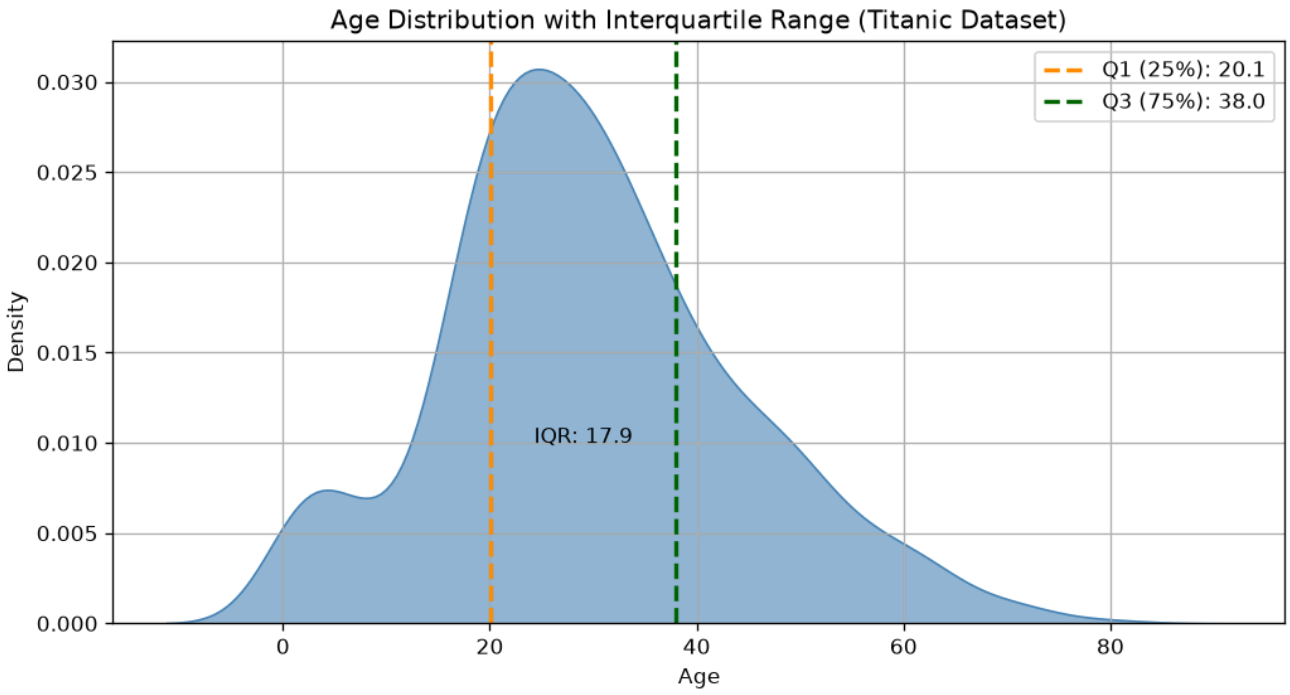
```
The interquartile range of sample 1 is: 2.0
The interquartile range of sample 2 is: 2.0
```

Let's see how to compute the IQR in the case of the Age sample:

```
q1,q3 = age.quantile([1/4,3/4])
print("The interquartile range of age is:",q3-q1)
```

```
The interquartile range of age is: 17.875
```

Graphically:



Variance and Standard Deviation

Variance (also known as **mean squared deviation**) provides an estimate of how much the observed data deviates from the mean. Variance calculates the mean of the squares of the deviations of the values from the mean, penalizing large deviations from the mean value (due to outliers) more heavily than small deviations:

$$s_n^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}$$

It can be shown that, when computing variance from smaller samples, it can lead to **biased estimates**. Hence, it is very common to use the **Sample variance** with $n-1$ in the denominator:

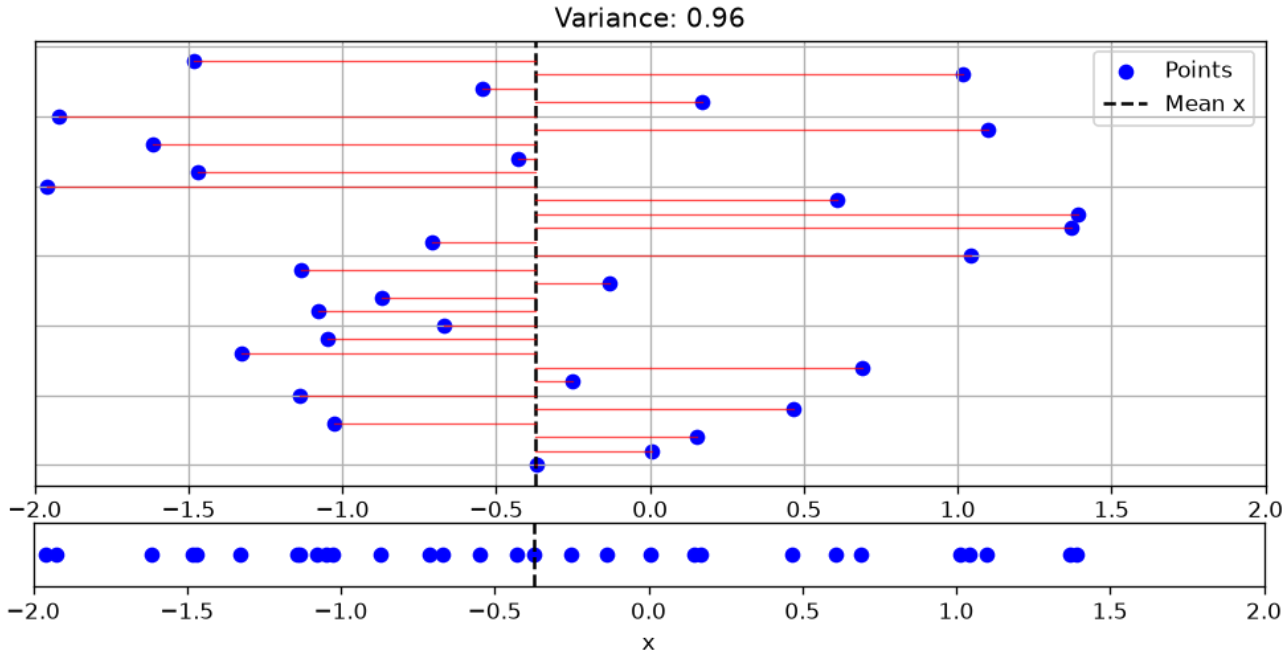
$$s_{n-1}^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}$$

We will see it better later in the course, but for now consider that:

- The variance formula can be used when we have access to the whole population;
- The sample variance formula should be used in all other cases;
- When sample size is large enough, the two give very similar result (for large values of n: $n-1 \approx n$).

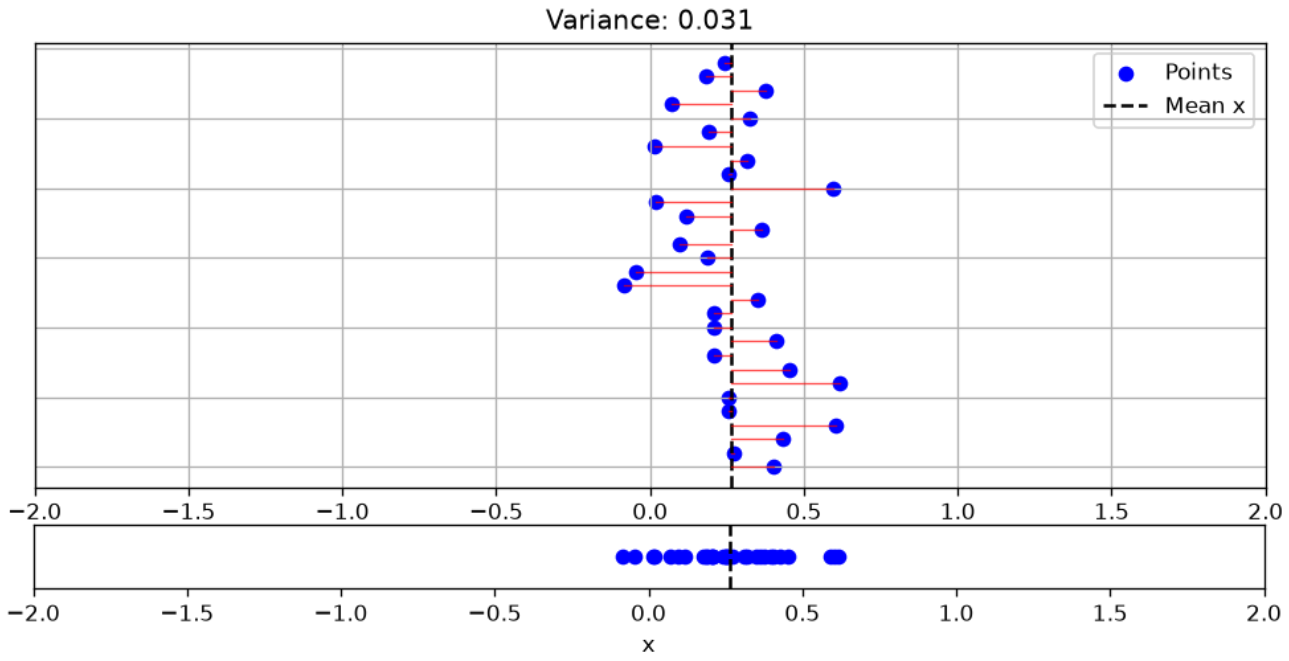
Here we use the symbol s instead of the symbol σ seen in the case of probability distributions, because s is a value "estimated" from the data and not "theoretical" as in the case of σ. It should be noted, however, that in practice, there is no universal convention, so in many cases σ is also used for estimated values.

We can attribute a geometric meaning to variance, as shown in the graph below:



In the graph, the bottom plot shows a univariate sample $\{x_i\}_i^N$. The top plot shows the same sample "exploded" on the y-axis for visualization purposes. In the top plot, the black dashed line indicates the sample mean, while the red lines are the terms $(x_i - \overline{x})$ that appear in the variance formula. The variance calculates the average of the lengths of these segments.

The following graph shows a less "dispersed" sample:



The variance of age will be:

```
age.var()
```

```
np.float64(211.01912474630802)
```

The dispersion indices seen so far (excluding variance) have the same **unit of measurement** as the input data. In the case of ages, the data are measured in years. It is therefore correct to say that **minimum**, **maximum**, **range**, **interquartile range**, and **mean absolute deviation** calculated on weights are measured in years.

The same does not apply to the variance, which will be measured in **years squared**. If we want to obtain a **commensurable** measure of dispersion, we can calculate the square root of the variance, thus obtaining the **standard deviation** (or **root mean square deviation**), which is defined as follows:

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

Similarly, we often use the **sample standard deviation**:

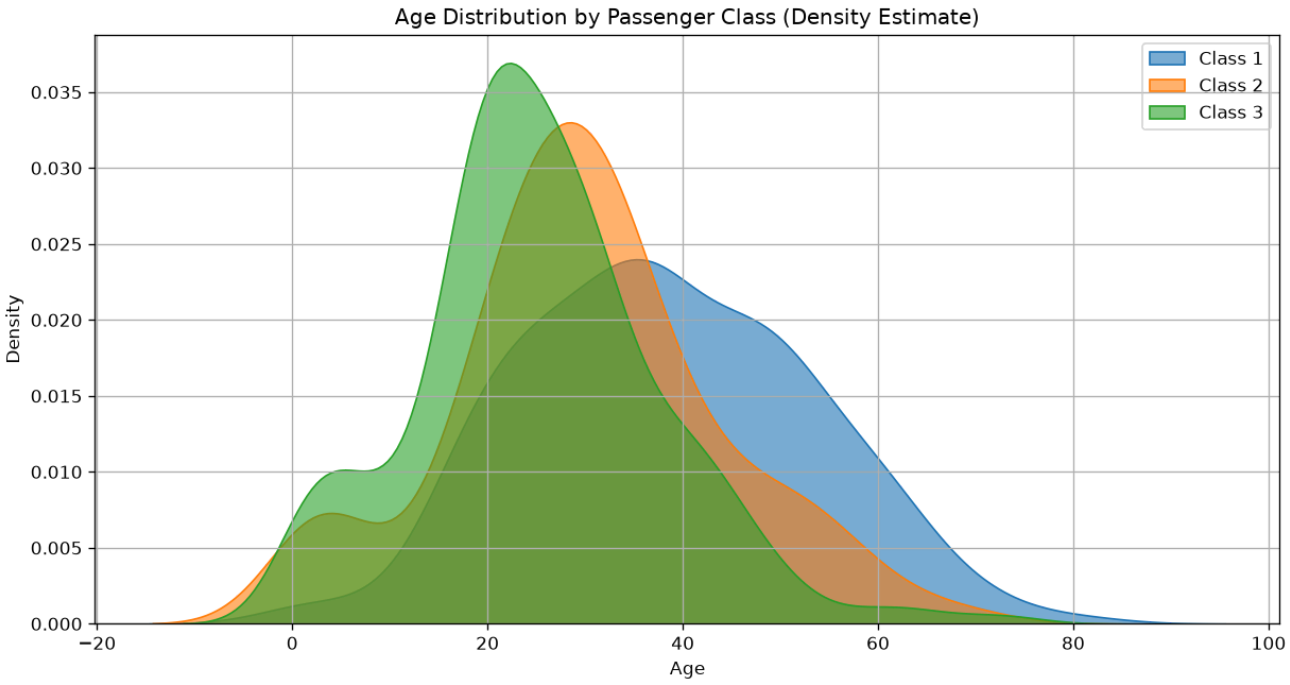
$$s_{n-1} = \sqrt{s_{n-1}^2} = \sqrt{\frac{\sum_{i=1}(x_i - \overline{x})^2}{n-1}}$$

We can compute the standard deviation as follows:

```
age.std()
```

```
np.float64(14.526497332334042)
```

Let's see how these measures of dispersion relate to distributions with different shapes. We will consider the ages of passengers in the three classes of the Titanic:




```
Class 1 ages:
Mean: 38.233440860215055
Min: 0.92
Max: 80.0
Range: 79.08
IQR: 22.0
Variance: 219.12454269107815
Std: 14.80285589645046

Class 2 ages:
Mean: 29.87763005780347
Min: 0.67
Max: 70.0
Range: 69.33
IQR: 13.0
Variance: 196.0301519088587
Std: 14.0010768124762

Class 3 ages:
Mean: 25.14061971830986
Min: 0.42
Max: 74.0
Range: 73.58
IQR: 14.0
Variance: 156.1349764510225
Std: 12.495398210982414
```

We can see a correspondence between the graphical forms of the distributions and the dispersion measures. Indeed, passengers in class 3 are “less dispersed” and “younger” than in the other classes.

Data Normalizations

The observed data dispersion indicators strongly depend on the nature of the data and their unit of measurement. For example, ages are measured in years, while weights are in Kg or pounds. Therefore, there are data normalization techniques that make data based on different units of measurement comparable to each other.

Normalization between 0 and 1

This normalization scales the data so that the minimum and maximum values are exactly equal to 0 and 1, using the following formula:

$$x_i^{norm} = (x_i - x_{min})/(x_{max} - x_{min})$$

In the case of weights:

```
weights_norm = (weights-weights.min())/(weights.max()-weights.min())
weights_norm
```

```
0       0.808081
1       0.429293
2       0.353535
3       0.378788
4       0.343434
          ...
4226    0.080808
4227    0.434343
4228    0.353535
4229    0.191919
4230    0.050505
Name: weight, Length: 4231, dtype: float64
```

Normalization between -1 and 1

In this case, the data are rescaled so that the new minimum and maximum are -1 and 1, using the following formula:

$$x_i^{norm} = (x_{max} + x_{min} - 2 \cdot x_i)/(x_{max} - x_{min})$$

We can perform this transformation in Pandas as follows:

```
weights_norm2 = (weights.max()+weights.min()-2*weights)/(weights.max()-weights.min())
weights_norm2
```

```
0      -0.616162
1       0.141414
2       0.292929
3       0.242424
4       0.313131
          ...
4226    0.838384
4227    0.131313
4228    0.292929
4229    0.616162
4230    0.898990
Name: weight, Length: 4231, dtype: float64
```

Standardization (z-scoring)

In many cases, it is useful to normalize data so that they have zero mean and unit standard deviation. This type of normalization is called "z-scoring" and is performed by subtracting the mean from the data and dividing by the standard deviation.

$$z_i = \frac{x_i - \overline{x}}{s_x}$$

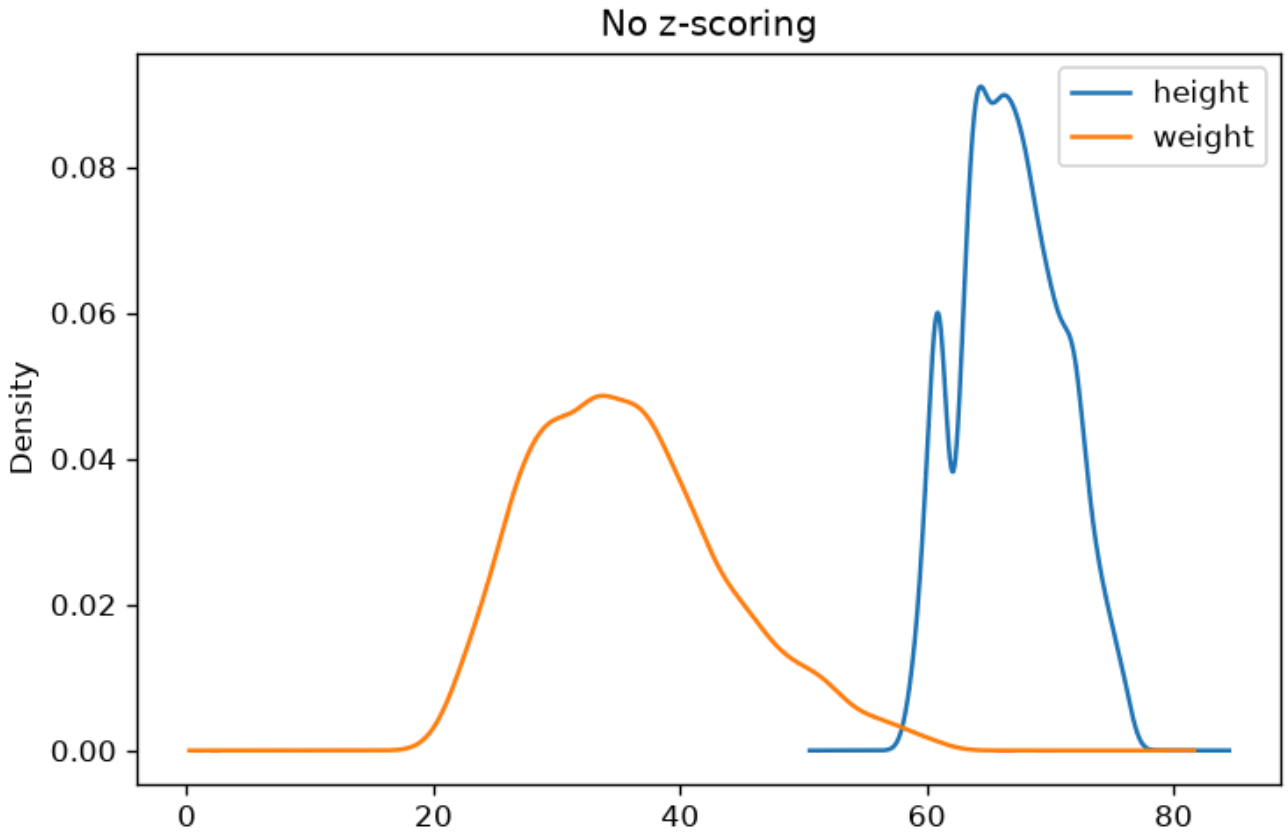
where s_x is the standard deviation of the population to which X belongs. Note that zeta scores are **dimensionless** (i.e., they have no unit of measurement).

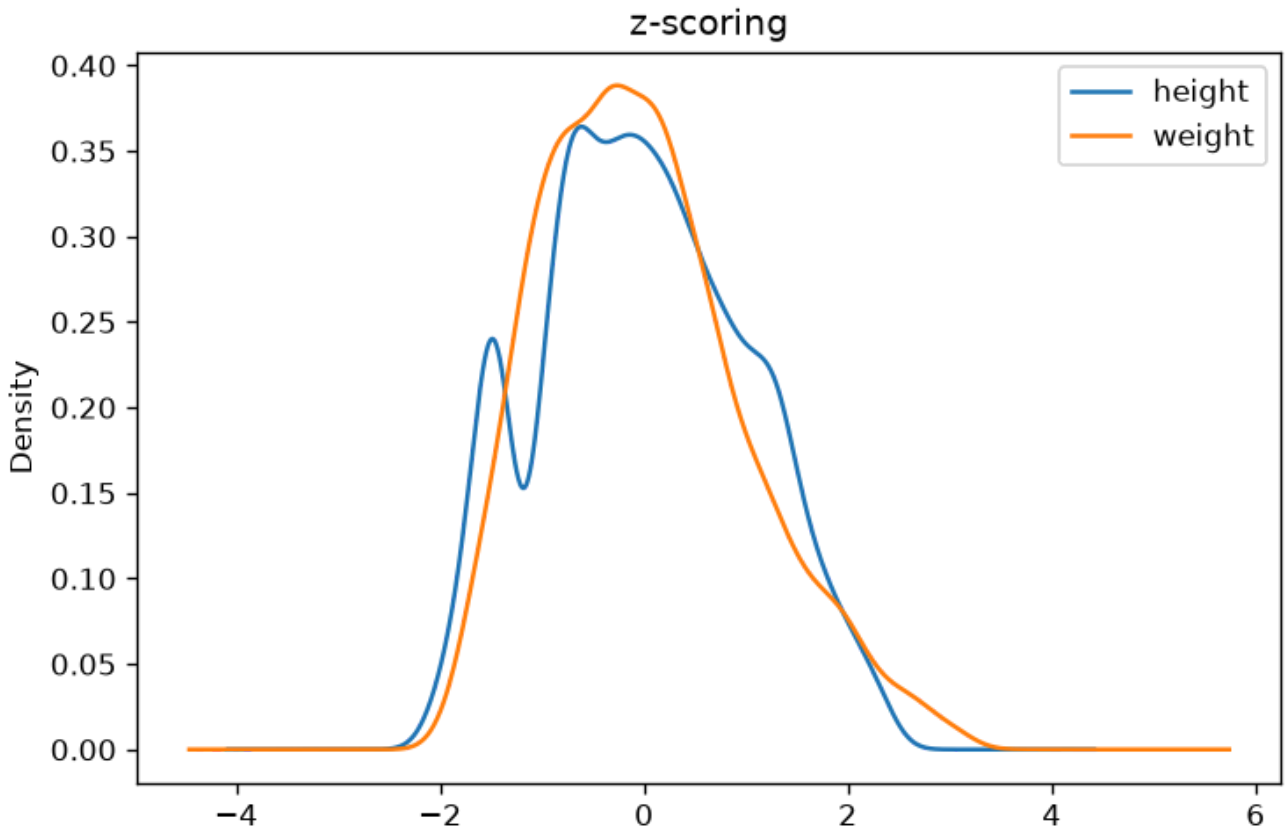
To understand the effect of this normalization, let's observe the density estimates of the samples before and after normalization:

```
from scipy.stats import zscore
hwz = hw.copy()
hwz['height'] = zscore(hwz['height'])
hwz['weight'] = zscore(hwz['weight'])

hw.plot.kde()
plt.title('No z-scoring')
plt.show()

hwz.plot.kde()
plt.title('z-scoring')
plt.show()
```





Shape Indicators

Let's now look at some indicators that allow us to get an idea of certain aspects of the "shape" of data distribution.

Asymmetry (skewness)

Skewness is an indicator of the "imbalance" to the left (negative value) or to the right (positive value) of a data sample with respect to the central value. The formula for skewness is as follows:

$$\sum_{i}^{n} \frac{(x_i - \overline{x})^3}{n \cdot s_x^3}$$

The skewness values will be:

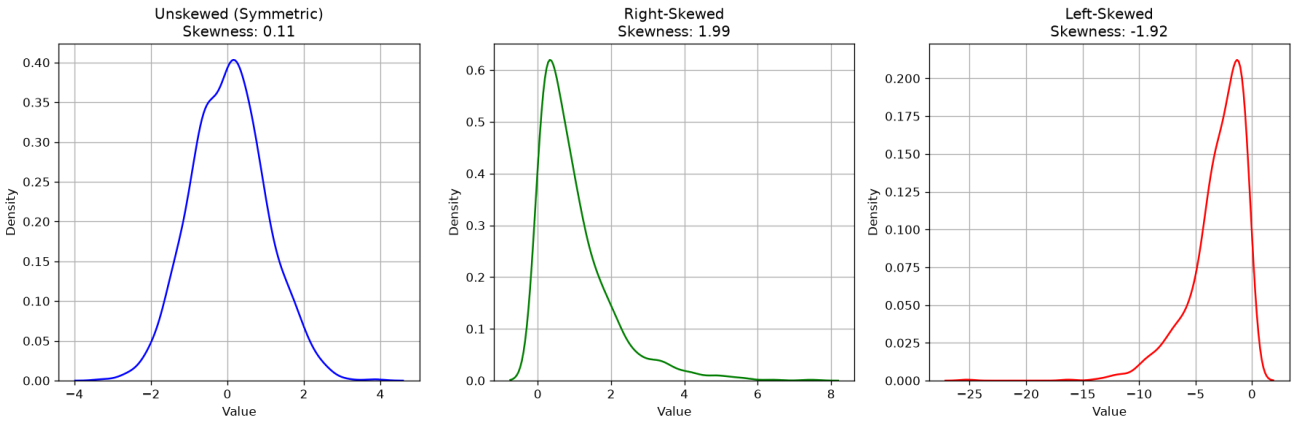
- **Negative** if the distribution is skewed to the left;
- **Positive** if the distribution is skewed to the right;
- **Close to zero** in the case of unskewed distributions.

We can compute the skewness as follows

```
age.skew()
```

```
np.float64(0.38910778230082704)
```

Let's look at some examples of distributions and skewness:



Kurtosis (kurtosis)

The kurtosis index measures the "thickness" of the tails of a density distribution. It is defined as follows:

$$K = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{X_i - \bar{X}}{s} \right)^4 - 3$$

The index is interpreted as follows:

- If it is greater than zero, the distribution is *leptokurtic*, i.e., more "peaked" than a Normal distribution;
- If it is less than zero, the distribution is *platykurtic*, i.e., more "flat" than a Normal distribution;

- If it is equal to zero, the distribution is *mesokurtic*, i.e., the tails are similar to those of a normal distribution.

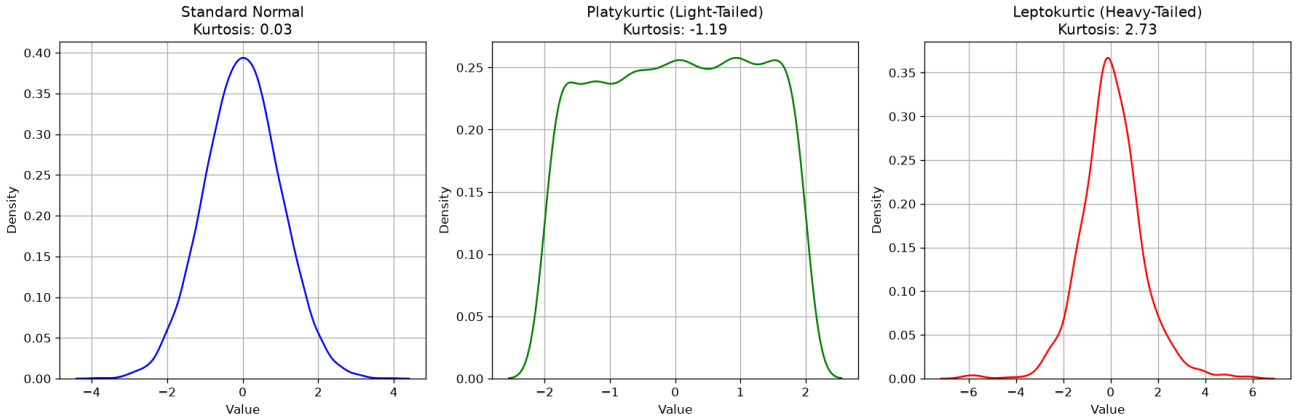
We will see what a normal distribution is later in the course.

We can compute the kurtosis as follows:

```
age.kurtosis()
```

```
np.float64(0.17827415364210353)
```

Let's look at some examples of Kurtosis values:



Statistical Summary

When we compute values like **mean**, **median**, **mode**, **minimum**, **maximum**, **range**, **standard deviation**, **variance**, and **interquartile range**, we're building what is known as a **statistical summary** of a dataset. These measures help us understand the **central tendency**, **spread**, and **shape** of a distribution—whether it's symmetric, skewed, concentrated, or dispersed.

In practice, we can obtain many of these summary statistics quickly using the `.describe()` method in **pandas**:

```
age.describe()
```

```
count    714.000000
mean      29.699118
std       14.526497
min        0.420000
25%       20.125000
50%       28.000000
75%       38.000000
max       80.000000
Name: Age, dtype: float64
```

This returns:

- **count**: number of non-missing values
- **mean**: average value
- **std**: standard deviation
- **min** and **max**: smallest and largest values
- **25%**, **50%**, **75%**: the first quartile (Q1), median (Q2), and third quartile (Q3)

If you want to include other measures like mode or variance, you can compute them separately:

```
print(f"Mode (most frequent age): {age.mode()[0]:.1f}")
print(f"Variance: {age.var():.2f}")
print(f"IQR (Interquartile Range): {age.quantile(0.75) - age.quantile(0.25):.2f}")
```

```
Mode (most frequent age): 24.0
Variance: 211.02
IQR (Interquartile Range): 17.88
```

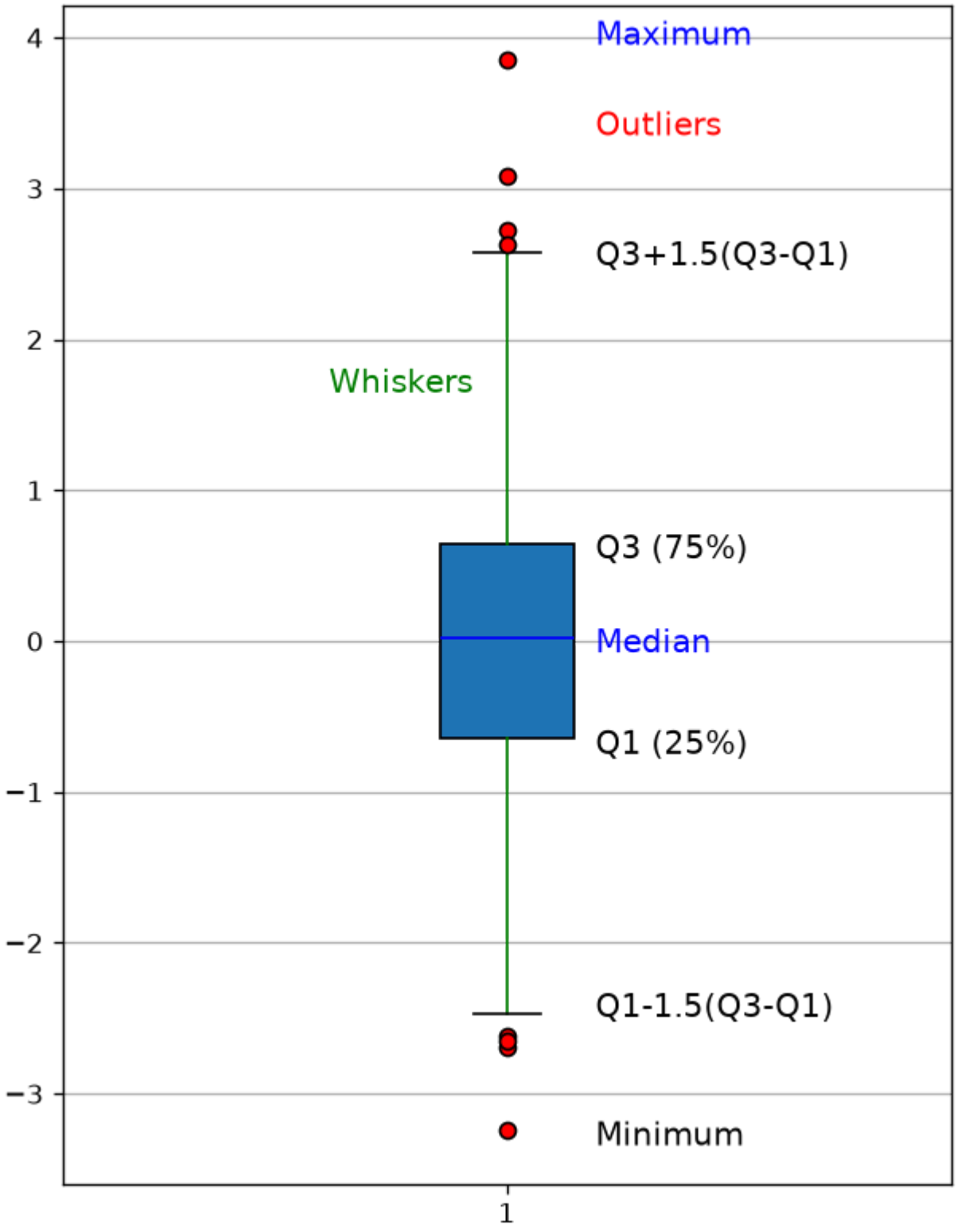
These tools are essential for both exploratory data analysis and communicating insights clearly—whether you're comparing groups, detecting outliers, or preparing data for modeling.

Boxplots

Boxplots are a compact visualization method for representing certain descriptive characteristics of the data under analysis. In particular, given a sample, a boxplot can effectively represent the following quantities:

- Median value;
- First and third quartile;
- Minimum and maximum (depending on the boxplot's "version," as discussed below).

A boxplot appears as follows:



A **boxplot** is a graphical summary of a dataset that displays its distribution using a rectangular "box" and two "whiskers." Specifically:

- **Bottom edge of the box**: marks the first quartile (Q1), the 25th percentile of the data.
- **Top edge of the box**: marks the third quartile (Q3), the 75th percentile.
- **Horizontal line inside the box**: represents the median (Q2), the 50th percentile.
- **Lower whisker**: extends to the smallest data point that is greater than or equal to (Q1 - 1.5 (Q3 - Q1)).

- **Upper whisker**: extends to the largest data point that is less than or equal to (Q3 + 1.5 (Q3 - Q1)).
- **Individual points beyond the whiskers**: represent outliers—values that fall outside the range defined by the whiskers.

To illustrate the usefulness of boxplots, we will consider the dataset of weights and heights seen previously:

```
hw
```

	sex	height	weight
0	M	74	53.484771
1	M	70	38.056472
2	F	61	34.970812
3	M	68	35.999365
4	F	66	34.559390
...	...	...	...
4226	F	69	23.862436
4227	M	69	38.262182
4228	F	64	34.970812
4229	F	64	28.388071
4230	F	61	22.628172

4231 rows × 3 columns

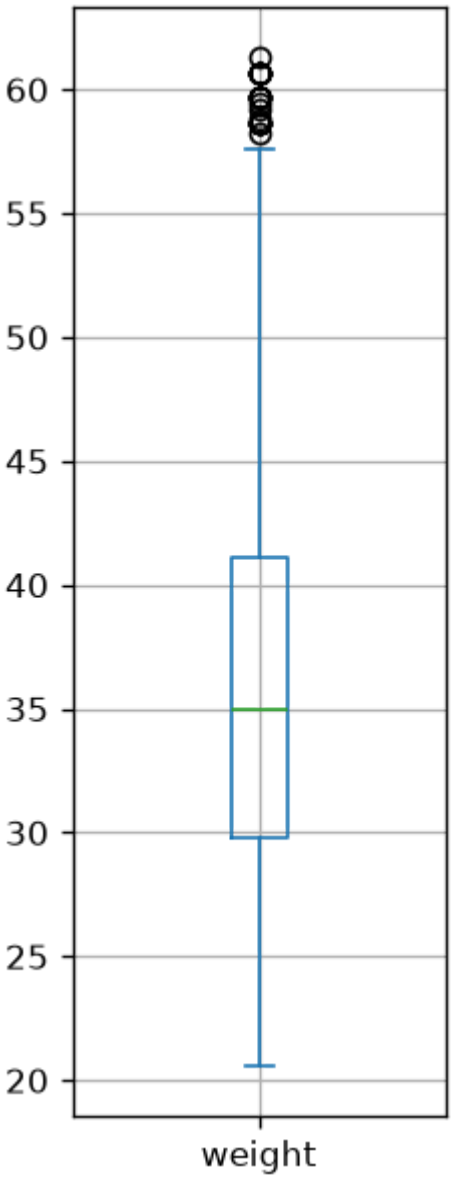
Let's choose the "weight" variable and show the **statistical summary**, that is, the list of all the descriptive statistics indicators discussed so far:

```
hw['weight'].describe()
```

```
count    4231.000000
mean       35.818062
std         7.987908
min        20.571066
25%        29.828045
50%        34.970812
75%        41.142132
max        61.301776
Name: weight, dtype: float64
```

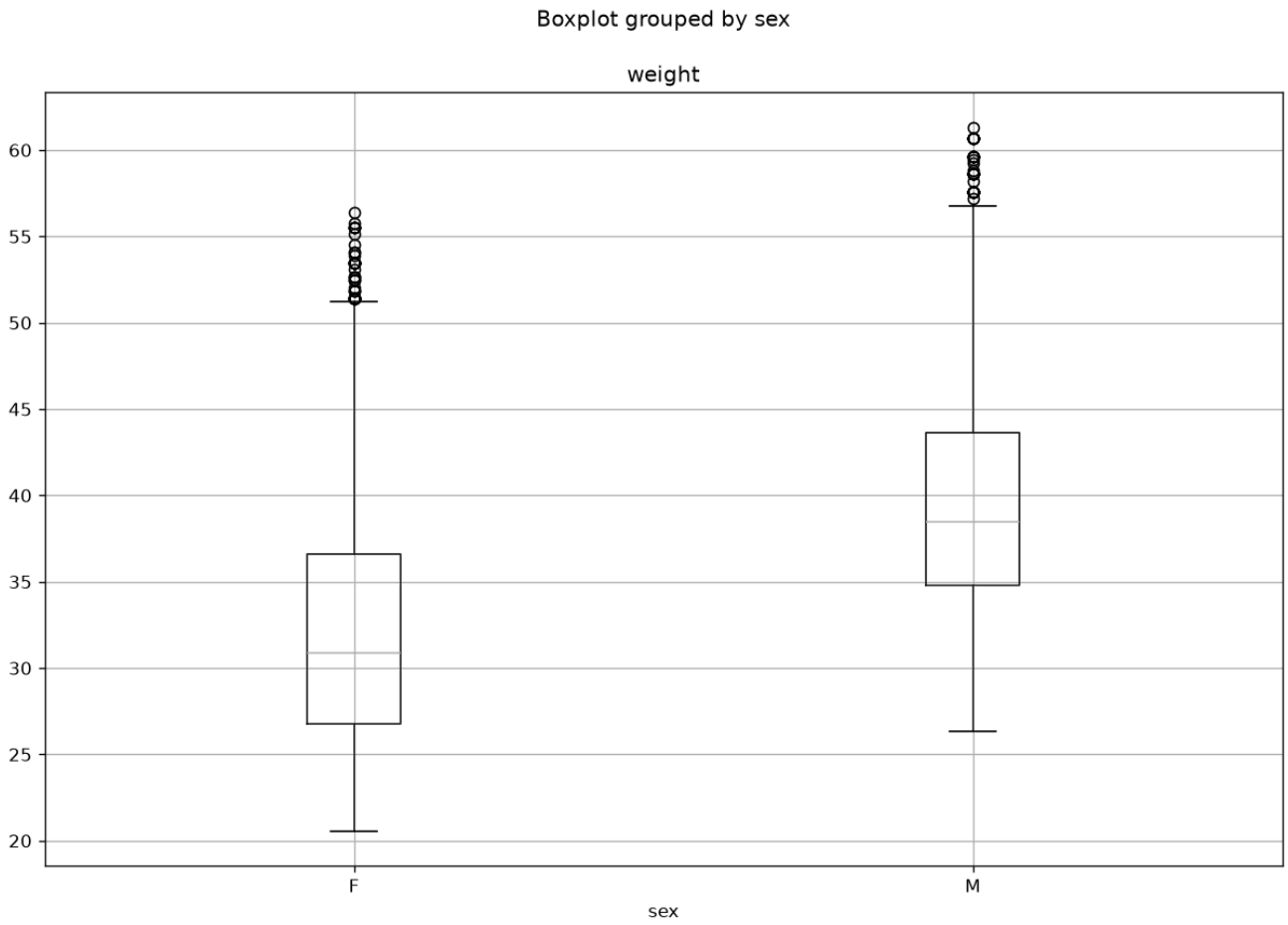
The boxplot of the weights is computed as follows:

```
from matplotlib import pyplot as plt
hw['weight'].plot.box(figsize=(2,6))
plt.grid()
plt.show()
```



Boxplots can be useful to compare samples. For example, the following boxplots compare the distributions of weights between men and women:

```
hw.boxplot(column='weight', by='sex', figsize=(12,8))
plt.show()
```



From the graph above, we can note that weight has different distributions in males and females, with males overall weighing more, as one would expect.

Exercises

Exercise 1

Considering the Titanic dataset, show the bar charts of absolute and relative frequencies of the values in the `Pclass` column. Randomly choosing among the passengers, what is the probability that they embarked in second class?

Exercise 2

Considering the Titanic dataset, show the histogram of the ages of passengers embarked in first class. Use an appropriate criterion to choose the number of bins.

Exercise 3

Modify the plot from the previous exercise to construct a histogram that answers the question "if I randomly choose a passenger boarded in first class, what is the probability that their age is between 20 and 30 years?"

Exercise 4

Considering the Titanic dataset, compare the cumulative age distributions of passengers embarked in the various classes. Are the distributions similar? In which class are the youngest subjects embarked?

Exercise 5

Considering the Titanic dataset, show with a stacked bar plot the percentages of passengers belonging to the three classes separately for the two sexes. Are there any differences in the distribution?

Exercise 6

Considering the Titanic dataset, calculate for each variable the count, mean, standard deviation, minimum, maximum, median, first quartile, and third quartile. After calculating the requested values individually, use the `describe` method to obtain these values. Which variable is the most dispersed?

Exercise 7

Considering the Titanic dataset, for each of the three classes, calculate the mean and variance of passenger ages. In which class are the ages least dispersed? Which class contains the youngest individuals? Complete the analysis with bar charts.

References

- Chapter 2 of: Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.
- Chapter 3 of: Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Probability for data analysis

Chapter resources

Slides (PDF) Open in Colab Practice quiz

When working with data, we often ask: **what values will variables assume?**

In some cases, we can predict these values with perfect accuracy—like solving a system of equations that describes the speed of an object using Newtonian physics. These are **deterministic systems**, where outcomes follow directly from initial conditions.

But in many real-world scenarios, we face **uncertainty**. Modeling the relationship between variables in a deterministic way is not always possible. This uncertainty can arise from several sources:

- **Inherent randomness**: Some events are truly stochastic, like rolling a die or drawing a card. Even with perfect knowledge, the outcome is unpredictable.
- **Incomplete observability**: Sometimes we can't see everything. In the Monty Hall problem, the game is deterministic, but the player lacks full information—so the outcome feels uncertain.
- **Incomplete modeling**: Even in deterministic systems, we may lack the tools to model them fully. For example, a robot with only an RGB camera may estimate object positions in 2D, but cannot reconstruct full 3D coordinates. The missing dimensions introduce uncertainty.

Real-World Examples

- **Tossing a coin or rolling a die**: These are often treated as random events, even if they could be modeled deterministically in theory. The complexity of initial conditions makes prediction intractable.
- **Diagnosing a medical condition**: Different pathologies may share symptoms. Even with detailed observations, we may not be able to determine the exact diagnosis with certainty.

Why Probability Theory Matters

Probability theory provides a **consistent framework** for reasoning about uncertainty. It allows us to:

- Quantify uncertain events
- Combine known and unknown information
- Derive new probabilistic statements using formal rules

In data analysis, probability is essential for:

- Modeling randomness and variability
- Making predictions under uncertainty

- Interpreting patterns and relationships in data

In this lecture, we'll revisit the core concepts of probability theory—starting with random variables—and apply them to real datasets like the Titanic. We'll use tools like the **sum rule**, **product rule**, and **Bayes' theorem** to answer questions such as:

- What is the probability that a passenger survived?
- What is the probability that a passenger was in 1st class, given that they survived?
- What is the probability that a passenger was in 3rd class, given that they died?

These examples will help us connect abstract probability rules to concrete data-driven reasoning.

Random Experiments

In practice, when data acquisition is affected by uncertainty, we refer to the process as a **random experiment**. Informally, we define a random experiment as:

> An experiment that can be repeated any number of times, potentially leading to different outcomes.

We introduce the following terminology:

- **Sample space** $\Omega = \{\omega_1, \dots, \omega_k\}$ The set of all possible outcomes of the experiment.
- **Simple event** ω_i: a single possible outcome.
- **Event** $A \subseteq \Omega$: a subset of the sample space representing a condition or outcome of interest. We usually denote $\overline{A} = \Omega \setminus A$ as the complementary event to A, i.e., the event that A does not occur.

By definition:

- The sample space Ω is called the **sure event** or **certain event**, since it includes all possible outcomes.
- The empty set $\emptyset$ is called the **impossible event**, as it contains no outcomes.

Example: Rolling a Die

Consider the random experiment of rolling a fair six-sided die. The outcome is the number shown on the top face when the die lands.

- **Sample space**: $\Omega = \{1, 2, 3, 4, 5, 6\}$
- **Simple event**: $\omega_1 = 1$ represents the outcome "we roll a 1".
- **Event examples**:
  - "We obtain a 1": $A = \{1\}$
  - "We obtain an even number": $A = \{2, 4, 6\}$
  - Complement of the above: $\overline{A} = \{1, 3, 5\}$ Represents "we obtain an odd number".

This framework allows us to describe uncertainty in a structured way, preparing us to assign probabilities to events and reason about their likelihood.

Random Variables

We have so far talked about "statistical variables". When dealing with uncertain events, we need to use the concept of 'random variables'. Informally (from wikipedia):

> A random variable is a variable whose values depend on outcomes of a random phenomenon.

A random variable is characterized by a set of possible values often called *sample space*, *probability space*, or *alphabet* (this last term comes from information theory, where we often deal with sources emitting symbols from an alphabet, in which case the values of X will be the symbols).

The definition of a random variable is very similar to that of a statistical variable. Formally, if Ω is the *sample space*, we will define a random variable as a function:

$$X : \Omega \rightarrow E$$

Where **E is a measurable space** and often $E = \mathbb{R}$. This definition is similar to the one of statistical variable we have given before.

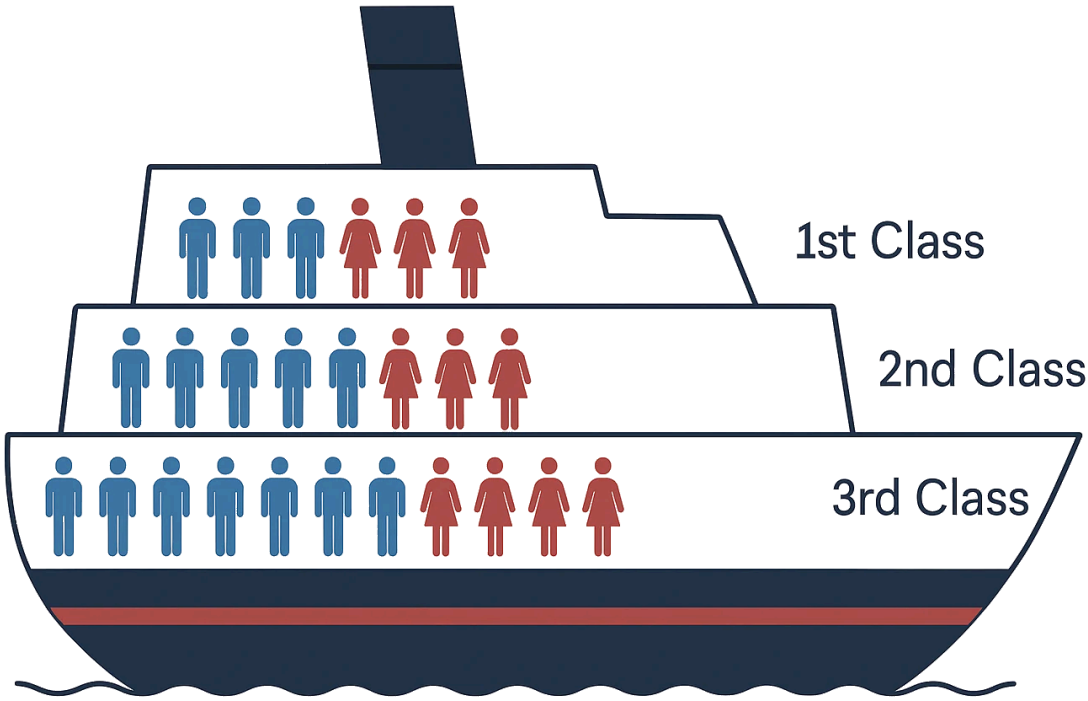
A random variable is generally denoted by a *capital letter*, such as X.

Similar to statistical variables, random variables can be *discrete* or *continuous*, *scalar* or *multi-dimensional*.

The following table lists some examples.

	Discrete	**Continuous**
Scalar	Tossing a coin	Height of a person
Multidimensional	Pair of dice	Coordinates of a car

Toy Titanic Example



We will now consider a toy example derived inspired to the Titanic dataset.

In this example, we will assume to have a very small ship with passengers in three classes and belonging to two possible genders (male or female).

To reason in terms of probability, we have to think of a random experiment first. To this aim, let's imagine the process of randomly picking a passenger from the list. This passenger will belong to one of the three classes and one of the two sexes. We will see how to use probability theory to answer different questions such as:

- Is it more likely to pick a woman or a man?
- If I pick a person from first class, is it more likely to pick a woman or a man?
- If I know I picked a woman, is it more likely that I am picking from first, second or third class?

As you can see, answering these questions allows us to better understand and characterize the data, but can also be useful for practical applications. For instance, we may deliver different services to the different classes depending on the people who populate the classes.

More technically, we will consider the experiment of randomly drawing a passenger from one of the three classes. This happens in two stages:

- We first randomly pick one of the three classes;
- Then we randomly pick one of the passengers in the class;

- After observing the sex of the passenger, we replace them in the same class.

The outcome of the experiment can be characterized by *two random variables*:

- C represents the class, which can take values $1, 2, 3$
- S represents the sex of the passenger, which can take values $male$ and $female$

If we pick a male passenger marble from the third class, then the outcome of the experiment can be characterized by the values $C = 3$, $S = male$;

Working Definition of Data

We will define "data" as follows:

> The values assumed by a random variable

Example

- For instance, if the sex of a passenger is $male$, then $S = male$ *is data*;
- It should be clear that the 'data' is *the pair* <random variable, value> and not just the value. Indeed $male$ alone would not be very useful (we don't know which phenomenon it is related to), whereas $S = male$ can be useful, as we know that S is the random variable describing the sex of the passenger;
- In this example, the data $S = male$ is representing a fact: 'I randomly picked a passenger and their sex was $male$'. This is also called 'an event';

Probability

Since random variables are related to stochastic phenomena, we cannot say much about the outcome of a single phenomenon.

However, we expect to be able to characterize the class of experiments related to a random variable, to infer rules on what values the random variable is likely to take.

For instance, in the case of coin tossing, we can observe that, if I toss a coin for a large number of times, the number of heads will be roughly similar to the number of tails.

This kind of observations is useful, as it can give us a prior on what values we are likely to encounter and what are not.

To formally express such rules, we can define the concept of probability on a random variable.

Specifically, it is possible to assign a probability value to a given outcome. This is generally represented with a capital P:

- For instance, $P(C = 1)$ represents the probability of picking a from the first class in the previous example;
- A probability $P(C = 1)$ is a number comprised between 0 and 1 which quantifies how likely we believe the event to be;
    - 0 means impossible;
    - 1 means certain;

When it is clear from the context which variable we are referring to, the probability can also be expressed simply as:

$$P(male) = P(S = male)$$

Probability Axioms

Kolmogorov in 1933 provided three axioms which define the "main rules" that probability should follow:

Axiom 1 Any event has a probability in the range $[0, 1]$:

$$0 \leq P(A) \leq 1, \forall A \subseteq \Omega$$

Axiom 2 The certain event has probability 1:

$$P(\Omega) = 1$$

Axiom 3 If A and B are disjoint event (i.e., $A \cap B = \emptyset$), then:

$$P(A \cup B) = P(A) + P(B)$$

From these axioms, the following corollaries follow:

Complement rule

$$P(\overline{A}) = 1 - P(A)$$

Probability of the impossible event

$$P(\emptyset) = P(\overline{\Omega}) = 1 - P(\Omega) = 0$$

General addition rule (for any events A and B)

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Monotonicity (if $A \subseteq B$)

$$A \subseteq B \Rightarrow P(A) \leq P(B)$$

Laplace Probability

When all outcomes in a random experiment are considered equally probable, this is called a **Laplace experiment**. In this case, we can calculate the probability of a given event A as the ratio between the favorable outcomes and the possible outcomes:

$$P(A) = \frac{|A|}{|\Omega|} = \frac{\text{number of favorable outcomes}}{\text{number of possible outcomes}}$$

For instance, if we are tossing a die:

- The probability of obtaining any of the faces will be: $\frac{1}{6}$;
- The probability of obtaining an even number will be: $\frac{|\{2,4,6\}|}{|\{1,2,3,4,5,6\}|} = \frac{3}{6} = \frac{1}{2}$.

Estimating probabilities from observations

What about the cases in which we cannot make any assumption on the equal probability of events? In those cases we would like to estimate probabilities from observations. There are two main approaches to do so: frequentist and Bayesian.

Frequentist approach

Probability theory was initially developed to analyze the frequency of events. For instance, it can be used to study events like drawing a certain hand of cards in a poker game. These events are repeatable and can be dealt with using frequencies. In this sense, when we say that an event has probability p of occurring, it means that if we repeat the experiment infinitely many times, then a proportion p of the repetitions would result in that outcome.

According to the frequentist approach, we can estimate probabilities by repeating an experiment for a large number of times and then computing:

- The number of trials: how many times we performed the experiment;
- The number of favorable outcomes: how many times the outcome of the experiment was favorable.

The probability is hence obtained by dividing the number of favorable outcomes by the number of trials.

For instance, let's suppose we want to estimate the probability of obtaining a 'head' by tossing a coin. Let's suppose we toss the coin 1000 times and obtain 499 heads and 501 tails. We can compute the probability of obtaining head as follows:

- Number of trials: 1000;
- Number of favorable outcomes: 499.

The probability of obtaining head will be 499/1000=0.49

This is the approach we have seen so far in the course when dealing with relative frequencies.

Examples

- The probability of obtaining ‘head’ when tossing a coin is 0.5. We know that because, if we toss a coin for a large number of times, about half of the times, we will obtain ‘head’;
- The probability of picking a red ball from a box with 40 red balls and 60 blue balls is 0.4. We know this because, if we repeat the experiment for a large number of times, we will observe that proportion.

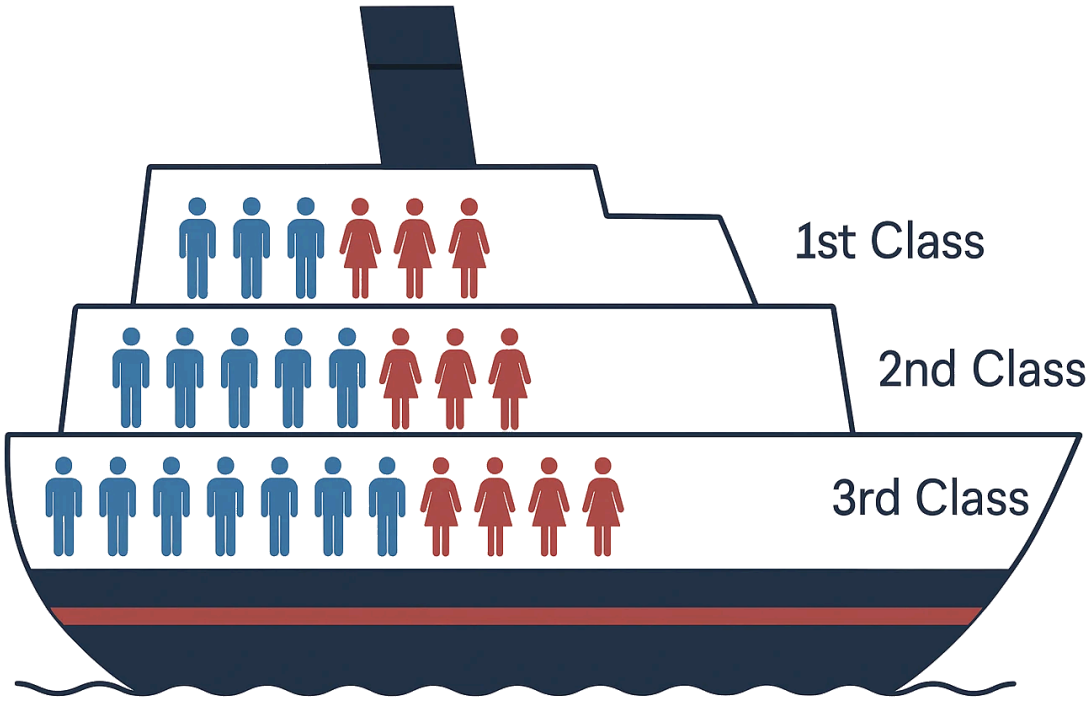
Bayesian Approach

The Bayesian approach to probability offers a different perspective on probability theory. Unlike the frequentist approach, which focuses on analyzing the frequency of events based on observations, the **Bayesian approach allows us to incorporate prior knowledge and update our beliefs as new information becomes available**.

In Bayesian probability, we view **probability as a measure of uncertainty or belief**. When we assign a probability to an event, it reflects our **subjective degree of belief in the event’s likelihood**. This approach is particularly useful when dealing with unique or one-time events where frequency-based analysis may not be applicable (e.g., “**what is the probability that the sun will extinguish in 5 billion years?**”).

We will discuss better the Bayesian approach when we’ll discuss Bayes theorem.

Example of Probability



Let's consider the previous example of randomly picking passengers.

Suppose we repeat this experiment for 100 times and observe that:

- We pick the first class 24 times;
- We pick the second class 32 times;
- We pick the third class 44 times;
- Once we selected a urn, we are equally likely to select any of the passengers contained in it, but we know that sexes are not distributed evenly in the classes (see figure).

Using a frequentist approach, we can define the probabilities:

- $P(C=1) = \frac{24}{100}$
- $P(C=2) = \frac{32}{100}$
- $P(C=3) = \frac{44}{100}$

This is done by using the formula:

- $P(X=x) = \frac{\#\ of\ times\ X=x}{\#\ trials}$

Computation of frequentist probability values

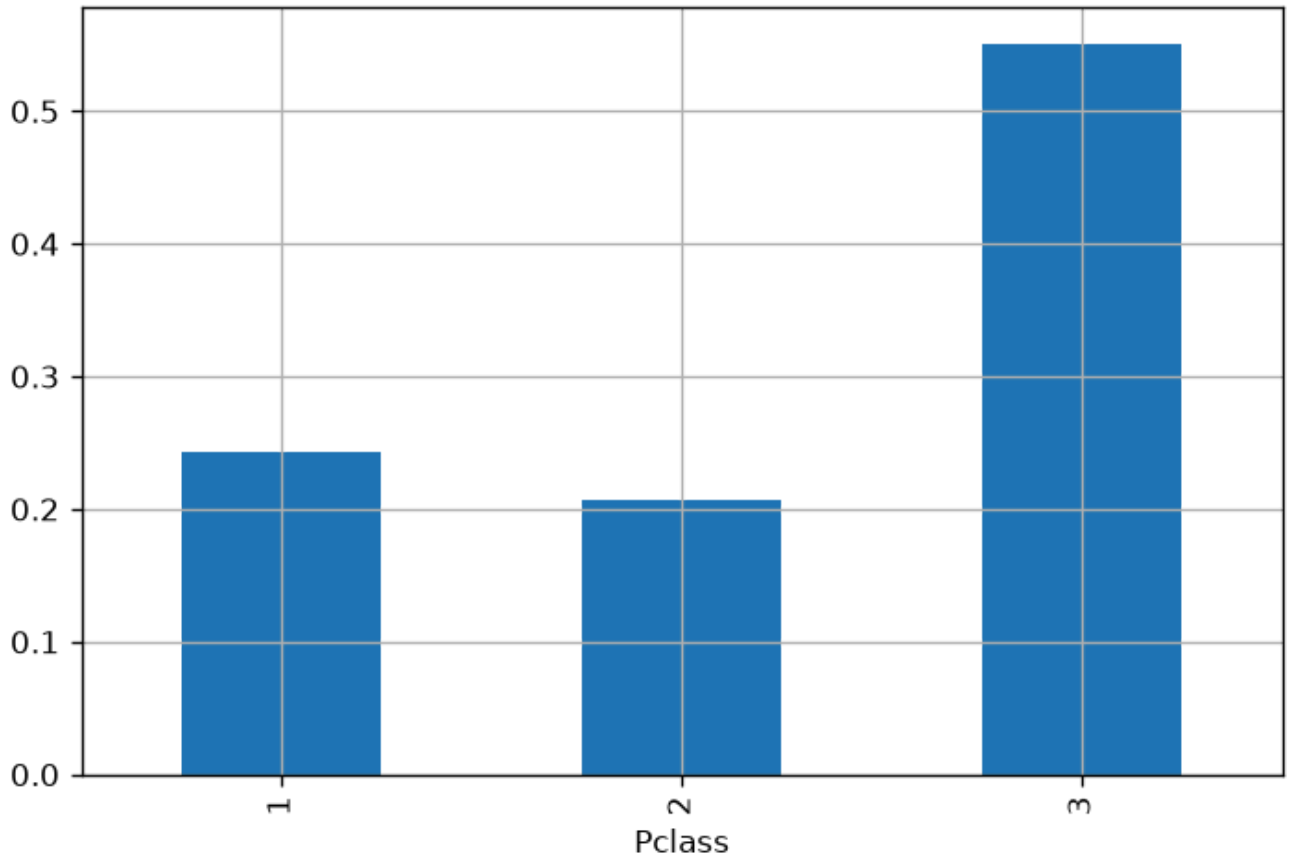
In frequentist terms, probabilities are the same as the relative frequencies of chapter 2. Let's see how to compute such probabilities on the real Titanic dataset:

```
import pandas as pd
titanic = pd.read_csv(DATA + "titanic.csv",
                      index_col='PassengerId')
titanic['Pclass'].value_counts(normalize=True).sort_index()  # relative frequencies
```

```
Pclass
1    0.242424
2    0.206510
3    0.551066
Name: proportion, dtype: float64
```

We can also visualize them for more clarity:

```
from matplotlib import pyplot as plt
titanic['Pclass'].value_counts(normalize=True).sort_index().plot.bar()  # relative
        frequencies
plt.grid()
plt.show()
```



Joint probability

We can define **univariate** (= with respect to only one variable) probabilities $P(C)$ as we have seen in the previous examples.

However, in many cases, it is useful to define probabilities on more than one variable at the time. For instance, we could be interested in studying the probability of picking a given fruit from a given box. In this case, we would be interested in the **joint probability** $P(C, S)$.

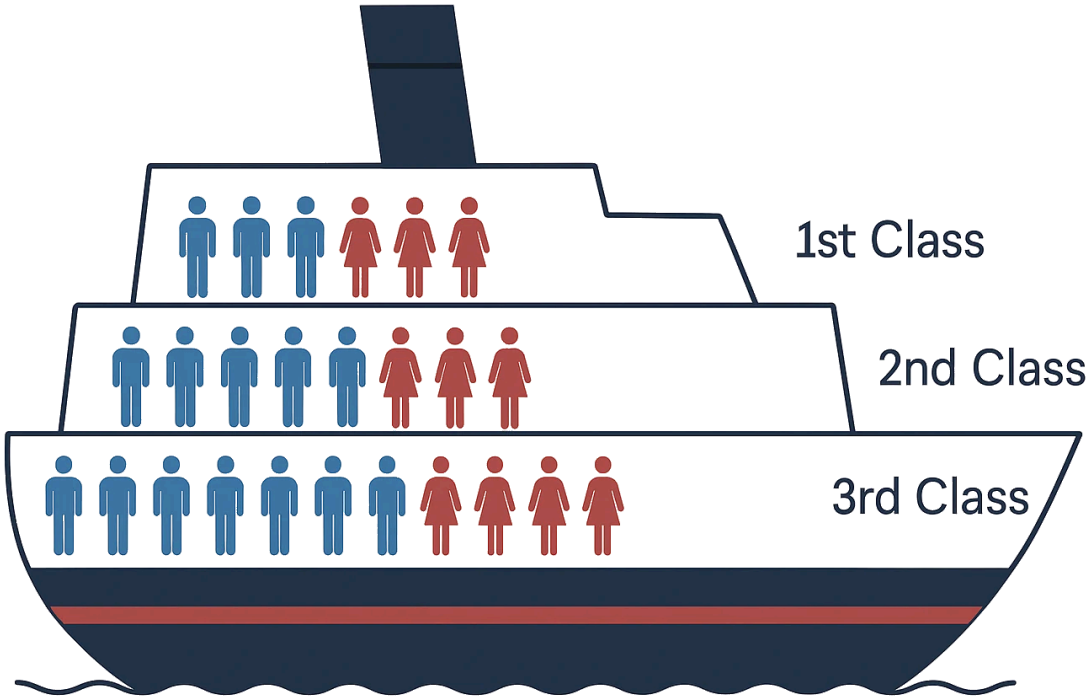
In general, we can have joint probabilities with arbitrary numbers of variable. For instance, $P\left(X_1, X_2, X_3, \ldots, X_n\right)$.

Joint probabilities are symmetric, i.e., $P(X, Y) = P(Y, X)$.

We should note that, when dealing with multiple unidimensional variables, we can always define a new multi-variate variable comprising all of them:

- $X = [X_1, X_2]$;
- $P(X) = P\left(X_1, X_2\right)$.

Example Joint Probability



We can see the concept of joint probability in the context of the Titanic example.

We have seen how to define the univariate probability $P(C)$ over the whole probability space of C.

However, we could be interested in the probability of both variables jointly: $P(C, S)$, i.e., the joint probability of C and S.

To 'measure' the joint probability, we could repeat the experiment many times and observe the outcomes.

We can then build a table which keeps track of how many times we observed a given combination:

	First Class	Second Class	Third Class	All
male	3	5	7	15
female	3	3	4	10
All	6	8	11	25

A table like this is called a "**contincency table**".

From the table above, we can easily derive the joint probability of a given pair of values using the **frequentist approach**. For instance:

$$P(C=1, S=male) = \frac{\#\ times\ (1, male)\ occurs}{\#\ trials} = \frac{3}{25}$$

Similarly, we can derive other values:

- $P(C=1, S=female) = \frac{3}{25}$
- $P(C=2, S=male) = \frac{5}{25}$
- $P(C=2, S=female) = \frac{3}{25}$
- $P(C=3, S=male) = \frac{7}{25}$
- $P(C=3, S=female) = \frac{4}{25}$

Note that we can also obtain the univariate probabilities by using the values in the "sum" row and column. For instance:

$$P(C=1) = \frac{\#\ times\ C=1\ occurred}{\#\ trials} = \frac{3+3}{25} = \frac{6}{25}$$

Similarly:

- $P(S=male) = \frac{15}{25}$
- $P(S=female) = \frac{10}{25}$
- $P(C=2) = \frac{8}{25}$
- $P(C=3) = \frac{11}{25}$

These univariate probabilities computed starting from joint probabilities are usually called **marginal probabilities**, because we are using the sums in the margin of the table.

We can obtain a joint probability table by dividing the table by the total number of trials (25):

	First Class	Second Class	Third Class	All
male	3/25	5/25	7/25	15/25
female	3/25	3/25	4/25	10/25
All	6/25	8/25	11/25	25/25

Alternatively, with decimal values:

	First Class	Second Class	Third Class	All
male	0.12	0.2	0.28	0.6
female	0.12	0.12	0.16	0.8
All	0.24	0.32	0.44	1

Computation of a Joint Probability Table

To compute a joint probability table from data, we should first compute a contingency table, which is also called a crosstab in Pandas:

```
pd.crosstab(titanic['Sex'], titanic['Pclass'])
```

Pclass	1	2	3
Sex			
female	94	76	144
male	122	108	347

The table above indicates, for example, that 94 passengers in class 1 were female. By specifying `margins=True`, we can obtain row and column sums, which will indicate the absolute frequencies of each variable:

```
pd.crosstab(titanic['Sex'], titanic['Pclass'],margins=True)
```

Pclass	1	2	3	All
Sex				
female	94	76	144	314
male	122	108	347	577
All	216	184	491	891

We can obtain the joint relative frequency table by passing `normalize=True` to the crosstab function:

```
pd.crosstab(titanic['Sex'], titanic['Pclass'], normalize=True, margins=True)
```

Pclass	1	2	3	All
Sex				
female	0.105499	0.085297	0.161616	0.352413
male	0.136925	0.121212	0.389450	0.647587
All	0.242424	0.206510	0.551066	1.000000

It should be noted that the marginal relative frequencies obtained are identical to the relative frequencies of the univariate samples:

```
titanic['Sex'].value_counts(normalize=True)
```

```
Sex
male      0.647587
female    0.352413
Name: proportion, dtype: float64
```

That are exactly the values in the last column of the previous table. Similarly, the univariate probabilities of Pclass are the same as the last row of the previous table:

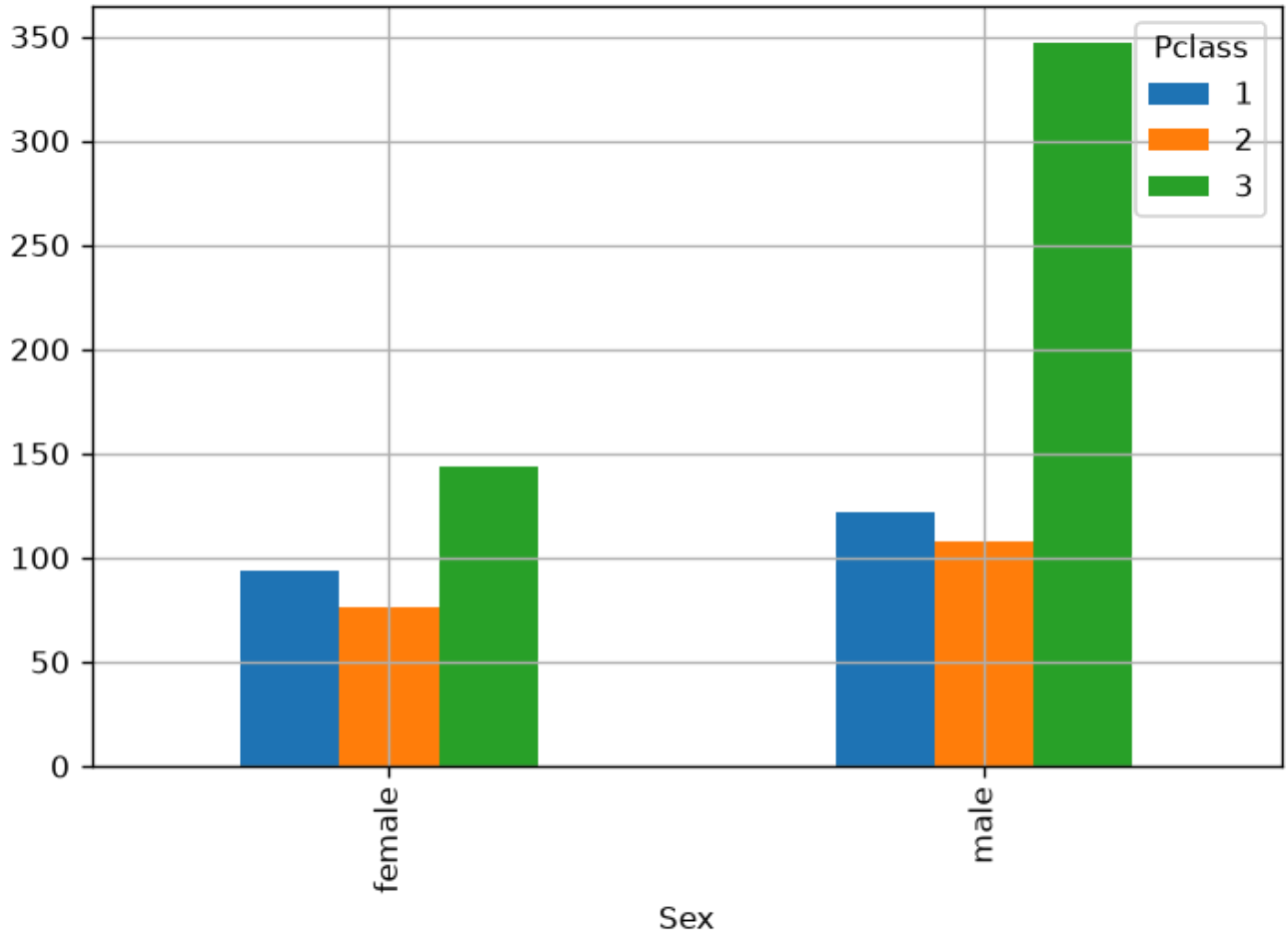
```
titanic['Pclass'].value_counts(normalize=True)
```

```
Pclass
3    0.551066
1    0.242424
2    0.206510
Name: proportion, dtype: float64
```

Graphical representations

We can use contingency tables and joint probability tables to obtain graphical representations. For example:

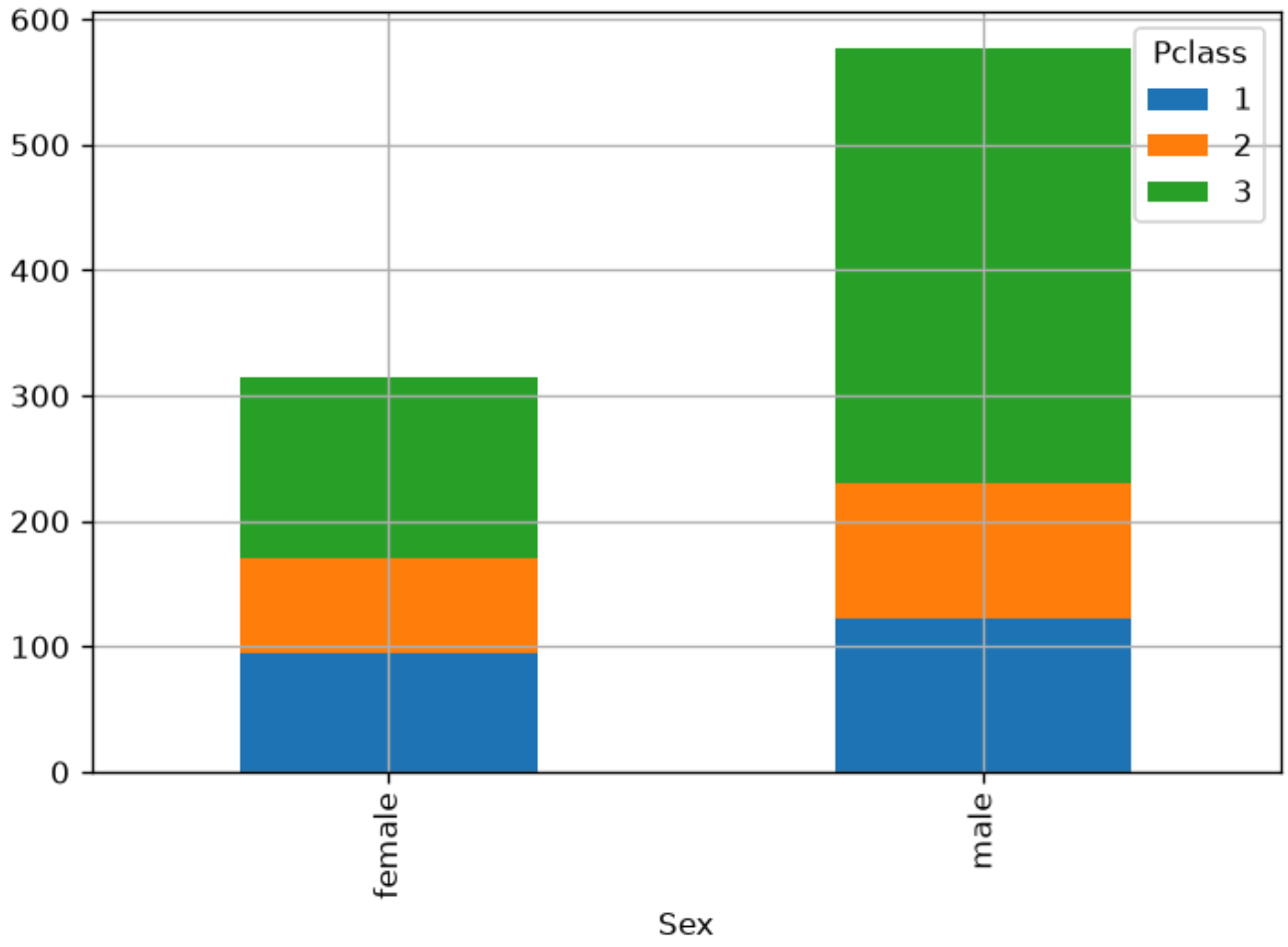
```
pd.crosstab(titanic['Sex'], titanic['Pclass']).plot.bar()
plt.grid()
plt.show()
```



The diagram above shows the distributions of people in the two classes, grouped by the two different sexes.

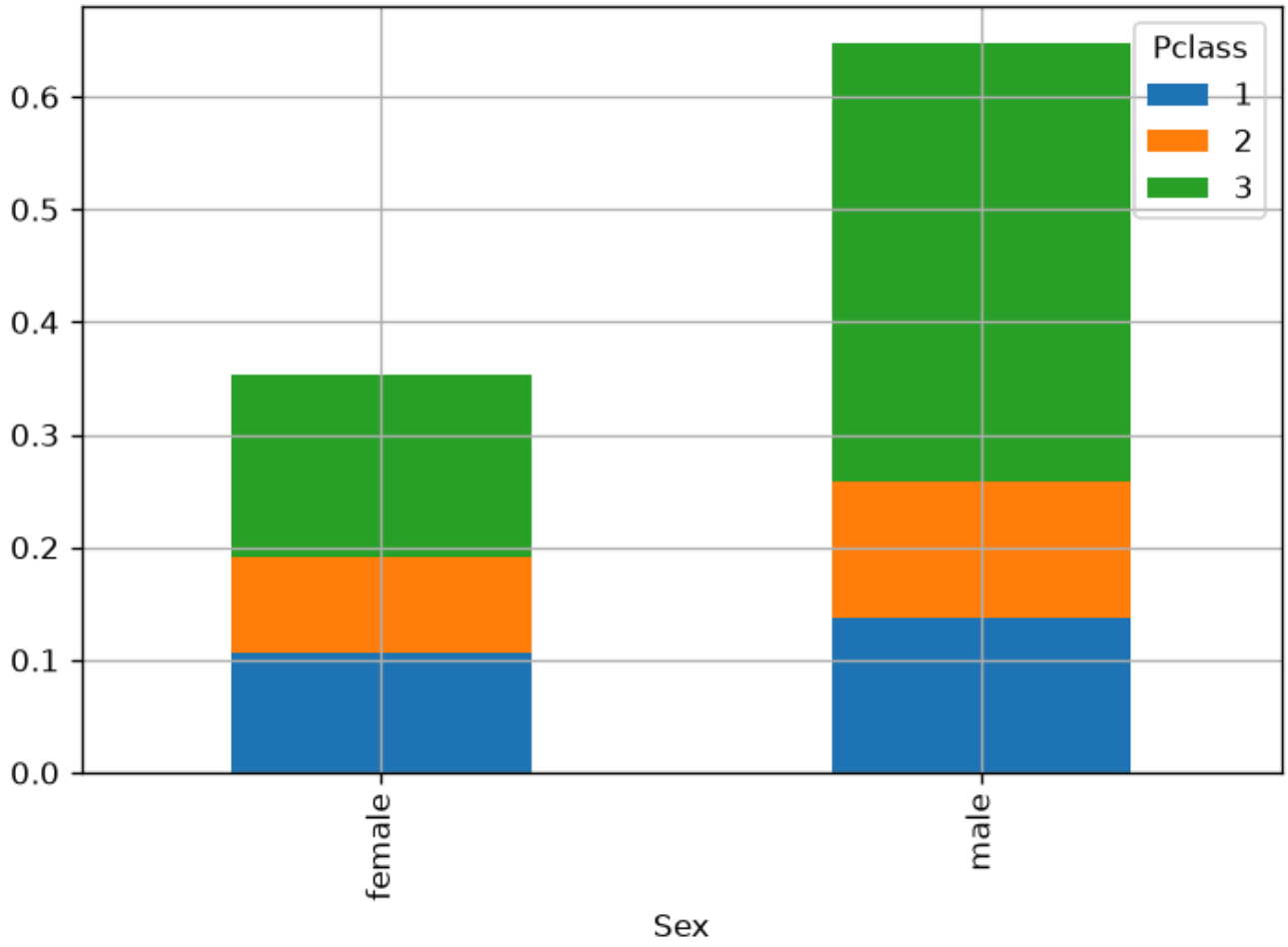
From this, we note that many passengers are men and in third class. In these cases, a stacked plot can sometimes be useful:

```
pd.crosstab(titanic['Sex'], titanic['Pclass']).plot.bar(stacked=True)
plt.grid()
plt.show()
```



We can reason in terms of joint probabilities by normalizing the counts:

```
pd.crosstab(titanic['Sex'], titanic['Pclass'], normalize=True).plot.bar(stacked=True)
plt.grid()
plt.show()
```



Sum Rule (Marginal Probability)

In the previous example, we have seen how we can compute **marginal (univariate) probabilities** from the contingency table. This is possible because the contingency table contains information on how the different possible outcomes distribute over the sample space.

In general, we can compute marginal probabilities form joint probabilities (i.e., we don't need to have the non-normalized frequency counts of the contingency table). Let us consider the general contingency table:

	Y= y_1	Y= y_2	...	Y= y_l	Total
X= x_1	n_{11}	n_{12}	...	n_{1l}	$n_{1\cdot}$
X= x_2	n_{21}	n_{22}	...	n_{2l}	$n_{2\cdot}$
...	...	...	...	...	...
X= x_k	n_{k1}	n_{k2}	...	n_{kl}	$n_{k\cdot}$
Total	$n_{\cdot 1}$	$n_{\cdot 2}$	...	$n_{\cdot l}$	n

where:

- n_{ij} is the number of times that we observe $X = x_i$ and $Y = y_j$ simultaneously (e.g., $C = 1$ and $S = male$);
- n_{i+} is the sum of all occurrences in which $X = x_i$ (i.e., we are summing all values in row i): $n_{i+} = \sum_j n_{ij}$;
- n_{+j} is the sum of all occurrences in which $Y = y_j$ (i.e., we are summing all values in column j): $n_{+j} = \sum_i n_{ij}$;

We can compute the joint probability $P(X = x_i, Y = y_j)$ with a frequentist approach using the formula:

$$P\left(X = x_i,\ Y = y_j\right) = \frac{n_{ij}}{n}$$

We note that we can define the marginal probabilities of X and Y as follows:

- $P\left(X = x_i\right) = \frac{n_{i+}}{n}$.
- $P\left(Y = y_j\right) = \frac{n_{+j}}{n}$.

We can hence write the marginal probability of X as follows:

$$P\left(X=x_i\right)=\frac{n_{i+}}{n}=\frac{\sum_j n_{ij}}{n}=\sum_j \frac{n_{ij}}{n}=\sum_j P(X=x_i,Y=y_j)$$

This result is known as the **sum rule of probability**, which allows to estimate marginal probabilities from joint probabilities. This can be seen in more general terms as:

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

The act of computing $P(X)$ from $P(X,Y)$ is also known as marginalization.

Conditional Probability and Product Rule

In many cases, we are interested in the probability of some event, given that some other event happened.

This is called **conditional probability** and is denoted as

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

and read as

> P of X=x given that Y=y

In this context, $Y=y$ is the condition, and we are interested in studying the probability of X only in the cases in which the condition is verified.

For instance, in our toy Titanic, we could be interested in $P(S=male|C=1)$, i.e., what is the probability of picking a male, given that we know that we are drawing from the first class?

Let's consider our example contingency table of two variables again:

	Y= y_1	Y= y_2	...	Y= y_l	Total
X= x_1	$n_{1:}$	$n_{1:}$	...	n_1	$n_{1\cdot}$
X= x_2	$n_{2:}$	$n_{2:}$	...	n_2	$n_{2\cdot}$
...	...	...	...	...	...
X= x_k	n_k	$n_{k:}$	...	n_k	$n_{k\cdot}$
Total	n_+	n_+	...	n_+	n

We can compute the conditional probability $P(X = x_i|Y = y_j)$ using the frequentist approach:

$$P\left(X = x_i|Y = y_j\right) = \frac{\#\ cases\ in\ which\ X = x_i\ and\ Y = y_i}{\#\ cases\ in\ which\ Y = y_j} = \frac{n_{ij}}{n_{+j}}$$

If we multiply the expression above by $1 = \frac{n}{n}$, we obtain:

$$P\left(X = x_i|Y = y_j\right) = \frac{n_{ij}}{n}\frac{n}{n_{+j}} = \frac{\frac{n_{ij}}{n}}{\frac{n_{+j}}{n}} = \frac{P\left(X = x_i, Y = y_j\right)}{P(Y = y_j)}$$

This leads us to the general definition of conditional probability:

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

The conditional probability is defined only when $P(Y = y) > 0$, that is, we cannot define a probability conditioned on an event that never happens. It should be noted that, in general $P\left(X|Y\right) \neq P(X)$.

Product Rule (Factorization)

We can see the definition of conditional probability:

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

As follows:

$$P(X = x, Y = y) = P\left(X = x | Y = y\right) P(Y = y)$$

which is often referred to as **the product rule**.

The product rule allows to compute joint probabilities starting from conditional probabilities and marginal probabilities. This is useful because measuring joint probabilities generally involves creating large tables, whereas conditional and marginal probabilities might be easier to derive.

This operation of expressing a joint probability in terms of two factors is known as *factorization*.

How to compute a conditional probability?

We just said that factorization can be useful for computing joint probabilities starting from conditional probabilities. However, two questions arise:

- How can we compute a conditional probability?
- Is it easier than computing a joint probability?

Since conditional probabilities are obtained by *restricting* the probability space to *a subset* of the events, we can compute conditional probabilities by considering the observations which satisfy the condition.

For example, let's say we want to compute the conditional probability:

$$P(S|C = 1)$$

That is to say, the probability of picking a person of a given sex, given that we know that we are considering the first class. Let's consider again our contingency table:

	First Class	Second Class	Third Class	All
male	3	5	7	15
female	3	3	4	10
All	6	8	11	25

To compute this probability, we can just *consider all the observations that satisfy the condition* $C = 1$, which is equivalent to taking the first column of the full contingency table and compute the probabilities in a frequentist way:

	First Class
male	3
female	3
All	6

$$P(S = male|C = 1) = \frac{3}{6}$$

$$P(S = female|C = 1) = \frac{3}{6}$$

Note that, in general, when the number of variables is large, this approach allows to save a lot of space and time as it is not necessary to even build the first contingency table, but only the second, restricted one is required (for instance, one may choose not to record all observations in which we where picking form the other classes).

Of course, the table above can be normalized to obtain a conditional probability table:

	First Class
male	0.5
female	0.5
All	1

In Python, it is possible to obtain the conditional relative frequencies by normalizing by the appropriate row. Rather than conditioning on class, let's condition on sex in this case. We can do so by normalizing by rows:

```
# normalize=0 indicates to condition on the first variable
pd.crosstab(titanic['Sex'], titanic['Pclass'], normalize=0, margins=True)
```

Pclass	1	2	3
Sex			
female	0.299363	0.242038	0.458599
male	0.211438	0.187175	0.601386
All	0.242424	0.206510	0.551066

From the table above, for example, we can infer:

- $f(Pclass = 1|Sex = female) = 0.299363$
- $f(Pclass = 2|Sex = female) = 0.242038$
- $f(Pclass = 3|Sex = female) = 0.458599$

These are the frequencies of women in the three classes. Similarly, the second row represents the conditional frequencies with respect to `Sex=male`. From the table, we notice that the distribution of passengers changes across the three classes. In particular, among men, the third class is more frequent than among women.

We can obtain the complementary perspective by conditioning on class instead:

```
# normalize=1 indicates conditioning on the first variable
pd.crosstab(titanic['Sex'], titanic['Pclass'], normalize=1, margins=True)
```

Pclass	1	2	3	All
Sex				
female	0.435185	0.413043	0.293279	0.352413
male	0.564815	0.586957	0.706721	0.647587

In this case, each column will be a probability distribution. For example:

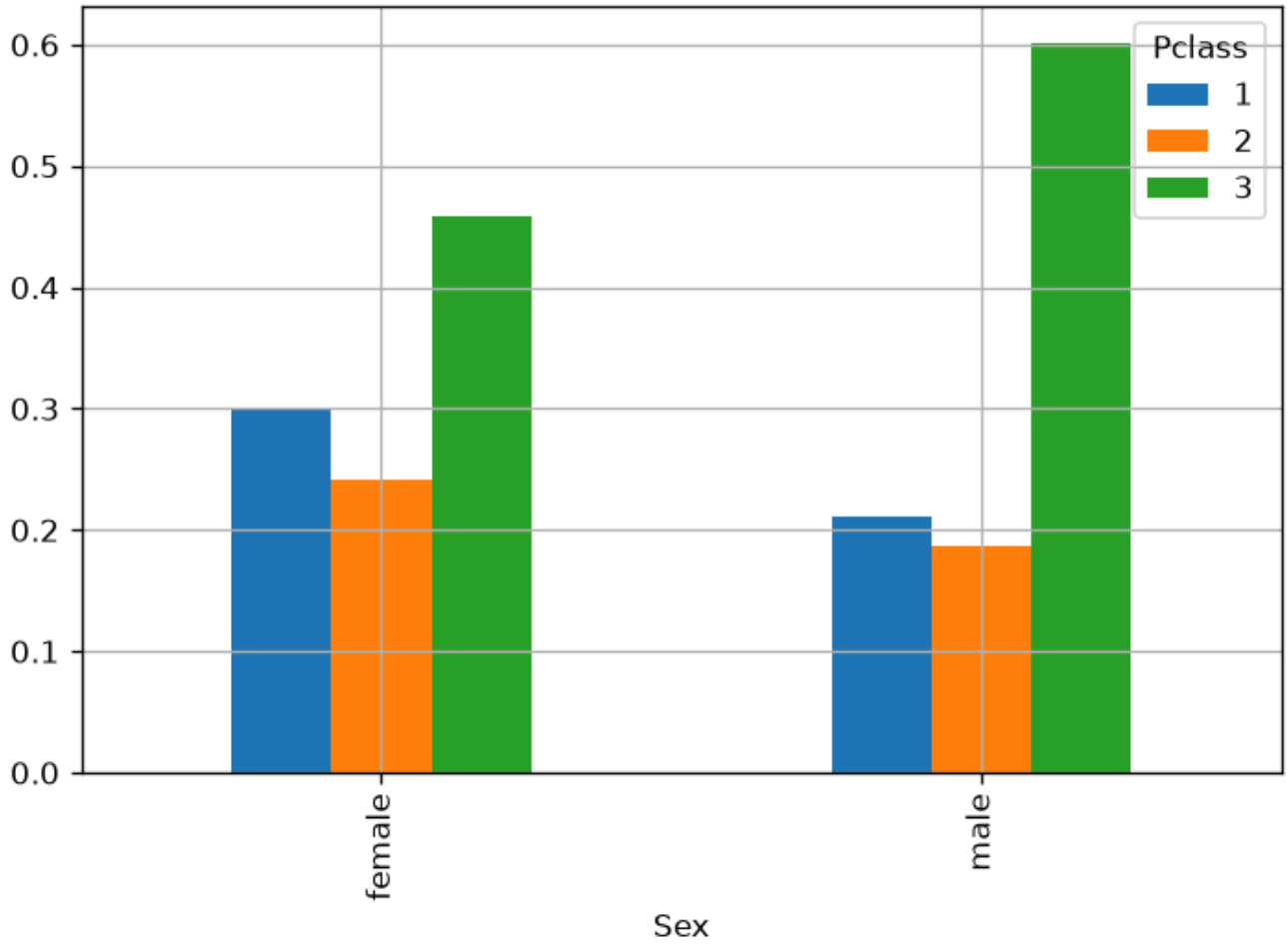
- $P(Sex = female|Pclass = 1) = 0.435185$
- $P(Sex = male|Pclass = 1) = 0.564815$

We note that the proportion between men and women changes in the three classes, and in particular in the third class there are many more men than women.

Graphical Representations

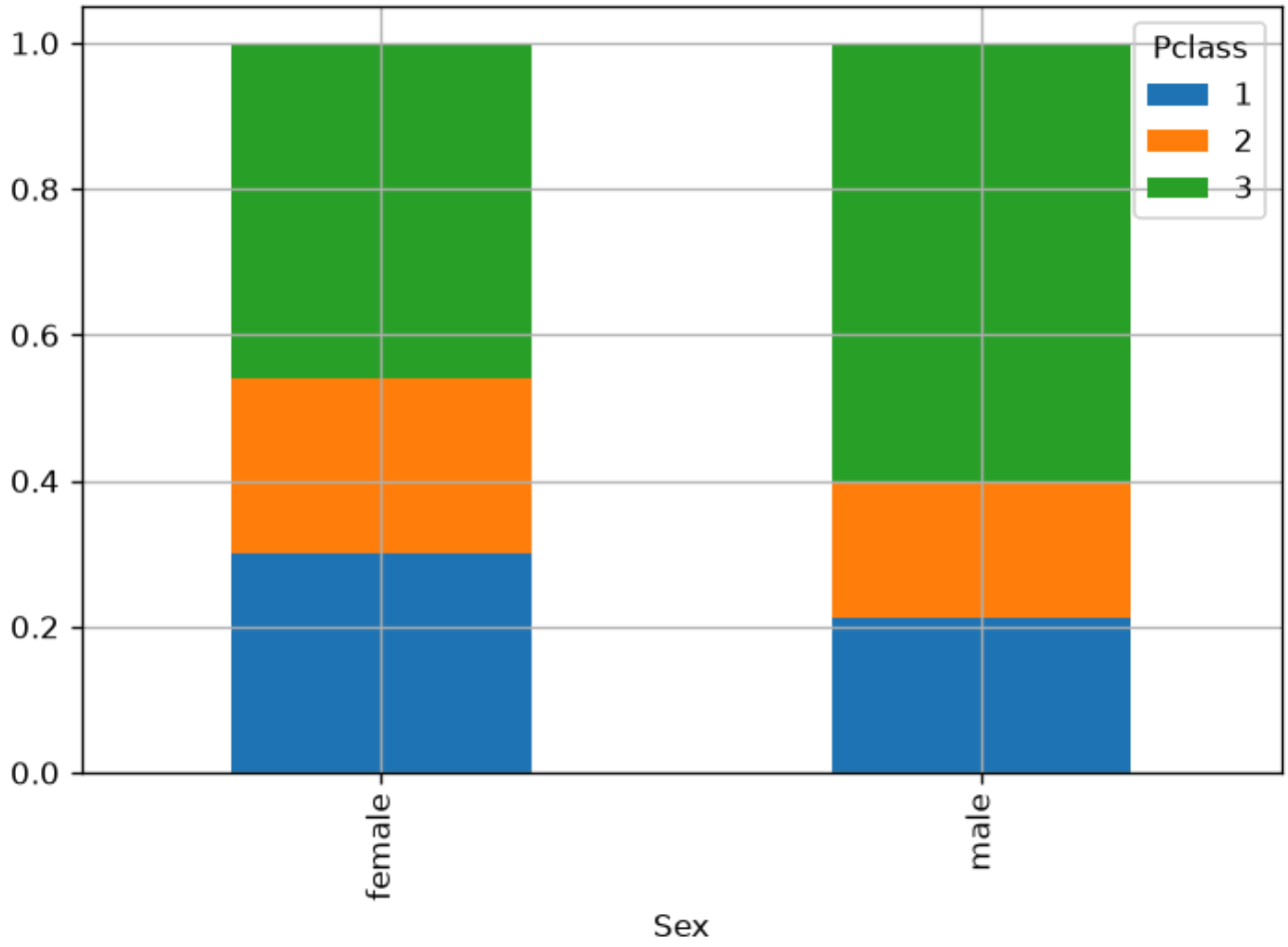
Similarly to joint probabilities, we can also obtain visualizations from conditional probabilities. For instance:

```
pd.crosstab(titanic['Sex'], titanic['Pclass'], normalize=0).plot.bar()
plt.grid()
plt.show()
```



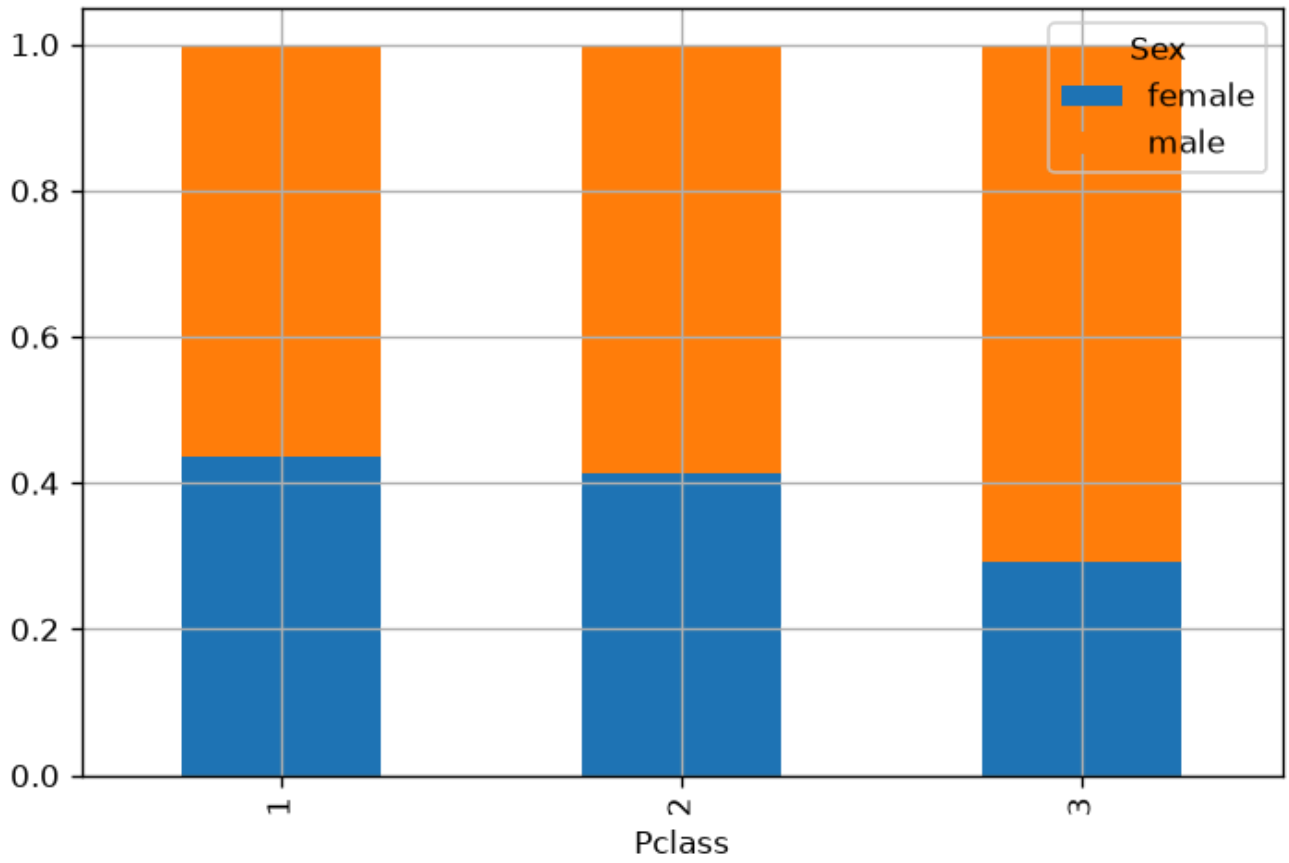
The diagram above shows the distributions of passengers in the three classes for the two sexes. Now, these are two valid probability distributions summing to one for all the three values (three classes). This is clearer with a stacked plot:

```
pd.crosstab(titanic['Sex'], titanic['Pclass'], normalize=0).plot.bar(stacked=True)
plt.grid()
plt.show()
```



We can obtain the graph of conditional frequencies with respect to the classes for a complementary view. In this case, to have a meaningful graph, we need to transpose the contingency table with a `.T`:

```
# normalize=1 indicates conditioning on the first variable
pd.crosstab(titanic['Sex'], titanic['Pclass'], normalize=1).T.plot.bar(stacked=True)
plt.grid()
plt.show()
```



This graph now shows the distribution of sexes in the three classes, clearly showing that class 3 contains mostly males.

The Chain Rule of Conditional Probabilities

When dealing with multiple variables, the product rule can be applied in an iterative fashion, thus obtaining the 'chain rule' of conditional probabilities.

For instance:

$$P(X, Y, Z) = P(X|Y, Z)\, P(Y, Z)$$

Since:

$$P(Y, Z) = P(Y|Z)\, P(Z)$$

We obtain:

$$P(X, Y, Z) = P(X|Y, Z)\, P(Y|Z)\, P(Z)$$

Since joint probabilities are symmetric, we could equally obtain:

$$P(X, Y, Z) = P(Z|Y, X)\, P(Y|X)\, P(X)$$

This rule can be formalized as follows:

$$P\left(X_1, \ldots, X_n\right) = P\left(X_1\right) \prod_{i=2}^{n} P(X_i | X_1, \ldots, X_{i-1})$$

Bayes' Theorem

Given two variables A and B, from the product rule, we obtain:

- $P(A, B) = P\left(A|B\right) P(B)$
- $P(B, A) = P\left(B|A\right) P(A)$

Since joint probabilities are symmetric, we have:

$$P(A, B) = P(B, A) \rightarrow P\left(A|B\right) P(B) = P\left(B|A\right) P(A)$$

Which implies:

$$P\left(A|B\right) = \frac{P\left(B|A\right) P(A)}{P(B)}$$

This last expression is known as Bayes' Theorem (or Bayes' rule).

Technically speaking, the Bayes' rule can be used to "turn" probabilities of the kind $P(A|B)$ into probabilities of the kind $P(B|A)$.

More formally, the Bayes' rule can be used to **update our expectation that some event will happen (event A) when we observe some evidence (event B)**. This links to the Bayesian interpretation of probability, according to which probability can be seen as **a *reasonable expectation* representing *the quantification of a degree of belief*** (i.e., how much we believe some event will happen).

The different terms in the Bayes' rule have specific names:

- $P(A)$ is called 'the prior' – this is our expectation that A happens when we do not have any other data to rely on
- $P(B|A)$ is called 'the likelihood' – this quantifies how likely it is to observe event B happening if we assume that event A has happened
- $P(B)$ is called 'the evidence' – this models the probability of observing event B
- $P(A|B)$ is called 'the posterior' – this is our updated probability, i.e., how likely it is A to happen once we have observed B happening

We'll see an example in the section below.

Bayesian Probability Example

Let's imagine we are trying to understand if a friend has COVID or not. If we do not know anything about our friend's symptoms (i.e., we don't know if they have any symptoms or not), then we would expect our friend to have COVID with a "prior" probability $P(C)$. If we know that currently one people over two has COVID, we expect:

$$P(C) = \frac{1}{2}$$

Now, if our friend tells us he has fever, things change a bit. We are now interested in modeling the probability: $P\left(C|F\right)$. We can try to model it using Bayes' rule:

$$P\left(C|F\right) = \frac{P\left(F|C\right)P(C)}{P(F)}$$

Note that it is not straightforward to estimate $P(C|F)$ in a frequentist way. Ideally, we should take all people with a fever on earth and check how many of them has COVID. This is not feasible as people with just fever may never do a COVID test. On the contrary, measuring $P(F|C)$ is easier: we take all people which we know have COVID (these may not be all people with COVID, but probably a large enough sample) and see how many of them have a fever. Let's suppose that one people with COVID out of three has a fever. Then we can expect:

$$P\left(F|C\right) = \frac{1}{3}$$

Now we need to estimate the evidence $P(F)$. This can be done by considering how frequent it is for people to have a fever. Let's say we use historical data and find out that one person out of five has fever. We finally have:

$$P\left(C|F\right) = \frac{\frac{1}{3}\frac{1}{2}}{\frac{1}{5}} = \frac{5}{6}$$

We can interpret this use of the Bayes' theorem as follows:

- Before knowing anything about symptoms, we could only guess that our friend had COVID with a prior probability of $P(C) = \frac{1}{2}$
- When we get to know that our friend has a fever, our expectation changes. We know that people with COVID often get a fever, but we also know that not all people with COVID get a fever, so we are not going to say that we are 100% certain that our friend has fever. Instead, we use Bayes' rule to obtain an updated estimate $P\left(C|F\right) = \frac{5}{6}$ based on our knowledge of the likelihood $P(F|C)$ (how likely it is for people with symptom to have COVID) and the evidence $P(F)$ (how common is this symptom – if it is too common, then it is not so informative)

Independence and Conditional Independence

Two variables X and Y are independent if the outcome of one of the two does not influence the outcome of the other. Formally speaking, two variables are **said to be independent** if and only if:

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

It should be noted that the expression above **is generally not true** as we cannot always assume that two variables are independent.

Independence can be denoted as:

$$X \perp Y$$

Moreover, if two variables X and Y are independent, then:

$$P\left(X|Y\right) = \frac{P(X,Y)}{P(Y)} = \frac{P(X)P(Y)}{P(Y)} = P(X)$$

This makes sense because it means that "the fact that Y happens does not influence the fact that X happens", which is what we would expect of two independent variables X and Y.

Examples

Intuitively, two variables are independent if the values of one of them do not affect the values of the other one:

- Weight and height of a person are **not independent**. Indeed, taller people are usually heavier.
- Height and richness are **independent**, as the richness does not depend on the height of a person.

Conditional independence

Two random variables X and Y are said to be **conditionally independent** given a random variable Z if:

$$P\left(X,Y|Z\right) = P\left(X|Z\right)P\left(Y|Z\right)$$

Conditional independence can be denoted as:

$$X \perp Y \mid Z$$

Example

Height and vocabulary are not independent: taller people are usually older, and hence they have a more sophisticated vocabulary (they know more words). However, if we condition on age, they become independent. Indeed, among people of the same age, height should not influence vocabulary. Hence, height and vocabulary are **conditionally independent** with respect to age.

Example of Probability Manipulation

Let's now consider a real probability manipulation example. This time we will consider the following variables: `Pclass` and `Survived`. We will use observed proportions to answer questions such as:

1. What is the overall probability that a passenger survived?
2. What is the probability that a passenger was in 1st class and they survived?
3. What is the probability that a passenger survived, given that they were in 3rd class?

We will start by defining two random variables:

- $Pclass \in \{1, 2, 3\}$: Passenger class
- $Survived \in \{0, 1\}$: Survival outcome (0 = died, 1 = survived)

What is the overall probability that a passenger survived?

To answer this question, we need to compute the overall probabilities of surviving ($P(Survived)$):

```
titanic['Survived'].value_counts(normalize=True)
```

```
Survived
0    0.616162
1    0.383838
Name: proportion, dtype: float64
```

From the table above, we can say:

- $P(Survived = 1) = 0.39$
- $P(Survived = 0) = 0.61$

What is the probability that a passenger was in 1st class and they survived?

To answer this question, we need to compute the joint probability:

```
pd.crosstab(titanic['Survived'], titanic['Pclass'], normalize=True, margins=True)
```

Pclass	1	2	3	All
Survived				
0	0.089787	0.108866	0.417508	0.616162
1	0.152637	0.097643	0.133558	0.383838
All	0.242424	0.206510	0.551066	1.000000

From the table above, we can answer our question as follows:

$$P(Pclass = 1, Survived = 1) = 0.15$$

Read the cell where the row $Survived = 1$ meets the column $Pclass = 1$. The cell just above it, 0.09, is the probability of being in first class **and dying** — a different question, and an easy one to answer by mistake.

What is the probability that a passenger survived, given that they were in 3rd class?

To answer this question, we need to compute a conditional distribution:

```
pd.crosstab(titanic['Survived'], titanic['Pclass'], normalize=1, margins=True)
```

Pclass	1	2	3	All
Survived				
0	0.37037	0.527174	0.757637	0.616162
1	0.62963	0.472826	0.242363	0.383838

To answer the question, we can say:

$$P(Survived = 1|Pclass = 3) = 0.24$$

Exercises

Suppose that we have three colored boxes r (red), b (blue), and g (green). Box r contains 3 apples, 4 oranges, and 3 limes, box b contains 1 apple, 1 orange, and 0 limes, and box g contains 3 apples, 3 oranges, and 4 limes. If a box is chosen at random with probabilities P(r)=0.2, P(b)=0.2, P(g)=0.6, and a piece of fruit is extracted from the box (with equal probability of selecting any of the items in the box), then what is the probability of selecting an apple? If we observe that the selected fruit is in fact an orange, what is the probability that it came from the green box?

References

- Parts of chapter 1 of [1];
- Most of chapter 3 of [2];
- Parts of chapters 5-7 of [3].

[1] Bishop, Christopher M. *Pattern recognition and machine learning*. springer, 2006. https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf

[2] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. https://www.deeplearningbook.org/

[3] Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Association between variables

Chapter resources

Slides (PDF) Open in Colab Practice quiz

We have seen how to describe and visualize "univariate" samples, i.e., samples composed of a single variable. More precisely, even when several variables were available, we analyzed the variables one by one. In practice, it often happens that the observations under analysis are composed of multiple variables. In these cases, it is useful to consider methods to describe and visualize the data in a "multivariate" manner, in order to study the interactions between the various factors that describe the phenomenon under analysis.

> In simple terms, we will say that two variables are associated if observing the value of one of the two variables tell us something about the value of the other variable.

As we will see, this is closely **related to the concept of dependence in probability**. In fact, **if two variables are independent, they typically show no statistical association**—knowing the value of one provides no information about the other.

Examples of associated variables are:

- Weight and height of a person: tall people tend to weigh more than short people;
- Education level and income: individuals with higher education often earn more;
- Age and blood pressure: older individuals tend to have higher blood pressure;
- Hours studied and exam score: more study time is often linked to better performance;

Note that correlation does not imply causation: two variables may be statistically associated without one directly causing the other. Confounding factors or reverse causality may be at play.

We will use again the Titanic dataset for our examples. Let's load it:

```
import pandas as pd
from matplotlib import pyplot as plt
titanic = pd.read_csv(DATA + "titanic.csv",
                      index_col='PassengerId')
titanic.head()
```

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId											
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Given the dataset above, we might ask ourselves if some variables influence the values of others. For example:

- does being in first, second, or third class (variable `Pclass`) somehow influence the probability of survival (variable `Survived`)?
- does age (variable `Age`) or the price paid (`Fare`) somehow influence the probability of being saved (`Survived`)?

In this lesson, we will see different ways to summarize the distributions of two variables and verify any associations (or correlations) between the variables.

Measures of Association between two Discrete Variables

Now let's look at measures that allow us to estimate if two variables are "associated" or not, meaning to what extent the values of one variable influence those of the other. We have seen that the variables `Sex` and `Pclass` seem to have some degree of association. Let's now look at a case of stronger association, by exploring the relationship between `Pclass` and `Survived`.

Let's consider three relative frequency distributions of `Survived` conditioning on the `Pclass` values:

- f(Survived|Pclass=1)
- f(Survived|Pclass=2)
- f(Survived|Pclass=3)

The table of conditional frequencies that follows summarizes these values

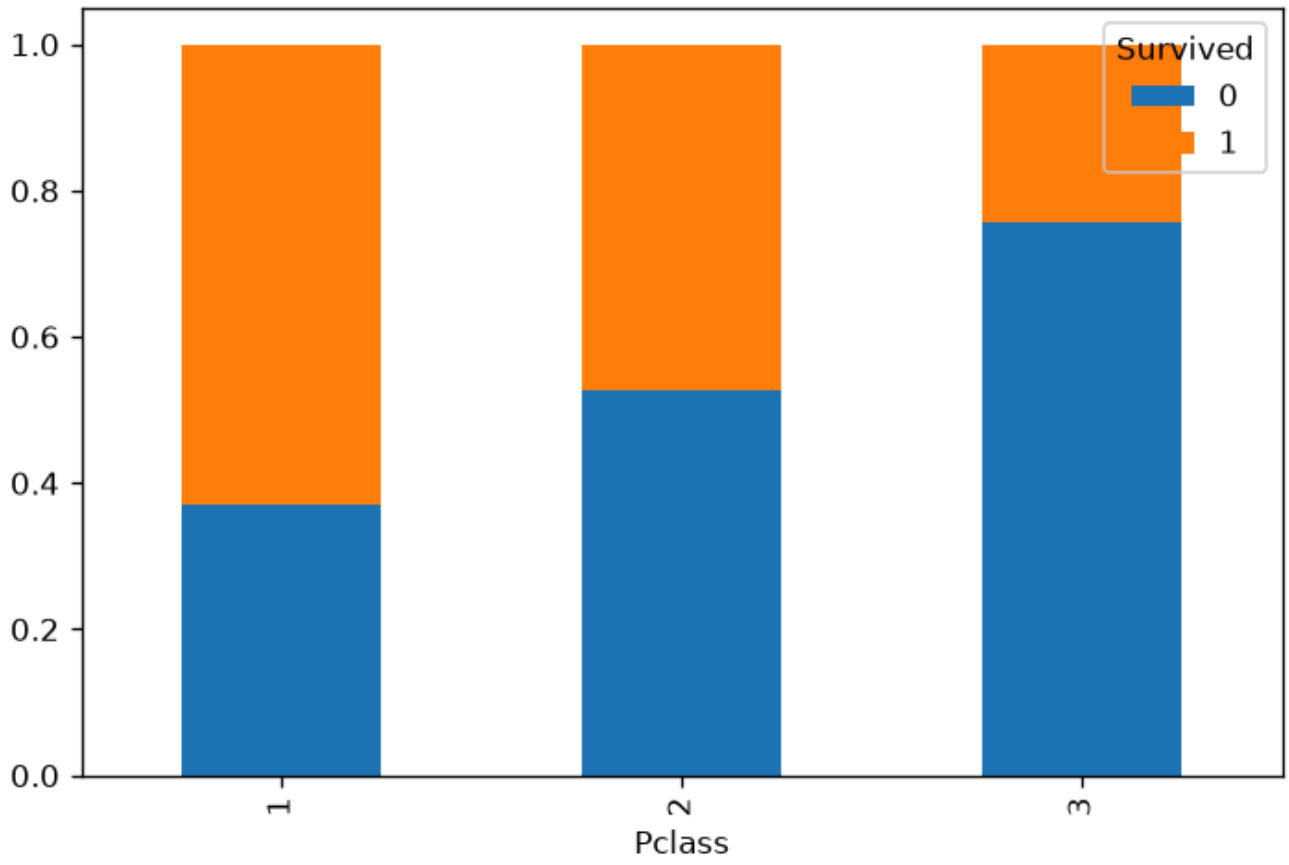
```
pd.crosstab(titanic['Pclass'], titanic['Survived'], normalize=0)
```

Survived	0	1
Pclass		
1	0.370370	0.629630
2	0.527174	0.472826
3	0.757637	0.242363

Looking at the numbers, we see that the distributions are very different, depending on the conditioning applied. This is not surprising, because we imagine that passengers in first and second class had different treatment compared to those in third class.

Let's visualize the distributions graphically:

```
pd.crosstab(titanic['Pclass'], titanic['Survived'],
        normalize=0).plot.bar(stacked=True)
plt.show()
```



The graph clearly shows the imbalance, from which we deduce that `Pclass` and `Survived` are in fact **associated** (or **correlated**). Indeed, knowing the class, tell us something about the **likely values of survived**.

Independence and Expected Frequencies

Before proceeding to illustrate some measures of association between variables, we must discuss the concept of **independence**, which is the opposite of association.

Two variables are said to be independent if the observation of one variable does not allow us to make any kind of prediction about the possible values of the other variable.

Imagine two randomly chosen people living in different, randomly chosen parts of the world. If we observe the way one of these two people dresses, this observation will tell us nothing about how the other might dress. The variables “clothes of the first person” and “clothes of the second person” are independent. The same does not apply if the people live in the same city. In that case, if it rains, for example, we expect the two people to wear similar clothes, so observing how one of the two people dresses can tell us something about how the other will dress.

Let’s consider the generic contingency:

	Y= y_1	**Y=** y_2	...	**Y=** y_l	**Total**
X= x_1	n_{11}	n_{12}	...	n_{1l}	$n_{1\cdot}$
X= x_2	n_{21}	n_{22}	...	n_{2l}	$n_{2\cdot}$
...	...	...	...	...	...
X= x_k	n_{k1}	n_{k2}	...	n_{kl}	$n_{k\cdot}$
Total	n_{+1}	n_{+2}	...	n_{+l}	n

Now suppose we cannot observe the co-occurrences, but can only observe the marginal values:

	Y= y_1	Y= y_2	...	Y= y_l	Total
X= x_1			...		$n_{1\cdot}$
X= x_2			...		$n_{2\cdot}$
...	...	...	...	...	...
X= x_k			...		$n_{k\cdot}$
Total	n_{+}	n_{+}	...	n_{+}	n

Now suppose we need to "reconstruct" the missing values. If the two variables are independent, then:

$$P(X = x_i, Y = y_j) = P(X = x_i)P(Y = y_j)$$

Recall that:

$$P(X = x_i) = \frac{n_{i+}}{n} \qquad P(Y = y_j) = \frac{n_{+j}}{n} \qquad P(X = x_i, Y = y_j) = \frac{n_{ij}}{n}$$

From here we obtain that the reconstructed frequencies $\tilde{n}_{ij}$ can be computed as follows:

$$\tilde{n}_{ij} = n \cdot P(X = x_i, Y = y_j) = n \cdot P(X = x_i) \cdot P(Y = y_j) = n \cdot \frac{n_{i+}}{n}\frac{n_{+j}}{n} = \frac{n_{i+}n_{+j}}{n}$$

From which:

$$\tilde{n}_{ij} = P(X = x_i)n_{+j} = P(Y = y_j)n_{+i}$$

Where the tilde on $\tilde{n}_{ij}$ indicates that this is an estimated value, not the actually observed value.

What has been seen above suggests that the frequencies n_{ij} follow the same distribution as the marginal frequencies in case of independence between the variables.

Example

Let's see an example on our contingency table putting `Pclass` and `Survived` in relation:

Survived	0	1	All
Pclass			
1	80	136	216
2	97	87	184
3	372	119	491
All	549	342	891

The marginal probability $P(Survived)$ is given by:

```
Survived
0    0.616162
1    0.383838
Name: proportion, dtype: float64
```

These are the probabilities $P(Survived = 0)$ and $P(Survived = 1)$.

If the two variables were not correlated, we would expect to have absolute frequencies proportional to the marginal probabilities:

$$\tilde{n}_{ij} = P(Y = y_j) \cdot n_{i+}$$

These values will be given by the following table:

Survived	0	1	All
Pclass			
1	133.090909	82.909091	216.0
2	113.373737	70.626263	184.0
3	302.535354	188.464646	491.0
All	549.000000	342.000000	891.0

Comparing these frequencies with the actual frequencies, we realize that there are discrepancies, reasonably due to the fact that the variables are not in fact independent.

Pearson's χ^2 Statistic

Pearson's χ^2 statistic measures these discrepancies with the following formula:

$$\chi^2 = \sum_{i=1}^{k} \sum_{j=1}^{l} \frac{(n_{ij} - \tilde{n}_{ij})^2}{\tilde{n}_{ij}}$$

The above formula calculates the squared differences between the observed and expected absolute frequencies (to eliminate negative signs) and scales the result by the expected frequencies. **Scaling the**

result ensures that small discrepancies obtained for small expected frequencies weigh more than small discrepancies obtained for large expected frequencies.

Note that if the expected frequencies are identical to the observed ones, the statistic is zero.

We can calculate Pearson's χ^2 statistic as follows:

```
from scipy.stats import chi2_contingency
contingency = pd.crosstab(titanic['Pclass'], titanic['Survived'])
chi2_contingency(contingency)
```

```
Chi2ContingencyResult(statistic=np.float64(102.88898875696056),
pvalue=np.float64(4.549251711298793e-23), dof=2, expected_freq=array([[133.09090909,
82.90909091],
       [113.37373737,  70.62626263],
       [302.53535354, 188.46464646]]))
```

The function returns:

- The statistic value (102.89);
- The p-value of an associated statistical test (we will see this in a future lecture);
- The number of degrees of freedom, a technical details of the distribution underlying the statistical test;
- The "expected frequencies", i.e., the frequencies we would expect in case of independence.

Please note that when computing the chi-squared statistics, **certain conditions must be met to ensure validity**. Most importantly, expected frequencies in each cell of the contingency table should be sufficiently large—typically at least 5. This helps approximate the sampling distribution to a chi-squared distribution. Additionally, observations should be independent, and the data should be categorical, organized into a contingency table with mutually exclusive categories.

For the moment, we will stop here. We will get back to this statistic when we'll talk about statistical tests.

Cramer's V Statistic

Pearson's χ^2 statistic is not normalized, in the sense that its maximum value depends on the sample size. To overcome this problem, Cramer's V statistic can be used, defined as follows:

$$V = \sqrt{\frac{\chi^2}{n(min(k,l)-1)}}$$

The result will be a value between 0 and 1.

Cramer's V statistic can be computed as follows:

```
from scipy.stats.contingency import association

print(f"{association(pd.crosstab(titanic['Pclass'], titanic['Survived'])):0.2f}")
```

```
0.34
```

This value indicates a small association between the two variables, because the value is greater than 0 but still not close to 1.

Associations Between Continuous Variables

When variables are continuous, it is no longer possible to construct contingency tables. In those cases, other tools are used, which we will see in this section.

We will consider the "diabetes" dataset for this part. The dataset was introduced in 1979 by Reaven and Miller and examines the relationship between blood measurements and insulin in 145 adults:

```
from statsmodels.datasets import get_rdataset
data = get_rdataset('Diabetes','heplots').data
data.head()
```

	relwt	glufast	glutest	instest	sspg	group
0	0.81	80	356	124	55	Normal
1	0.95	97	289	117	76	Normal
2	0.94	105	319	143	105	Normal
3	1.04	90	356	199	108	Normal
4	1.00	90	323	240	143	Normal

The 6 variables have the following meanings:

- `relwt` relative weight, expressed as the ratio of actual weight to expected weight, given the person's height, a numeric vector
- `glufast` fasting plasma glucose level, a numeric vector
- `glutest` test plasma glucose level, a measure of glucose intolerance, a numeric vector
- `instest` plasma insulin during test, a measure of insulin response to oral glucose, a numeric vector
- `sspg` steady state plasma glucose, a measure of insulin resistance, a numeric vector
- `group` diagnostic group, a factor with levels `Normal` `Chemical_Diabetic` `Overt_Diabetic`

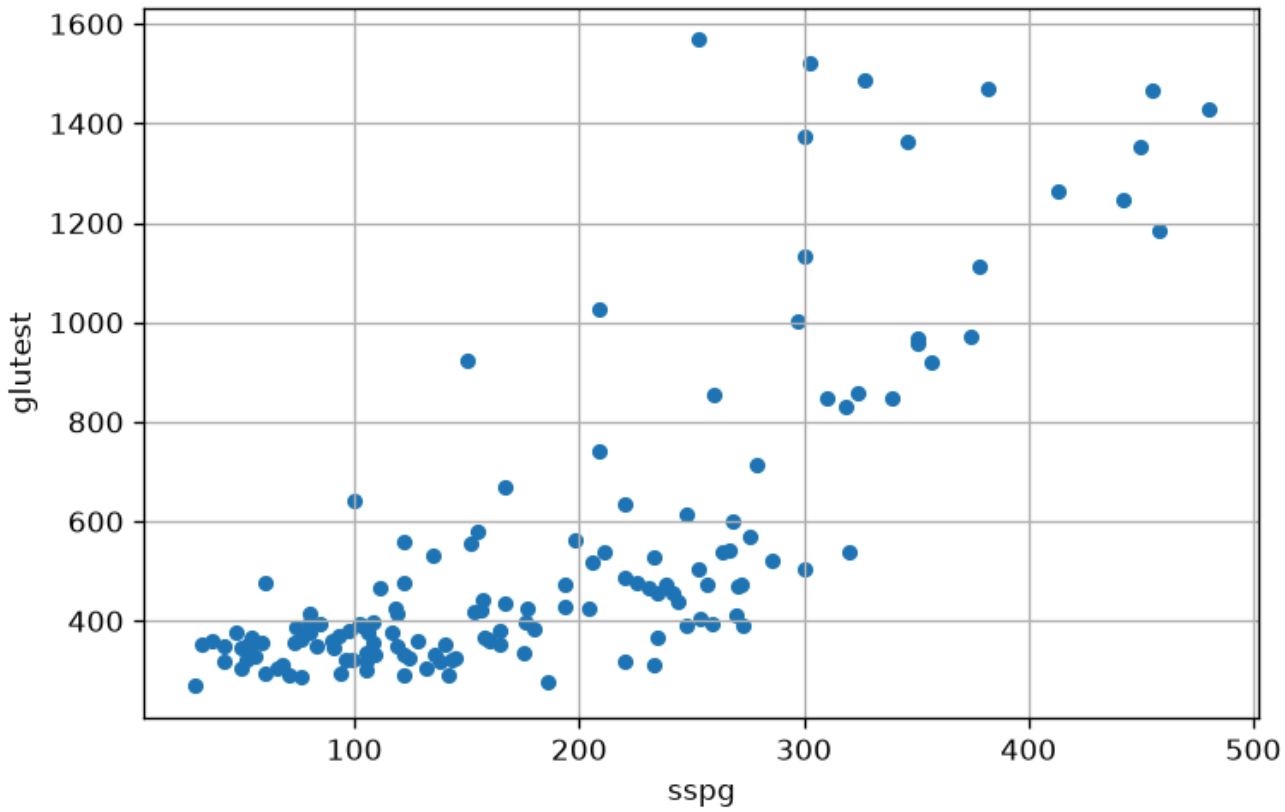
Graphical Representation of Two Continuous Variables

Let's start by considering the variables `sspg` and `glutest`.

Scatter plots

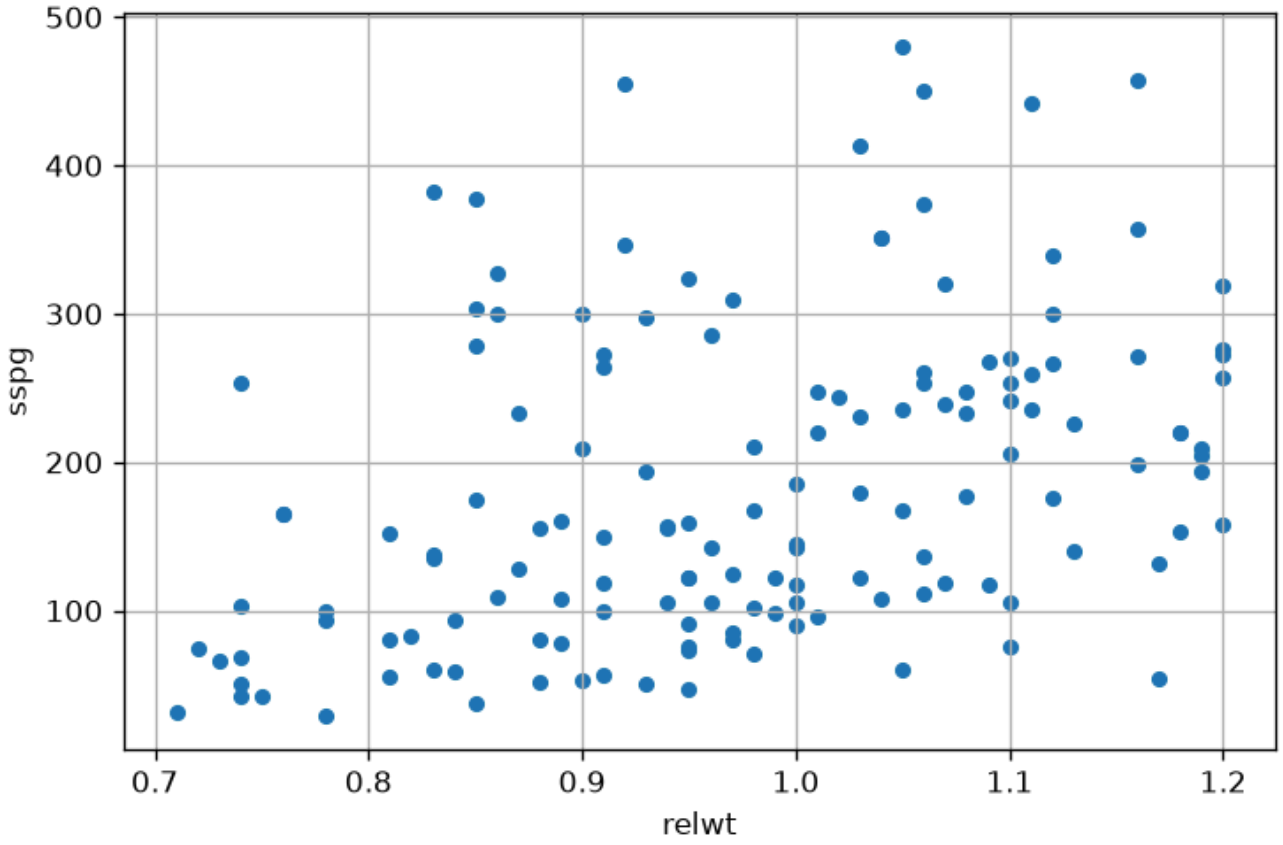
The first useful form of representation is the **scatter plot**, which visualizes observations as two-dimensional data:

```
from matplotlib import pyplot as plt
data.plot.scatter(x='sspg',y='glutest')
plt.grid()
```



We note that there seems to be a relationship between the two variables. Indeed, as `sspg` increases, the values of `glutest` tend to increase. There appears to be a positive **correlation** (or **association**). Let's now consider the pair `relwt` and `sspg`:

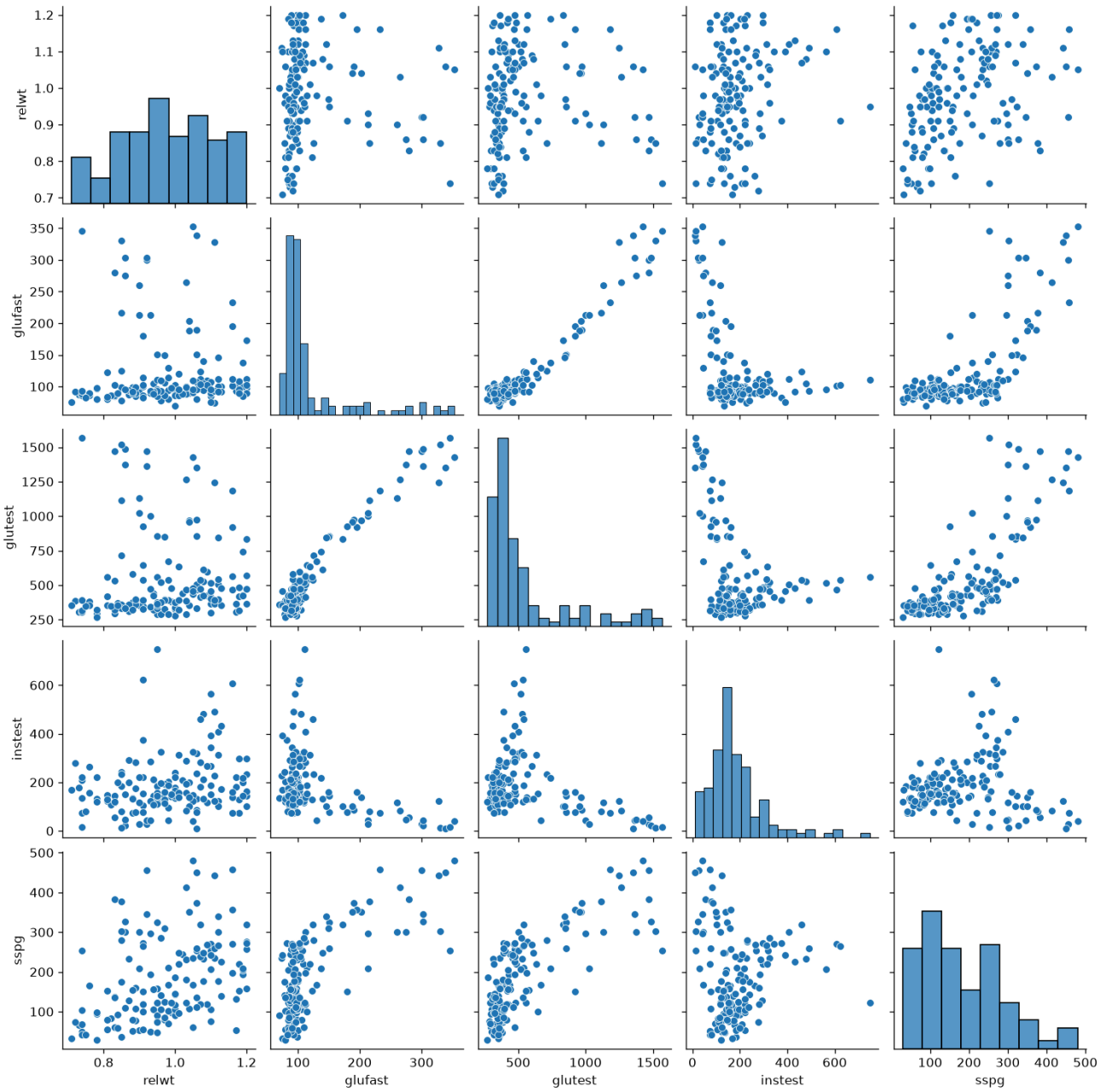
```
data.plot.scatter(x='relwt',y='sspg')
plt.grid()
```



Scatter Matrix

We note that in this case the relationship appears less marked, but we still note an ascending trend of `sspg` as `relwt` grows. It is often useful to try to study correlations with a **scattermatrix** that shows all possible scatterplots. In the case of our example:

```
from matplotlib import pyplot as plt
import seaborn as sns
sns.pairplot(data)
plt.show()
```

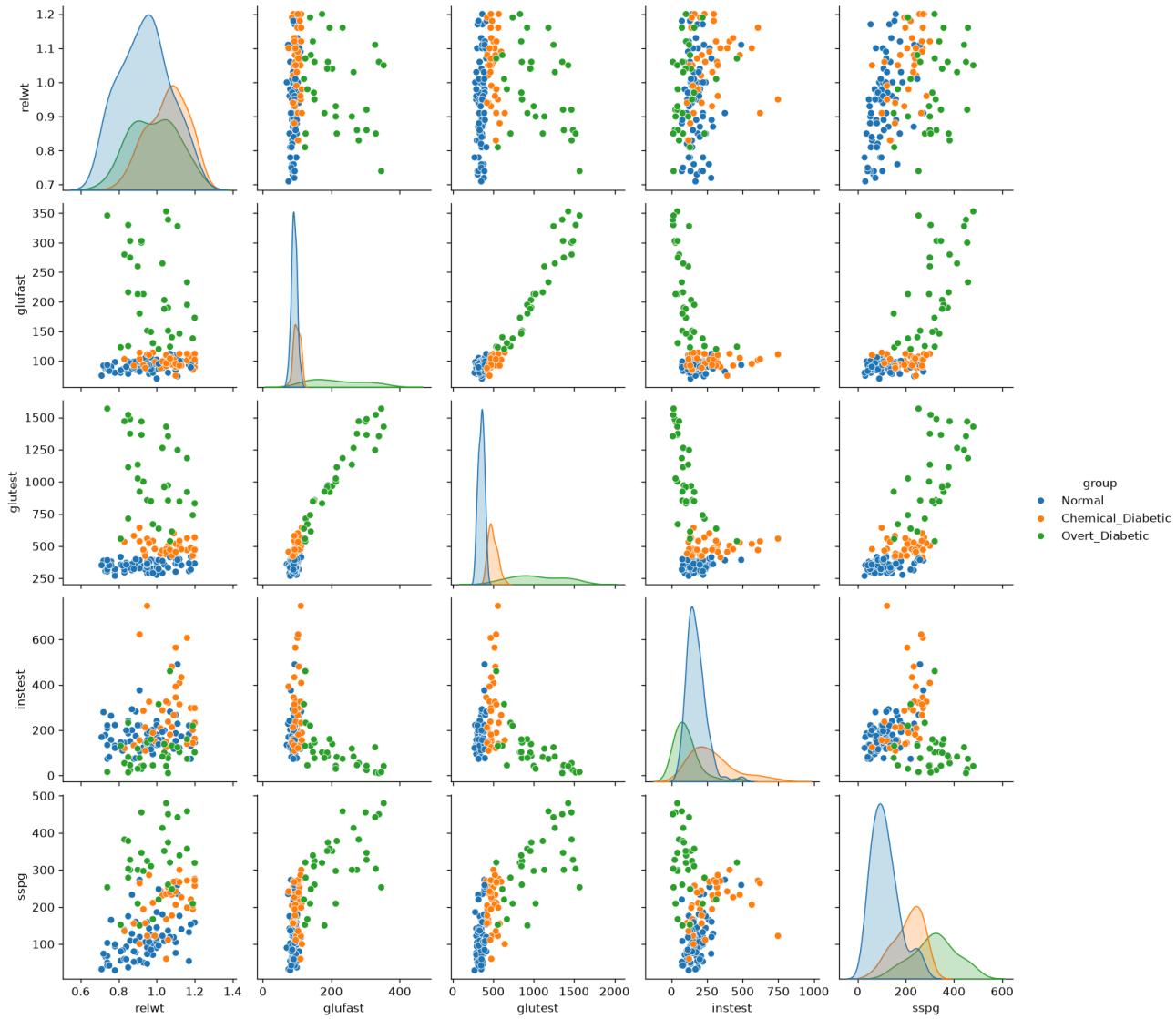


From the visualization above, we can deduce a few things:

- some pairs of variables are positively correlated (e.g., `relwt` - `sspg`), while others are negatively so (`glutest` - `instest`);
- some relationships are "linear" (e.g., `glutest` and `glufast`), while others tend not to be (e.g., `glufast` and `sspg`):
- some pairs of variables (e.g., `sspg` and `relwt`) do not appear to be correlated.

In our case, the data also contain a discrete variable `group` dividing the data into groups. We can reflect this in our visualization by specifying `hue='group'`:

```
from matplotlib import pyplot as plt
import seaborn as sns
sns.pairplot(data, hue='group')
plt.show()
```



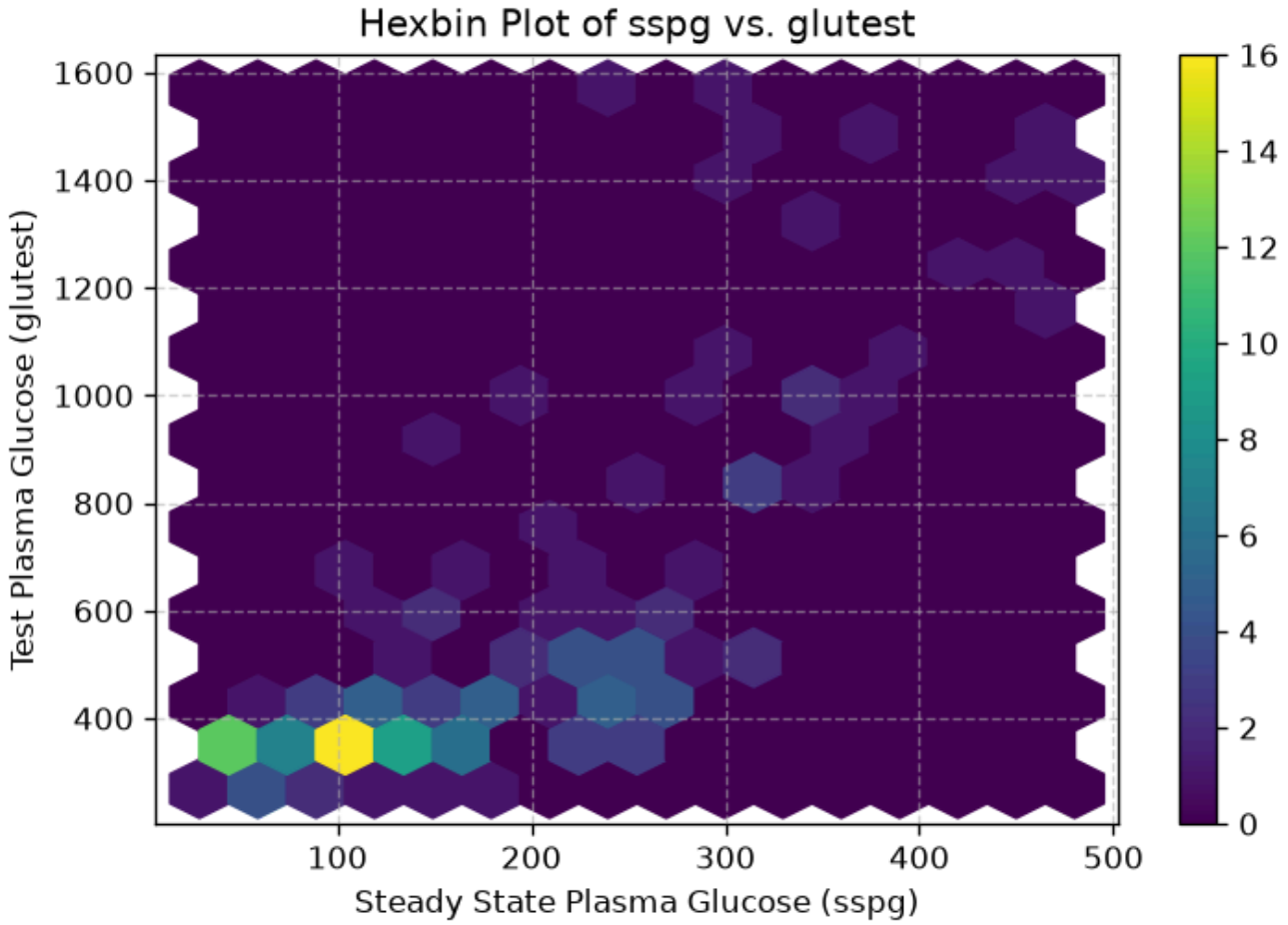
Hexbin Plots: A 2D Histogram

When a scatterplot contains a large number of data points, we can encounter the problem of **overplotting**, where points are plotted on top of each other. This makes it difficult to discern the density of the data, as heavily populated areas can look the same as moderately populated ones.

A **hexbin plot** is an excellent alternative that addresses this issue[cite: 48]. It divides the two-dimensional space into a grid of hexagonal bins and then colors each hexagon based on the number of data points it contains. Think of it as a two-dimensional histogram, which effectively visualizes data density.

Let's generate a hexbin plot for the `sspg` and `glutest` variables. We can adjust the `gridsize` parameter to control the number of bins (and thus their size).

```
from matplotlib import pyplot as plt
# let's use the hexbin function
# cmap is the color map
# gridsize regulates the number of hexagons
data.plot.hexbin(x='sspg', y='glutest', gridsize=15, cmap='viridis')
plt.xlabel('Steady State Plasma Glucose (sspg)')
plt.ylabel('Test Plasma Glucose (glutest)')
plt.title('Hexbin Plot of sspg vs. glutest')
plt.grid(linestyle='--', alpha=0.6)
plt.show()
```



This plot clearly shows that the highest concentration of data points (the darkest hexagons) is where both `sspg` and `glutest` values are low. This is a level of detail about the data's distribution that might be obscured in a dense, standard scatterplot.

Density and Contour Plots: A Smoother View of Distribution

Just as we used Kernel Density Estimation (KDE) to create a smooth curve representing the distribution of a single variable, we can apply the same principle to two variables. This gives us a smoother alternative to the binned approach of the hexbin plot.

The result is a **2D density plot**, which visualizes the estimated probability density of data points using a continuous color gradient. We can also overlay **contour lines**, which connect points of equal density, much like a topographical map shows lines of equal elevation. 🗺️

This method is excellent for identifying the shape and peaks of a bivariate distribution without being constrained by the grid of a histogram or hexbin plot. We'll revisit the theory behind **Kernel Density Estimation** in a later lecture, but for now, let's focus on its application.

Let's create a density plot for the `sspg` and `glutest` variables using the `seaborn` library.

```
import seaborn as sns
from matplotlib import pyplot as plt

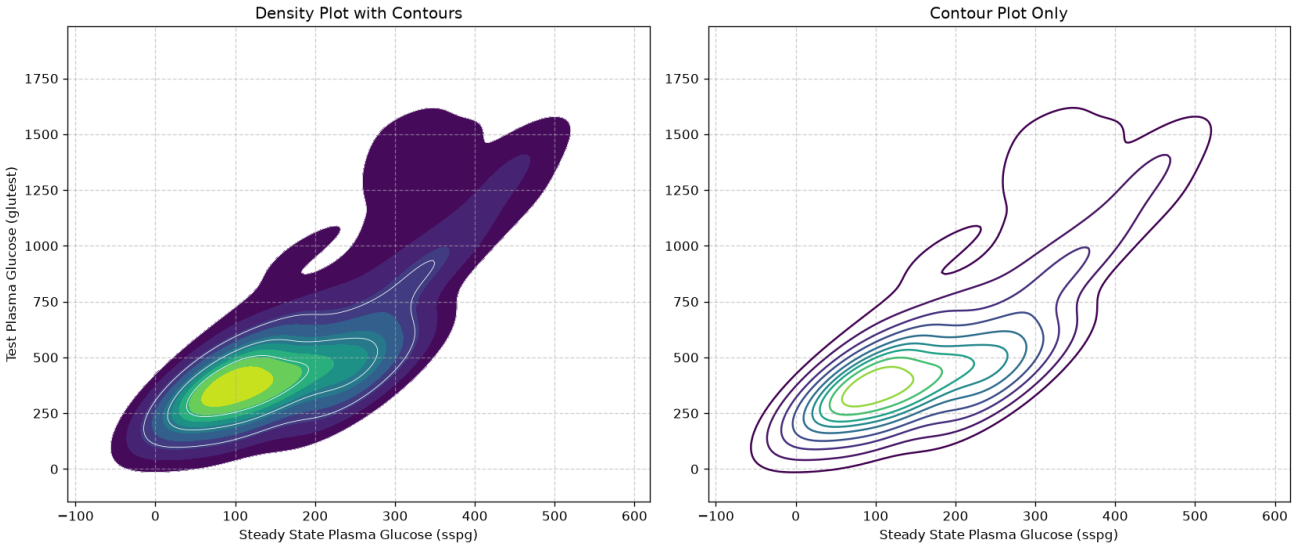
# Create a figure and a set of subplots (1 row, 2 columns)
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# --- Plot 1: Left Subplot (Density + Contour) ---
# we'll use kdeplot from seaborn
# we use it twice: first to fill the density, then to draw the contours
sns.kdeplot(data=data, x='sspg', y='glutest', fill=True, cmap='viridis', ax=axes[0])
# levels regulates the number of contour lines
# color and linewidths regulate the appearance of the contour lines
sns.kdeplot(data=data, x='sspg', y='glutest', levels=5, color='white',
        linewidths=0.5, ax=axes[0])
axes[0].set_title('Density Plot with Contours')
axes[0].set_xlabel('Steady State Plasma Glucose (sspg)')
axes[0].set_ylabel('Test Plasma Glucose (glutest)')
axes[0].grid(linestyle='--', alpha=0.6)

# --- Plot 2: Right Subplot (Contour Only) ---
# only the contours
sns.kdeplot(data=data, x='sspg', y='glutest', cmap='viridis', ax=axes[1])
axes[1].set_title('Contour Plot Only')
axes[1].set_xlabel('Steady State Plasma Glucose (sspg)')
axes[1].set_ylabel('') # Hide y-axis label for a cleaner look
axes[1].grid(linestyle='--', alpha=0.6)

# Adjust the layout to prevent titles/labels from overlapping
plt.tight_layout()

# Display the plots
plt.show()
```



In the plot on the left, the color-filled areas provide an intuitive map of the data's density; the darkest regions represent the highest concentration of points, while the contour lines enclose areas of progressively lower density. This visualization clearly highlights a single, dense cluster in the lower-left corner.

The **contour-only plot** on the right is particularly useful when you need to **compare multiple distributions on the same axes**. While overlaying several filled density plots would result in colors obscuring each other, using only the contour lines for each group allows you to clearly see how the centers and shapes of different distributions overlap or diverge without the visual clutter.

Covariance

We have seen that **variance** indicates how dispersed a univariate sample is around its mean. We can use a similar idea to quantify to what extent **two variables are distributed similarly around their mean**. We expect that, if the variables are correlated, then they will be distributed similarly.

The concept of **variance** is generalized by the concept of **covariance** in the case of bivariate data. Covariance measures how much the two variables vary together and is measured as follows:

$$Cov(X,Y) = \frac{1}{n}\sum_{i}^{n}(x^{(i)} - \overline{x})(y^{(i)} - \overline{y})$$

where X and Y are the two variables of interest, $x^{(i)}$ is the value of X for the i-th observation, $y^{(i)}$ is the value of Y for the i-th observation, and $\overline{x}$, $\overline{y}$ are the means of the observations for the two variables X and Y.

Note that we can obtain an unbiased estimation of the covariance similar to the variance as follows:

$$Cov(X,Y) = \frac{1}{n-1}\sum_{i}^{n}(x^{(i)} - \overline{x})(y^{(i)} - \overline{y})$$

This is important especially with small samples.

The main "actors" in the formula above are the terms:

- $(x^{(i)} - \overline{x})$: measures the distance between a point $x^{(i)}$ and its respective mean $\overline{x}$. If the value of the point in question is above the mean, then this difference will be positive, otherwise negative.
- $(y^{(i)} - \overline{y})$: measures the distance between a point $y^{(i)}$ and its respective mean $\overline{y}$. If the value of the point in question is above the mean, then this difference will be positive, otherwise negative.

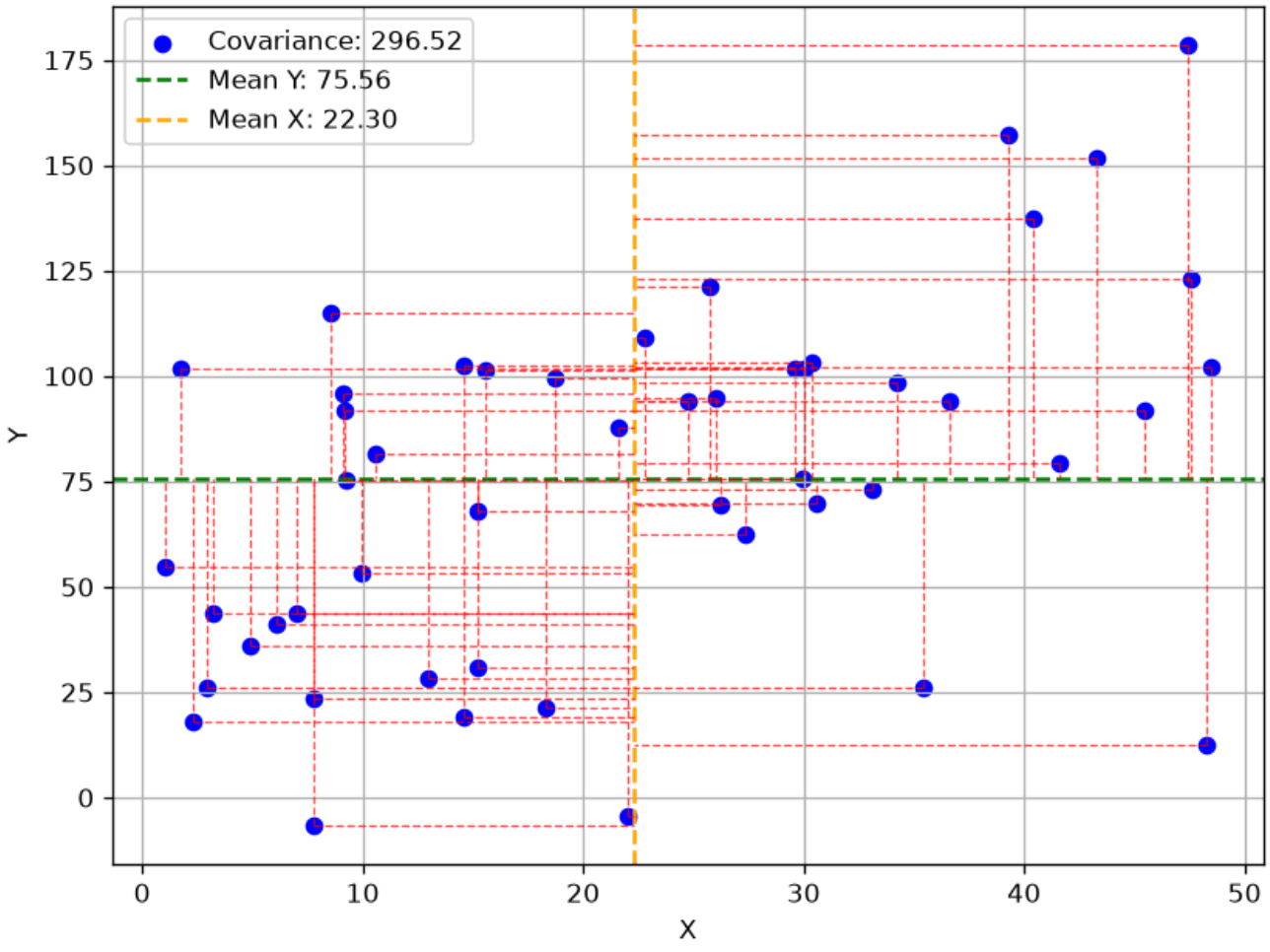
Each term of the sum in the covariance formula performs the products between these two terms:

$$(x^{(i)} - \overline{x})(y^{(i)} - \overline{y})$$

Which will be:

- **positive** if the elements $x^{(i)}$ and $y^{(i)}$ are "concordant", i.e., if they are both above their respective means or both below their respective means;
- **negative** if the elements $x^{(i)}$ and $y^{(i)}$ are "discordant", i.e., if one of the two is above its respective mean, while the other is below its respective mean;
- **zero or close to zero** if the elements $x^{(i)}$ and $y^{(i)}$ are very close to their respective means.

These quantities have a geometric interpretation. Let's consider the following plot:



The graph shows a sample of points and their respective means with a horizontal line (for Y) and a vertical line (for X). The two means divide the graph into four quadrants. For each point, the red dashed lines indicate the values of the magnitudes $(x^{(i)} - \overline{x})$ and $(y^{(i)} - \overline{y})$. Specifically:

- The points in the upper right quadrant will have both these magnitudes positive. The products $(x^{(i)} - \overline{x})(y^{(i)} - \overline{y})$ will be positive.
- The points in the lower right quadrant will have both these magnitudes negative. The products $(x^{(i)} - \overline{x})(y^{(i)} - \overline{y})$ will be positive.
- The points in the other two quadrants will have one of the magnitudes positive and the other negative. The products $(x^{(i)} - \overline{x})(y^{(i)} - \overline{y})$ will be negative.

In practice:

- points in the first and third quadrants contribute **positively** to the correlation coefficient;
- points in the second and fourth quadrants contribute **negatively** to the correlation coefficient.
- Points close to the mean (the intersection between the two means) will correspond to very small magnitudes and **will not contribute** to changing the covariance value.

Intuitively, the covariance will take on very positive values when X and Y vary in agreement (i.e., if X takes high values, Y takes high values, and if X takes low values, Y takes low values). Indeed, in such a case, the signs of the two factors within the summation will agree, and their product will have a positive sign. If, conversely, X and Y vary in disagreement, the product will have a negative sign, and the covariance will take on negative values.

We note that the covariance of a variable X with itself is equal to the variance:

$$Cov(X, X) = \frac{1}{n}\sum_{i}^{n}(x_i - \overline{x})(x_i - \overline{x}) = s_X$$

It should be noted that:

- Negative values indicate a negative correlation (when one of the two variables increases, the other decreases);
- Positive values indicate a positive correlation (the two variables increase or decrease together);
- Zero values (or close to zero) indicate that the two variables are not correlated (or are weakly correlated).

It should also be noted that covariance values are not normalized and depend on the ranges of the individual variables, therefore it is not possible to compare covariances. For example, the covariance between `relwt` and itself (thus the variance of `relwt`) is close to zero; however, this does not mean that `relwt` is not correlated with itself!

In practice, in the presence of multiple variables, the covariances of all possible pairs of variables are calculated, somewhat as seen in the case of the scatter matrix. We can obtain all these covariances with the cov function, as follows:

```
data.drop('group',axis=1).cov() #drop the group column to compute covariance only on
        numerical columns
```

	relwt	glufast	glutest	instest	sspg
relwt	0.016702	-0.072815	0.982426	3.473373	5.266255
glufast	-0.072815	4087.097031	19546.064080	-3063.463649	4849.905651
glutest	0.982426	19546.064080	100457.849808	-12918.162739	25908.490182
instest	3.473373	-3063.463649	-12918.162739	14625.312548	101.482519
sspg	5.266255	4849.905651	25908.490182	101.482519	11242.331897

The matrix seen above is also called "covariance matrix" and denoted by the letter Σ. Given the variables $X = (X_i, \dots, X_k)$, Σ is a $[k \times k]$ matrix whose general term is:

$$\Sigma_{ij} = Cov(X_i, X_j)$$

Pearson Correlation Coefficient

The Pearson correlation coefficient attempts to solve the problems of covariance by providing a normalized quantitative score. Given a bivariate sample $\{(x^{(i)}, y^{(i)})\}$, the Pearson correlation coefficient is defined as the covariance between the two variables, divided by the product of their standard deviations:

$$\rho(x, y) = \frac{Cov(X, Y)}{s_X s_Y}$$

We note that:

$$\rho(x, y) = \frac{Cov(X, Y)}{s_X s_Y} = \frac{1}{n} \sum \frac{(x_i - \overline{x})}{s_X} \frac{(y_i - \overline{y})}{s_X} = Cov(z(X), z(Y))$$

Where

$$z(X) = \frac{X - \overline{X}}{s_x}$$

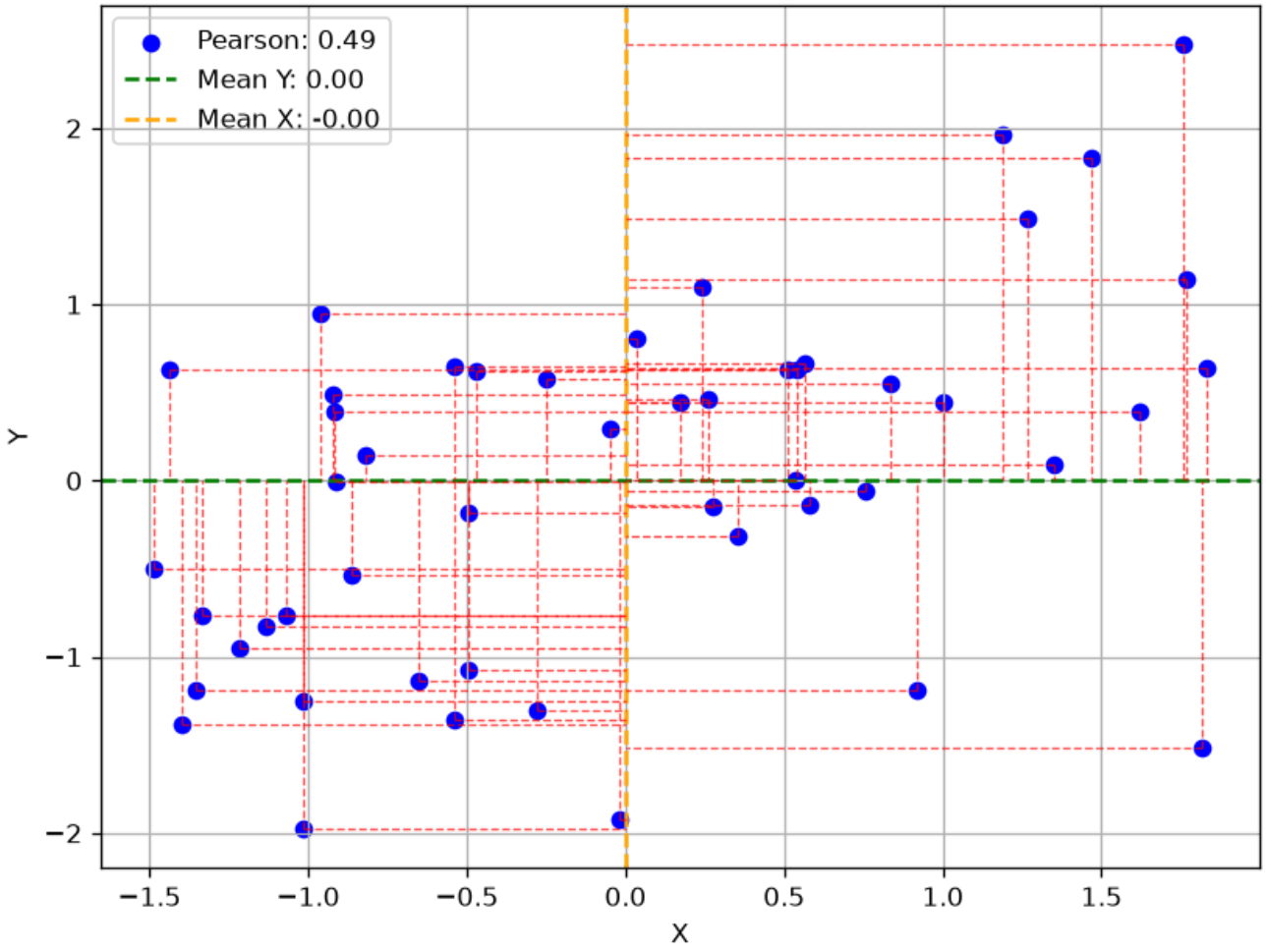
is the z-scoring function.

We can therefore view the Pearson correlation coefficient as the covariance of variables normalized with z-scoring. It can be easily seen that the Pearson correlation coefficient can also be written as follows:

$$\rho(x, y) = \frac{\sum_{i=1}^{n}(x^{(i)} - \overline{x})(y^{(i)} - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x^{(i)} - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y^{(i)} - \overline{y})^2}}$$

Where $\overline{x}$ and $\overline{y}$ are respectively the means of the values of the two variables.

The following graph shows a plot analogous to the previous one after performing z-scoring. Note how the ranges of x and y are now normalized and the mean is at the origin.



Pearson's correlation coefficient is a number between -1 (maximum anticorrelation) and 1 (maximum correlation).

In practice, you will have:

- $\rho(x, -x) = -1$
- $\rho(x, x) = 1$

The correlation coefficient between the variables `relwt` and `sspg` can be computed with the pearsonr function:

```
from scipy.stats import pearsonr
pearsonr(data['relwt'], data['sspg'])
```

```
PearsonRResult(statistic=np.float64(0.38431980355592876),
pvalue=np.float64(1.820843846790297e-06))
```

This function returns both the statistic (0.38) and a p-value, associated to a statistical test, as we will see in a future lecture.

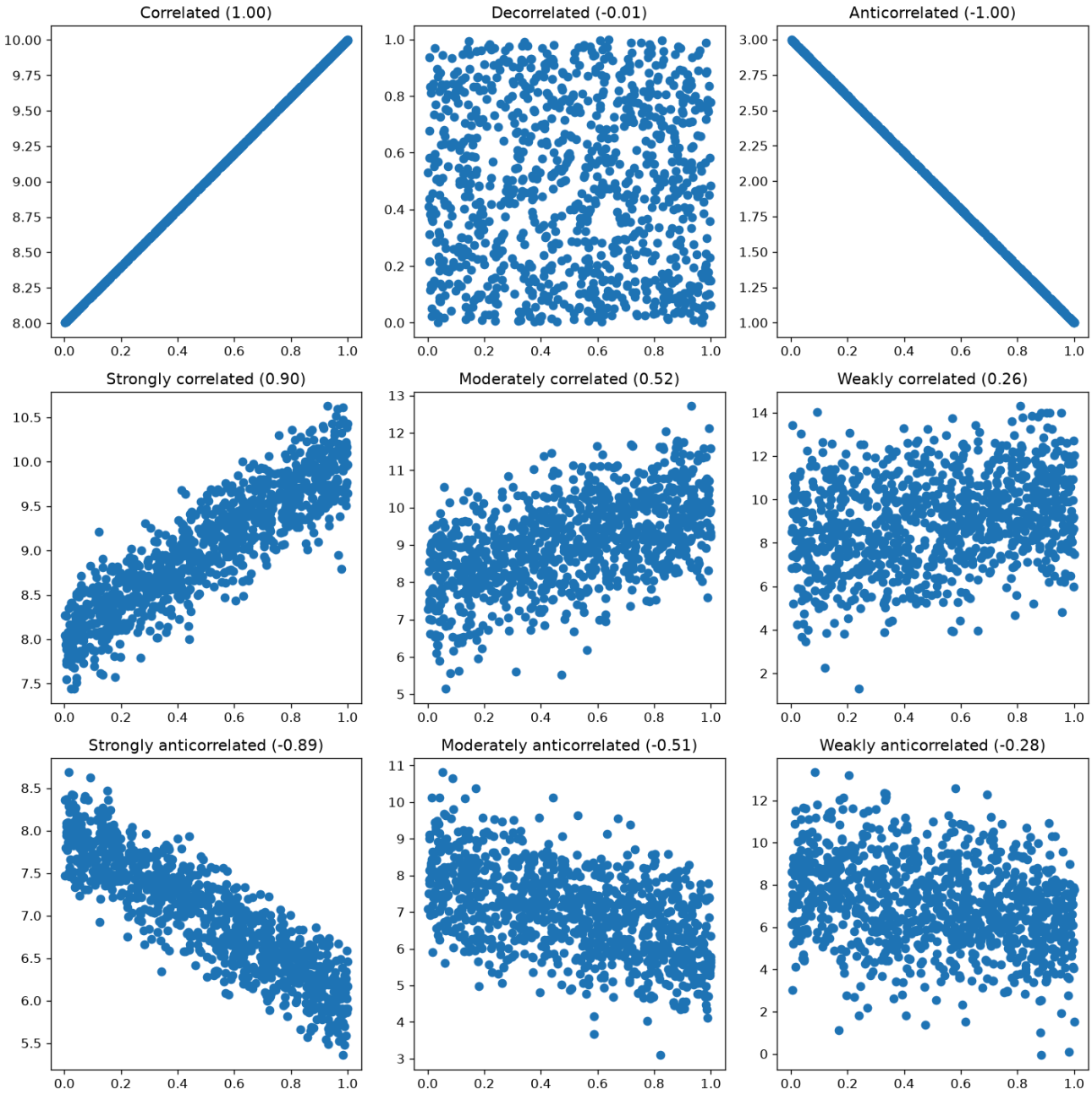
Interpretation of Pearson's Correlation Coefficient Values

Pearson's correlation coefficient is a number between -1 and 1 which indicates whether there is a correlation (i.e., when the values of the first variable increase, the values of the second variable also increase) or an anticorrelation (i.e., when the values of the first variable increase, the values of the second variable decrease and vice versa). Values close to -1 indicate that the variables are anticorrelated; values close to 1 indicate that the variables are correlated; values close to 0 indicate that the variables are decorrelated. In practice:

- The sign of the coefficient indicates the "direction" of the correlation:
  - Positive: the variables are correlated;
  - Negative: the variables are anticorrelated;
- The absolute value of the coefficient indicates how strong the correlation (or anticorrelation) is:
  - for values between 0 and 0.3 it indicates a **weak** correlation (or anticorrelation);
  - for values between 0.3 and 0.7 it indicates a **moderate** correlation (or anticorrelation);
  - for values between 0.7 and 1 it indicates a **strong** correlation (or anticorrelation).

Note that Pearson's correlation coefficient only measures **linear correlations**. Therefore, a non-linear correlation might exist between two variables even if the obtained correlation coefficient is low.

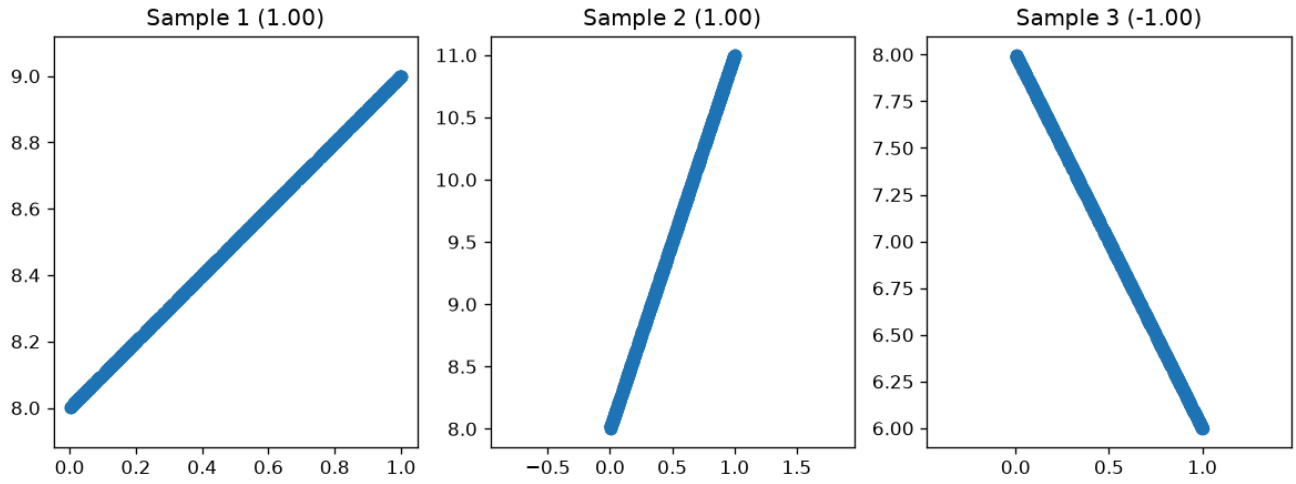
The following graph shows the scatterplots of some example bivariate samples with different degrees of correlation, together with their respective Pearson correlation coefficients:



Observing the graphs shown above, we can note that the following relationships exist between index and scatterplot:

- A positive index identifies an upward trend (a line with a positive slope);
- A negative index identifies a downward trend (a line with a negative slope);
- The absolute value of the index is correlated with the “width” of the “corridor” formed by the data.

Note that there is no relationship between Pearson’s correlation coefficient and the slope of the line. We can see this in the following graph:



Pearson Correlation Coefficient and Discrete Variables

It's important to remember that the **Pearson correlation coefficient is designed to measure the linear relationship between two continuous variables**. It is not suitable for nominal categorical variables (e.g., "City" or "Favorite Color"), as the categories have no inherent numerical order.

However, a notable exception exists for the specific case where you have **one continuous variable and one dichotomous** variable (a discrete variable with only two categories, like "Yes/No" or "Survived/Died"). In this scenario, you can convert the two categories into the numbers `0` and `1`. By applying the standard Pearson formula to this data, you calculate what is known as the **point-biserial correlation coefficient (r_{pb})**.

The example below shows how to compute the point-biserial correlation coefficient between `Sex` and `Age`:

```
import pandas as pd
from scipy.stats import pointbiserialr, pearsonr

# Load the dataset
titanic = pd.read_csv(DATA + "titanic.csv")

# Drop rows where 'Age' is missing to perform the correlation
titanic_cleaned = titanic.dropna(subset=['Age'])

# Extract the two variables of interest
age = titanic_cleaned['Age']
survived = titanic_cleaned['Sex'].replace({'male':'1','female':'0'}).astype(int)

# --- Method 1: Using the specific pointbiserialr function ---
pbr_result = pointbiserialr(age, survived)
print(f"Point-Biserial Correlation: {pbr_result.correlation:.4f}")

# --- Method 2: Verifying with the standard Pearson correlation function ---
# The result should be identical
pearson_result = pearsonr(age, survived)
print(f"Pearson Correlation (on the same data): {pearson_result[0]:.4f}")
```

```
Point-Biserial Correlation: 0.0933
Pearson Correlation (on the same data): 0.0933
```

We see a small positive association: males tend to be older. This is also confirmed by the boxplot below:

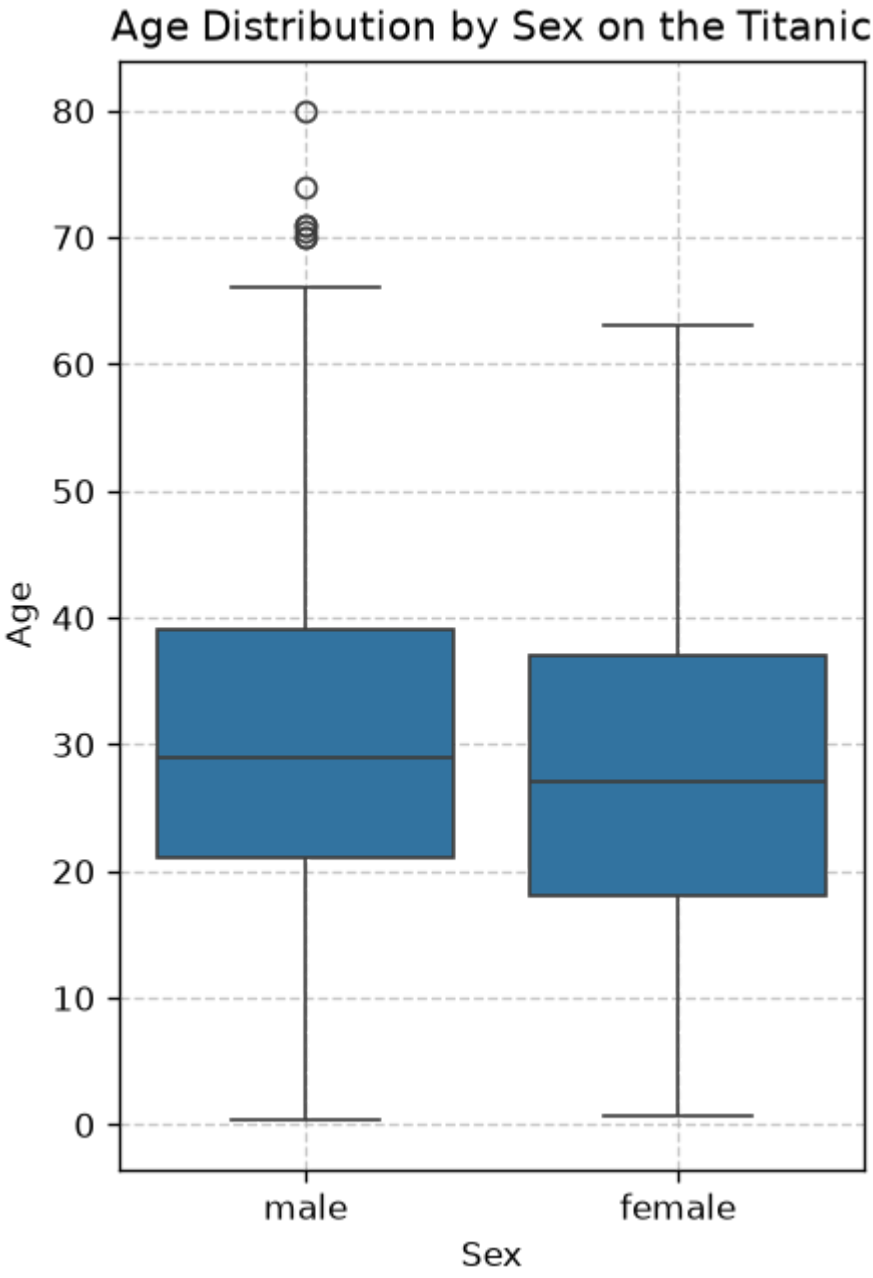
```
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt

# Set up the plot
plt.figure(figsize=(4, 6))

# Create boxplots to compare the age distribution by sex
sns.boxplot(data=titanic_cleaned, x='Sex', y='Age')

# Add titles and labels for clarity
plt.title('Age Distribution by Sex on the Titanic')
plt.xlabel('Sex')
plt.ylabel('Age')
plt.grid(linestyle='--', alpha=0.7)

# Display the plot
plt.show()
```



Spearman's Rank Correlation Coefficient

Pearson's coefficient captures linear relationships between variables. However, there are cases where the relationship between two variables may not be linear, but it still makes sense to try and understand if a correlation exists.

Imagine we have a series of users who assign scores to products they review. Each user will use a different and probably non-linear subjective scale. Pearson's correlation coefficient would not be the most appropriate to check if the scores assigned by two users are correlated.

Spearman's correlation coefficient tries to solve this problem by moving from scores to ranks. Instead of reasoning about individual scores, Spearman's correlation coefficient orders the products by score and checks if the resulting orderings are similar.

Consider the sample $\{(x^{(i)}, y^{(i)})\}_i$. Let $R(x^{(i)})$ and $R(y^{(i)})$ be the ranks associated with the values of the two variables, and let $d_i = R(x^{(i)}) - R(y^{(i)})$. Spearman's rank correlation coefficient is defined as follows:

$$R = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)}$$

The values of Spearman's coefficient are normalized between -1 and $+1$ and have a similar interpretation to Pearson's coefficient.

Consider the following example. We ask Alice, Bob, Charlie, and David to assign a score from 1 to 5 to four ice cream flavors, obtaining these results:

```
import pandas as pd
from scipy.stats import spearmanr

# Create a DataFrame with ice cream flavor scores
data = {
    'Participant': ['Alice', 'Bob', 'Charlie', 'David'],
    'Chocolate': [5, 3, 4, 2],   # Scores out of 5
    'Vanilla': [4, 5, 3, 3],      # Scores out of 5
    'Strawberry': [3, 4, 5, 2],   # Scores out of 5
    'MintChip': [2, 2, 2, 5]      # Scores out of 5
}

df = pd.DataFrame(data)
df
```

	Participant	Chocolate	Vanilla	Strawberry	MintChip
0	Alice	5	4	3	2
1	Bob	3	5	4	2
2	Charlie	4	3	5	2
3	David	2	3	2	5

Given that each child can use a different scale, we use Spearman's coefficient, obtaining the following values:

```
# Calculate the Spearman correlation coefficients
spearman_corr_chocolate_vs_vanilla, _ = spearmanr(df['Chocolate'], df['Vanilla'])
spearman_corr_chocolate_vs_strawberry, _ = spearmanr(df['Chocolate'],
        df['Strawberry'])
spearman_corr_chocolate_vs_mintchip, _ = spearmanr(df['Chocolate'], df['MintChip'])

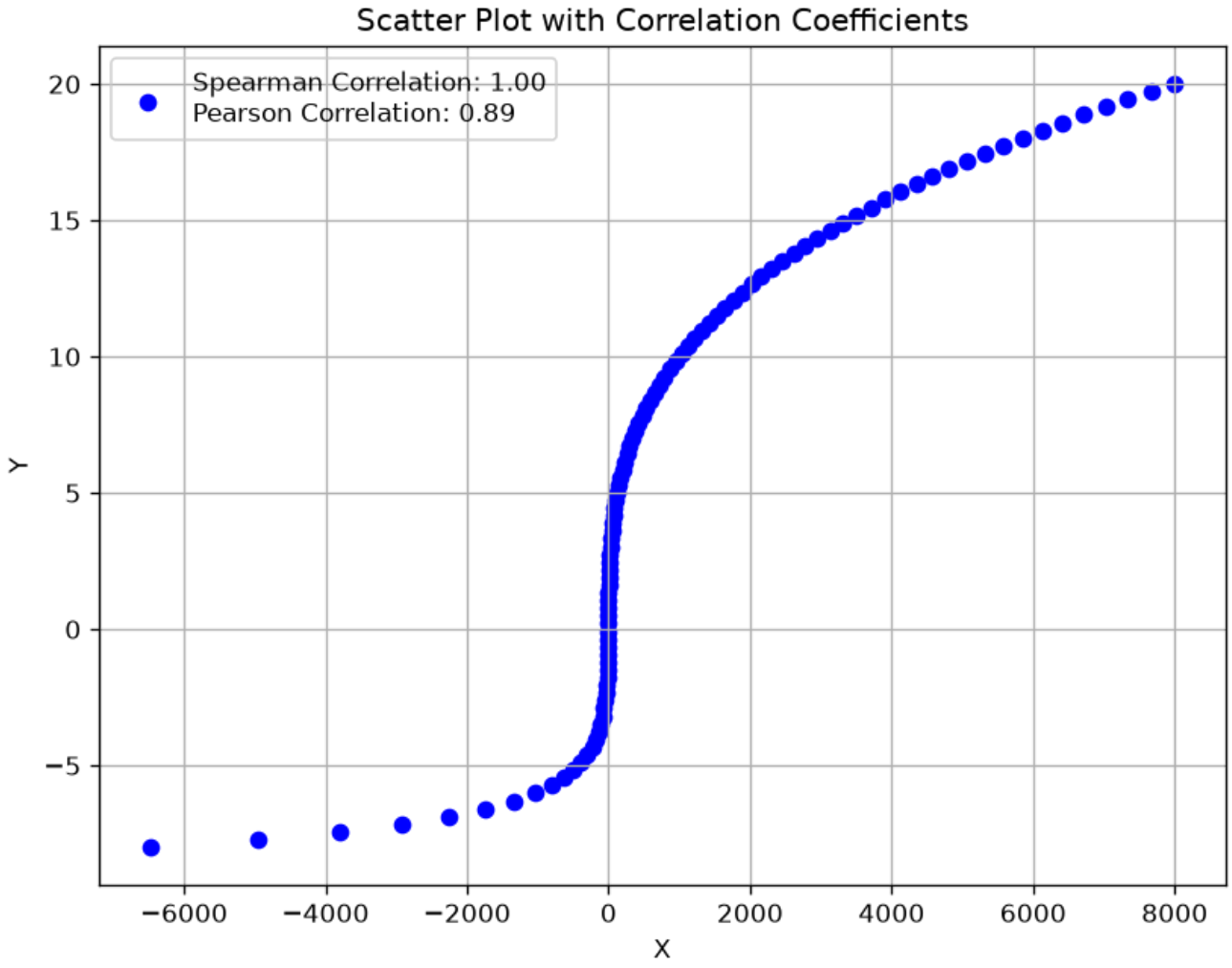
# Calculate and display the Spearman correlation coefficients
print(f"Spearman Correlation (Chocolate vs. Vanilla):
        {spearman_corr_chocolate_vs_vanilla:.2f}")
print(f"Spearman Correlation (Chocolate vs. Strawberry):
        {spearman_corr_chocolate_vs_strawberry:.2f}")
print(f"Spearman Correlation (Chocolate vs. MintChip):
        {spearman_corr_chocolate_vs_mintchip:.2f}")
```

```
Spearman Correlation (Chocolate vs. Vanilla): 0.21
Spearman Correlation (Chocolate vs. Strawberry): 0.40
Spearman Correlation (Chocolate vs. MintChip): -0.77
```

Spearman correlation values are computed using the `spearmanr` function.

These values tell us that the ice cream flavor that received scores “most similar” to chocolate is strawberry.

Unlike the Pearson coefficient, Spearman’s index can capture non-linear relationships, as shown in the following example:



Kendall's Correlation Coefficient

Kendall's rank correlation coefficient, also known as **Kendall's τ**, is also used to measure the level of association between two variables, considering their ranks.

Let's consider the sample $\{(x^{(i)}, y^{(i)})\}_i$. Each pair of observations (x_i, y_i) and (x_j, y_j), with $i < j$ is called:

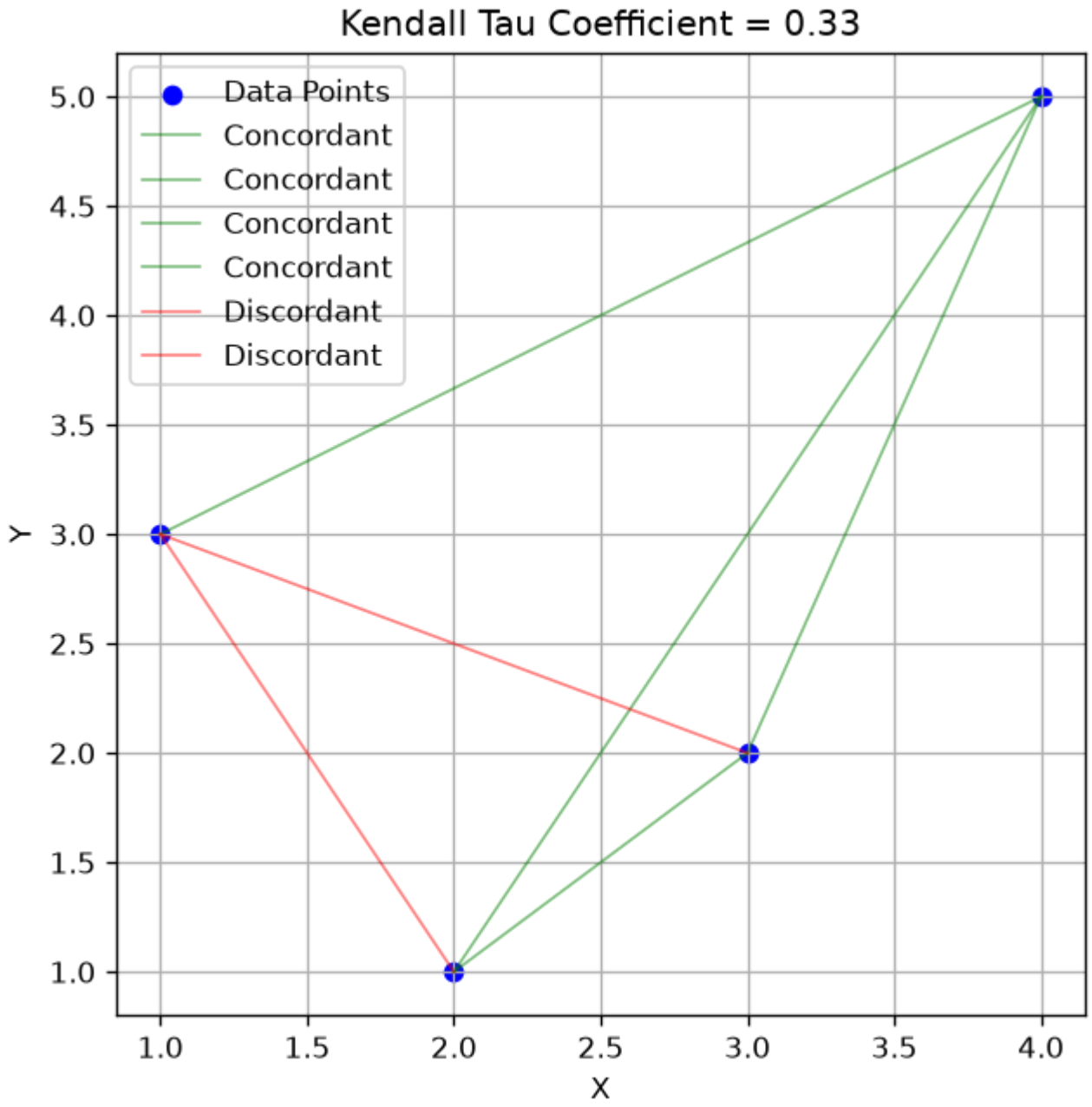
- **concordant** if: $x_i > x_j$ and $y_i > y_j$ or $x_i < x_j$ and $y_i < y_j$;
- **discordant** if: $x_i < x_j$ and $y_i > y_j$ or $x_i > x_j$ and $y_i < y_j$;

Kendall's τ coefficient is defined as follows:

$$\tau = \frac{\text{concordant pairs} - \text{discordant pairs}}{\text{pairs}}$$

The following graph shows a sample with six points. Each line shows a possible pair of points. The green lines show the concordant pairs, while the red ones show the discordant pairs. Kendall's coefficient for this example is given by:

$$\tau = \frac{4-2}{6} = \frac{1}{3}$$



The kendall correlation coefficients for our earlier examples can be computed as follows:

```
from scipy.stats import kendalltau

# Calculate the Kendall correlation coefficients
kendall_corr_chocolate_vs_vanilla, _ = kendalltau(df['Chocolate'], df['Vanilla'])
kendall_corr_chocolate_vs_strawberry, _ = kendalltau(df['Chocolate'],
        df['Strawberry'])
kendall_corr_chocolate_vs_mintchip, _ = kendalltau(df['Chocolate'], df['MintChip'])

# Display the Kendall correlation coefficients
print(f"Kendall Correlation (Chocolate vs. Vanilla):
        {kendall_corr_chocolate_vs_vanilla:.2f}")
print(f"Kendall Correlation (Chocolate vs. Strawberry):
        {kendall_corr_chocolate_vs_strawberry:.2f}")
print(f"Kendall Correlation (Chocolate vs. MintChip):
        {kendall_corr_chocolate_vs_mintchip:.2f}")
```

```
Kendall Correlation (Chocolate vs. Vanilla): 0.18
Kendall Correlation (Chocolate vs. Strawberry): 0.33
Kendall Correlation (Chocolate vs. MintChip): -0.71
```

Correlation Matrix

Similarly to what was seen in the case of scatterplots, when we have multiple variables, we can calculate the correlation indices between all variables in the dataset. In this way, we obtain a "correlation matrix". We can compute a correlation matrix with the `corr` function. The correlation matrix calculated using Pearson's coefficient, in the case of our `diabetes` dataset, will be:

```
from statsmodels.datasets import get_rdataset
data = get_rdataset('Diabetes','heplots').data
data.drop('group',axis=1).corr()
```

	relwt	glufast	glutest	instest	sspg
relwt	1.000000	-0.008813	0.023984	0.222238	0.384320
glufast	-0.008813	1.000000	0.964628	-0.396235	0.715480
glutest	0.023984	0.964628	1.000000	-0.337020	0.770942
instest	0.222238	-0.396235	-0.337020	1.000000	0.007914
sspg	0.384320	0.715480	0.770942	0.007914	1.000000

We can specify spearman or kendall coefficient as follows:

```
data.drop('group',axis=1).corr(method='spearman')
```

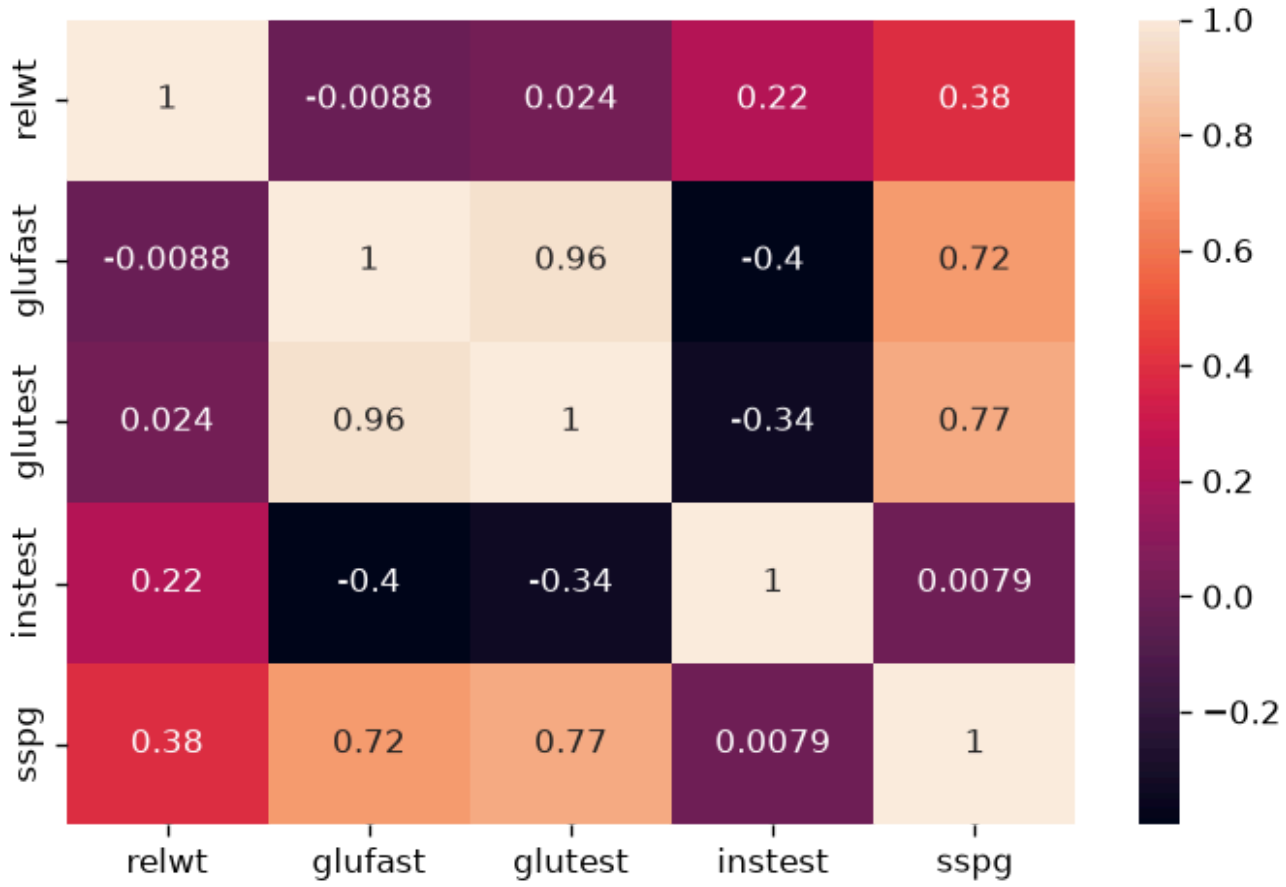
	relwt	glufast	glutest	instest	sspg
relwt	1.000000	0.204332	0.244952	0.231419	0.421998
glufast	0.204332	1.000000	0.767299	-0.265128	0.615134
glutest	0.244952	0.767299	1.000000	-0.164513	0.747564
instest	0.231419	-0.265128	-0.164513	1.000000	0.019898
sspg	0.421998	0.615134	0.747564	0.019898	1.000000

```
data.drop('group',axis=1).corr(method='kendall')
```

	relwt	glufast	glutest	instest	sspg
relwt	1.000000	0.137786	0.162601	0.158357	0.295005
glufast	0.137786	1.000000	0.608882	-0.175293	0.452355
glutest	0.162601	0.608882	1.000000	-0.091613	0.552304
instest	0.158357	-0.175293	-0.091613	1.000000	0.051423
sspg	0.295005	0.452355	0.552304	0.051423	1.000000

There is also a graphical representation of the correlation matrix, often called “correlation plot”:

```
sns.heatmap(data.drop('group',axis=1).corr(), annot=True)
plt.show()
```



Additional Measures of Correlation Between Discrete Variables (Optional)

This section reports optional material on two measures of correlation between discrete variables which are often used in epidemiology and biostatistics.

Relative Risk

Relative risk is one of the most common methods for measuring the degree of correlation between two discrete variables in epidemiology.

Let's look at an example (from here): suppose that in a village of 1000 people, the incidence of a rare disease has increased. Investigating, we discover that recently a part of the population has been exposed to radiation.

The data relating to diseased and healthy subjects exposed or not exposed to the risk (the radiation) is summarized by the following contingency table that relates the "Diseased/Healthy" variable to the "Exposed/Non Exposed" one:

```
contingency = pd.DataFrame({
    "Exposed": [20, 380],
    "Non Exposed": [6,594]
}, index=["Diseased", "Healthy"])
contingency
```

	Exposed	Non Exposed
Diseased	20	6
Healthy	380	594

The table indicates the number of people who have contracted a given disease or not, in relation to whether they have been exposed to a certain risk or not (e.g., to a pollutant).

The risk of developing the disease if exposed can be calculated with the following probability:

$$P(Diseased|Exposed) = \frac{\#\left(Diseased, Exposed\right)}{\#\ Exposed} = \frac{20}{20+380} = 0.05$$

We see that we have a 5% risk of developing the disease if exposed to radiation. For example, among 100 people exposed to the risk, we expect to find 5 sick.

The risk calculated above, by itself, does not tell us much about the association between the variables "Exposed" and "Non Exposed". Let's now consider the probability of contracting the disease if not exposed to the risk:

$$P(Diseased|Non\ Exposed) = \frac{\#\left(Diseased, Non\ Exposed\right)}{\#\ Non\ Exposed} = \frac{6}{6+594} = 0.01$$

This value tells us that we have a risk of about 1% of contracting the disease if not exposed. In practice, out of 100 subjects not exposed to the risk, 1 gets sick.

It is useful to compare these two risk values. Indeed, although the 5% risk of getting sick among exposed subjects is not a very high number in absolute terms, it must also be said that the proportion of sick subjects not exposed to the risk is much lower (1%). We therefore define relative risk as the ratio between these two risks (or probabilities):

$$RR = \frac{P(Diseased|Exposed)}{P(Diseased|Non\ Exposed)} = \frac{\frac{\#\left(Diseased, Exposed\right)}{\#\ Exposed}}{\frac{\#\left(Diseased, Non\ Exposed\right)}{\#\ Non\ Exposed}} = \frac{0.05}{0.01} = 5$$

We interpret this value as follows:

> The proportion of subjects who contract the disease is 5 times greater among those who have been exposed to the risk, compared to the group of those who have not.

For example, if among 100 people not exposed to the risk, generally 10 get sick (a risk of 10%), then we expect that among 100 people exposed to the risk, as many as 50% will get sick!

Relative risk tells us nothing about the absolute risk (the proportion of people who get sick), but only about how observing subjects exposed to the risk, compared to observing subjects not exposed to the risk, can influence the proportion of sick subjects.

Generally:

- $RR = 1$ indicates that exposure to the risk is irrelevant in the development of the disease (the variables are independent);
- $RR > 1$ indicates that exposure to the risk is positively associated with the development of the disease (there are more sick people among subjects exposed to the risk);
- $RR < 1$ indicates that exposure to the risk is negatively associated with the development of the disease (there are fewer sick people among subjects exposed to the risk). In this case, it is said that the risk "protects" against the onset of the disease (useful if the "risk" is actually taking a drug).

Note that relative risk measures a correlation between variables, not a cause-and-effect relationship, so it is not always correct to say that in the presence of a relative risk greater than 1, exposure to the risk "causes" the disease. However, there are tools (causal analysis) that allow us to verify in which cases we can give this interpretation. These tools are fundamental for ascertaining whether a particular drug causes (or is co-responsible for) a recovery or whether a particular habit (e.g., smoking) causes (or is co-responsible for) the onset of a disease. We will see more examples later in the course.

Odds Ratio

Relative risk cannot always be calculated. In fact, in the previous case, we assumed we had access to all population data: for all 1000 inhabitants, we knew who had been exposed to the risk and who had not, and who had developed the disease and who had not. In many cases, we might not have all this data. For example, imagine that the incident occurred 50 years ago and that at the time only a random sample of 50% of the 1000 inhabitants of the village was interviewed. Suppose we obtain the following contingency table:

```
contingency2 = pd.DataFrame({
    "Exposed": [20, 10],
    "Non Exposed": [6,16]
}, index=["Diseased", "Healthy"])
contingency2
```

	Exposed	Non Exposed
Diseased	20	6
Healthy	10	16

Note that this case (that of sampling) will be the most frequent, as we often do not have access to the entire population, but only to a sample.

In this case, we cannot calculate the relative risk as the real numbers of people exposed and not exposed to risk, which were in the denominator of the risk calculation expressions, are not available. If we still apply the formula, we will get an estimate biased by our sampling:

$$RR = \frac{P(Diseased|Exposed)}{P(Diseased|Non\ Exposed)} = \frac{\frac{\#\ (Diseased, Exposed)}{\#\ Exposed}}{\frac{\#\ (Diseased, Non\ Exposed)}{\#\ Non\ Exposed}} = \frac{20/(20+10)}{6/(6+16)} = 2.45$$

Although we cannot calculate the relative risk as we do not know the absolute number of sick and exposed people, **we expect our sample to be a good representation of the population**, therefore we can reason in terms of "odds".

Odds are a probability concept widely used in betting, because they allow the bettor to easily calculate the payout to collect. For example, if team A's victory over team B in a soccer match is given as $5:1$, the bookmaker expects team A to be 5 times more likely to win than team B. If I bet 2 euros on team B and team B wins, then I will win $5 \times 2 = 10$ euros (I bet on a rare event). The odd in this case would be defined as:

$$odd = \frac{5}{1} = 5$$

Alternatively, I can reason in terms of probability. Given the scheme above, I expect that, if we played the match 100 times, then the probability of victories for team A would be equal to $P(A\ wins) = \frac{5}{5+1}$, while the probability of loss would be equal to $P(A\ loses) = \frac{1}{5+1}$. The odd can be defined in terms of these probabilities:

$$odd = \frac{P(A\ wins)}{P(A\ loses)} = \frac{\frac{5}{5+1}}{\frac{1}{5+1}} = 5$$

In general, the odd for an event E is the probability that the event occurs, divided by the probability that the event does not occur:

$$odd = \frac{P(E)}{1 - P(E)}$$

The odd is a concept similar to risk because it tells us how much greater the frequency of an event occurring (A wins) is compared to the frequency of the event not occurring. Let's apply this concept to our example. We define two odds (just as we defined two risks):

- The odds of getting sick if exposed: $\frac{P(Diseased|Exposed)}{P(Healthy|Exposed)} = \frac{\frac{20}{20+10}}{\frac{10}{20+10}} = \frac{20}{10} = 2$
- The odds of getting sick if not exposed: $\frac{P(Diseased|Non\ Exposed)}{P(Healthy|Non\ Exposed)} = \frac{\frac{6}{6+16}}{\frac{16}{6+16}} = \frac{6}{16} = 0.375$

It should be noted that in the formulas above, the absolute counts, i.e., those we do not trust given the sampling, "cancel out".

The odds ratio will be defined as follows:

$$odd = \frac{\frac{P(Diseased|Exposed)}{P(Healthy|Exposed)}}{\frac{P(Diseased|Non\ Exposed)}{P(Healthy|Non\ Exposed)}} = \frac{2}{0.375} \approx 5.3$$

We note that the obtained odds ratio is very close to the previously calculated relative risk of 5. **In practice, it can be demonstrated that, if it is assumed that the disease is rare (rare-disease assumption), the odds ratio and the relative risk converge to similar values.** It is therefore a fairly common practice to make this assumption and calculate the odds ratio instead of the relative risk.

Relative Risk and Odds Ratio in Python

Let's consider this contingency table:

```
import pandas as pd
contingency = pd.DataFrame({
    "Exposed": [20, 380],
    "Non Exposed": [6,594]
}, index=["Diseased", "Healthy"])
contingency
```

	Exposed	Non Exposed
Diseased	20	6
Healthy	380	594

We can calculate the relative risk as follows:

```
from scipy.stats.contingency import relative_risk
relative_risk(contingency['Exposed']
        ['Diseased'],contingency['Exposed'].sum(),contingency['Non Exposed']
        ['Diseased'], contingency['Non Exposed'].sum())
```

```
RelativeRiskResult(relative_risk=5.0, exposed_cases=20, exposed_total=400,
control_cases=6, control_total=600)
```

The function requires that the various necessary values be passed rather than the contingency matrix to avoid ambiguity.

The odds ratio is calculated as follows:

```
from scipy.stats.contingency import odds_ratio
odds_ratio(contingency)
```

```
OddsRatioResult(statistic=5.202179487843029)
```

Exercises

Exercise 1

Considering the Titanic dataset, calculate for each variable the count, mean, standard deviation, minimum, maximum, median value, first and third quartile. After calculating the requested values individually, use the `describe` method to obtain these values. Which variable is the most dispersed?

Exercise 2

Considering the Titanic dataset, for each of the three classes, calculate the means and variances of passenger ages. In which class are the ages least dispersed? Which class contains the youngest individuals? Complete the analysis with bar charts.

Exercise 3

Add a new column “Old” to the Titanic dataset, which takes a value of 1 for observations with an age greater than the average value and 0 otherwise. Then, calculate a crosstab that computes the number of “elderly” subjects (subjects for whom old is equal to 1) with respect to the Pclass and Sex variables. Display a barplot based on the table. In which class do men tend to be much older than women?

Exercise 4

Load the mpg dataset using the seaborn library (use the instruction `t=sns.load_dataset('mpg')`). Visualize the scatter matrix by differentiating the colors of the points according to the values of the `origin` variable. Who produces the heaviest cars? Are there pairs of variables that are a decisive factor in distinguishing one origin from the others?

Exercise 5

Considering the `mpg` dataset, choose a group of variables for which to obtain a parallel coordinates plot. What can we infer from the plot?

References

- Chapter 3 of: Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.
- https://en.wikipedia.org/wiki/Odds_ratio

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Data distributions

Chapter resources

Slides (PDF) Open in Colab Practice quiz

We have seen how it is possible to assign a probability value to a given outcome of a random variable.

In practice, it is often useful to assign probability values to **all the values** that the random variable can assume.

To do so, we can define a **function**, which we will call **probability distribution** which assigns a probability value to each of the possible values of a random variable.

In the case of discrete variables, we will talk about "**probability mass functions**", whereas in the case of continuous variables, we will refer to "**probability density functions**".

A probability distribution characterizes the random variable and defines which outcomes it is more likely to observe.

Once we find that a given random variable X is characterized by a probability distribution $P(X)$, we can say that **"X follows P"** and write:

$$X \sim P$$

Discrete Variables

We will start by focusing on discrete variables.

Probability Mass Functions (PMF)

If X is discrete, $P(X)$ is called a "probability mass function" (PMF).

P can be seen as a function mapping the values of X to real numbers indicating whether a given value is more or less likely.

A PMF on a random variable X is a function:

$$P : \Omega \rightarrow [0, 1]$$

which satisfies the following property:

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

where Ω is the sample space X,

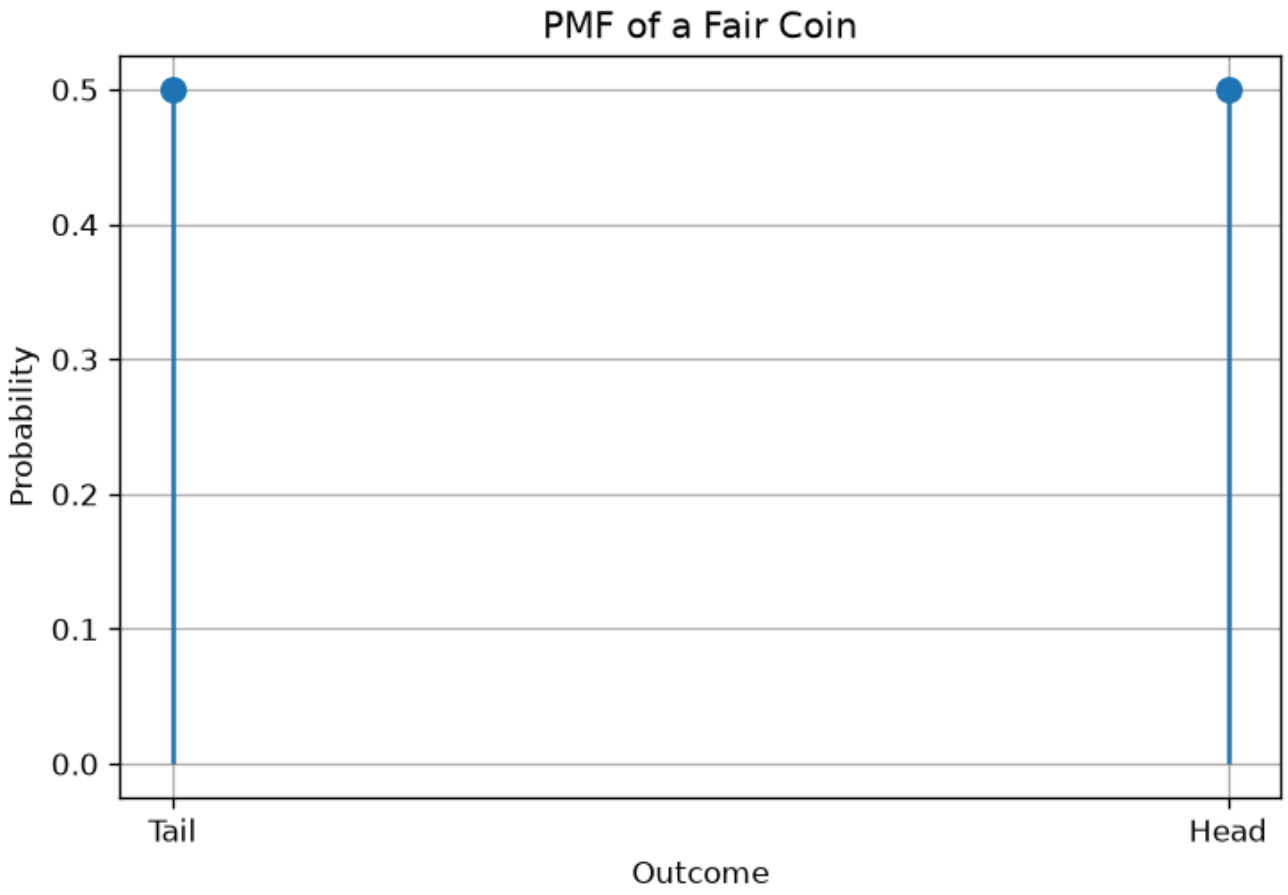
This condition implies that the probability distribution is normalized (and probability axioms are satisfied).

Example: Probability Mass Function for a Fair Coin

Let X be the random variable indicating the outcome of a coin toss.

- The space of all possible outcomes (the domain of $P(X)$) is $\{head,\ tail\}$.
- The probabilities $P(head)$ and $P(tail)$ must be l**arger than or equal to zero and smaller than or equal to 1**.
- Also, $P(head) + P(tail) = 1$. This is obvious, as one of the two outcomes will always happen. Indeed, if we had $P(tail) = 0.3$, this would mean that, 30 times out of 100 times we toss a coin, the outcome will be tail. What will happen in all other cases? The outcome will be head, hence, $P(head) = 0.7$, so $P(head) + P(tail) = 1$.
- In the case of a fair coin, we can characterize $P(X)$ as a **"discrete uniform distribution"**, i.e., a distribution which maps any value $x \in X$ to a constant, such that the properties of the probability mass functions are satisfied.
- If we have N possible outcomes, the discrete uniform probability will be $P(X = x) = \frac{1}{N}$, which means that all outcomes have the same probability.
- This definition satisfies the constraints. Indeed:
  - $\frac{1}{N} \geq 0,\ \forall N$
  - $\sum_i P\left(X = x_i\right) = 1$

A probability mass function can be visualized as a 2D diagram where the values of the function ($P(x)$) is plotted against the values of the independent variable x. This is the diagram associated to the PMF of the previous example, where $P(head) = P(tail) = 0.5$.



Example: Probability Mass Function for a Biased Coin

Now suppose we tossed our coin for 10000 times and discovered that 6000 times the outcome was "head", whereas 4000 times it was "tail". We deduce the coin is not fair.

Using a **frequentist** approach, we can manually assign values to our PMF using the general formula:

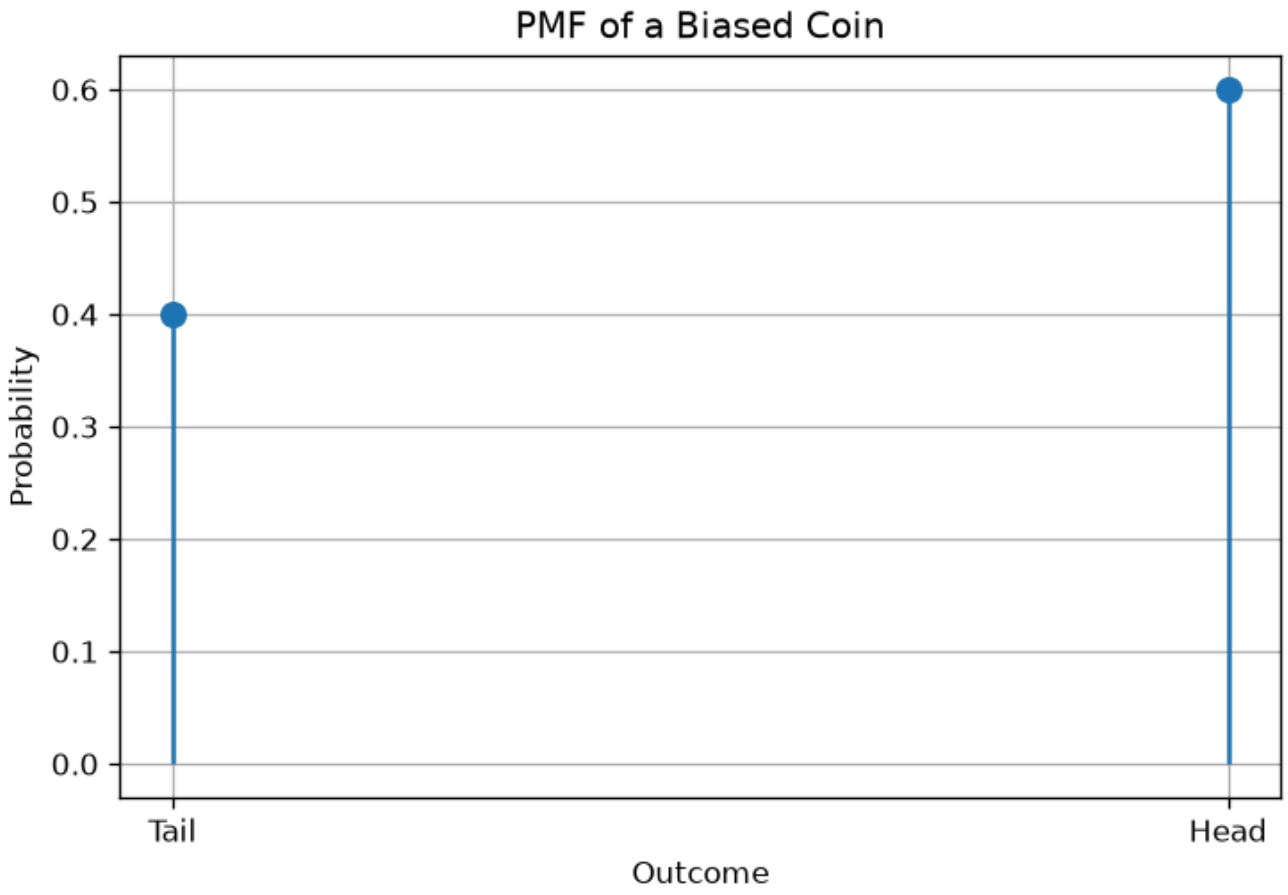
$$P(x) = \frac{\#trials\ in\ which\ X = x}{\#\ trials}$$

That is, in our case:

$$P(head) = \frac{6000}{10000} = 0.6; P(tail) = \frac{4000}{10000} = 0.4$$

We shall note that the probability we just defined satisfies all properties of probabilities, i.e.:

- $0 \leq P(x) \leq 1\ \forall x$
- $\sum_x P(x) = 1.$



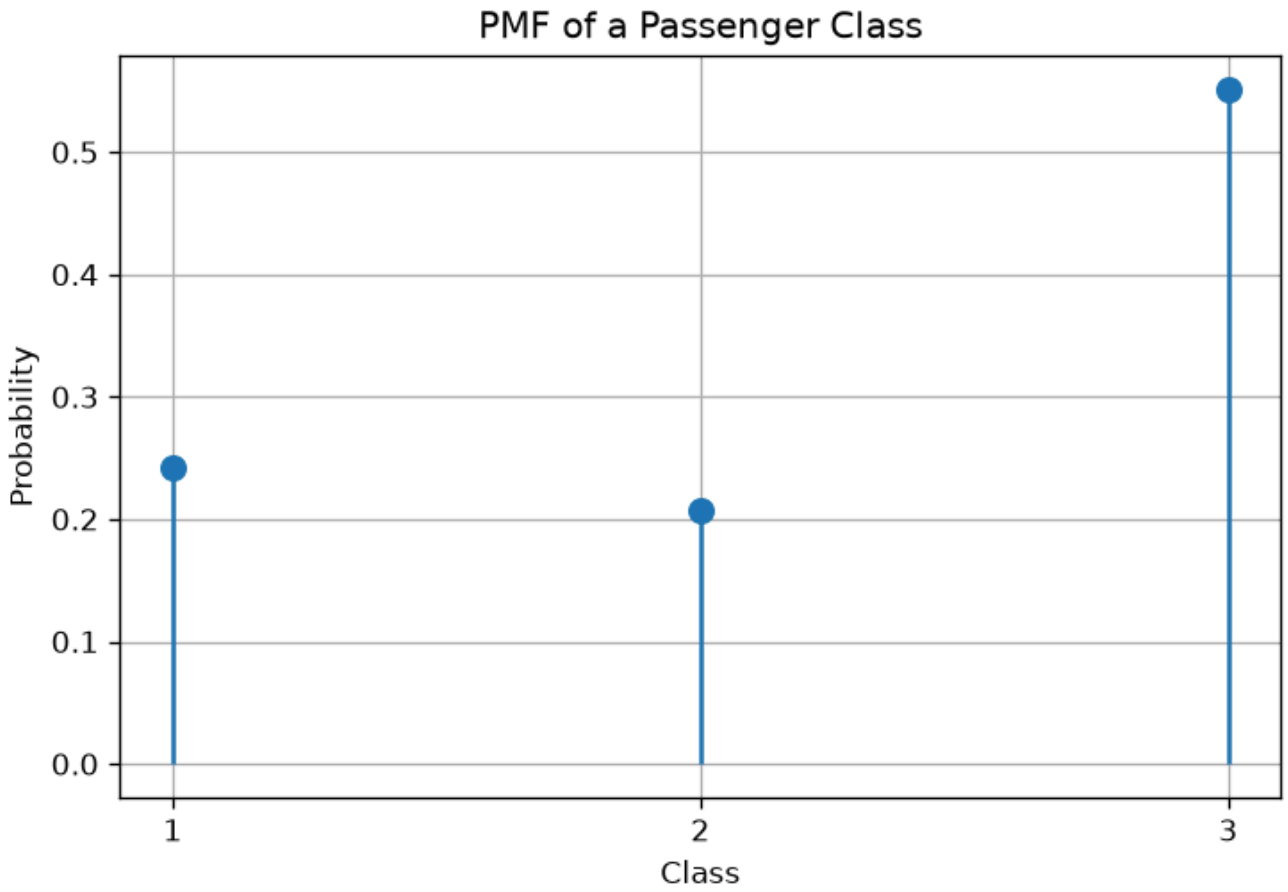
Computing Probability Mass Functions from Data

In practice, Probability Mass Functions are computed exactly as relative frequencies. Let's see a quick example:

```
import pandas as pd
titanic = pd.read_csv(DATA + "titanic.csv",
                      index_col='PassengerId')
pclass = titanic['Pclass'].value_counts(normalize=True).sort_index()

plt.vlines(pclass.index.values, ymin=0, ymax=pclass.values,  linestyles='solid')
plt.plot(pclass.index.values, pclass.values, 'o', markersize=8)
plt.xlabel('Class')
plt.ylabel('Probability')
plt.title('PMF of a Passenger Class')
plt.xticks(pclass.index.values)
plt.grid(True)

# Show the plot
plt.show()
```



Exercise: Probability Mass Function

Let X be a random variable representing the outcome of rolling a fair dice with 6 faces:

- What is the space of possible values of X?
- What is its cardinality?
- What is the associated probability mass function $P(X)$?
- Suppose the dice is not fair and $P(X = 1) = 0.2$, whereas all other outcomes are equally probable. What is the probability mass function of $P(X)$?
- Draw the PMF obtained for the dice.

Cumulative Distribution Functions of Discrete Variables

If the discrete variable X can be ordered (i.e., it is at least ordinal), we can define the **Cumulative Distribution Function (CDF)** as:

$$F(x) = P(X \leq x) = \sum_{y \leq x} P(y)$$

The CDF records the cumulative probability of all values less than or equal to a given value. For example, if H measures the height of people in centimeters, $F(180)$ indicates the **probability of**

selecting a person with height at most 180 cm.

Example – PMF and CDF of a fair die

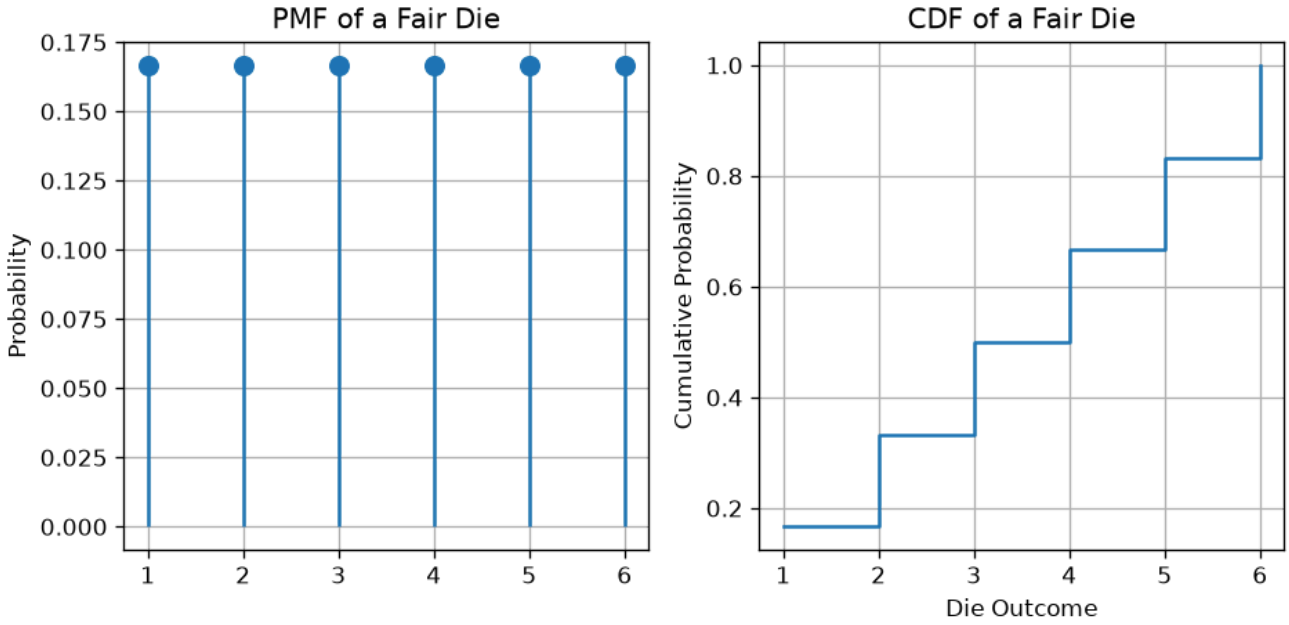
For a fair die, the PMF is:

$$f(x) = \begin{cases} \frac{1}{6} & \text{if } x \in \{1,2,3,4,5,6\} \\ 0 & \text{otherwise} \end{cases}$$

The CDF is:

$$F(x) = \begin{cases} 0 & \text{if } x < 1 \\ \frac{1}{6} & \text{if } 1 \leq x < 2 \\ \frac{2}{6} & \text{if } 2 \leq x < 3 \\ \frac{3}{6} & \text{if } 3 \leq x < 4 \\ \frac{4}{6} & \text{if } 4 \leq x < 5 \\ \frac{5}{6} & \text{if } 5 \leq x < 6 \\ 1 & \text{if } x \geq 6 \end{cases}$$

The diagram below shows an example:



Computing CDFs for Discrete Variables from Data

Empirical Cumulative Distribution Functions (ECDFs), introduced in the lecture on data description, allow us to estimate the CDF directly from observed data. For a discrete variable, the ECDF at a value x is simply the proportion of data points less than or equal to x.

Here is a quick example:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

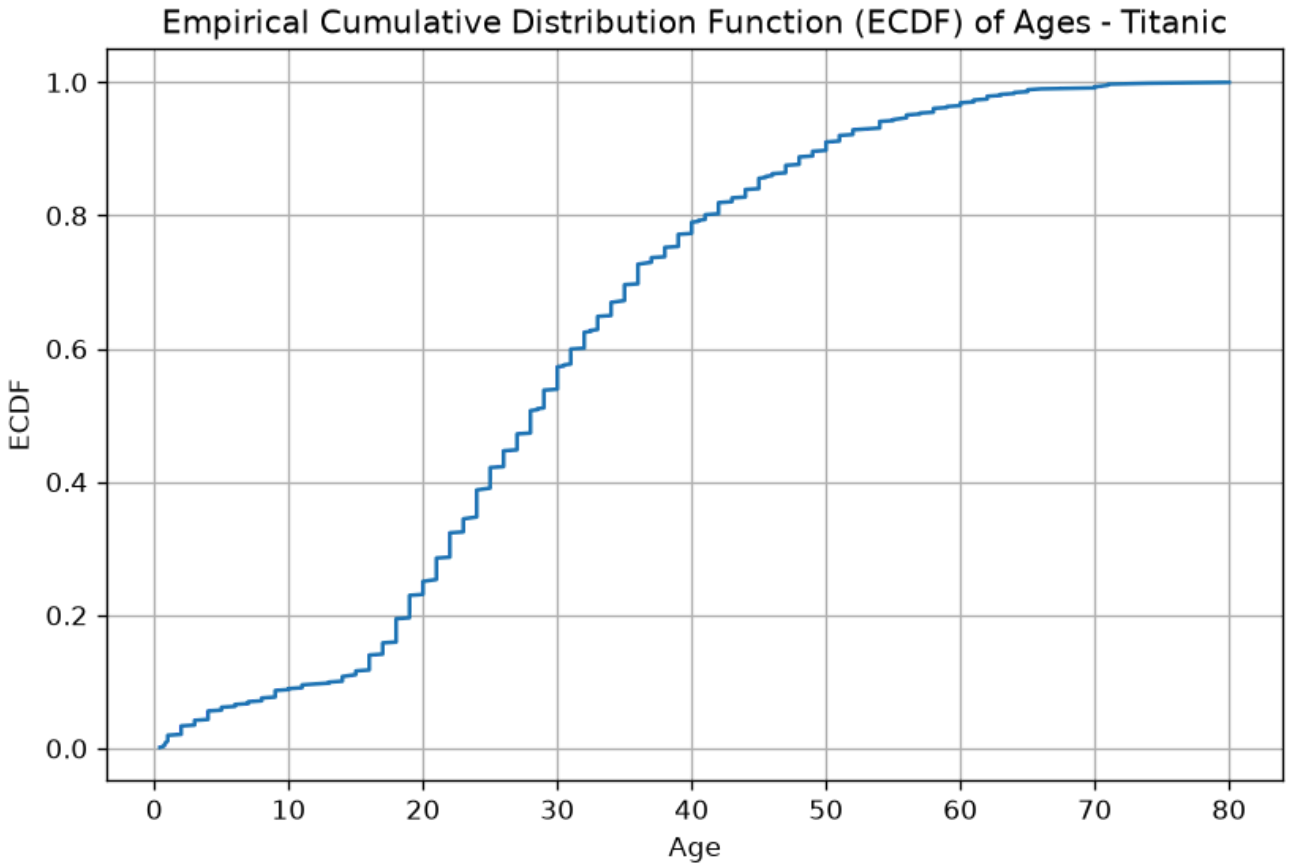
# Load the Titanic dataset
titanic = pd.read_csv(DATA + "titanic.csv", index_col='PassengerId')

# Extract the age column, removing missing values
ages = titanic['Age'].dropna()

# Sort the ages
ages_sorted = np.sort(ages)

# Compute the ECDF
ecdf = np.arange(1, len(ages_sorted)+1) / len(ages_sorted)

# Plot the ECDF
plt.figure(figsize=(8, 5))
plt.plot(ages_sorted, ecdf)
plt.xlabel('Age')
plt.ylabel('ECDF')
plt.title('Empirical Cumulative Distribution Function (ECDF) of Ages - Titanic')
plt.grid(True)
plt.show()
```



Continuous Variables

We will now focus on the case in which X is a continuous variable. In this case, we cannot use the same definitions as for discrete variables and we have to resort to new definition.

Probability Density Functions (PDF)

In order to define a probability distribution over a continuous variable, we first have to introduce the concept of "probability density function".

A probability density function over a variable X is defined as follows:

$$f : \Omega \rightarrow [0, 1]$$

and must satisfy the following property:

$$\int f(x)\, dx = 1$$

This condition is equivalent to $\sum P(x) = 1$ in the case of a discrete variable. The sum becomes an integral for continuous variables.

We can then define the concept of probability based on the density as follows:

$$P(a \leq x \leq b) = \int_a^b f(x)\, dx$$

Intuitively, the density function $f(x)$ describes how likely it is to observe values near x. The higher the density at a point, the more likely it is to find the variable in a small interval around that point. In other words, the density can be seen as a "concentration" of probability: regions where $f(x)$ is high correspond to intervals where the variable is more frequently observed. To obtain an actual probability, we must integrate the density over an interval.

> For a continuous random variable, the probability of observing exactly x is zero: $P(X = x) = 0$. However, the density function $f(x)$ at x is generally not zero; it represents the probability per unit interval around x. Probability is obtained by integrating $f(x)$ over an interval, not at a single point.

Example: Uniform PDF

Let us consider a random number generator which outputs numbers comprised between a and b.

Let X be a random variable assuming the values generated by the random number generator.

The PDF of X will be a uniform distribution such that:

- $P(x) = 0 \; \forall x < a \;\; or \; x > b$;

- $P(x) = \frac{1}{b-a} \; \forall a \leq x \leq b$;

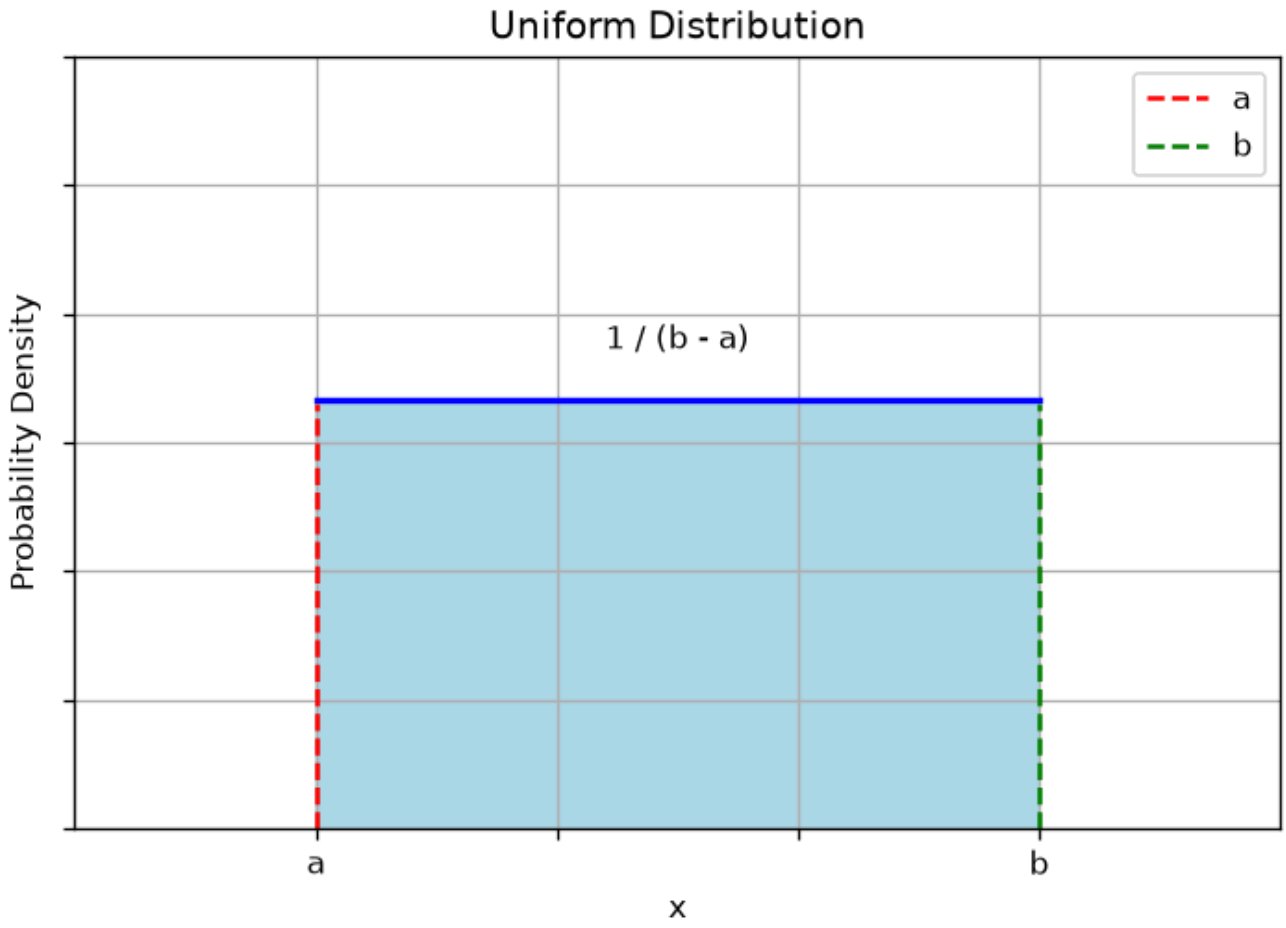
We can see that this PDF satisfies all constraints:

- $P(x) \geq 0 \; \forall x.$
- $\int P(x)dx = 1$ (prove that this is true as an exercise).

The diagram below shows an illustration of a uniform PDF with bounds a and b:

$$U(a, b)$$

Of course, continuous distributions can be (and generally are) much more complicated than that.



Approximating a PDF with Density Histograms and Density Estimates

When working with real data, we can try to approximate the PDF of a continuous variable with histograms.

Let the height and width of bin b_j be h_j and w_j respectively. In a **normalized histogram**, we define the height of bin b_j as follows:

$$h_j = \frac{c_j}{n}$$

where c_j is the number of observations that fall in bin b_j.

Note that, while **normalized histograms** are such that each bar height is comprised between 0 and 1, they do not satisfy the properties of PDFs.

Let the area of bin b_j be:

$$A_j = w_j \cdot h_j$$

The integral over histogram H (i.e., the area under the histogram) will be:

$$\int_{-\infty}^{\infty} H(x)dx = \sum_j A_j = \sum_j w_j \frac{c_j}{n} = \frac{1}{n} \sum_j w_j \cdot c_j = \sum w_j \neq 1$$

To ensure that this property is satisfied, we define a new kind of histogram, the **density histogram**, where bin heights are defined as:

$$h_j = \frac{c_j}{w_j \cdot n}$$

In this case:

$$\sum_j A_j = \sum_j w_j \frac{c_j}{w_j \cdot n} = \frac{1}{n} \sum_j \frac{w_j}{w_j} \cdot c_j = 1$$

To create a density histogram in Python, simply pass `density=True` to the `hist` function in matplotlib or pandas.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Load the Titanic dataset and extract the 'Age' column, dropping missing values
titanic = pd.read_csv(DATA + "titanic.csv")
ages = titanic['Age'].dropna()

# Create a figure with two subplots for side-by-side comparison
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# --- Plot 1: Normalized Histogram (Probability) ---
# The sum of bar heights will equal 1
weights = np.ones_like(ages) / len(ages)
ax1.hist(ages, bins=20, weights=weights, color='skyblue', edgecolor='black')
ax1.set_title('Normalized Histogram (Probability)')
ax1.set_xlabel('Age')
ax1.set_ylabel('Probability (Sum of heights = 1)')
ax1.grid(True, alpha=0.6)

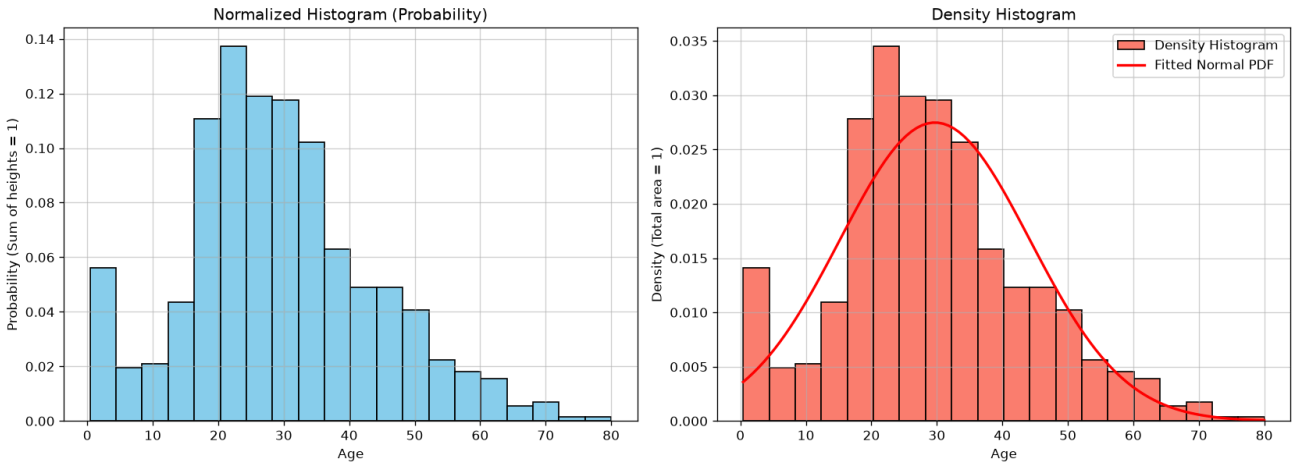
# --- Plot 2: Density Histogram ---
# The total area of the bars will equal 1
ax2.hist(ages, bins=20, density=True, color='salmon', edgecolor='black',
         label='Density Histogram')

# Fit a normal distribution and overlay its PDF
mu_hat, sigma_hat = norm.fit(ages)
x = np.linspace(ages.min(), ages.max(), 100)
pdf_fitted = norm.pdf(x, loc=mu_hat, scale=sigma_hat)
ax2.plot(x, pdf_fitted, 'r-', lw=2, label='Fitted Normal PDF')

ax2.set_title('Density Histogram')
ax2.set_xlabel('Age')
ax2.set_ylabel('Density (Total area = 1)')
ax2.grid(True, alpha=0.6)
ax2.legend()

# Add an overall title and show the plot
plt.suptitle('Normalized vs. Density Histograms for Passenger Ages', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust layout to make space for suptitle
plt.show()
```

Normalized vs. Density Histograms for Passenger Ages



The figure above contrasts two important ways to normalize a histogram using Titanic passenger ages, each with a distinct but related interpretation:

- **Normalized Histogram (Left)**:
  The y-axis represents **probability**. Each bar's height shows the proportion of passengers in that age bin, so the **sum of all bar heights equals 1**. This treats the binned data like a discrete probability distribution and allows direct reading of probabilities (e.g., the chance of being 20–24 years old is roughly 0.12 or 12%).

- **Density Histogram (Right)**:
  The y-axis represents **density**—probability per unit on the x-axis (e.g., per year of age). Here, it's the **area** of each bar (height × width) that reflects probability. The **total area of all bars equals 1**, mirroring the behavior of a continuous Probability Density Function (PDF).

Key takeaway:
The density histogram is the correct empirical approximation of a continuous PDF. That's why the smooth red PDF curve aligns with the right plot's y-axis but would be mis-scaled on the left. While both histograms are normalized, only the density version properly reflects continuous probability.

Cumulative Distribution Functions (CDF)

Just like in the case of discrete variables, we can define Cumulative Distribution Functions (CDFs) for continuous random variables using their density functions. The CDF of a random variable is generally defined as:

$$F(x) = \int_{-\infty}^{x} f(x)\, dx$$

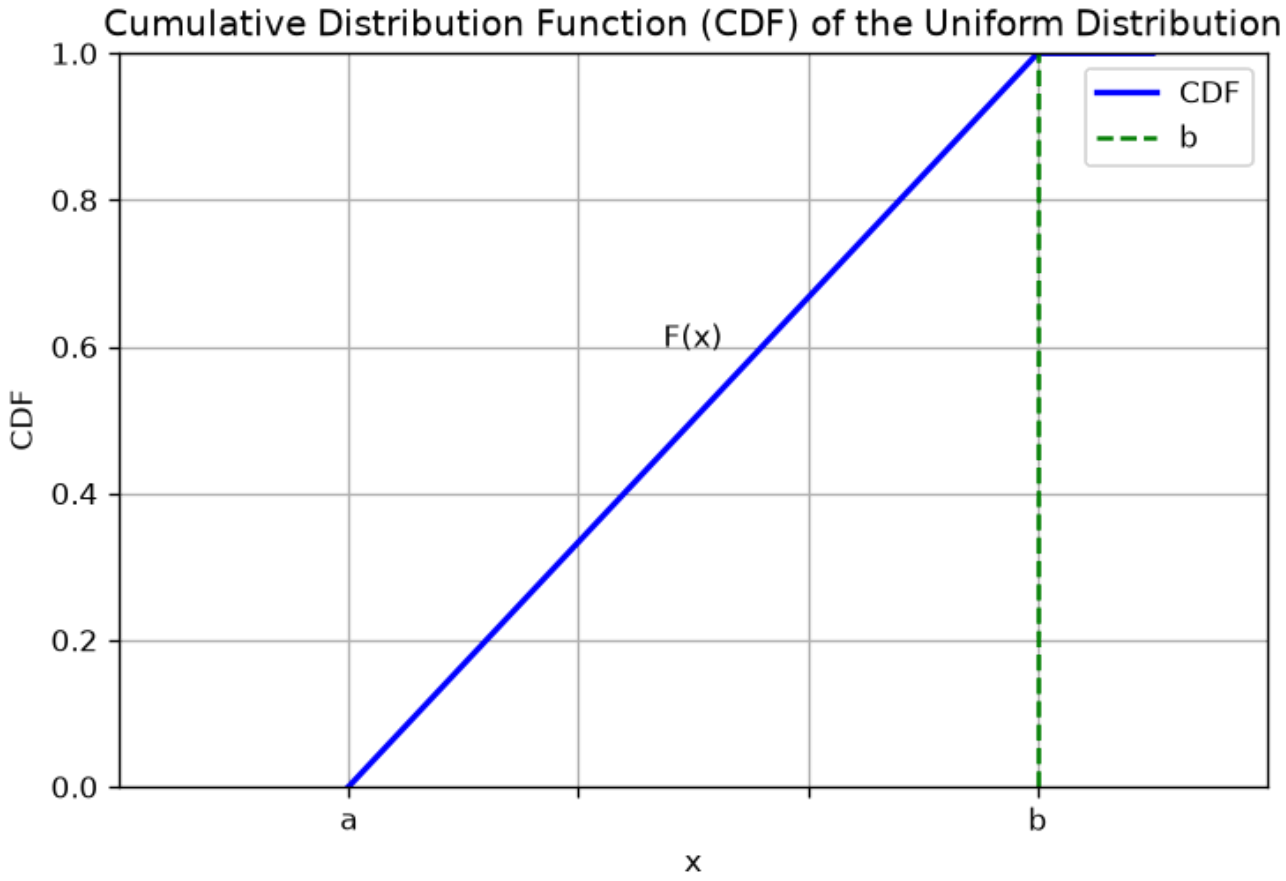
Example

The CDF of the uniform distribution will be given by:

- $F(x) = 0$ for $x < a$

- $F(x) = \frac{x-a}{b-a}$ for $a \leq x \leq b$
- $F(x) = 1$ for $x > b$

The plot below shows a diagram:



Common Probability Distributions

There are several common probability distributions which can be used to describe random events. **These distributions have an analytical formulation which depends generally on one or more parameters.**

When we have **enough evidence that a given random variable is well described by one of these distributions**, we can simply "fit" the distribution to the data (i.e., choose the correct parameters for the distribution) and use the analytical formulation to deal with the random variable.

It is hence useful to know the **most common probability distributions** so that we can recognize the cases in which they can be used.

Discrete Uniform Distribution

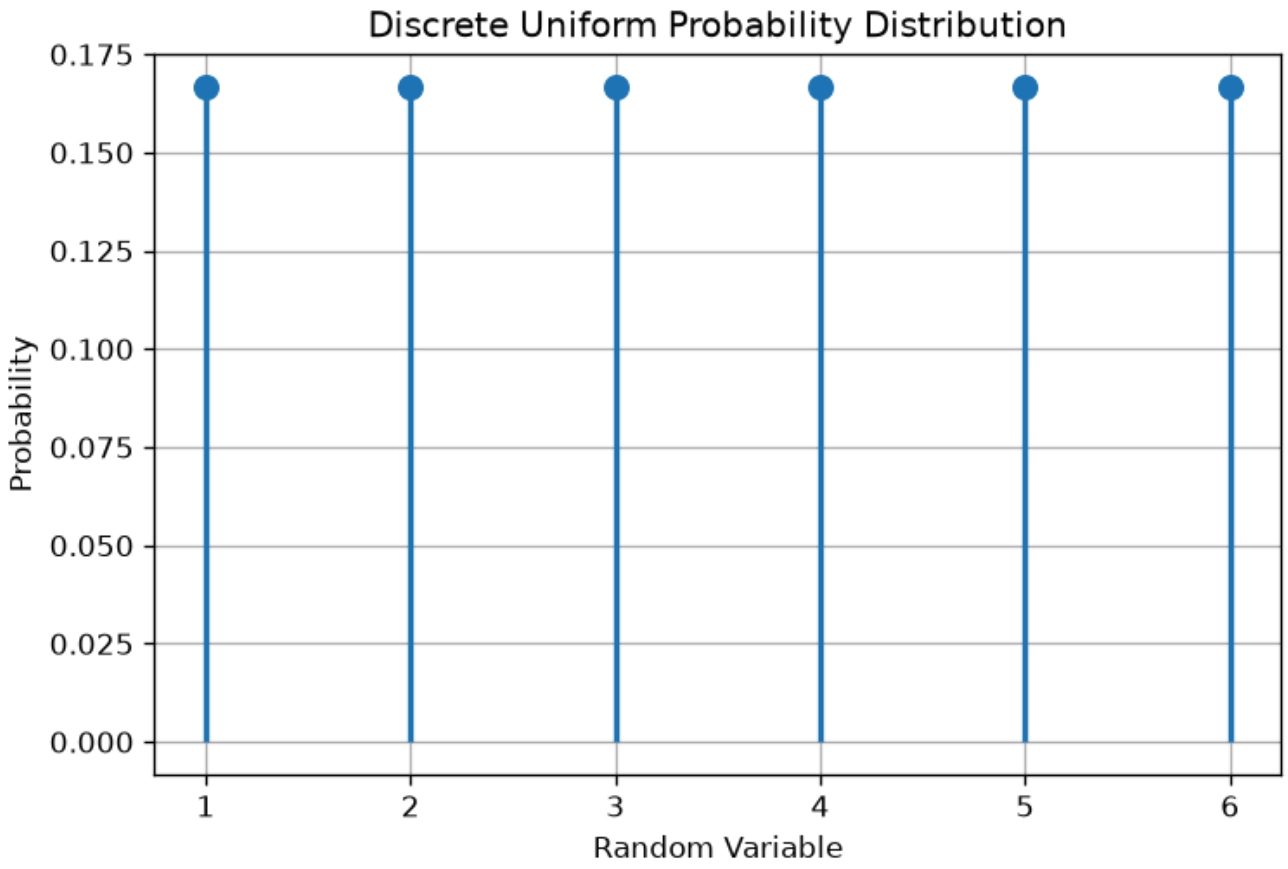
The discrete uniform distribution is controlled by a parameter $k \in \mathbb{N}$ and assumes that all outcomes have the same probability of occurring:

$$P(X = a_i) = \frac{1}{k}$$

Where $\Omega = \{a_1, \ldots, a_k\}$.

Example

The outcomes of rolling a fair die follow a uniform distribution with $k = 6$, as shown in the diagram below:



Bernoulli Distribution

The Bernoulli distribution is a distribution over a single binary random variable, i.e., the variable X can take only two values: $\{0, 1\}$.

The distribution is controlled by a single parameter $\phi \in [0, 1]$, which gives the probability of the variable to be equal to 1.

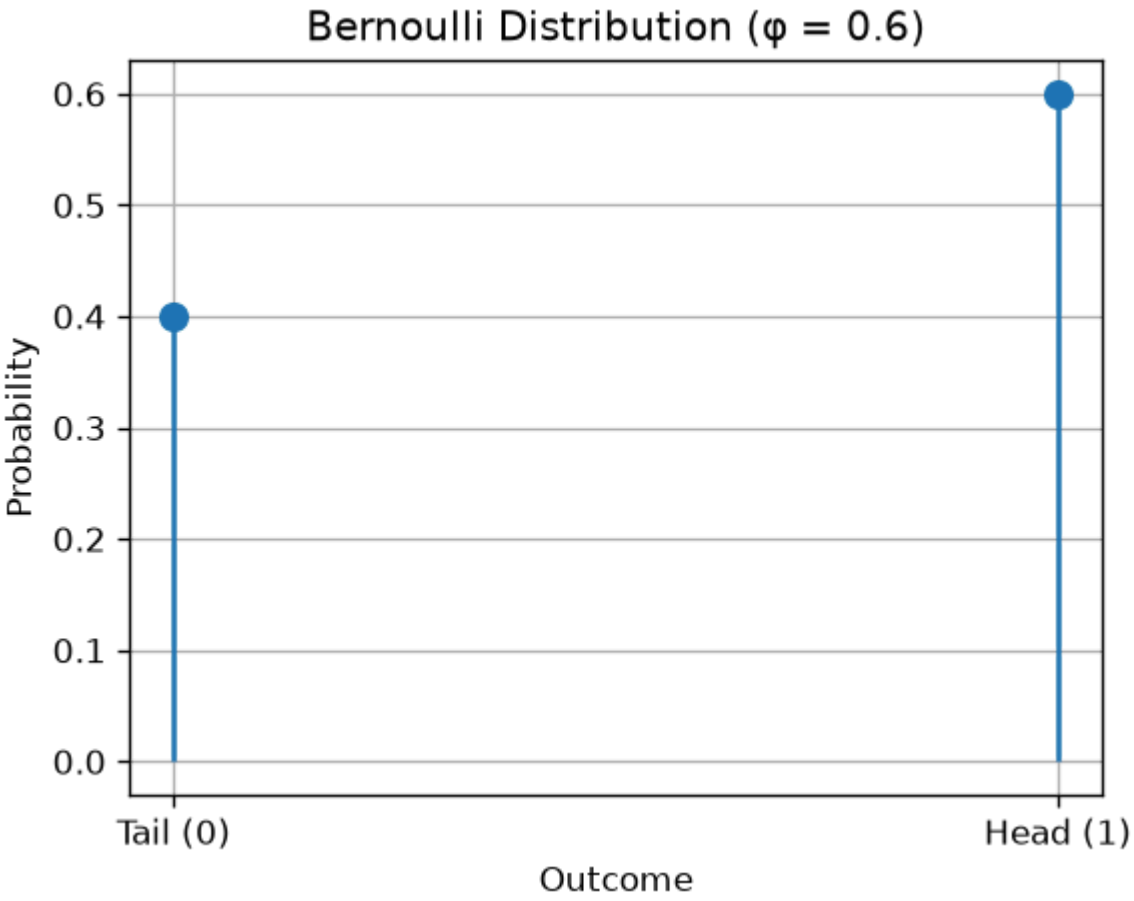
The analytical formulation of the Bernoulli distribution is very simple:

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

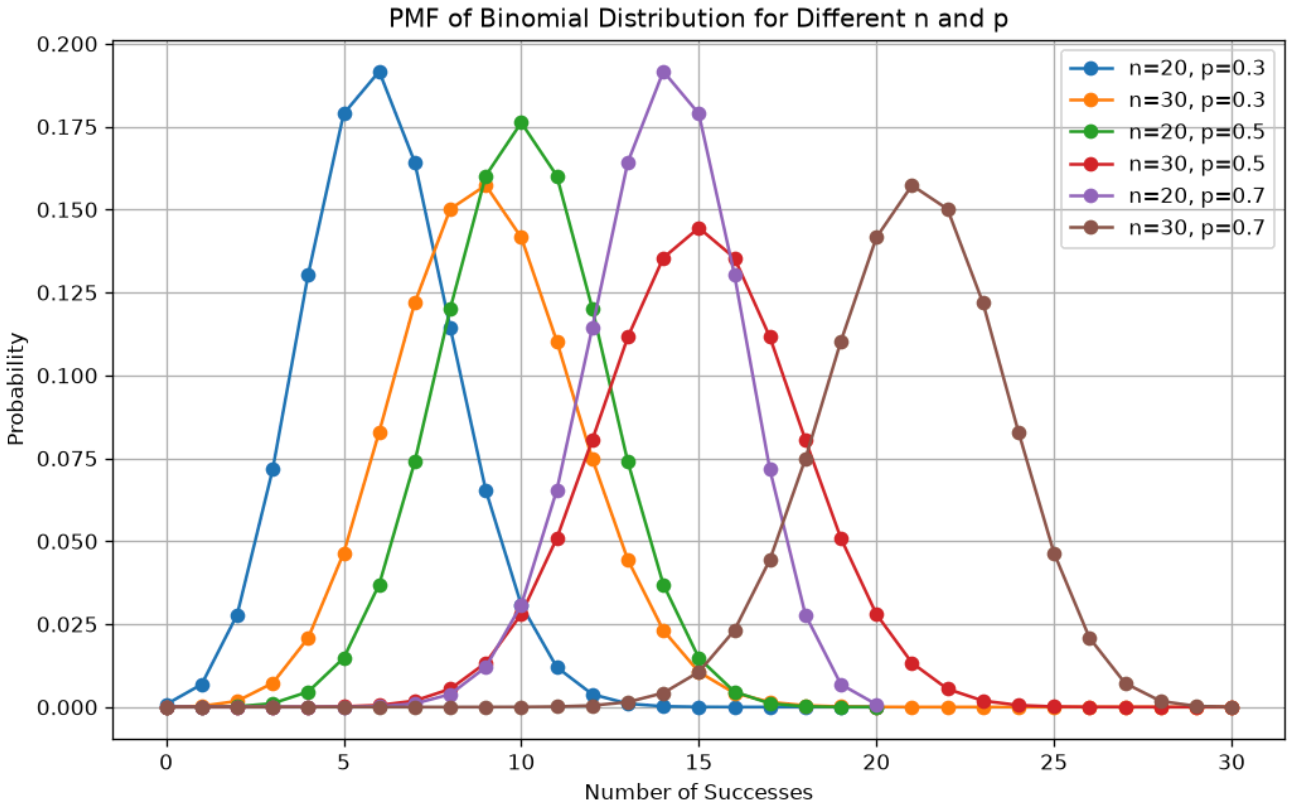
Example

A skewed coin lands on "head" 60% of the times. If we define $X = 1$ when the outcome is head and $X = 0$ when the outcome is tail, then the variable follows a Bernoulli distribution with $\phi = 0.6$.

The diagram below gives an example:



Binomial Distribution



The binomial distribution is a discrete probability distribution (PMF) over natural numbers with parameters $\mathbf{n}$ and $\mathbf{p}$

It models **the probability of obtaining $\mathbf{k}$ successes in a sequence of $\mathbf{n}$ independent experiments which follow a Bernoulli distribution with parameter $\mathbf{p}$ ($\phi = \mathbf{p}$);**

The probability mass function of the distribution is given by:

$$P(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Where:

- k is the number of successes
- n is the number of independent trials
- p is the probability of a success in a single trial

Example

What is the probability of tossing a coin three times and obtaining three heads? We have:

- $k = 3$: number of successes (three times head)
- $n = 3$: number of trials

- $p = 0.5$: the probability of getting a head when tossing a coin

The required probability will be given by:

$$P(3) = \binom{3}{3} 0.5^3 (1 - 0.5)^{3-3} = 0.5^3 = 0.125$$

Exercise

What is the probability of tossing an unfair coin ($P\left('head'\right) = 0.6$) 7 times and obtaining 2 tails?

Categorical Distribution

The **multinoulli** or **categorical** distribution is a distribution of a *single discrete variable with k different states*, where k is finite.

- The distribution is parametrized by a vector $\mathbf{p} \in [0, 1]^k$, where p_i gives the probability of the ith state.
- $\mathbf{p}$ must be such that $\sum_{i=1}^{k} p_i = 1$ to obtain a valid probability distribution.
- The analytical form of the distribution is given by: $p(x = i) = p_i$;

This distribution is the **generalization of the Bernoulli distribution to the case of multiple states**.

Example: Rolling a Biased Die

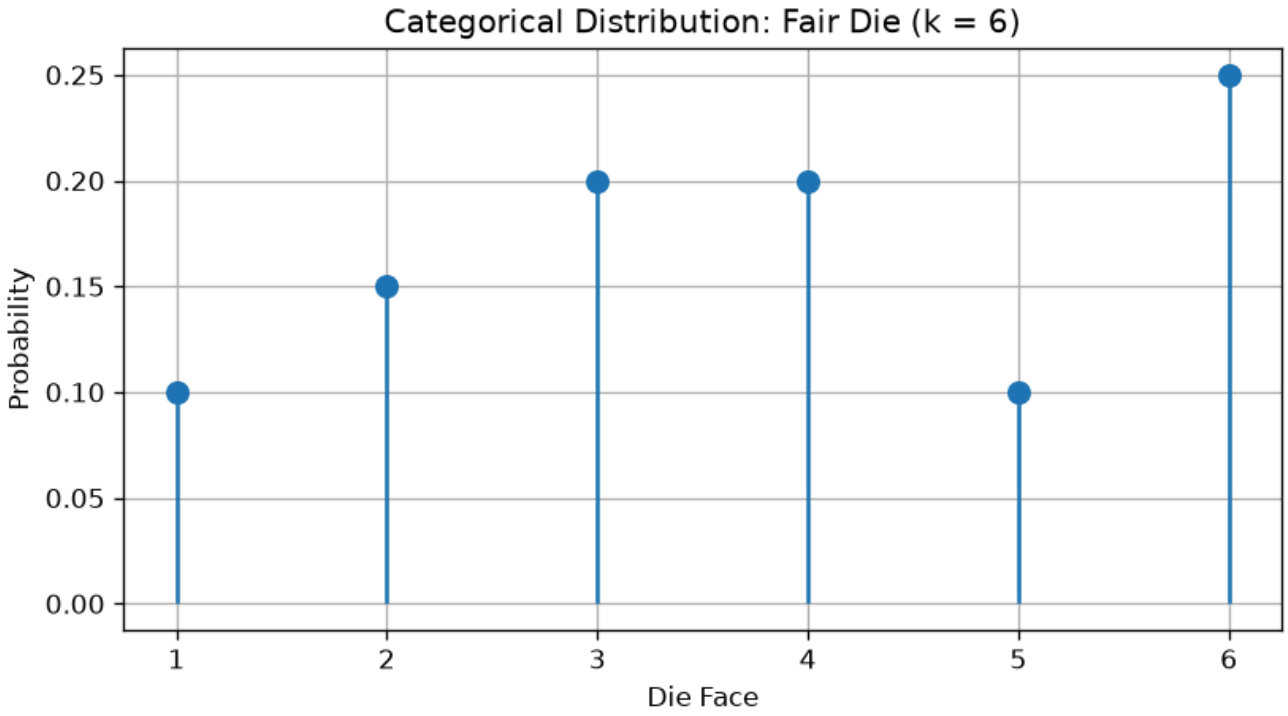
Consider a single roll of a biased six-sided die. The outcomes are discrete and finite:
$X \in \{1, 2, 3, 4, 5, 6\}$.

If the die is biased such that the probabilities are:

- $P(X = 1) = 0.10$
- $P(X = 2) = 0.15$
- $P(X = 3) = 0.20$
- $P(X = 4) = 0.20$
- $P(X = 5) = 0.10$
- $P(X = 6) = 0.25$

Then the outcome follows a **Categorical distribution** with $k = 6$ and the specified probabilities $\{p_1, p_2, \ldots, p_6\} = \{0.1, 0.15, 0.2, 0.2, 0.1, 0.25\}$.

The figure below shows an example:



Multinomial Distribution

The multinomial distribution **generalizes the binomial distribution to the case in which the experiments are not binary**, but they can have multiple outcomes (e.g., *a dice vs a coin*).

In particular, the multinomial distribution models the probability of obtaining exactly $(n_1, \ldots, n_k)$ occurrences (with $n = \sum_i n_i$) for each of the k possible outcomes in a sequence of n independent experiments which follow a Categorial distribution with probabilities $p_1, \ldots, p_k$.

The parameters of the distribution are:

- n: the number of trials
- k: the number of possible outcomes
- $p_1, \ldots, p_k$ the probabilities of obtaining a given class in each trial (with $\sum_{i=1}^{k} p_i = 1$)

The PMF of the distribution is:

$$P(n_1, \ldots, n_k) = \frac{n!}{n_1! \ldots n_k!} p_1^{n_1} \cdot \ldots \cdot p_k^{n_k}$$

Example

Given a fair die with 6 possible outcomes, what is the probability of getting 3 times 1, 2 times 2, 4 time 3, 5 times 4, 0 times 5, and 1 time 6, rolling the dice for 15 times?

We have:

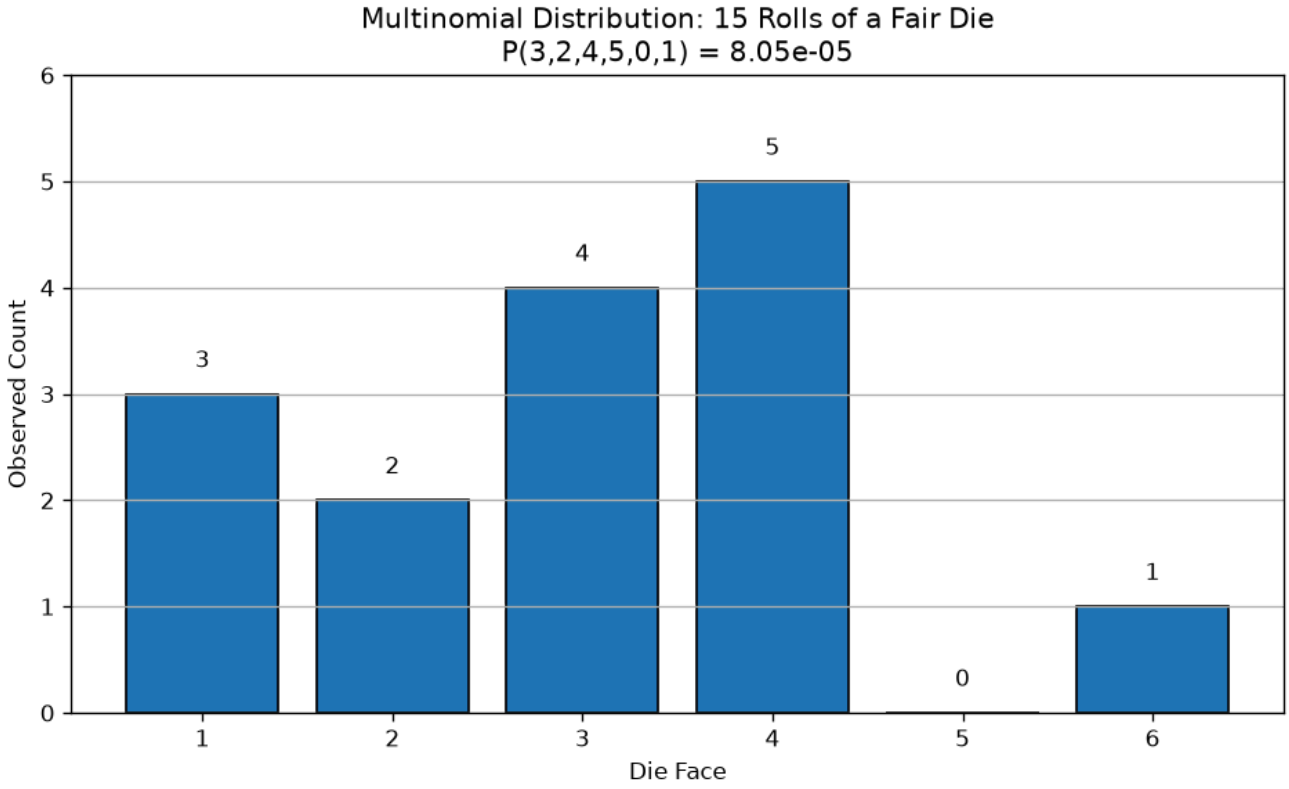
- $n = 15$
- $k = 6$
- $p_1 = p_2 = \ldots p_6 = \frac{1}{6}$

The required probability is given by:

$$P(3,2,4,5,0,1) = \frac{15!}{3!2!4!5!0!1!} \cdot \frac{1}{6^3} \cdot \frac{1}{6^2} \cdot \frac{1}{6^4} \cdot \frac{1}{6^5} \cdot \frac{1}{6^0} \cdot \frac{1}{6^1} = 8.04 \cdot 10^{-5}$$

The bar chart below illustrate the example, showing the observed counts and the estimated probability.

This low probability reflects the combinatorial nature of the multinomial distribution, which accounts for both the number of ways the outcomes can be arranged and the likelihood of each individual outcome.



Gaussian Distribution

The Bernoulli and Categorical distributions are PMF, i.e., distributions over discrete random variables.

A common PDF when dealing with real values is the **Gaussian distribution,** also known as **Normal Distribution**.

The distribution is characterized by two parameters:

- The mean $\mu \in \Re$

- The standard deviation $\sigma \in (0, +\infty)$

In practice, the distribution is often *seen in terms of* μ *and* σ^2 rather than σ, where σ^2 is called **the variance**.

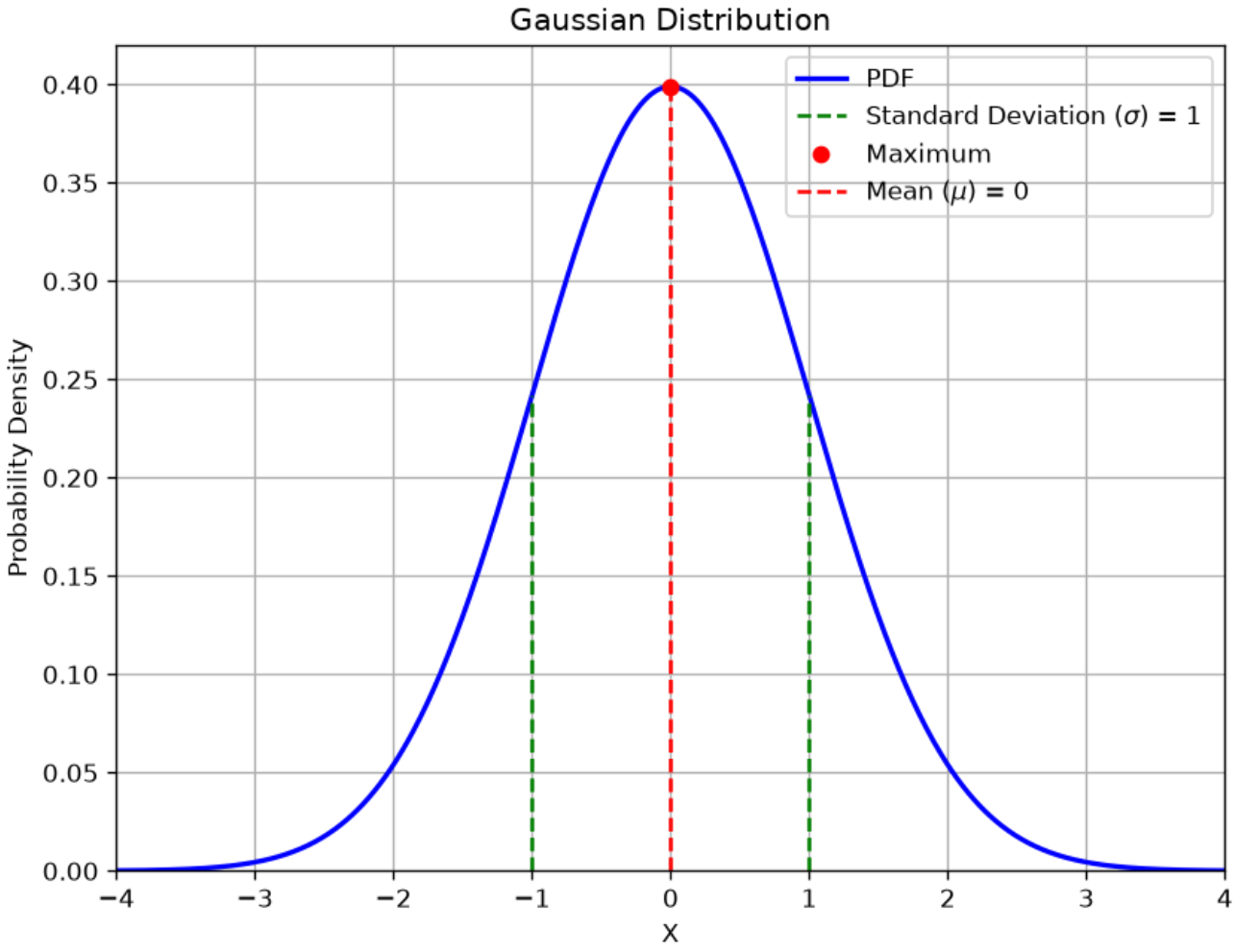
The analytical formulation of the Normal distribution is as follows:

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

The term under the square root is a normalization term which ensures that the distribution integrates to 1.

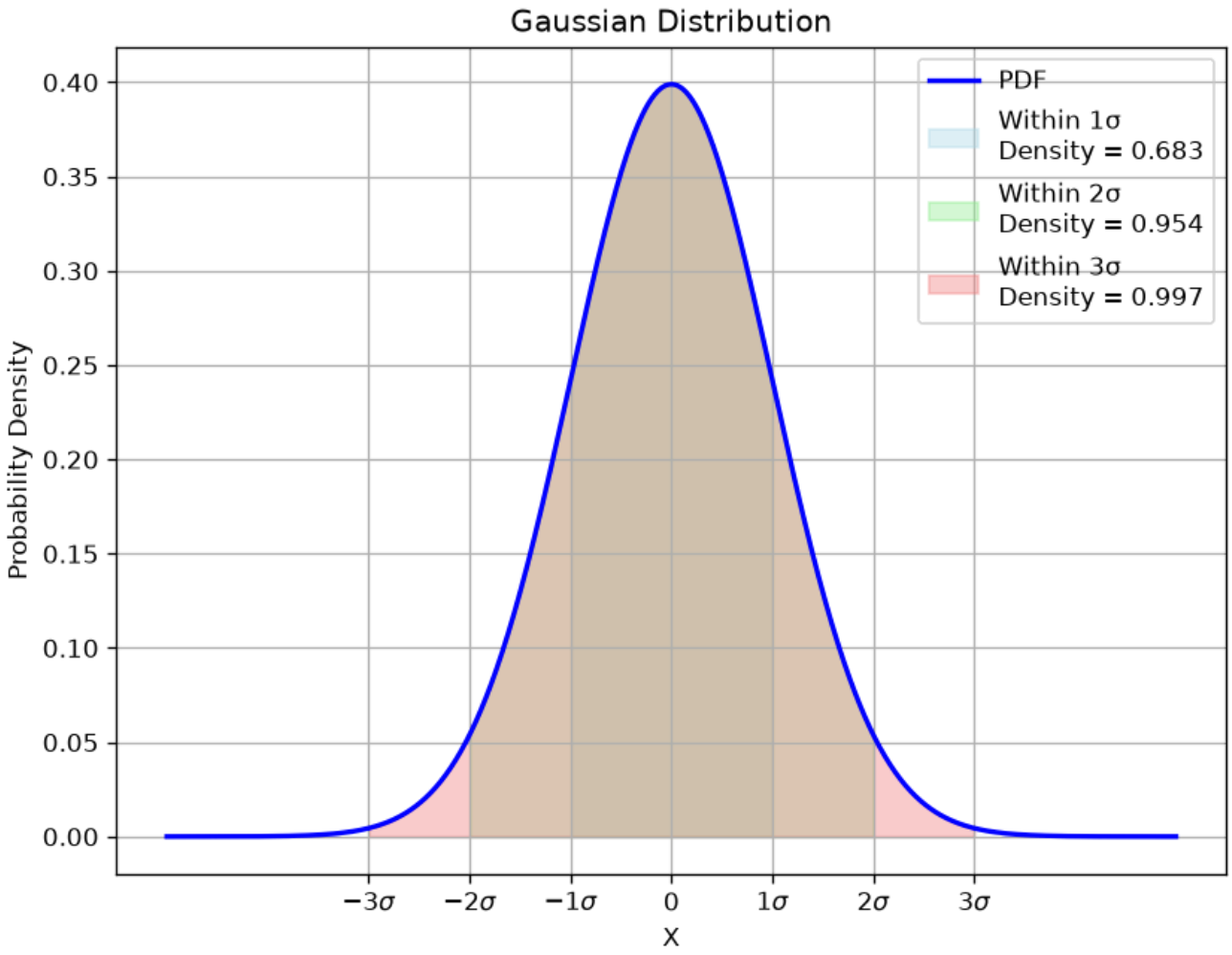
The Gaussian distribution is very used when we do not have much prior knowledge on the real distribution we wish to model. This in mainly due to the **central limit theorem**, which states that the sum of many independent random variables with the same distribution is approximately normally distributed.

Interpretation



If we plot the PDF of a Normal distribution, we can find that it is easy to interpret the meaning of its parameters:

- The resulting curve has a maximum (highest probability) when $x = \mu$
- The curve is symmetric, with the inflection points at $x = \mu \pm \sigma$
- The example shows a normal distribution for $\mu = 0$ and $\sigma = 1$



Another notable property of the Normal distribution is that:

- About 68% of the density is comprised in the interval $[-\sigma, \sigma]$;
- About 95% of the density is comprised in the interval $[-2\sigma, 2\sigma]$;
- Almost 100% of the density is comprised in the interval $[-3\sigma, 3\sigma]$.

Central Limit Theorem

The **Central Limit Theorem (CLT)** is a fundamental statistical principle stating that the distribution of the sum (or average) of a large number of independent and identically distributed (i.i.d.) random variables $\{X_i\}_{i=1}^n$ tends toward a normal (Gaussian) distribution as $n \to \infty$, regardless of the original distribution of each X_i.

This result is crucial because it explains why the Gaussian distribution is so commonly observed in data analysis and natural phenomena, even when the underlying data are not normally distributed.

Key points:

- The CLT applies to sums or averages of i.i.d. random variables.
- The convergence to the normal distribution becomes more accurate as the number of variables n increases.
- The original distribution of the variables can be arbitrary (e.g., uniform, binomial, etc.).

Illustration:
The plot below shows the distributions of the average outcome when rolling different numbers of dice. Each die represents an independent random variable. As the number of dice increases, the distribution of the average outcome becomes increasingly similar to a Gaussian curve.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns  # Import seaborn for KDE

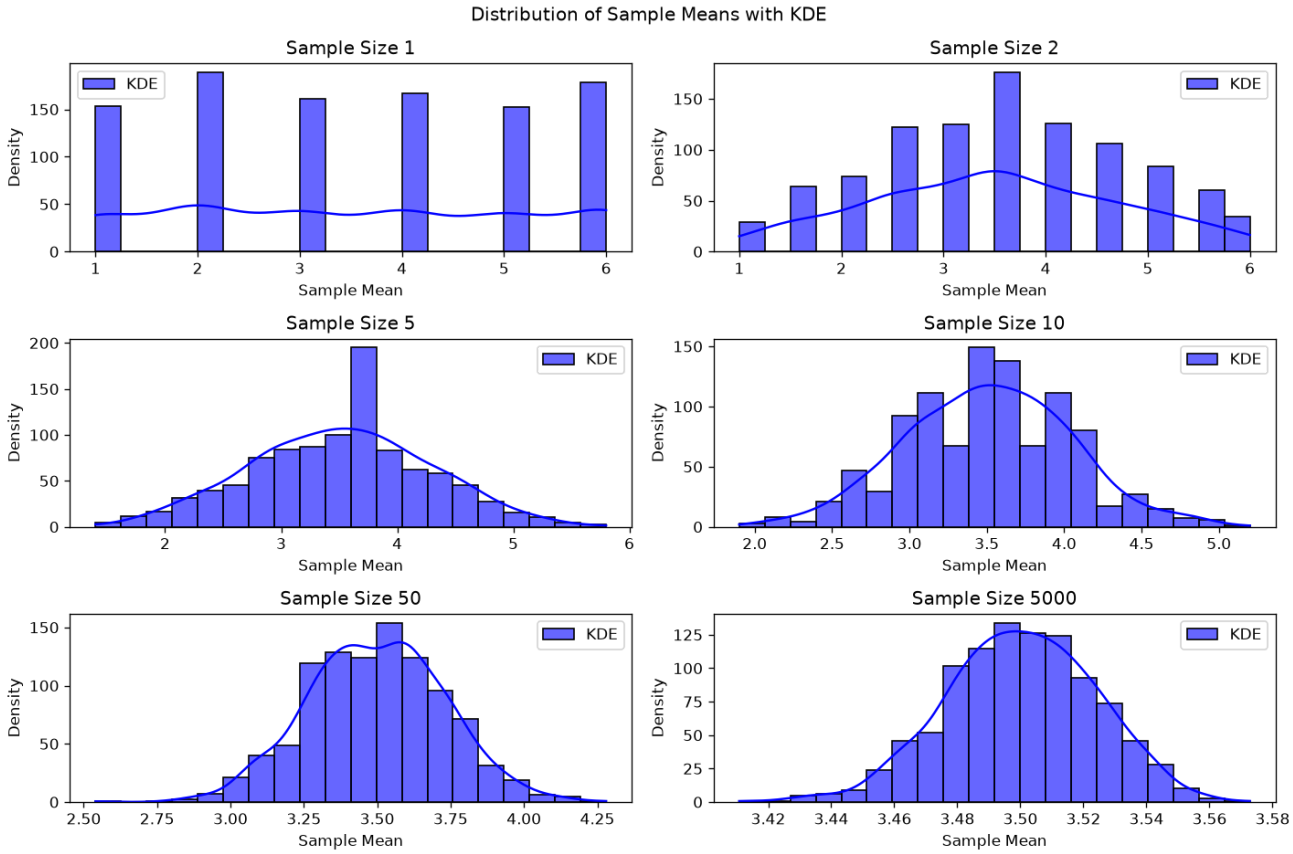
# Simulate rolling a fair six-sided die
num_rolls = 1000
sample_sizes = [1, 2, 5, 10, 50, 5000]

# List to store the means of each sample
sample_means = []

for sample_size in sample_sizes:
    means = [np.mean(np.random.randint(1, 7, sample_size)) for _ in range(num_rolls)]
    sample_means.append(means)

# Plot the sample means for different sample sizes with KDE
plt.figure(figsize=(12, 8))
for i, sample_size in enumerate(sample_sizes):
    plt.subplot(3, 2, i + 1)
    sns.histplot(sample_means[i], bins=20, kde=True, color='blue', label='KDE',
        alpha=0.6)
    plt.title(f'Sample Size {sample_size}')
    plt.xlabel("Sample Mean")
    plt.ylabel("Density")
    plt.legend()

plt.suptitle('Distribution of Sample Means with KDE')
plt.tight_layout()
plt.show()
```



Multivariate Gaussian

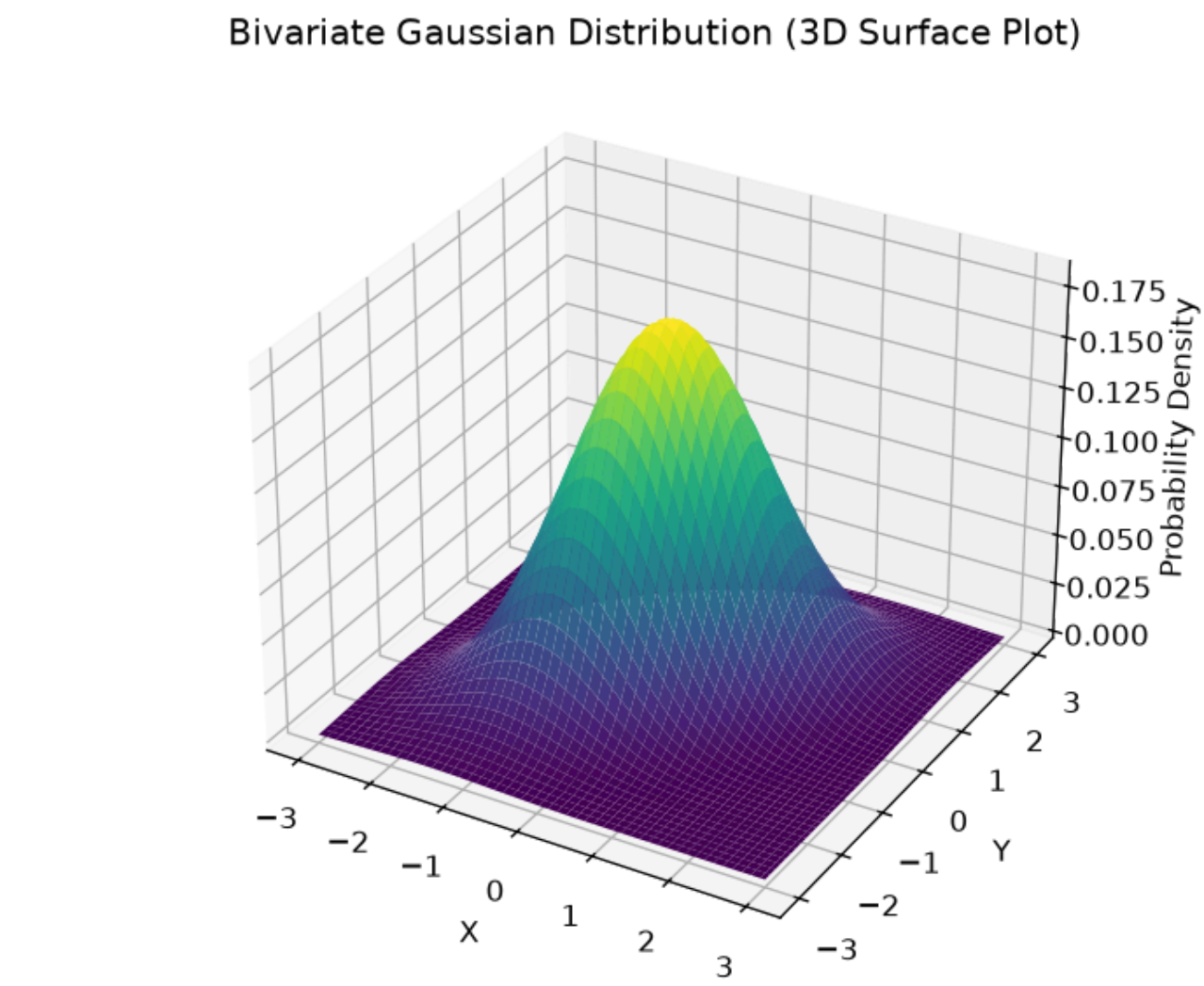
The formulation of the Gaussian distribution generalizes to the multivariate case, i.e., the case in which X is d-dimensional.

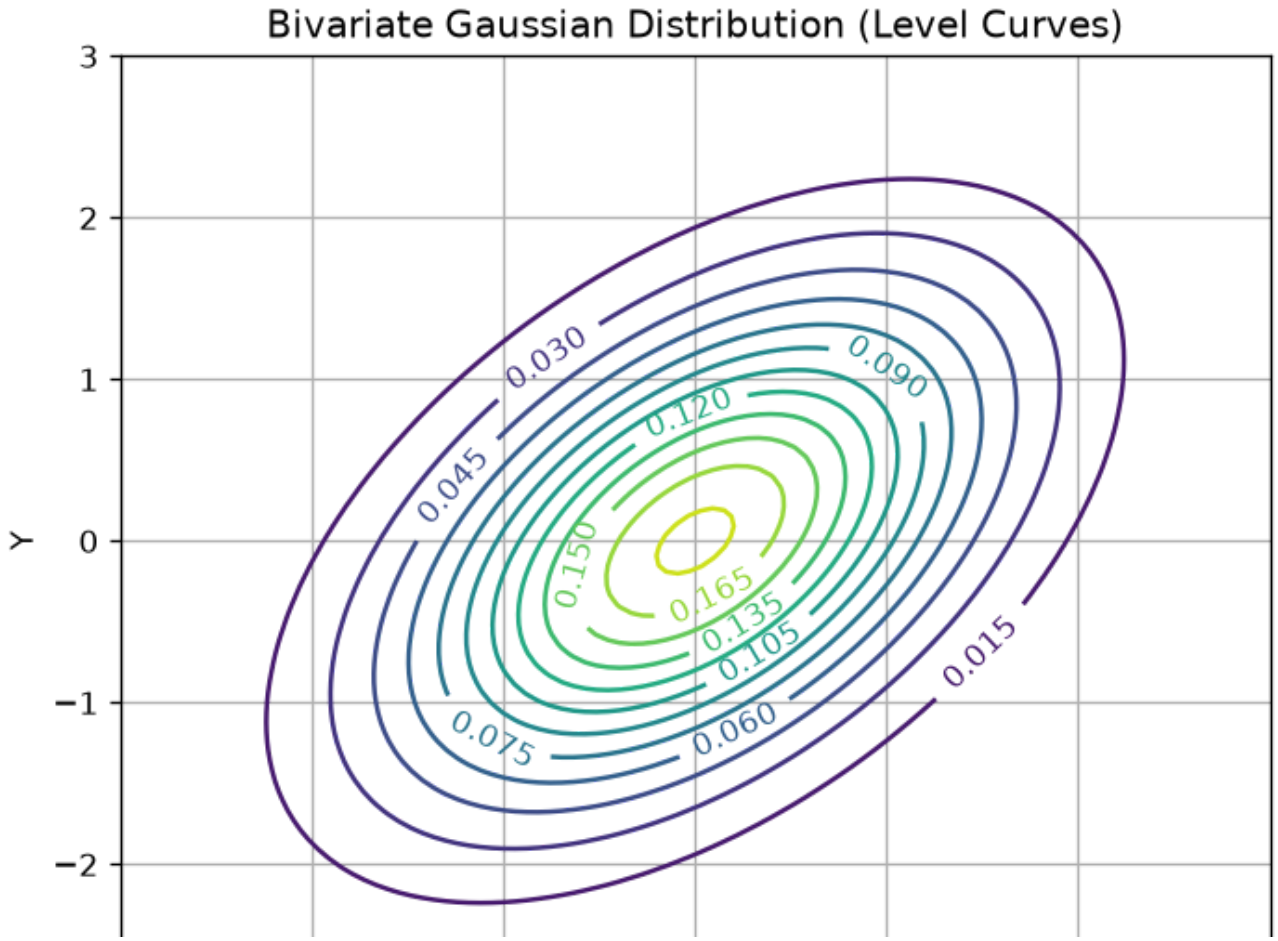
In that case, the distribution is parametrized by a **d-dimensional vector** μ and a $\mathbf{d} \times \mathbf{d}$ **positive definite symmetric matrix** $\mathbf{\Sigma}$. The formulation of the multi-variate Gaussian is:

$$N\left(\mathbf{x}; \mu, \mathbf{\Sigma}\right) = \sqrt{\frac{1}{(2\pi)^d \det(\Sigma)}} e^{\left(-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)\right)}$$

In the 2D case, μ is a 2D point representing the center of the Gaussian (the position of the mode), whereas the matrix Σ influences the "shape" of the Gaussian.

Examples of bivariate Gaussian distributions are shown below.


Bivariate Gaussian Distribution (3D Surface Plot)
Probability Density
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
X
Y
−3
−2
−1
0
1
2
3


Bivariate Gaussian Distribution (Level Curves)
Y
3
2
1
0
−1
−2
0.030
0.045
0.090
0.120
0.150
0.165
0.135
0.105
0.075
0.060
0.015



The two plots above are common representations for bivariate continuous distributions:

- The plot on the top shows a 3D representation of the PDF in which the X and Y axes are the values of the variables, while the third axis reports the probability density.
- Since it's often hard to draw 3D graphs, we often use a contour plot to represent the 3D curves. In the 3D plot, curves of the same color represent points which have the same density in the 3D plot.

Effect of Σ

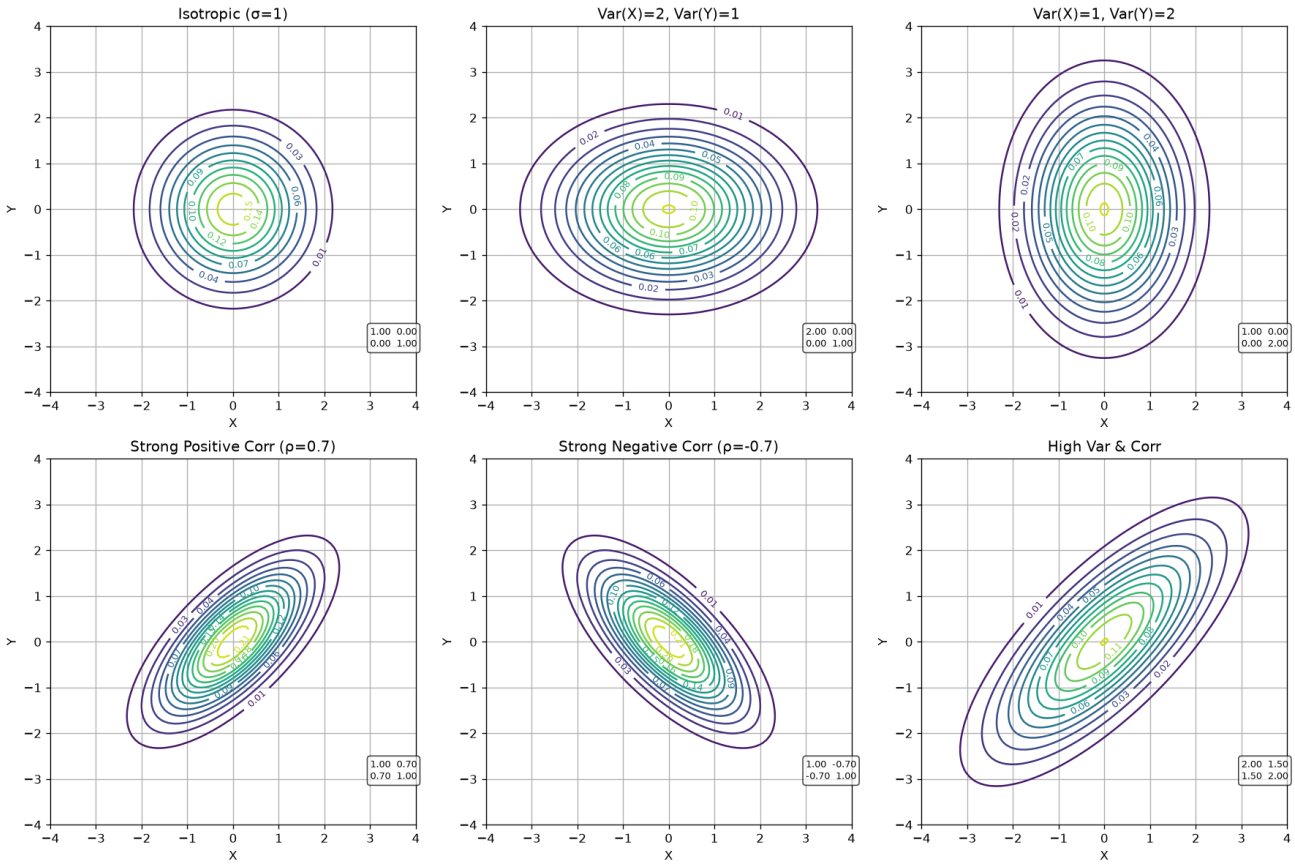
Similar to how variance affects the dispersion of a 1D Gaussian, the covariance matrix Σ affects the dispersion in both axes. As a result, changing the values of the matrix will affect the shape of the distribution. Let's consider the general covariance matrix:

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{yx} & \sigma_y^2 \end{bmatrix}$$

The covariance matrix Σ determines the shape, orientation, and spread of a bivariate Gaussian distribution:

- When Σ is diagonal ($\sigma_{xy} = \sigma_{yx} = 0$), the distribution is symmetric and spreads equally along both axes, resulting in circular contours.
- If the variances along the axes differ, the distribution becomes elongated along the axis with higher variance.
- Non-zero off-diagonal elements introduce correlation: a positive covariance causes the distribution to tilt so that higher values of one variable are associated with higher values of the other, while a negative covariance produces an opposite tilt.
- Larger absolute values of covariance lead to more pronounced stretching along a diagonal direction. When both variances and covariances are large, the distribution is highly dispersed and strongly correlated. These effects are clearly visible when comparing different covariance matrices in contour plots.

Some examples are shown below:



Estimation of the Parameters of a Gaussian Distribution

We have noted that in many cases we can assume a random variable follows a Gaussian distribution. However, it is not yet clear how to choose the parameters of the Gaussian distribution.

Given some data (remember, data is values assumed by random variables!), we can obtain the parameters of the Gaussian distribution related to the data with a **maximum likelihood** estimation.

This consists in computing the mean and variance parameters using the following formula (in the univariate case):

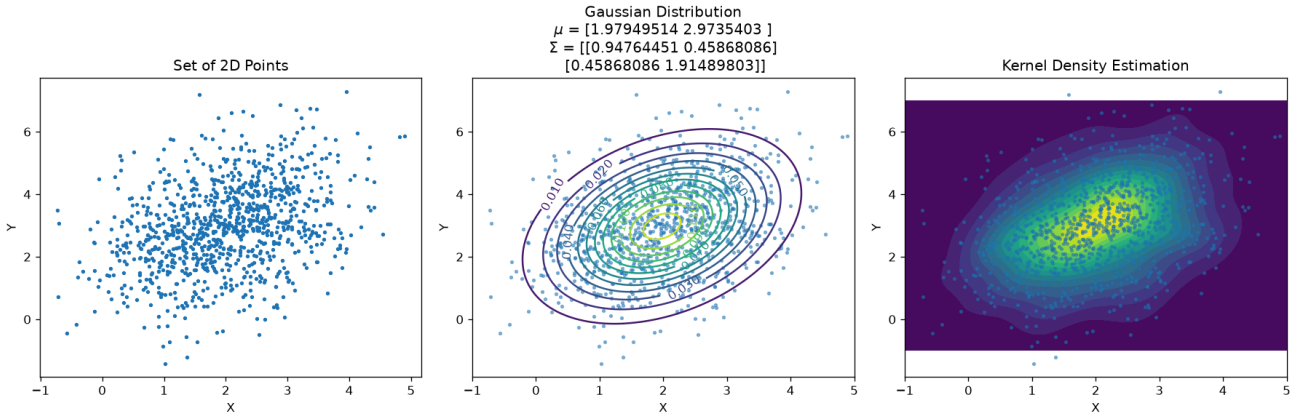
- $\mu = \frac{1}{n} \sum_j x_j$
- $\sigma^2 = \frac{1}{n-1} \sum_j (x_j - \mu)^2$

Where x_j represent the different data points.

In the multi-variate case, the computation of the multi-dimensional μ vector is similar:

- $\mu = \frac{1}{n} \sum_j \mathbf{x}_j$
- Σ is instead computed as the covariance matrix related to X: $\Sigma = Cov(\mathbf{X})$, i.e., $\Sigma_{ij} = Cov(\mathbf{X}_i, \mathbf{X}_j)$

The diagram below shows an example in which we fit a Gaussian to a set of data and compare it with a 2D KDE of the data.



Describing Data Distributions

To effectively summarize and interpret the properties of a probability distribution, it is useful to rely on a set of descriptive statistics. Measures such as **expectation** (mean), **variance**, and **covariance** allow us to capture the central tendency, variability, and relationships between variables, while **standardization** allows us to compare variables on a common scale. These concepts provide essential tools for understanding the behavior of random variables and for analyzing data distributions in practice.

Expectation

When it is known that a random variable follows a probability distribution, **it is possible to characterize that variable** (and hence the related probability distribution) **with some statistics**.

The most straightforward of them is the expectation. **The concept of expectation is very related to the concept of mean.** When we compute the mean of a given set of numbers, we usually sum all the numbers together and then divide by the total.

Since a probability distribution tell us which values will be more frequent than others, we compute this mean with a **weighted average**, where the weights are given by the probability distribution.

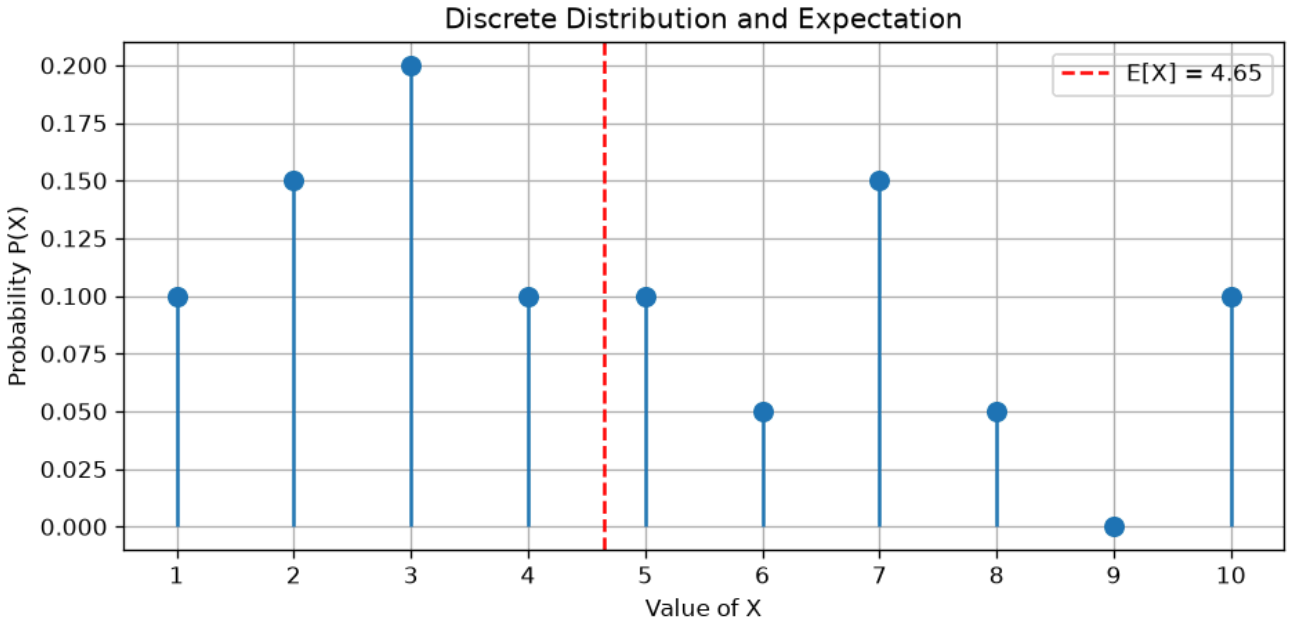
Specifically, we can define the expectation of a **discrete random variable** X as follows:

$$E_{X\sim P}[X] = \sum_{x\in\Omega} xP(x)$$

In the case of **continuous variables**, the expectation takes the form of an integral:

$$E_{X\sim P}[X] = \int_{x\in\Omega} xf(x)dx$$

The diagram below shows an example of expectation:



Variance and Standard Deviation

The variance gives a measure of how much variability there is in a variable X around its mean $E[X]$.

Similarly to the concept of variance as a descriptive measure, we can define the variance in this context as follows:

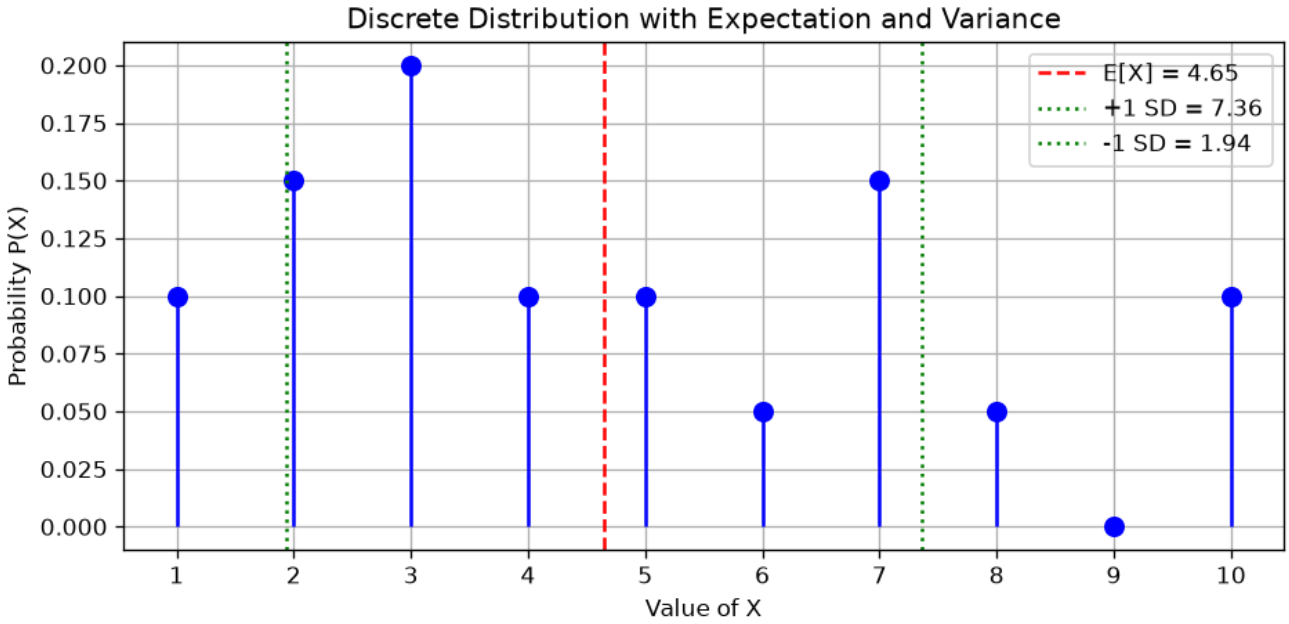
$$Var_{X\sim P}[X] = E_{X\sim P}[(X - E_{X\sim P}[X])^2]$$

As seen with data description, to make the measure of spread more interpretable, we often use the **standard deviation**, which is simply the square root of the variance:

$$Std_{X\sim P}[X] = \sqrt{Var_{X\sim P}[X]}$$

Standard deviation tells us, on average, how far values tend to deviate from the mean, using the same units as the original variable.

The diagram below shows an example:



Covariance

The covariance gives a measure of how two variables are linearly related to each other. It allows to **measure to what extent the increase of one of the variables corresponds to an increase of the value of the other one**.

Given two random variables $X \sim P_X$ and $Y \sim P_Y$, the covariance is defined as follows:

$$Cov_{X \sim P_X, Y \sim P_Y}(X, Y) = E_{X \sim P_X, Y \sim P_Y}[(X - E_{X \sim P_X}[X])(Y - E_{Y \sim P_Y}[Y])]$$

We can distinguish the following terms:

- $E[X]$ and $E[Y]$ are the expectations of X and Y.
- $(X - E[X])$ and $(Y - E[Y])$ are the differences between the samples and the expected values.
- $(X - E[X])(Y - E[Y])$ computes the product between the differences.

We have:

- If the signs of the terms agree, the product is positive.
- If the signs of the terms disagree, the product is negative.

In practice, if when X is larger than the mean, then Y is larger than the mean and vice versa, when X is lower than the mean then Y is lower than the mean, then the two variables are *correlated*, and the covariance is high.

If X is a multi-dimensional variable $X = [X_1, X_2, \ldots, X_n]$, we can compute all the possible covariances between variable pairs: $Cov[X_i, X_j]$. This allows to create a matrix, which is generally referred to as **the covariance matrix**. The general term of the covariance matrix $Cov(X)$ is given by:

$$Cov(X)_{i,j} = \Sigma_{ij} = Cov(X_i, X_j)$$

Entropy (Optional)

Another fundamental way to characterize a probability distribution is through **entropy**, which measures the average uncertainty or unpredictability associated with the possible values of a random variable.

Self-Information

To understand entropy, we first introduce the concept of **self-information**. The self-information of an event x with probability $P(x)$ is defined as:

$$I(x) = -\log_2 P(x)$$

(we will use $\log$ in the future in this context to denote $\log_2$ for brevity)

This definition is motivated by several properties of the logarithm:

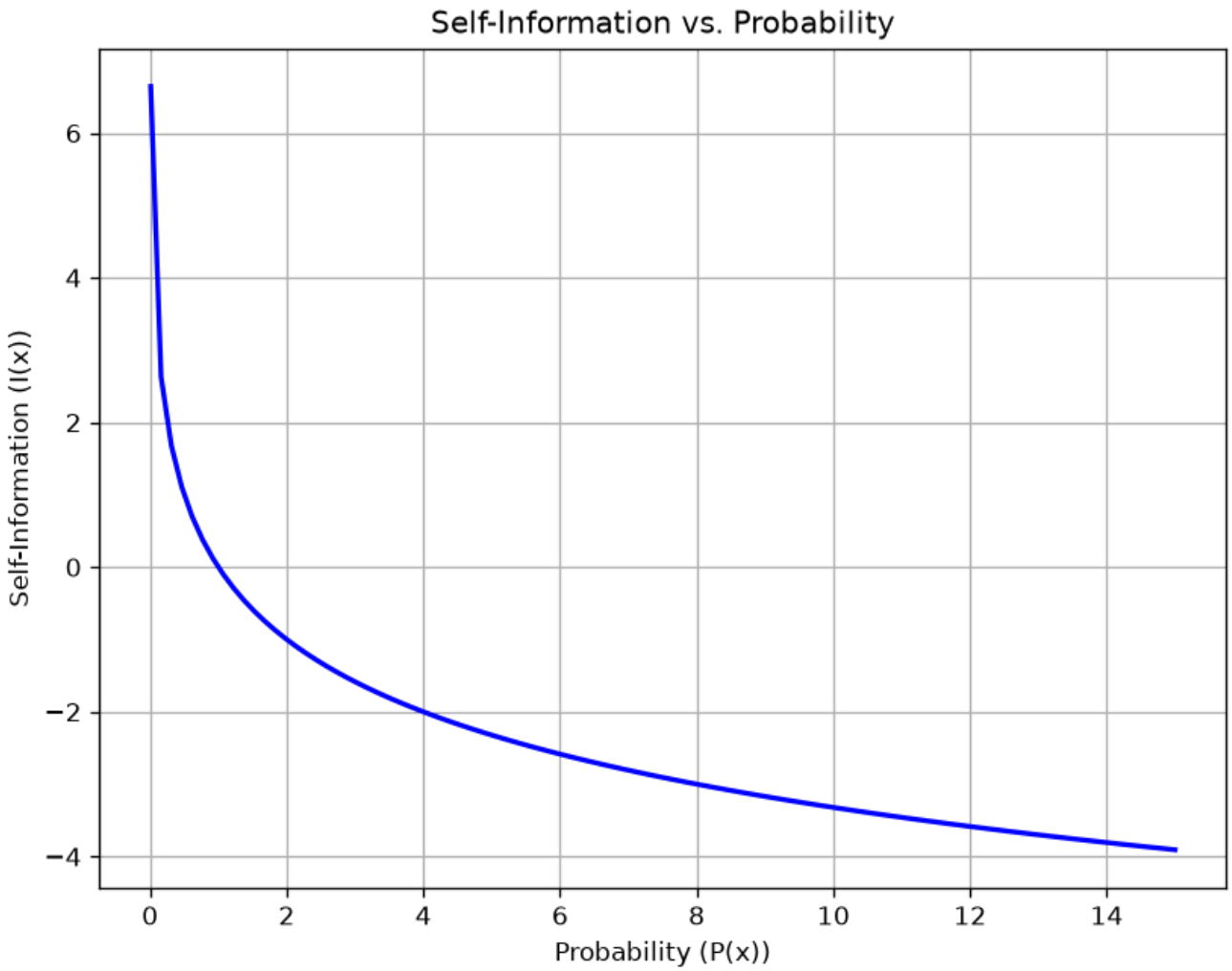
- The logarithm makes self-information **additive** for independent events: the information gained from observing two independent events is the sum of their individual self-informations.

$$I(X = x, Y = y) = -\log\left[P(X = x, Y = y)\right] =$$

$$= \log\left[P(X = x)P(Y = y)\right] = -logP(X = x) - \log P(Y = y) = I(x) + I(y)$$

- The negative logarithm reflects the intuition that **rare events are more informative**: as $P(x)$ decreases, $I(x)$ increases (see plot below)
- The base 2 logarithm measures information in **bits**, which is standard in information theory

For example, if an event is certain ($P(x) = 1$), $I(x) = 0$ (no new information). If an event is very unlikely ($P(x) \to 0$), $I(x) \to \infty$ (maximum surprise).



Entropy of a Distribution

The **entropy** of a random variable X is the average self-information over all possible values of X, weighted by their probabilities (i.e., the expected self-information):

- For discrete variables:

$$H(X) = -\sum_{x \in \Omega} P(x) \log_2 P(x)$$

- For continuous variables (differential entropy):

$$h(X) = -\int_{x \in \Omega} f(x) \log_2 f(x) dx$$

Entropy is highest when the distribution is uniform (maximum uncertainty) and lowest when the distribution is concentrated on a few values (minimum uncertainty).

Note: Entropy is measured in bits if the logarithm base 2 is used and nats if we use a natural logarithm (base e).

Example

- A fair coin ($P(head) = 0.5, P(tail) = 0.5$): $H(X) = 1$ bit.
- A highly biased coin ($P(head) = 0.99, P(tail) = 0.01$): $H(X) \approx 0.08$ bits.

Entropy thus provides a synthetic measure of the uncertainty of a distribution, useful for comparing different random variables and for applications in information theory and machine learning.

Entropy of a Bernoulli variable

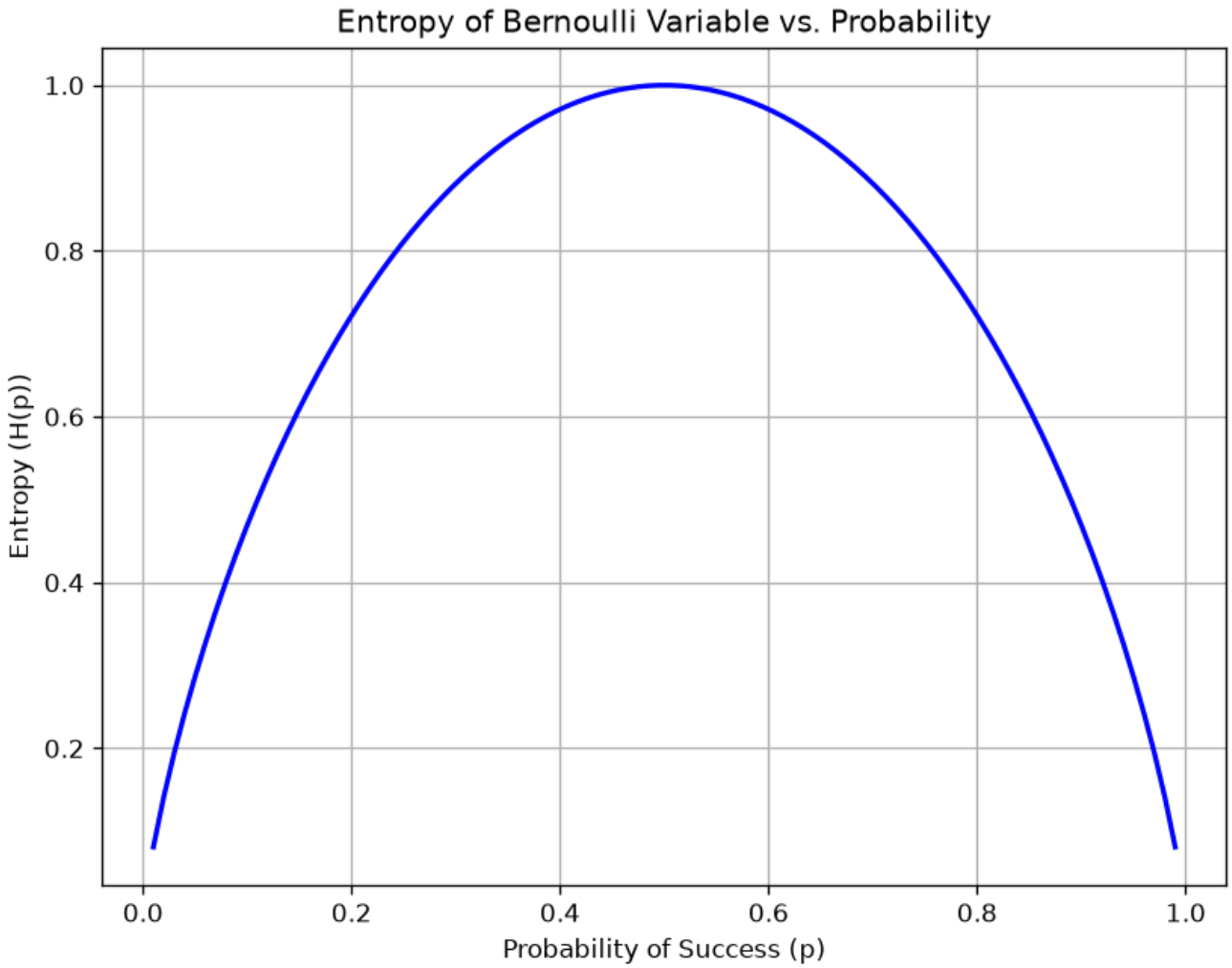
Let's consider a variable X following a Bernoulli distribution with probability p, i.e.:

$$X_p = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases}$$

Then the entropy of this variable will be:

$$H(X_p) = -p \log p - (-1-p) \log(1-p)$$

If we plot $H(X_p)$ with respect to p, we obtain the following graph:



Standardization

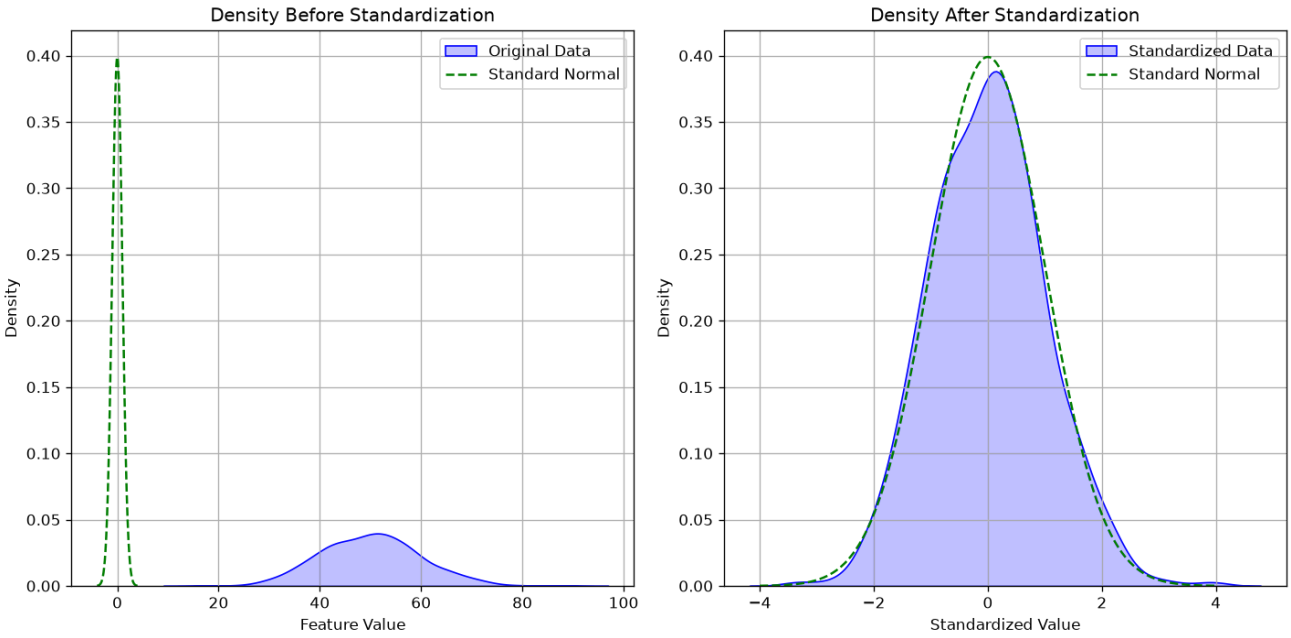
Standardization transforms a random variable X into a variable Z so that it has:

- Expectation equal to zero: $E(Z) = 0$.
- Variance equal to one: $Var(Z) = 1$.

The standardized variable will be:

$$Z = \frac{X - \mu_X}{\sigma_X} = \frac{X - E[X]}{\sqrt{Var[X]}}$$

The plot below shows a data distribution before and after standardization.



Fitting Distributions to Data

Often, we want to fit a theoretical probability distribution to observed data. This means estimating the parameters of the distribution so that it best describes the data. In Python, this can be done easily using `scipy.stats`.

Below is an example of fitting a normal (Gaussian) distribution to a dataset:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Load the Titanic dataset
titanic = pd.read_csv(DATA + "titanic.csv", index_col='PassengerId')

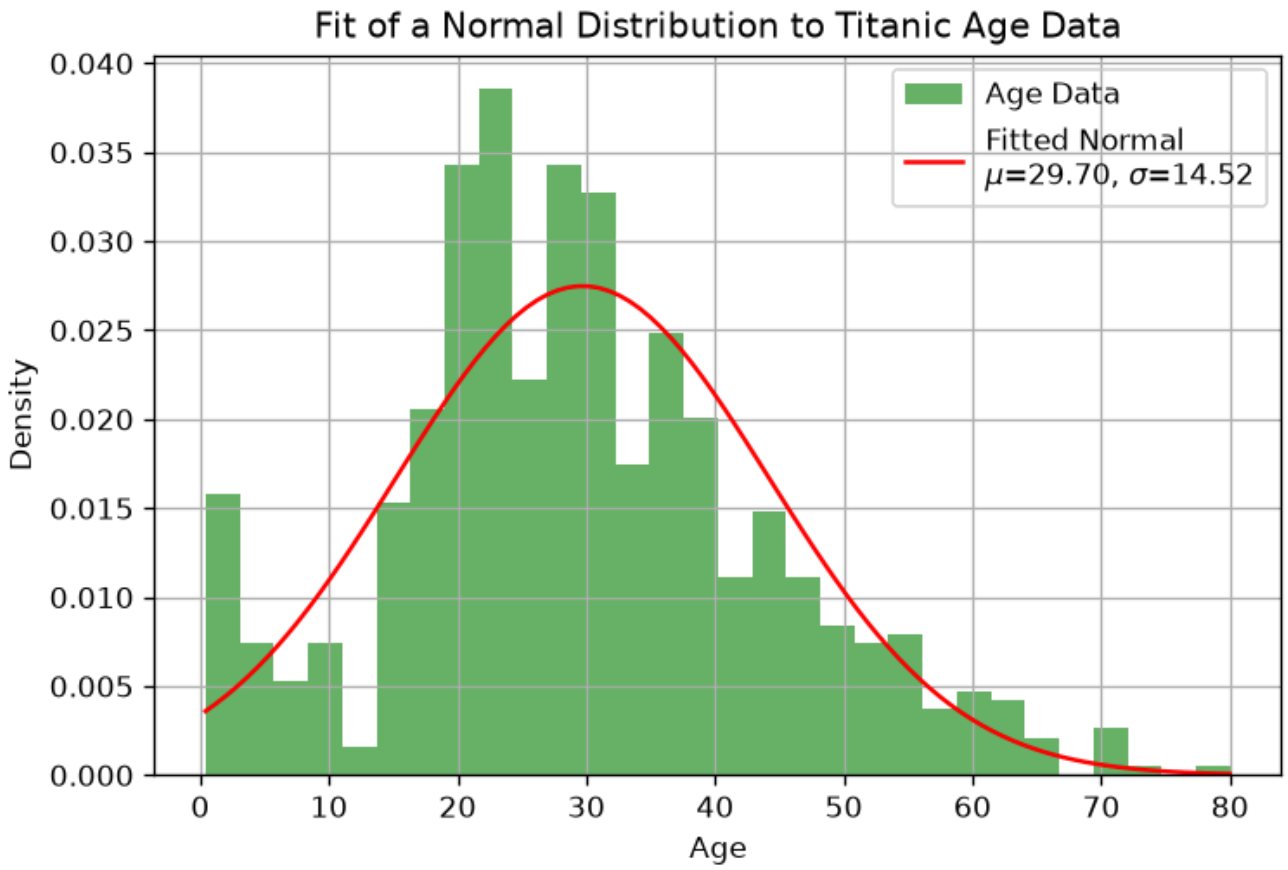
# Extract the Age column and drop missing values
ages = titanic['Age'].dropna()

# Fit a normal distribution to the Age data
mu_hat, sigma_hat = norm.fit(ages)

# Plot the histogram of Age
plt.hist(ages, bins=30, density=True, alpha=0.6, color='g', label='Age Data')

# Plot the fitted normal distribution
x = np.linspace(ages.min(), ages.max(), 100)
pdf_fitted = norm.pdf(x, loc=mu_hat, scale=sigma_hat)
plt.plot(x, pdf_fitted, 'r-', label=f'Fitted Normal\n$\mu$={mu_hat:.2f}, $\sigma$=
        {sigma_hat:.2f}')

plt.xlabel('Age')
plt.ylabel('Density')
plt.title('Fit of a Normal Distribution to Titanic Age Data')
plt.legend()
plt.grid(True)
plt.show()
```



We can do the same with other distribution. For instance, with the binomial distribution:

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import binom

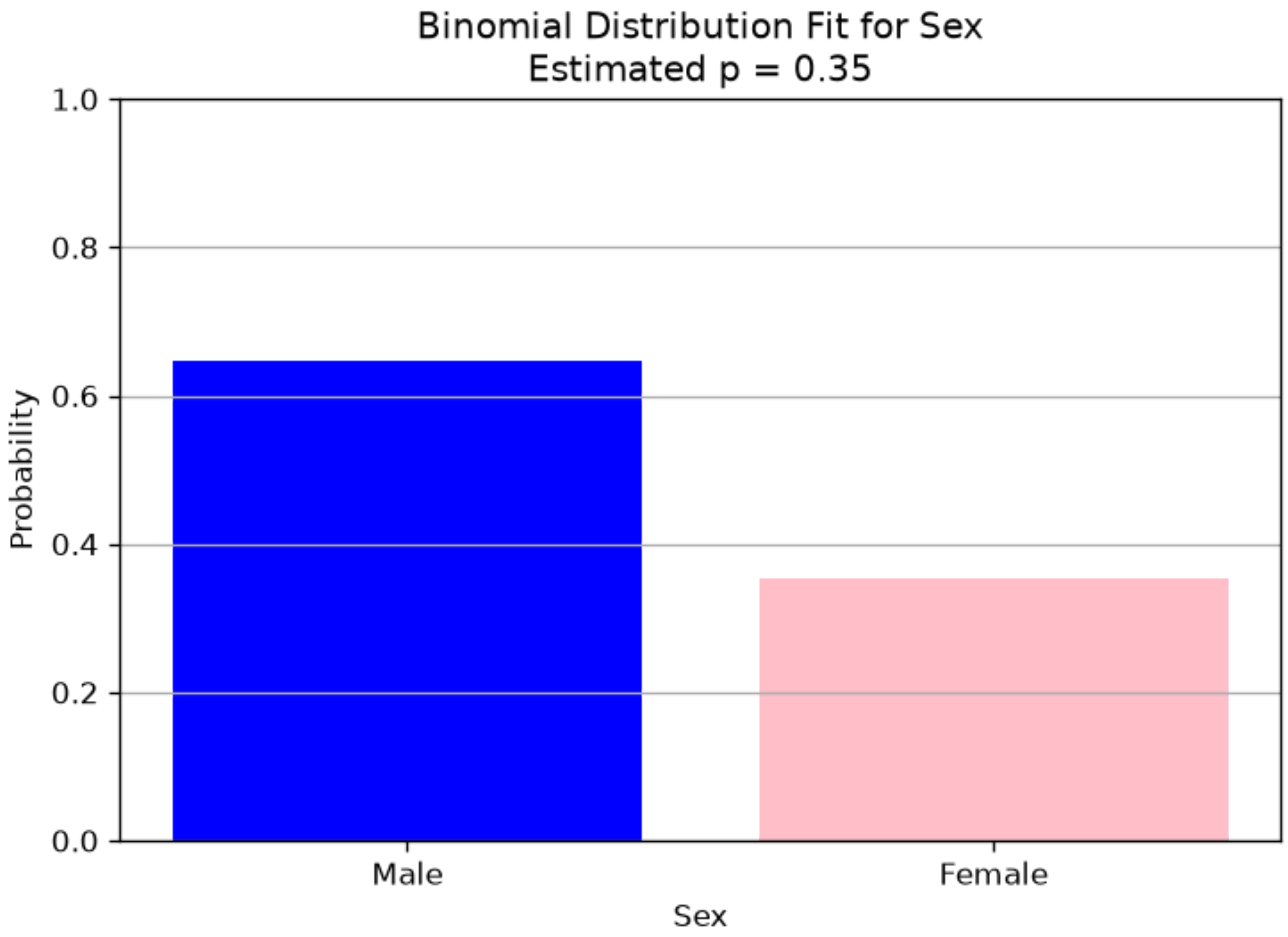
# Load the Titanic dataset
titanic = pd.read_csv(DATA + "titanic.csv", index_col='PassengerId')

# Encode 'Sex': female=1, male=0
sex_numeric = (titanic['Sex'] == 'female').astype(int)

# Estimate p (probability of being female)
p_hat = sex_numeric.mean()
n = len(sex_numeric)

# Binomial PMF for n trials (here, n=1 for each passenger)
x = [0, 1]
pmf = binom.pmf(x, n=1, p=p_hat)

# Plot the PMF
plt.bar(['Male', 'Female'], pmf, color=['blue', 'pink'])
plt.title(f'Binomial Distribution Fit for Sex\nEstimated p = {p_hat:.2f}')
plt.ylabel('Probability')
plt.xlabel('Sex')
plt.ylim(0, 1)
plt.grid(True, axis='y')
plt.show()
```



Exercises

1. **Recognise the shape.** Plot the distribution of `Fare` and of `Age` in the Titanic data. Describe each in words: symmetric or skewed, unimodal or not, heavy-tailed or not. Which summary statistic would you report for each, and why?
2. **Fit and check.** Fit a normal distribution to `Age` and overlay it on the histogram. Then produce a Q-Q plot. Where does the fit fail, and does it matter for a statement about the mean age?
3. **The log-normal.** Plot `Fare` on a logarithmic axis. Does the transformed variable look normal? What does it mean, in the language of the problem, if fares are log-normally distributed?
4. **Simulate the central limit theorem.** Draw 1000 samples of size n from a strongly skewed distribution (an exponential, say) for $n \in \{2, 5, 30, 100\}$, and plot the distribution of the sample means. At which n does it start to look normal?
5. **Discrete distributions.** `SibSp` counts siblings and spouses aboard. Fit a Poisson distribution to it and compare the expected and observed counts. Where does the Poisson model break down, and what feature of the data explains it?

References

- Parts of chapter 1 of [1];

- Most of chapter 3 of [2];
- Parts of chapter 8 of [3].

[1] Bishop, Christopher M. *Pattern recognition and machine learning*. springer, 2006. https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf

[2] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. https://www.deeplearningbook.org/

[3] Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Statistical inference

Chapter resources

Slides (PDF) Open in Colab Practice quiz

So far, we have seen methods for describing a sample of data (descriptive statistics) and we have reasoned on abstract concepts using basic probability theory concepts. In practice, we are often interested in the properties of a **population**, rather than a sample or some abstract quantities. Examples are:

- What is the percentage of votes each candidate will get at an election?
- What is the proportion of defective goods in a manufacturing process?
- Is there a relationship between smoking and developing a given disease in the world population?

One approach to answer these questions would be to collect the whole population, but this is often unfeasible (e.g., interviewing **all voters**) and sometimes impossible.

The statistician's approach is instead to **sample** a subset of the whole population and try to **infer some of the properties of the population from the sample**. This part of statistics is called **statistical inference**. Analyzing data using such techniques is often called an **inferential analysis**. In this part of the course, we will review different statistical tools for inferential analysis and show some concrete examples, without giving a formal definition of such tools, which is left to other courses.

Sampling

The first step towards an inferential analysis is the **sampling process**. When we acquire a pre-made dataset, sampling is already done, while when we collect data, we are actually sampling from the population. In both cases, it is important to reason on the properties of the sample we will work on.

Simple random sample

The easiest way to sample from a population is **randomly**. A simple random sample makes two assumptions:

- **Unbiasedness**: each element of the population has the same probability of being selected;
- **Independence**: selecting one of the elements of the population does not affect the selection of the other elements in any way.

This approach guarantees that, if we collect a large number of elements, the obtained sample will be a good representative of the population. For instance, if in the population of interest we have 10% of people over 70, we expect this proportion to be roughly represented in the sample as well.

An example of bad sampling:

We want to ask the inhabitants of a city whether they are satisfied with the quality of life in that city. To sample a large quantity of subjects, we go to the main square and ask passengers to reply to a few questions. If a ground of friends stops we interview all of them to maximize the number of examples we can obtain.

The sampling design outlined above has two important issues:

- **Unbiasedness**: the selection process is biased (**selection bias**). We selected a single location in the city (the main square) and hence we are **oversampling** people who tend to spend time there (e.g., because they work in the city center), versus people who do not spend much time there (e.g., because they work in the periphery).
- **Independence**: when we interview groups of people, in fact, we are breaking the independence assumption, Indeed, selecting one of the people is not independent of selecting others (the members of the same group).

Another example of flawed sampling:

We want to check how many people believe a given conspiracy theory. To do so, I send a message to all my contacts (500). About 200 of them reply to my message and 180 of them say they do believe that theory. 80% of people actually believe it!

Also here there are important issues:

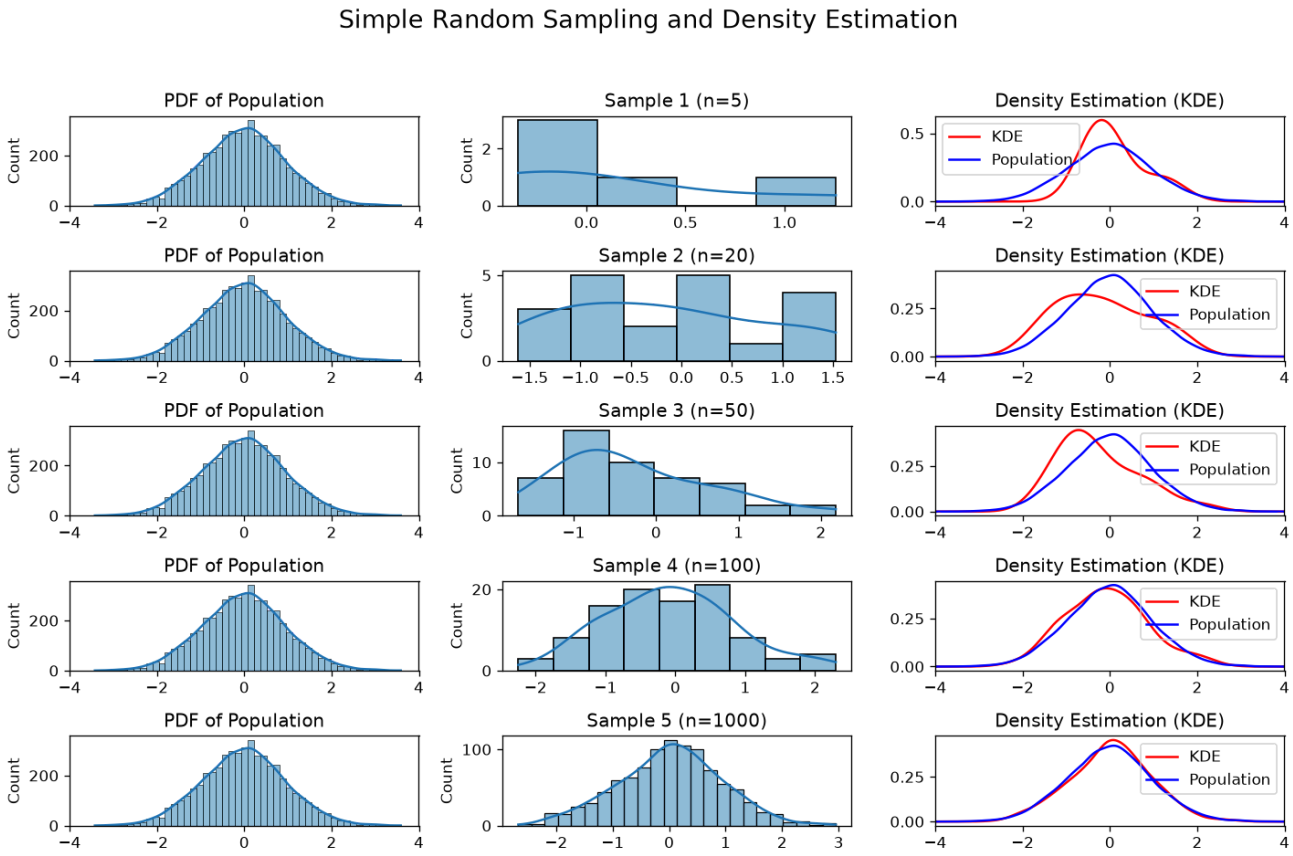
- **Selection bias**: I am not randomly sampling. Instead, I am choosing among my contacts.
- **Response bias**: Only 200 people replied. Chances are that only people who are very motivated will reply. Maybe most of the believers did, while the others just ignored my message.

Another example:

We interview people on their voting preferences by dialing random phone numbers.

While this may seem sound, we will not end up with a simple random sample because we will not select people without a phone number and we will oversample people with more than one numbers (e.g., work and home).

The plot below shows the importance of sample size:



Stratified Sampling

Sometimes, the population under study is **heterogeneous**, meaning it contains distinct subgroups with different characteristics. In such cases, **simple random sampling** may fail to capture the diversity of the population, leading to **under-representation** of certain groups.

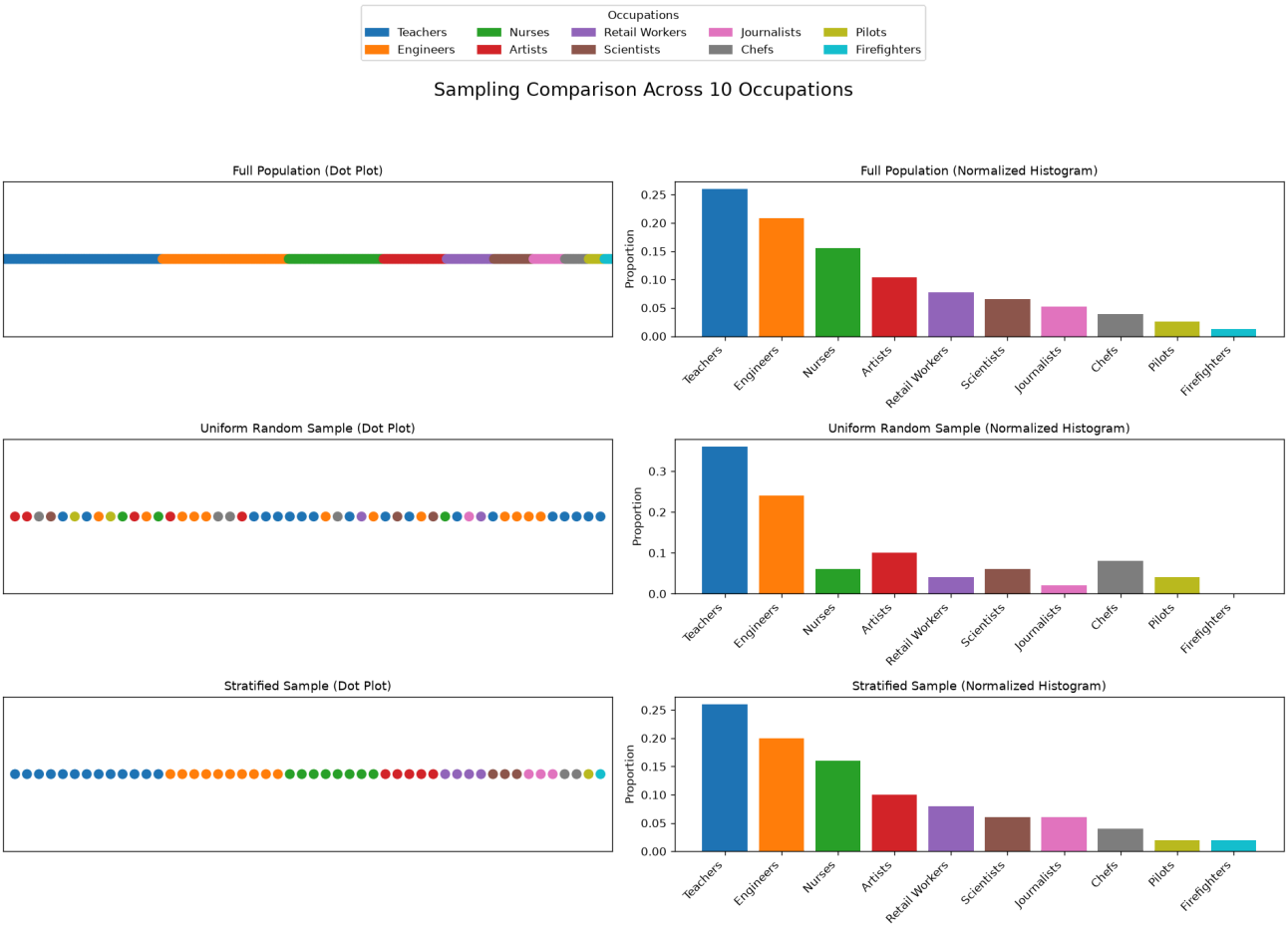
For example, suppose we hypothesize that the variable `Occupation` influences voting preferences. If we randomly sample from the entire population, we might unintentionally select too few individuals from certain occupations, skewing our results.

To address this, we can use **stratified sampling**. This involves:

- Dividing the population into **homogeneous subgroups** (*strata*) based on a relevant variable, such as `Occupation`
- Performing **random sampling within each stratum**, ensuring proportional representation

This method improves representativeness but requires **prior knowledge or assumptions** about the population structure.

The figure below shows a heterogeneous population, a sample obtained via uniform random sampling, and a sample obtained via stratified sampling.



Sampling in Python

In Python we can easily sample from known distributions using numpy `random` module, as shown in the example below:

```python
import numpy as np
import matplotlib.pyplot as plt

# Set the sample size
sample_size = 1000

# Sample from the Normal (Gaussian) Distribution
mean_normal = 0
std_dev_normal = 1
samples_normal = np.random.normal(mean_normal, std_dev_normal, sample_size)

# Sample from the Exponential Distribution
rate_exponential = 0.5
samples_exponential = np.random.exponential(1/rate_exponential, sample_size)

# Sample from the Uniform Distribution
low_uniform = 0
high_uniform = 1
samples_uniform = np.random.uniform(low_uniform, high_uniform, sample_size)

# Sample from the Laplace Distribution
loc_laplace = 0
scale_laplace = 1
samples_laplace = np.random.laplace(loc_laplace, scale_laplace, sample_size)

# Sample from the Poisson Distribution
lam_poisson = 3
samples_poisson = np.random.poisson(lam_poisson, sample_size)

# Create histograms to visualize the distributions
plt.figure(figsize=(15, 3))

plt.subplot(1, 5, 1)
plt.hist(samples_normal, bins=30, color='blue', alpha=0.7)
plt.title('Normal Distribution')

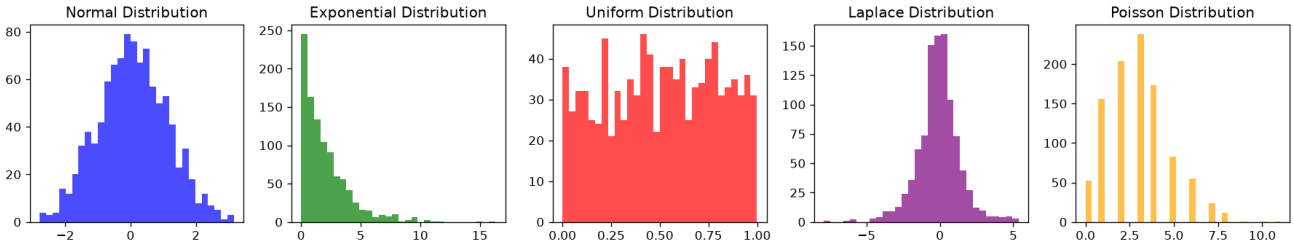
plt.subplot(1, 5, 2)
plt.hist(samples_exponential, bins=30, color='green', alpha=0.7)
plt.title('Exponential Distribution')

plt.subplot(1, 5, 3)
plt.hist(samples_uniform, bins=30, color='red', alpha=0.7)
plt.title('Uniform Distribution')

plt.subplot(1, 5, 4)
plt.hist(samples_laplace, bins=30, color='purple', alpha=0.7)
plt.title('Laplace Distribution')
```

```
plt.subplot(1, 5, 5)
plt.hist(samples_poisson, bins=30, color='orange', alpha=0.7)
plt.title('Poisson Distribution')

plt.tight_layout()
plt.show()
```



We can also sample from an existing sample to obtain a **subsample**. This can be done starting from a dataset. For instance, let us consider the *weight-height* dataset:

```
import pandas as pd
data=pd.read_csv(DATA + "height_weight.csv")
data.info()
data.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4231 entries, 0 to 4230
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   sex     4231 non-null   object
 1   BMI     4231 non-null   float64
 2   height  4231 non-null   float64
 3   weight  4231 non-null   float64
dtypes: float64(3), object(1)
memory usage: 132.3+ KB
```

	sex	BMI	height	weight
0	M	33.36	187.96	117.933920
1	M	26.54	177.80	83.914520
2	F	32.13	154.94	77.110640
3	M	26.62	172.72	79.378600
4	F	27.13	167.64	76.203456

We can obtain a subsample of 1000 observations as follows:

```
sample1=data.sample(1000)
sample1.info()
sample1.head()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1000 entries, 129 to 3443
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   sex     1000 non-null   object
 1   BMI     1000 non-null   float64
 2   height  1000 non-null   float64
 3   weight  1000 non-null   float64
dtypes: float64(3), object(1)
memory usage: 39.1+ KB
```

	sex	BMI	height	weight
129	M	33.90	182.88	113.398000
3065	F	22.32	162.56	58.966960
2755	M	31.68	185.42	108.862080
3505	F	21.65	165.10	58.966960
1785	F	20.92	162.56	55.338224

Sampling will be random and uniform. As can be seen the indexes of the rows are shuffled. By default, Pandas will sample without replacement. We can sample with replacement specifying `replace=True`. For instance, we can obtain a larger sample (with repetitions) as follows:

```
sample2=data.sample(5000, replace=True)
sample2.info()
sample2.head()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 5000 entries, 1637 to 1037
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   sex     5000 non-null   object
 1   BMI     5000 non-null   float64
 2   height  5000 non-null   float64
 3   weight  5000 non-null   float64
dtypes: float64(3), object(1)
memory usage: 195.3+ KB
```

	sex	BMI	height	weight
1637	F	26.53	157.48	65.77084
4082	M	26.54	177.80	83.91452
1890	F	28.33	152.40	65.77084
3205	M	24.39	177.80	77.11064
477	F	24.02	162.56	63.50288

We can perform stratified sampling as follows:

```
# Define the stratification variable
strata_variable = 'sex'

# Number of samples per stratum
samples_per_stratum = 500

# Draw the same number of rows from every stratum. `.sample()` on a groupby does
# this directly, and unlike `.apply()` it keeps the grouping column.
stratified_sample = (
    data.groupby(strata_variable, group_keys=False)
        .sample(samples_per_stratum)
        .reset_index(drop=True)
)

# Inspect the result
stratified_sample.info()
stratified_sample.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   sex     1000 non-null   object
 1   BMI     1000 non-null   float64
 2   height  1000 non-null   float64
 3   weight  1000 non-null   float64
dtypes: float64(3), object(1)
memory usage: 31.4+ KB
```

	sex	BMI	height	weight
0	F	26.31	170.18	76.203456
1	F	27.45	167.64	77.110640
2	F	28.29	165.10	77.110640
3	F	30.03	162.56	79.378600
4	F	29.16	162.56	77.110640

Note that, with this method, sampled observations will be sorted by `sex`. To get rid of this bias, we can shuffle the dataframe as follows:

```
stratified_sample = stratified_sample.iloc[np.random.choice(len(stratified_sample),
        len(stratified_sample), replace=False)]
stratified_sample.head()
```

	sex	BMI	height	weight
898	M	26.55	175.26	81.646560
452	F	29.01	162.56	76.657048
665	M	28.89	187.96	102.058200
200	F	31.26	152.40	72.574720
378	F	21.92	170.18	63.502880

Males and females will be perfectly balanced by design in this stratified sample:

```
stratified_sample['sex'].value_counts()
```

```
sex
M    500
F    500
Name: count, dtype: int64
```

Sampling Distribution of the Mean

Let's consider this problem:

> A bakery sells packages of cookies labeled as weighing **1kg**. Due to small variations in the production process, the actual weight of each package may differ slightly. To ensure quality, the bakery wants to verify that the average weight is close to the target.

We take a **random sample of** $n = 1000$ packages and measure their weights $x_1, x_2, \ldots, x_n$. We compute the sample mean:

$$\overline{x} = \frac{x_1 + x_2 + \ldots + x_n}{n} = 1000.2\text{g}$$

This value is close to the target of 1000g, so we might conclude that the production process is accurate. But what if we took another sample? Would we get the same result?

$$\overline{x} = \frac{x_1 + x_2 + \ldots + x_n}{n} = 999.7\text{g}$$

This is still close to the target, but slightly lower than before. If we repeated this process many times, we would get a **distribution of sample means**.

We treat each package's weight as a **random variable** X_i, with:

$$E[X_i] = \mu$$
$$Var[X_i] = \sigma^2$$

where μ and σ are the **true mean and standard deviation** of the population.

The sample mean $\overline{X}$ is also a random variable:

$$\overline{X} = \frac{X_1 + X_2 + \ldots + X_n}{n}$$

Note that, while we cannot make assumptions for the individual X_i, for the **central limit theorem**, we know that:

> For large n, the distribution of $\overline{X}$ will be approximately **normal**, regardless of the shape of the original data.

We can derive the expected value, variance and standard deviation of X as follows:

$$E[\overline{X}] = E\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\sum_{i=1}^{n} E[X_i] = \mu$$

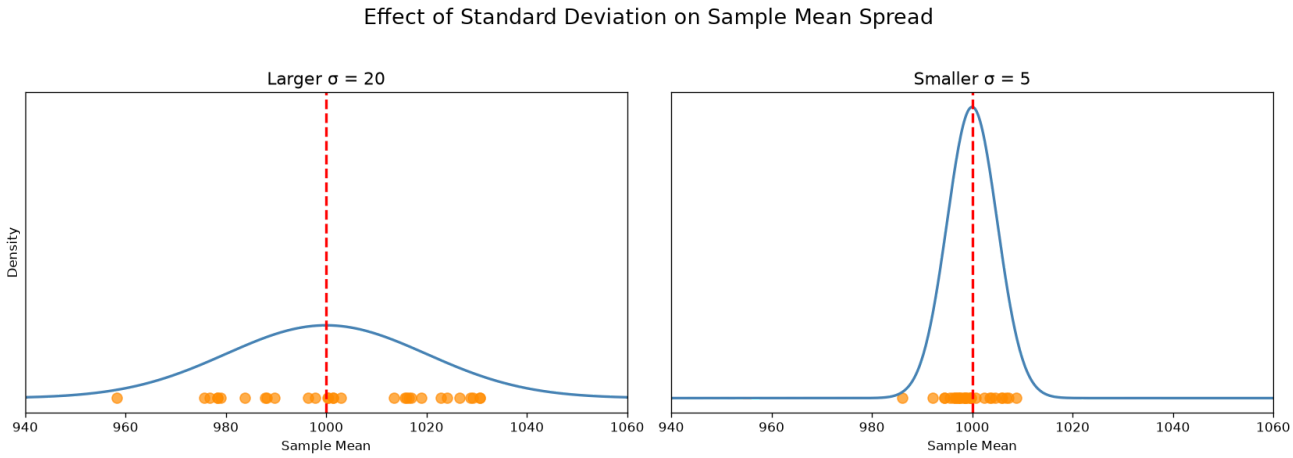
$$Var[\overline{X}] = Var\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n^2}\sum_{i=1}^{n} Var[X_i] = \frac{\sigma^2}{n}$$

$$Std[\overline{X}] = \frac{\sigma}{\sqrt{n}}$$

We note that:

- The **expected value of the sample means** (the mean of the means) converges to the **mean of the population**. This is the value we want to estimate: the actual average weight of packages.
- The **standard deviation of the distribution** quantifies the precision according to which we can measure the mean. A small standard deviation indicates that sample means have small variability, so a single estimate will likely be closer to the true mean and hence more reliable.

This is exemplified by the following figure:



Standard Error

While useful to quantify our precision in the estimation of the true mean, the standard deviation depends on the variance of the population, which is unknown.

In practice, we estimate it using the sample standard deviation:

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

and define the Standard Error (SE) as follows:

$$SE(\overline{x}) = \frac{s_{n-1}}{\sqrt{n}}$$

Similar to the standard deviation, **the standard error quantifies our uncertainty in the estimation of the true mean**.

Note that the standard error is **inversely proportional to the square root of the sample size**:

$$SE(\overline{x}) \propto \frac{1}{\sqrt{n}}$$

This relationship is common in many statistical estimators. It means that:

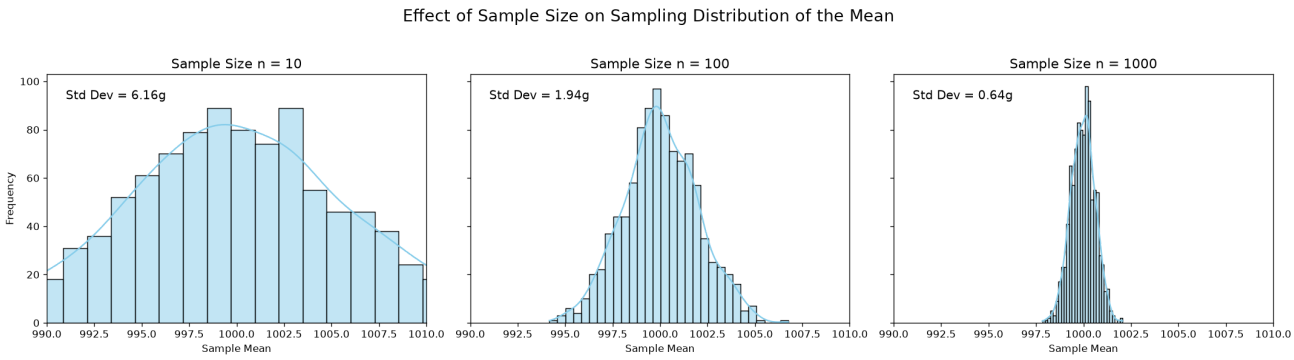
- If we **double the sample size**, the error **reduces by about 30%**.

- To **halve the error**, we need to **quadruple the sample size**.

This is a key insight: reducing uncertainty requires much more data than we might expect.

The standard error tells us how close our sample mean $\overline{x}$ is likely to be to the true mean μ. A small standard error means that repeated samples will yield similar results, and our estimate is precise. A large standard error means more variability and less confidence in the estimate.

This is shown in the graph below:



This figure illustrates how **sample size affects the precision of the sample mean**. Each panel shows the distribution of sample means obtained by repeatedly sampling from the same population (mean = 1000g, standard deviation = 20g), but with different sample sizes:

- **Left (n = 10)**: The distribution is wide, indicating high variability. Small samples lead to more fluctuation in the estimated mean.
- **Center (n = 100)**: The distribution narrows, showing improved stability. The sample means are more tightly clustered around the true mean.
- **Right (n = 1000)**: The distribution is very narrow. Large samples yield highly consistent estimates, with minimal deviation from the true mean.

This visual reinforces the key idea that **larger samples reduce the standard error**, making our estimates more reliable. The spread of the sampling distribution shrinks proportionally to $1/\sqrt{n}$.

t-Student Distribution

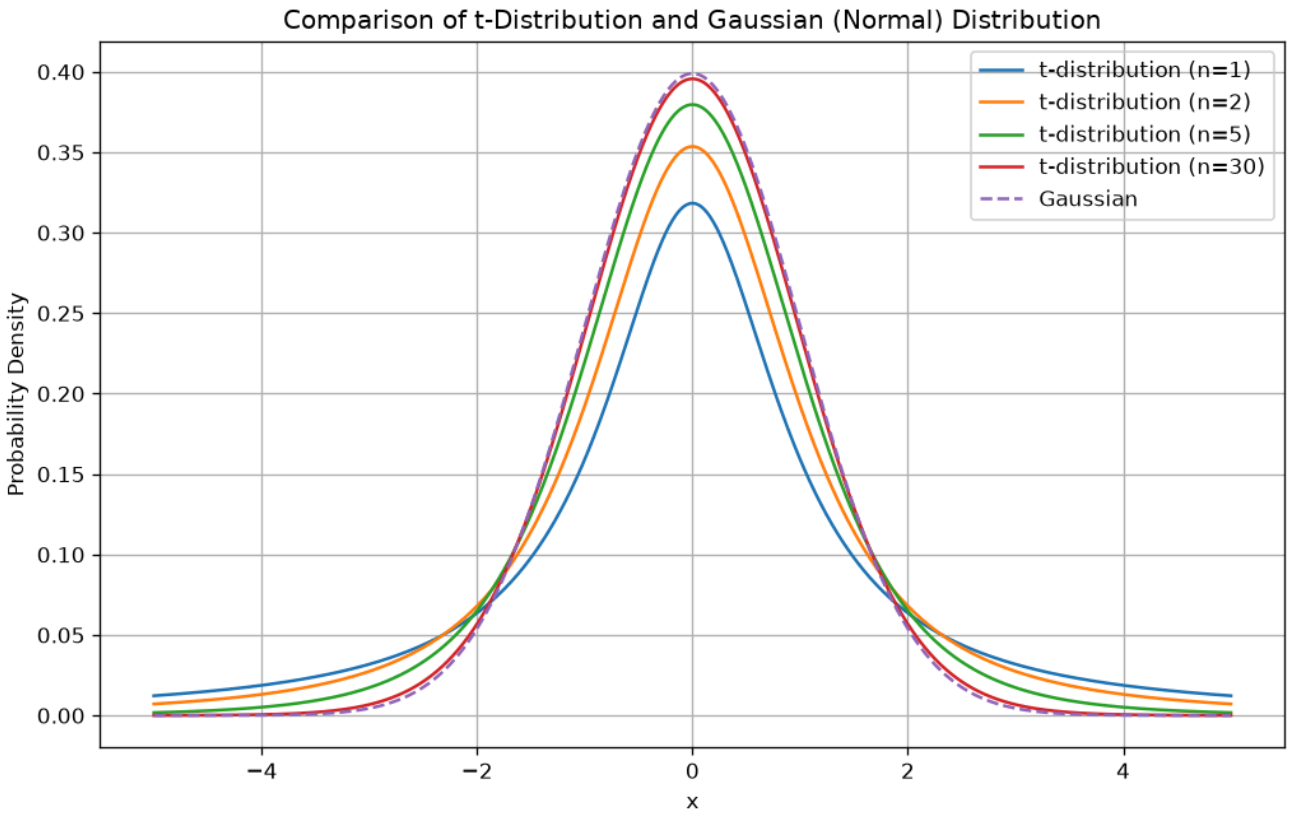
When n is small, the distribution of $\overline{X}$ is **not exactly normal**. Instead, we use the **t-Student distribution**, which accounts for extra uncertainty:

$$t_{n-1} = \frac{\overline{X} - \mu}{s_{n-1}/\sqrt{n}}$$

This follows a t-distribution with $n - 1$ degrees of freedom. The t-distribution looks like a Gaussian but has **heavier tails**, meaning more room for extreme values. As n increases, the t-distribution **converges to the normal distribution**.

Note that the t-Student distribution is obtained by applying standardization using the standard error rather than the standard deviation (which is unknown).

The relationship between a t-Student and Gaussian distribution is shown in the figure below:



As n gets larger, the t-Distribution approximates a Gaussian distribution.

We can now characterize how the means will distribute. In particular, **for large values of n, sample means will distribute according to a Gaussian distribution with standard deviation equal to the sample standard deviation**.

Confidence Intervals

Let's return to our bakery example. We want to estimate the **true average weight** μ of cookie packages, which are labeled as 1kg. We take a sample of $n = 1000$ packages and compute:

$$\overline{x} = 1000.2\text{g}$$

This is our estimate of the true mean. However, we know that this value may vary depending on the sample we draw. If we took another sample, we might get a slightly different result. It is interesting to define the **bounds of such variation**, to understand where the true mean μ is likely to be found relative to our estimate $\overline{x}$.

Remember that the sample mean $\overline{X}$ follows a **Gaussian distribution** centered at μ due to the **Central Limit Theorem**.

We also know that about 68.3% of the density of the Gaussian distribution will be within $\mu - \sigma$ and $\mu + \sigma$, so we can write:

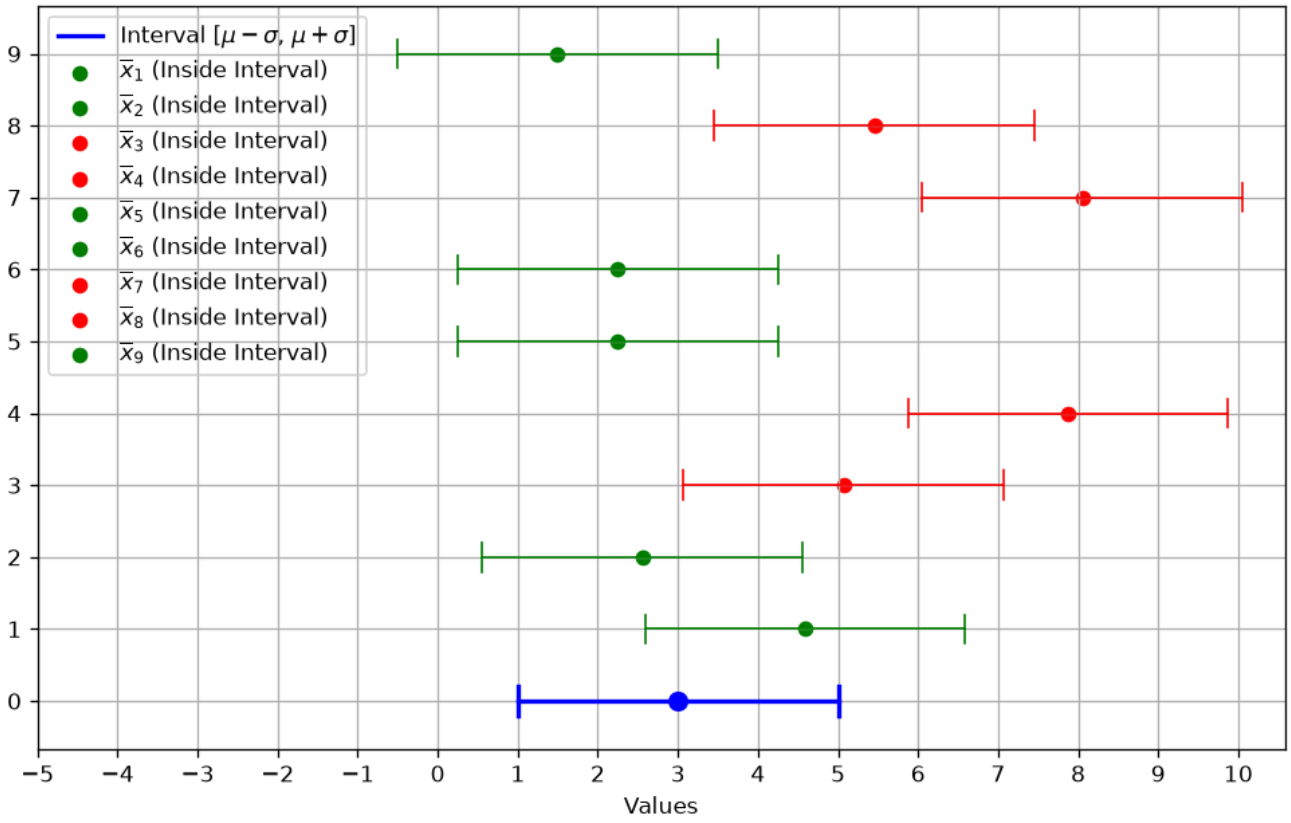
$$P\left(\mu - \sigma \leq \overline{x} \leq \mu + \sigma\right) = 0.683$$

Which means that, if we perform several independent samplings, with sample size n, the probability of obtaining a sample mean $\overline{x}$ in the range $[\mu - \sigma, \mu + \sigma]$ is 68.3%.

It is easy to show that:

$$\overline{x} \in [\mu - \sigma, \mu + \sigma] \Leftrightarrow \mu \in [\overline{x} - \sigma, \overline{x} + \sigma]$$

This is graphically shown in the plot below. The blue segment is the one of bounds $[\mu - \sigma, \mu + \sigma]$. Note that, all times a point $\overline{x}$ happens to be in the blue segment centered around μ, then μ is in the segment centered around $\overline{x}$.



This allows us to write:

$$P\left(\overline{x} - \sigma \leq \mu \leq \overline{x} + \sigma\right) = 0.683$$

which has a powerful interpretation:

If we draw many independent samples of size n and compute $\overline{x}$ from the samples, the true mean μ will lie in the interval $[\overline{x} - \sigma, \overline{x} + \sigma]$ 68.3% of the times

Alternatively

We can say with a confidence of 68.3% that the true mean will be in the $[\overline{x} - \sigma, \overline{x} + \sigma]$ interval.

In this context, $[\overline{x} - \sigma, \overline{x} + \sigma]$ is called a **confidence interval**.

We still have to compute actual numbers for our confidence interval, but we don't have the standard deviation σ. In practice, we replace it with the standard error and obtain the confidence interval:

$$[\overline{x} - SE(\overline{x}), \overline{x} + SE(\overline{x})]$$

We note that 68.3 is a very peculiar number and also a relatively low probability. In general, given a chosen percentage p, that we call a **confidence level**, we can obtain a number β such that:

$$P\left(\overline{x} - \beta\sigma \leq \mu \leq \overline{x} + \beta\sigma\right) = p$$

leading to the following confidence interval:

$$[\overline{x} - \beta SE(\overline{x}), \overline{x} + \beta SE(\overline{x})]$$

Alternatively, we will say that we chose a **significance level** $\alpha = 0.05$, which quantifies the number of times we admin the true mean will be out of the confidence interval.

It is very common to set $\alpha = 0.05$, which leads to $\beta = 1.96$ and the following confidence interval:

$$[\overline{x} - 1.96 \cdot SE(\overline{x}), \overline{x} + 1.96 \cdot SE(\overline{x})]$$

Example: Confidence Interval for Cookie Weights

Let's suppose we want to estimate the **true average weight** μ of cookie packages sold by a bakery. We take a sample of $n = 1000$ packages and compute the sample mean:

$$\overline{x} = 1000.2\text{g}$$

We also compute the sample standard deviation:

$$s_{n-1} = 0.1\text{g}$$

Our estimate $\overline{x}$ is close to the labeled weight of 1000g, but we know it may vary depending on the sample. It is useful to define the **bounds of such variation**, to understand where the true mean μ is

likely to be found relative to our estimate.

To do this, we build a **confidence interval**. We first choose a **confidence level**. For example, if we want to be **95% confident**, we define:

$$\alpha = 0.05$$

This means we accept a 5% chance that the true mean lies outside our interval. Since the sample mean $\overline{X}$ follows a **Gaussian distribution**, we know that **95% of the density** lies within 1.96 standard deviations from the mean. So we write:

$$P\left(\overline{X} - 1.96 \cdot SE(\overline{X}) \leq \mu \leq \overline{X} + 1.96 \cdot SE(\overline{X})\right) = 0.95$$

We compute the standard error:

$$SE(\overline{x}) = \frac{s_{n-1}}{\sqrt{n}} = \frac{0.1}{\sqrt{1000}} = 0.0032\text{g}$$

Then the confidence interval becomes:

$$[1000.2 - 1.96 \cdot 0.0032,\ 1000.2 + 1.96 \cdot 0.0032] = [1000.194,\ 1000.206]$$

This means we are **95% confident** that the true average weight of cookie packages lies between **1000.194g and 1000.206g**.

Since the interval is narrow, we can say that our estimate is precise. If the sample size were smaller, the interval would be wider, reflecting greater uncertainty in the estimate.

Computing Confidence Intervals in Practice

We have seen how to compute confidence interval “by hand” in the case of the estimation of proportions (defect rate).In practice, depending on the quantities for which we want to estimate confidence bounds, we will need to use different distributions. For instance, when estimating means, we will have to use the t-Student distribution with $n - 1$ degrees of freedom. We will not see all methods in detail, but the main libraries implement all confidence bounds estimation procedure for us.

The main estimation procedures are related to:

- Estimation of confidence bounds for means;
- Estimation of confidence bounds for variances;
- Estimation of confidence bounds for proportions.

We will see how to compute these practically in the laboratory sessions.

Confidence Intervals for Means

We can compute confidence intervals for the estimation of means with `scipy`. Let us see how to compute confidence intervals for the mean of `height`:

```
from scipy import stats
mean = data['height'].mean()
std = np.std(data['height'])
standard_error = std/np.sqrt(len(data))

confidence_level = 0.95

interval = stats.norm.interval(confidence_level, loc=mean, scale=standard_error)

print(f"Estimated mean: {mean:0.2f}")
print(f"Standard deviation of the sample: {std:0.2f}")
print(f"Standard error of the sample: {standard_error:0.2f}")
print(f"Confidence interval: [{interval[0]:0.2f}, {interval[1]:0.2f}]")
```

```
Estimated mean: 169.89
Standard deviation of the sample: 9.96
Standard error of the sample: 0.15
Confidence interval: [169.59, 170.19]
```

Let us set the confidence level to 0.99:

```
from scipy import stats
mean = data['height'].mean()
std = np.std(data['height'])
standard_error = std/np.sqrt(len(data))

confidence_level = 0.99

interval = stats.norm.interval(confidence_level, loc=mean, scale=standard_error)

print(f"Estimated mean: {mean:0.2f}")
print(f"Standard deviation of the sample: {std:0.2f}")
print(f"Standard error of the sample: {standard_error:0.2f}")
print(f"Confidence interval: [{interval[0]:0.2f}, {interval[1]:0.2f}]")
```

```
Estimated mean: 169.89
Standard deviation of the sample: 9.96
Standard error of the sample: 0.15
Confidence interval: [169.50, 170.29]
```

Confidence Intervals for Variances

To compute confidence intervals for the estimation of variances, we have to use the χ^2 distribution:

```
import numpy as np
from scipy import stats

# Set the desired confidence level
confidence_level = 0.95  # Change this to the desired confidence level (e.g., 0.95
        for 95% confidence)

# Calculate the confidence interval for the population variance
confidence_interval = stats.chi2.interval(confidence_level, df=len(data) - 1)

# Calculate the sample variance
sample_variance = np.var(data['height'], ddof=1)  # ddof=1 for sample variance

# Calculate the lower and upper bounds of the confidence interval
variance_lower = (len(data) - 1) * sample_variance / confidence_interval[1]
variance_upper = (len(data) - 1) * sample_variance / confidence_interval[0]

print(f"Sample Variance: {sample_variance:.2f}")
print(f"Confidence Interval for Variance: ({variance_lower:.2f},
        {variance_upper:.2f})")
```

```
Sample Variance: 99.13
Confidence Interval for Variance: (95.04, 103.49)
```

Confidence Intervals for Proportion

Let us see how to compute confidence intervals for the proportion of females over males:

```
import statsmodels.api as sm
import numpy as np

females = data['sex'].value_counts()['F']
total = len(data)

# Set the desired confidence level
confidence_level = 0.95  # Change this to the desired confidence level (e.g., 0.95
        for 95% confidence)

# Calculate the proportion (sample proportion)
proportion = females / total

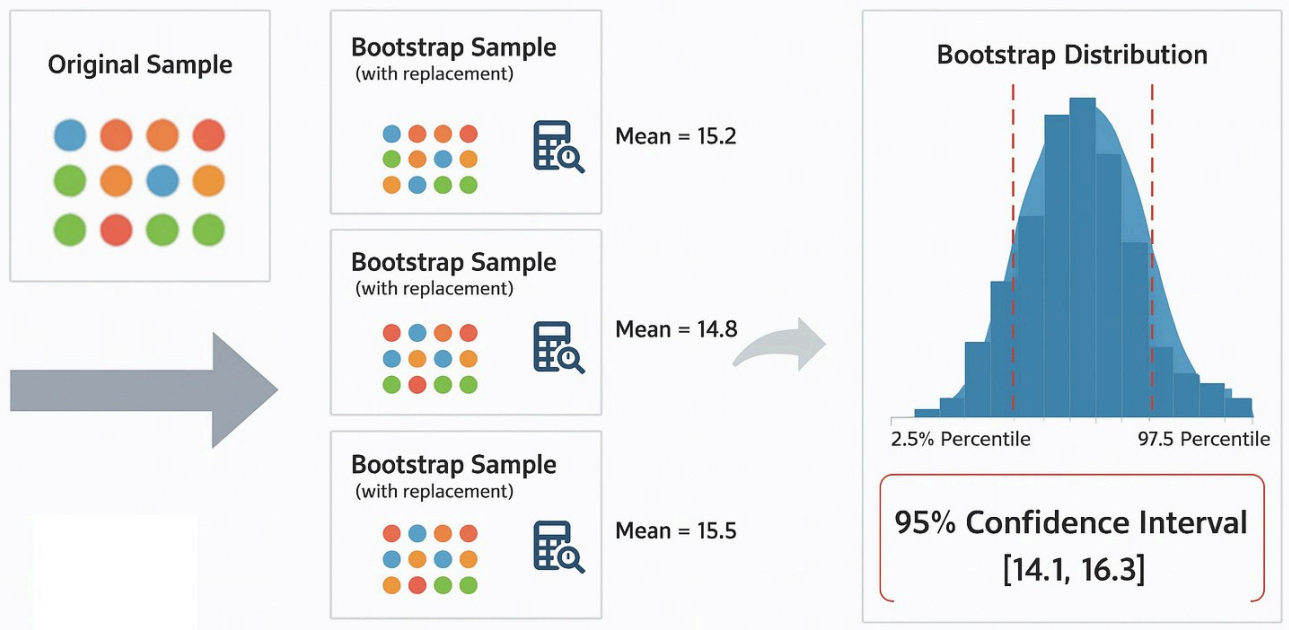
# Compute the confidence interval for the proportion
conf_interval = sm.stats.proportion_confint(females, total, alpha=1 -
        confidence_level, method='normal')

print(f"Sample Proportion: {proportion:.2f}")
print(f"Confidence Interval for Proportion: ({conf_interval[0]:.2f},
        {conf_interval[1]:.2f})")
```

```
Sample Proportion: 0.54
Confidence Interval for Proportion: (0.53, 0.56)
```

Bootstrapping: Inference Through Resampling

So far, we've built confidence intervals using formulas that rely on the Central Limit Theorem and assumptions about the underlying distribution (e.g., normality). But **what if our sample size is small**, the distribution is clearly not normal, or **we want to find a confidence interval for a statistic where no simple formula exists (like the median)**?



This is where **bootstrapping** comes in. It's a powerful computational method that allows us to estimate the sampling distribution of any statistic using only our original sample. The name comes from the phrase "to pull oneself up by one's own bootstraps", reflecting the idea that we use the sample itself to learn about its own uncertainty.

The core idea is to treat the **sample as a stand-in for the population**. We then simulate the process of drawing new samples by resampling *from our own sample*.

The process is straightforward and relies on computation rather than complex formulas:

1. **Start with your original sample** of size n.
2. **Create a "bootstrap sample"** by drawing n observations from your original sample **with replacement**. This new sample will be the same size as the original, but some data points may appear multiple times, while others may not appear at all.
3. **Calculate the statistic of interest** (e.g., mean, median, standard deviation) for this bootstrap sample.
4. **Repeat steps 2 and 3 thousands of times** (e.g., 10,000 times), recording the statistic each time.
5. **Build the bootstrap distribution**. The collection of all the statistics you calculated forms an empirical sampling distribution. We can now use this distribution to estimate properties like the standard error or confidence intervals.

For a 95% confidence interval, we can simply take the 2.5th and 97.5th percentiles of our bootstrap distribution.

Example: Bootstrap Confidence Interval for Mean Height

Let's use bootstrapping to find the 95% confidence interval for the mean height in our `height_weight` dataset.

```
import numpy as np
import pandas as pd
from scipy import stats

# Load the dataset
data = pd.read_csv(DATA + "height_weight.csv")
original_sample = data['height']

# The data must be passed as a sequence (e.g., a tuple)
data_tuple = (original_sample,)

# Use scipy.stats.bootstrap to perform the resampling and calculate the interval
# We pass the data, the statistic to compute (np.mean), and the number of resamples.
bootstrap_result = stats.bootstrap(data_tuple, np.mean, n_resamples=10000,
        confidence_level=0.95)

# Extract the confidence interval from the result object
confidence_interval = bootstrap_result.confidence_interval

print(f"Original sample mean: {original_sample.mean():.2f}")
print(f"SciPy Bootstrap 95% Confidence Interval for the Mean:
        [{confidence_interval.low:.2f}, {confidence_interval.high:.2f}]")
```

```
Original sample mean: 169.89
SciPy Bootstrap 95% Confidence Interval for the Mean: [169.60, 170.19]
```

Bias and Variance of Estimators

Let's return to our bakery example. We want to estimate the **true average weight** μ of cookie packages. To do this, we take a sample of n packages and compute the sample mean:

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

This formula is called an **estimator** of the population mean. The value we obtain from a specific sample is called an **estimate**. Since the sample changes each time we repeat the experiment, the estimate will vary.

It is useful to study how this estimator behaves across different samples—specifically, we want to understand its **bias** and **variance**.

Bias of an Estimator

Let X be a random variable and let $x = (x_1, x_2, \ldots, x_n)$ be a sample from the population. Let $T(X)$ be an estimator of a population quantity ϕ (for instance, the mean weight). Then:

$$T(X) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

is our estimator for ϕ. Since the sample changes, $T(X)$ is a **random variable**.

The **bias** of the estimator is defined as:

$$Bias_\phi(T(X)) = E[T(X)] - \phi$$

This measures whether the estimator systematically overestimates or underestimates the true value. If:

$$E[T(X)] = \phi$$

then the estimator is **unbiased**, meaning that, on average, it gives the correct result.

In our case, the sample mean $\overline{x}$ is an **unbiased estimator** of the population mean. If we repeated the sampling many times and averaged all the sample means, we would get a value close to μ.

Unbiased Estimator for the Variance

To estimate the variance of cookie weights, we may compute:

$$s_n^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

This formula is **biased**—it tends to underestimate the true variance σ^2. Specifically:

$$E[s_n^2] = \frac{n-1}{n}\sigma^2$$

To correct this, we use the **unbiased estimator**:

$$s_{n-1}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

This is the version we use when estimating the standard error and building confidence intervals. For large n, the difference between s_n^2 and s_{n-1}^2 becomes negligible.

Variance of an Estimator

The **variance** of an estimator tells us how much the estimate fluctuates across different samples. It is defined as:

$$Var(T(X)) = E[(T(X) - E[T(X)])^2]$$

A low variance means that repeated samples give similar results. A high variance means that estimates are unstable and vary widely.

Bias-Variance Tradeoff

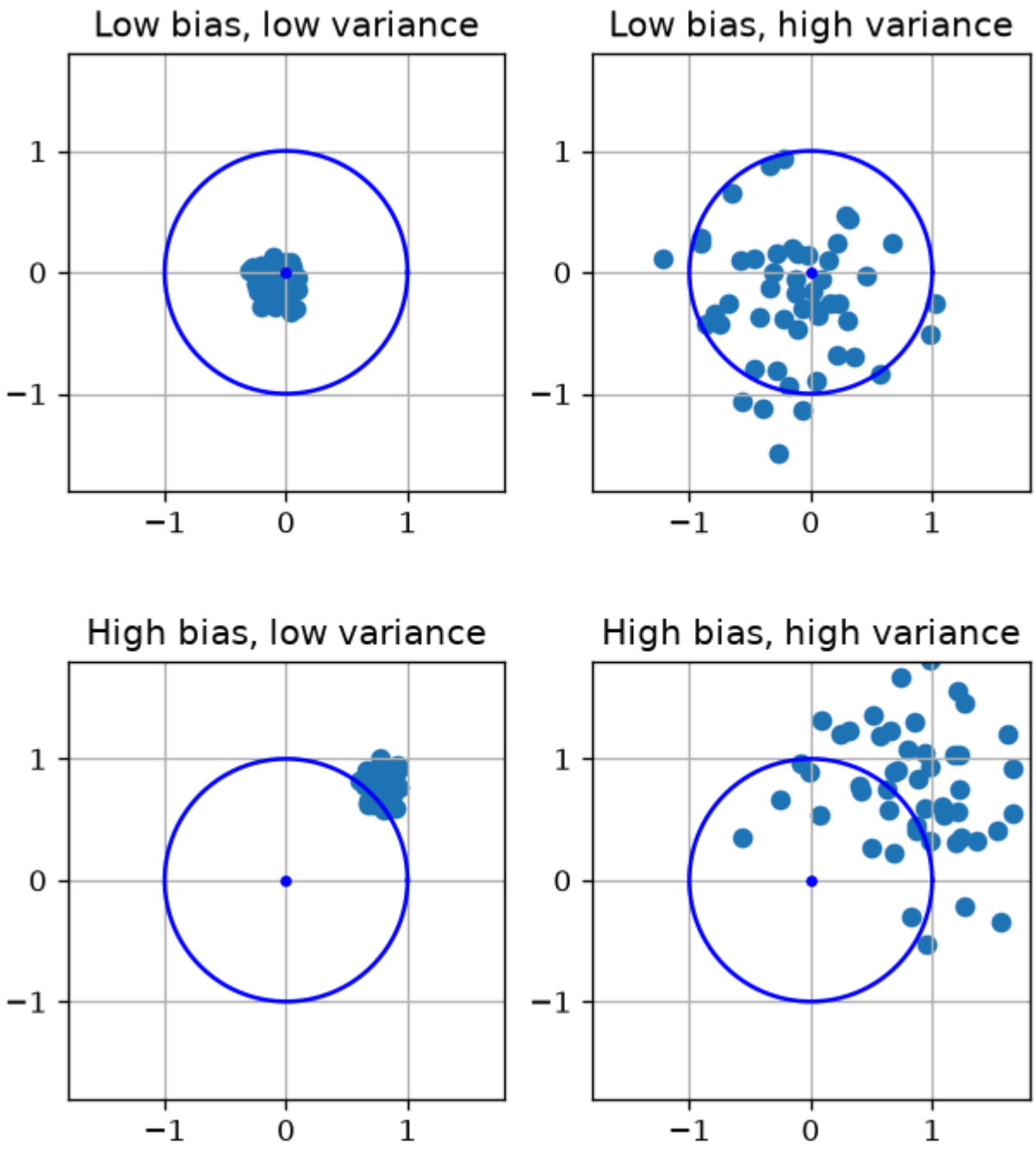
Ideally, we want an estimator with **low bias** and **low variance**. This means that:

- The estimates are close to each other (low variance),
- And they are close to the true value (low bias).

In practice, we can visualize four scenarios:

- **Low bias, low variance**: estimates are tightly clustered around the true value.
- **Low bias, high variance**: estimates are scattered but centered correctly.
- **High bias, low variance**: estimates are consistent but systematically wrong.
- **High bias, high variance**: estimates are scattered and off-target.

This is often illustrated as a target with darts: the true value is the bullseye, and each dart is an estimate. The goal is to hit close to the center, consistently.



The four cases are:

- Low bias, low variance: all estimates will be close to the true value;
- Low bias, high variance: in average, estimates will be close to the true value, but different estimates may greatly differ;
- High bias, low variance: while different estimates will be similar, they all are very far away from the true value;
- High bias, high variance: we don't have many guarantees - estimates will all be different, but also far from the real value, even in average.

It's clear that having low bias and low variance is desirable, but, as we will see, this is not always easy to achieve. In practice, we'll see that in many cases it there is a trade-off between bias and variance, meaning that **we can tweak our estimator to find a balance between these two properties**.

Hypothesis Testing

Confidence intervals provide a range of plausible values for a population parameter based on a sample. A **hypothesis test**, instead, is used to **challenge a specific claim** about that parameter. Examples of hypotheses we might want to test include:

- The average weight of cookie packages is exactly 1000g.
- Two different ovens produce cookies with the same average weight.
- The proportion of underweight packages is below a regulatory threshold.

If the hypothesis is rejected, we conclude that the claim is likely false. Otherwise, we do not have enough evidence to reject it, and we act as if it were true.

Hypothesis Testing for Cookie Weights

Let's return to our bakery. The packaging machine is set to produce cookie packages weighing **exactly 1000g**. We take a sample of $n = 1000$ packages and compute:

$$\overline{x} = 1000.0005\text{g}, \quad s_{n-1} = 0.01\text{g}$$

We assume the small deviation is due to measurement noise or natural variation, and we are inclined to believe that the **true mean is still** $\mu = 1000$**g**.

However, our quality control manager raises a concern: what if the population mean is **not** 1000g? She proposes a formal test to challenge the assumption. We define:

- **Null hypothesis** (H_0): the population mean is $\mu_0 = 1000$ g
- **Alternative hypothesis** (H_a): the population mean is different from 1000g

The null hypothesis is the hypothesis we are trying to reject (what we are trying to prove). If we do so, then we embrace the alternative hypothesis.

Before proceeding, we ask what margin of error is acceptable. The manager says she can tolerate a **5% chance of wrongly rejecting** H_0. This defines our **significance level**:

$$\alpha = 0.05$$

We now ask: **how much does our sample mean** $\overline{x} = 1000.0005g$ **deviate from the assumed mean** $\mu_0 = 1000g$? More precisely, **what is the probability of observing a difference this large (or larger) just by chance?**

To answer this, we compute the **test statistic** using the t-distribution:

$$t = \frac{\overline{x} - \mu_0}{s_{n-1}/\sqrt{n}} = \frac{1000.0005 - 1000}{0.01/\sqrt{1000}} = 1.58$$

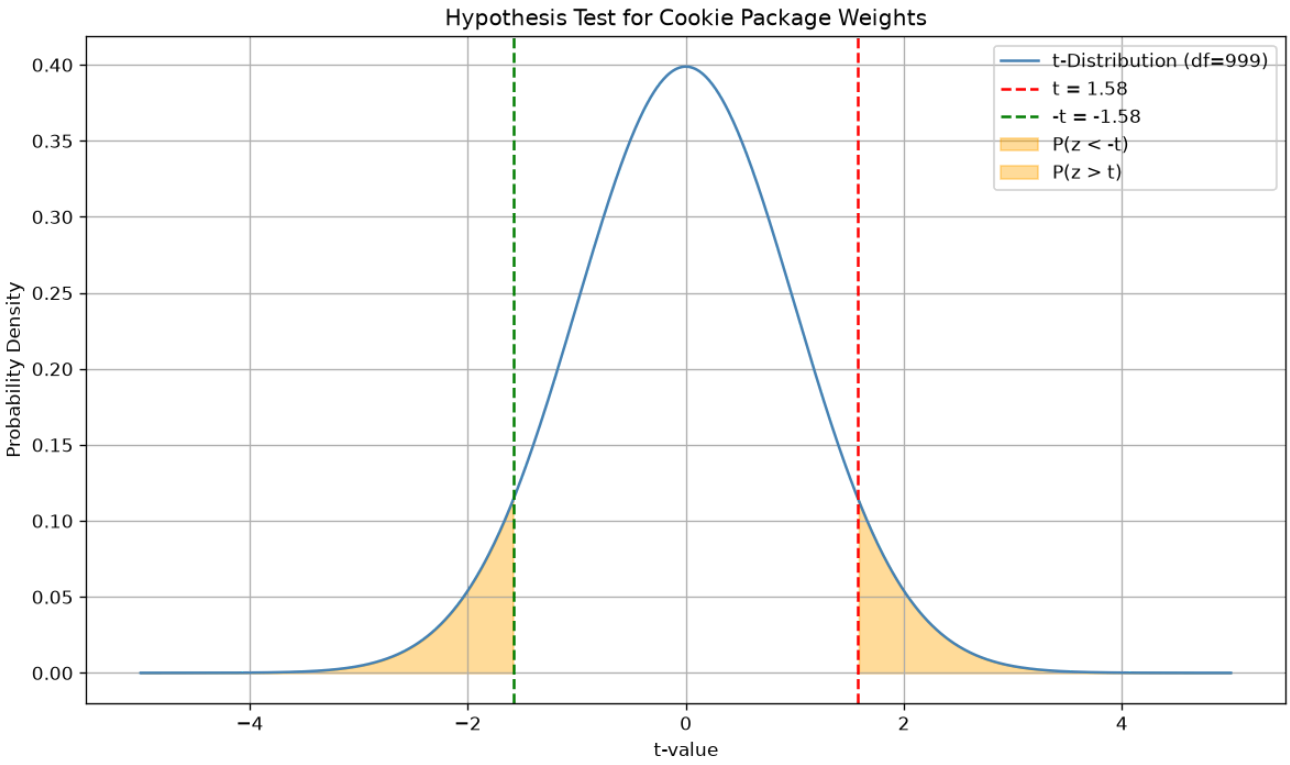
This tells us how many standard errors our estimate is away from the hypothesized mean.

We now ask: **what is the probability of observing a value this extreme or more extreme, assuming H_0 is true?** This is the **p-value**, defined as:

$$P(|z| > |t|)$$

This is the area under the tails of the t-distribution beyond t and $-t$:

```
Test statistic (t): 1.5811
```



Since the t-Student distribution is symmetrical, we can easily compute the p-value as:

$$\text{p-value} = 2(1 - CDF_t(t))$$

In our case, we obtain:

```
t_stat = (x_bar - mu_0) / (s / np.sqrt(n))
print(f"{2*(1-t.cdf(t_stat, df, loc=0, scale=1)):0.2f}")
```

```
0.11
```

This is a large number! How to interpret it?

> If the true mean is $\mu = 1000$ and we repeat sampling many times ($n = 1000$), then 11% of

the times we obtain a deviation more extreme than the observed one.

We now compare this number to the significance level of 5%. If we reject the null hypothesis, we risk to make a mistake 11% of times (i.e., if we reject with this deviation, we should also reject for the other 11% of the times that the deviation is larger and we know that H_0 is true), which is above the threshold of 5%. Hence, **we cannot reject the null hypothesis** under these circumstances.

Does this mean that the two means are the same? **We don't know, the test does not tell us what to do in this case!** But we may try to collect more measurements hoping to reduce uncertainty.

We suspect that our sample is too small, so we collect a total of $n = 5000$ examples and obtains:

$$\overline{x} = 1000.0001$$

$$s_{n-1} = 0.0031$$

The mean has decreased a little and the standard deviation has decreased. We recompute the statistic and obtain:

```
2.28
```

We have a larger (more extreme) statistic. We hence compute our p-value:

```
0.02
```

This p-value is now below the threshold of 5%. We can now reject the null hypothesis and conclude that the population mean is different than $1000g$.

Types of Errors in Statistical Tests

Our statistical test provides evidence to either **reject** the null hypothesis in favor of the alternative or **fail to reject** it. However, since our decision is based on a sample, not the entire population, we can never be 100% certain. This uncertainty leads to the possibility of making two types of errors.

Type I Error (False Positive)

A **Type I Error** occurs when we **incorrectly reject a true null hypothesis**. In other words, we conclude there is an effect or a difference when, in reality, there isn't one.

The probability of committing a Type I error the **significance level alpha (α)**. When we set a significance level of $\alpha = 0.05$, we are accepting a 5% risk of making a Type I error.

Type II Error (False Negative)

A **Type II Error** occurs when we **fail to reject a false null hypothesis**. This means we miss a real effect or difference that actually exists.

Summary of Decisions

We can summarize the relationship between our decision and reality in a table:

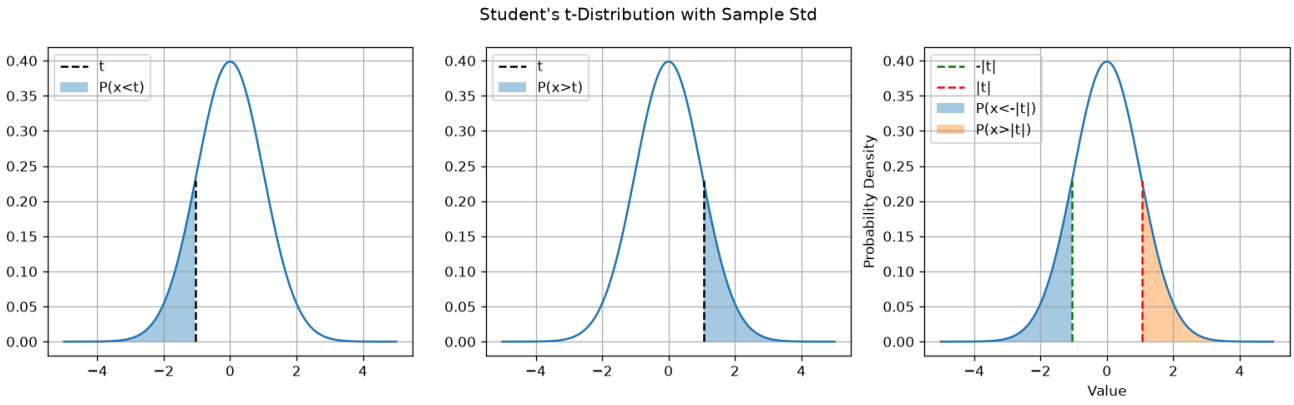
	Null Hypothesis (H_0) is True	**Null Hypothesis (H_0) is False**
Fail to Reject H_0	✅ Correct Decision (True Negative)	❌ **Type II Error** (False Negative)
Reject H_0	❌ **Type I Error** (False Positive)	✅ Correct Decision (True Positive)

One-tailed vs Two-tailed Tests

The tests seen above is a "two-tailed test" in which we summed the areas in the two tails of the distribution. Depending on the form of the alternative hypothesis, in particular:

- If the alternative hypothesis has the form $\mu \neq \mu_0$, then we want to check when the deviation from the assumed mean is larger than the observed one: $P(|x| > |z|)$;
- If the alternative hypothesis has the form $\mu > \mu_0$, then we want to check when the deviation from the assumed mean is positive and larger than the observed one: $P(x > z)$;
- If the alternative hypothesis has the form $\mu < \mu_0$, then we want to check when the deviation from the assumed mean is negative and smaller than the observed one: $P(x < z)$.

This will affect the computation of the p-value as shown in the following figure:



Hypotheses Tests in General

A hypothesis test generally includes:

- H_0: the **null hypothesis**, e.g., the means of two populations are equal;
- H_a: the **alternative hypothesis**, e.g., the means of two populations are not equal (this determines if the test is one- or two-tailed);

- a **test statistics** which quantifies how likely it is to reject the null hypothesis. The test statistics follows a specific distribution which depends on the type of statistical tests we are performing. E.g., it can follow a t-Student distribution;
- a **significance level** α which defines the sensitivity of the test. A common value is $\alpha = 0.05$, which means that we can wrongly reject the null hypothesis 5% of the times when it is in fact true. It represents the degree of error that we are willing to accept when performing hypothesis testing. Common values are $0.1, 0.05, 0.01$;
- the **p-value**: this quantifies the probability of sampling a test statistics at least as extreme as the one observed under the null hypothesis. In practice, the p-value measures the probability that the null hypothesis is true but we are observing test statistic leading to rejection nevertheless.

The null hypothesis is rejected if the **p-value is smaller than the chosen significance level** α. We will not see in details all the possible hypothesis tests, but all of them follow a similar scheme.

Other Important Tests

In this section, we briefly see the main statistical tests which can be used in practice, besides the one for means. We will not see how they are formulated, but we will see how to interpret them. We will see a few other tests when we'll talk about linear regression.

One Sample T-Test

This is the test for sample means we have previously seen. It is used to **determine whether the mean of a single sample is significantly different from a known or hypothesized value.**

This test allows to assess whether a sample has a given mean. Let us check if the average height in our dataset is equal to $170cm$:

```python
import numpy as np
from scipy import stats

# Define the null hypothesis mean (population mean you want to test against)
null_hypothesis_mean = 170

# Perform a one-sample t-test
t_stat, p_value = stats.ttest_1samp(data['height'], null_hypothesis_mean)

# Set the desired significance level (alpha)
alpha = 0.05

# Check the p-value against the significance level
if p_value < alpha:
    print(f"Reject the null hypothesis. The data provides enough evidence to conclude
        that the population mean is different from {null_hypothesis_mean}.")
else:
    print(f"Fail to reject the null hypothesis. The data does not provide enough
        evidence to conclude that the population mean is different from
        {null_hypothesis_mean}.")

print(f"t-statistic: {t_stat:.2f}")
print(f"P-value: {p_value:.4f}")
```

```
Fail to reject the null hypothesis. The data does not provide enough evidence to
conclude that the population mean is different from 170.
t-statistic: -0.69
P-value: 0.4872
```

Let us now check if the average height is 170 among males only:

```python
import numpy as np
from scipy import stats

# Define the null hypothesis mean (population mean you want to test against)
null_hypothesis_mean = 170

data2 = data[data['sex']=='M']

# Perform a one-sample t-test
t_stat, p_value = stats.ttest_1samp(data2['height'], null_hypothesis_mean)

# Set the desired significance level (alpha)
alpha = 0.05

# Check the p-value against the significance level
if p_value < alpha:
    print(f"Reject the null hypothesis. The data provides enough evidence to conclude
        that the population mean is different from {null_hypothesis_mean}.")
else:
    print(f"Fail to reject the null hypothesis. The data does not provide enough
        evidence to conclude that the population mean is different from
        {null_hypothesis_mean}.")

print(f"t-statistic: {t_stat:.2f}")
print(f"P-value: {p_value:.4f}")
```

```
Reject the null hypothesis. The data provides enough evidence to conclude that the
population mean is different from 170.
t-statistic: 45.52
P-value: 0.0000
```

Two Sample T-Test

A two-sample t-test is used to determine if there is a significant difference between the means of two independent samples. It's often used when you want to compare the means of two different populations or treatments. The test assesses whether the difference between the sample means is statistically significant or if it could have occurred due to random chance. Also in this case, the test statistic will follow a t-Student distribution.

Let us now compare the average heights in two samples we previously obtained via random sampling. We expect no differences:

```
import scipy.stats as stats

h1 = sample1['height']
h2 = sample2['height']

# Perform independent two-sample t-test
t_stat, p_value = stats.ttest_ind(h1, h2)

# Define significance level (alpha)
alpha = 0.05

print(f"Test statistic: {t_stat:0.2f}")
print(f"Significance level: {alpha:0.2f}")
print(f"P-value: {p_value:0.2f}")

# Comment on result
if p_value < alpha:
    print("Conclusion: there is a significant difference between the mean exam scores
        of the two classes.")
else:
    print("Conclusion: there is no significant difference between the mean exam
        scores of the two classes.")
```

```
Test statistic: -0.47
Significance level: 0.05
P-value: 0.64
Conclusion: there is no significant difference between the mean exam scores of the
two classes.
```

```
import pandas as pd
hw=pd.read_csv(DATA + "height_weight.csv")
hw['height'] = (hw['height']/2.54).astype(int)
hw['weight'] = hw['weight']/2.205
hw
```

	sex	BMI	height	weight
0	M	33.36	74	53.484771
1	M	26.54	70	38.056472
2	F	32.13	61	34.970812
3	M	26.62	68	35.999365
4	F	27.13	66	34.559390
...	...	...	...	...
4226	F	17.12	69	23.862436
4227	M	27.47	69	38.262182
4228	F	29.16	64	34.970812
4229	F	23.68	64	28.388071
4230	F	20.12	61	22.628172

4231 rows × 4 columns

If we compute the average `BMI` for males and females we obtain:

```
hw.groupby('sex')['BMI'].mean()
```

```
sex
F    26.929287
M    27.684959
Name: BMI, dtype: float64
```

We see a small difference. Is this due to chance or is it significant? If we run a two-sample t-test, we obtain the following results:

```
import scipy.stats as stats

male_bmi = hw[hw['sex']=='M']['BMI']
female_bmi = hw[hw['sex']=='F']['BMI']

# Perform independent two-sample t-test
t_stat, p_value = stats.ttest_ind(male_bmi, female_bmi)

# Define significance level (alpha)
alpha = 0.05

print(f"Test statistic: {t_stat:0.2f}")
print(f"Significance level: {alpha:0.2f}")
print(f"P-value: {p_value:0.2f}")

# Comment on result
if p_value < alpha:
    print("Conclusion: there is a significant difference between the two means.")
else:
    print("Conclusion: there is no significant difference between the two means.")
```

```
Test statistic: 4.64
Significance level: 0.05
P-value: 0.00
Conclusion: there is a significant difference between the two means.
```

Chi-Square Test for Independence

The Chi-Square Test for Independence is a statistical test used to determine whether there is an association or independence between two or more categorical variables. This test is particularly useful when we want to assess whether changes in one categorical variable are related to changes in another categorical variable. The typical scenario is to set up a contingency table to compare the observed frequencies (counts) of the joint categories of the two variables to the expected frequencies that would occur under the assumption of independence.

The null hypothesis for the Chi-Square Test for Independence is that **there is no association between the two categorical variables** (they are independent), while the alternative hypothesis suggests that there is an association (they are dependent).

The test statistics follows a Chi-square distribution (we won't see the details) in this case.

Let's consider the Titanic dataset:

```
import pandas as pd
titanic = pd.read_csv(DATA + "titanic.csv",
                      index_col='PassengerId')
titanic
```

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId											
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...
887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

891 rows × 11 columns

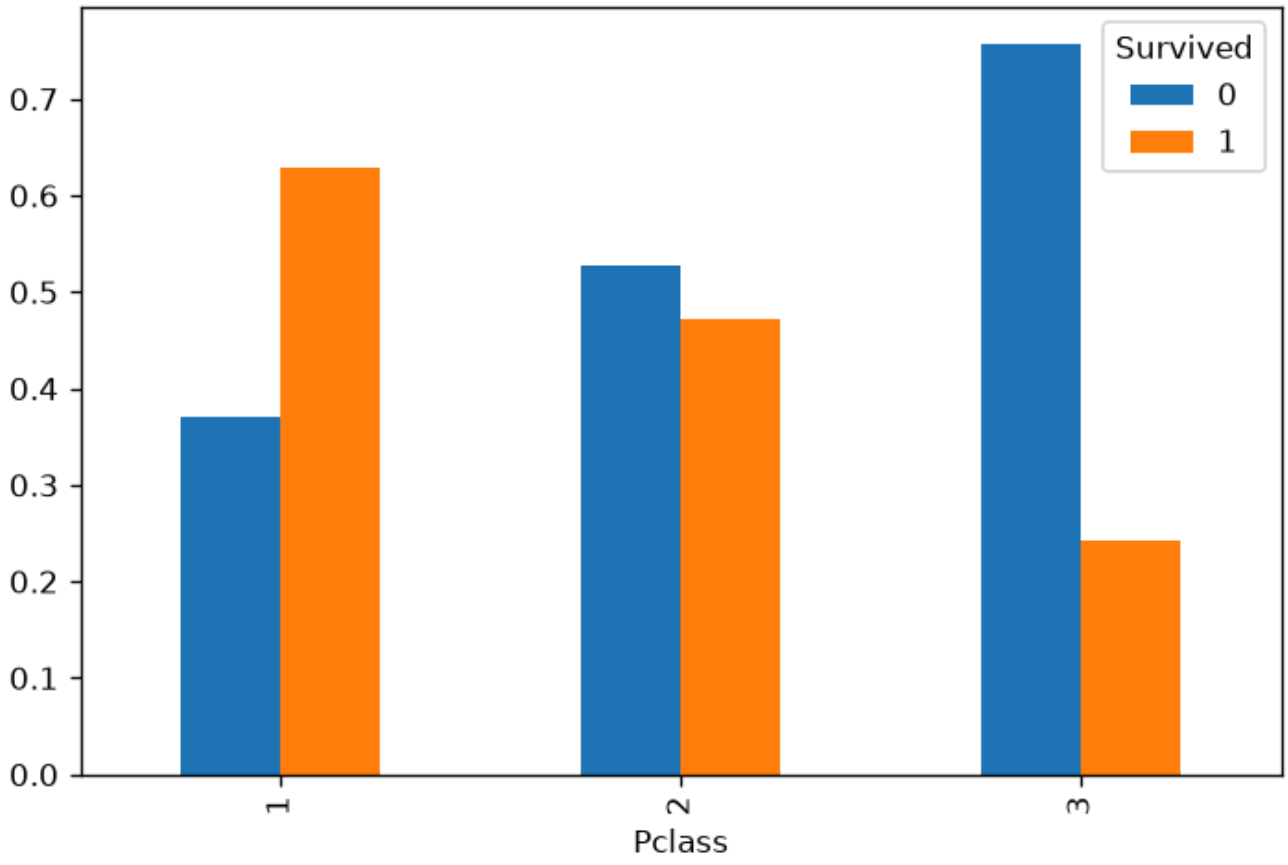
Let's consider the following contingency table:

```
contingency_table = pd.crosstab(titanic['Pclass'], titanic['Survived'])
contingency_table
```

Survived	0	1
Pclass		
1	80	136
2	97	87
3	372	119

We can visualize the distributions of Survived in the three classes for more clarity:

```
from matplotlib import pyplot as plt
pd.crosstab(titanic['Pclass'], titanic['Survived'], normalize=0).plot.bar()
plt.show()
```



We expect some form of correlation between the two variables. Indeed, the chi-square statistics and Cramer V statistics are:

```
from scipy.stats import chi2_contingency
from scipy.stats.contingency import association

print(f"Chi-square statistic: {chi2_contingency(contingency_table).statistic:0.2f}")
print(f"Cramer V statistic: {association(contingency_table):0.2f}")
```

```
Chi-square statistic: 102.89
Cramer V statistic: 0.34
```

We have numbers different from zero, but **is this due to chance or is it statistically significant**? If we run a chi-square contingency test:

```
import seaborn as sns
import pandas as pd
from scipy.stats import chi2_contingency

# Perform the Chi-Square Test for Independence
chi2, p, _, _ = chi2_contingency(contingency_table)

# Set the significance level (alpha)
alpha = 0.05

# Print the results
print("Chi-Square Statistic:", chi2)
print("p-value:", p)

# Interpret the results
if p < alpha:
    print("\nThere is a significant association between 'Pclass' and 'survived'.")
else:
    print("\nThere is no significant association between 'Pclass' and 'survived'.")
```

```
Chi-Square Statistic: 102.88898875696056
p-value: 4.549251711298793e-23

There is a significant association between 'Pclass' and 'survived'.
```

Chi-Square Goodness-of-Fit Test

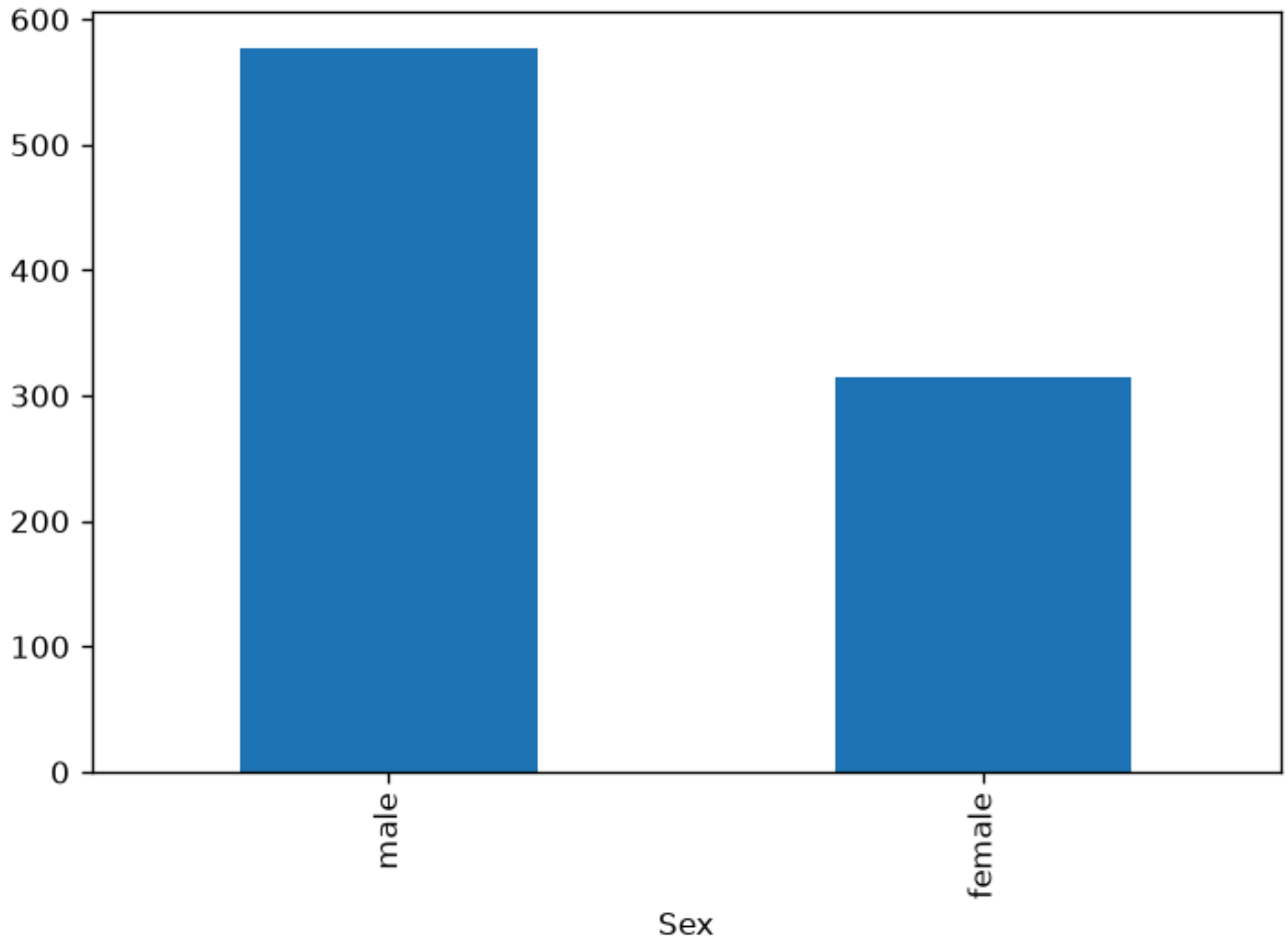
The Chi-Square Goodness of Fit test is a statistical test used to determine whether observed categorical data (frequencies) fit a specified distribution or expected frequencies. This test is often used to assess whether the observed data deviates significantly from a hypothesized distribution. The typical scenario is to compare observed frequencies with expected frequencies based on a theoretical model or prior knowledge.

The null hypothesis for the Chi-Square Goodness of Fit test is that **there is no significant difference between the observed and expected frequencies**, meaning the observed data fits the specified distribution. The alternative hypothesis suggests that there is a significant difference.

The test statistics follows a Chi-square distribution in this case.

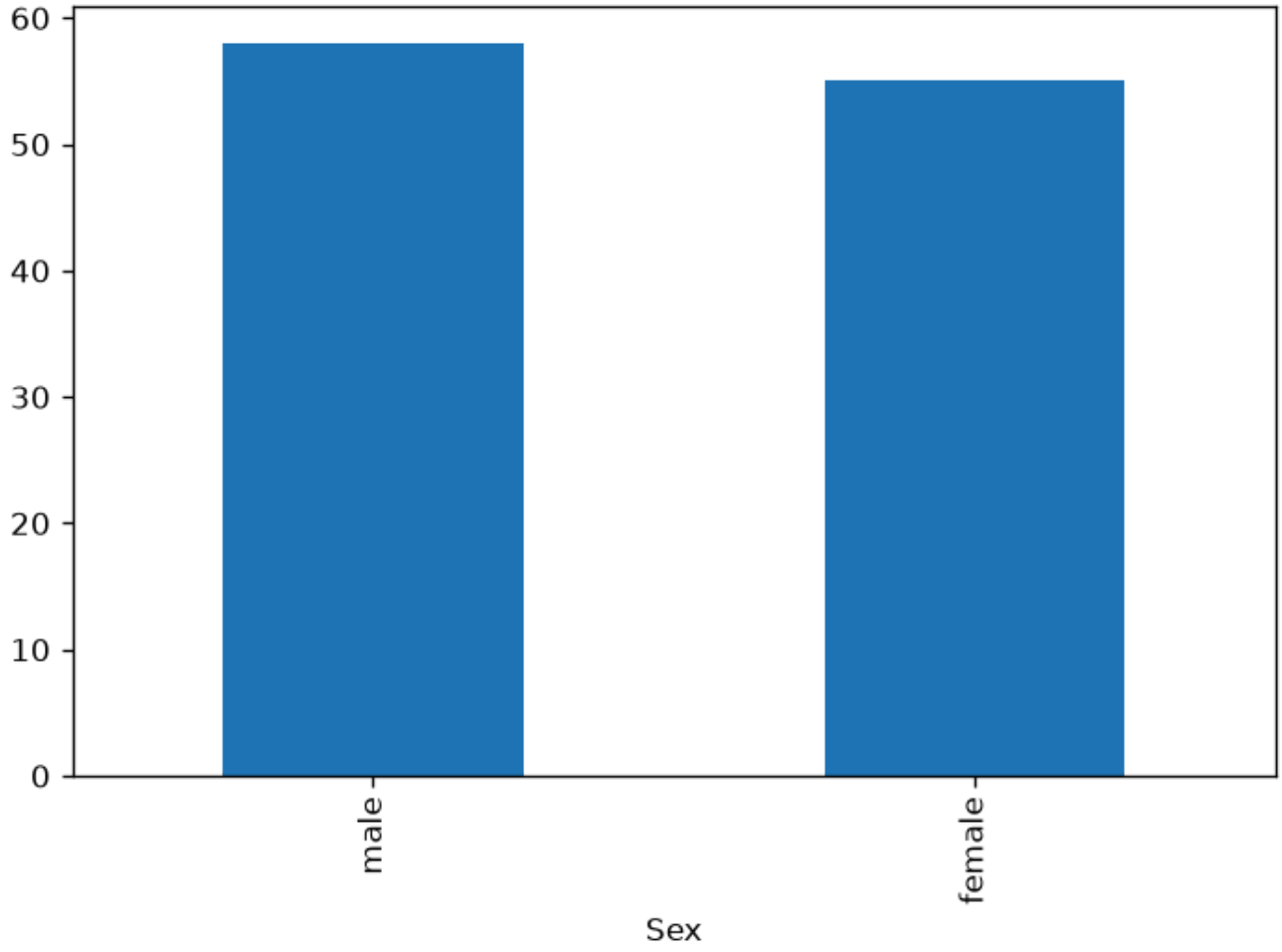
Let us consider the Titanic dataset again. We know that the distribution of `Sex` among passengers is biased:

```
titanic['Sex'].value_counts().plot.bar()
plt.show()
```



We now consider the distribution of `Sex` among passengers less than 18 years old:

```
minor=titanic[titanic['Age']<18]
observed_frequencies = minor['Sex'].value_counts()
observed_frequencies.plot.bar()
plt.show()
observed_frequencies
```




```
Sex
male      58
female    55
Name: count, dtype: int64
```

This looks less biased, but there are still minor differences between the counts. Are these due to chance? If `Sex` was distributed uniformly (as we hypothesize), we would have the following frequencies:

```
expected_frequencies = [113/2, 113/2]
expected_frequencies
```

```
[56.5, 56.5]
```

We can run a Goodness-of-fit test to check if the observed frequencies match the expected ones:

```
import seaborn as sns
import pandas as pd
from scipy.stats import chisquare

# Load the Titanic dataset
titanic_data = sns.load_dataset("titanic")

observed_frequencies = minor['Sex'].value_counts().values

# Define expected frequencies that closely match the observed data
expected_frequencies = [113/2, 113/2]

chi2, p_value = chisquare(f_obs=observed_frequencies, f_exp=expected_frequencies)

# Set the significance level (alpha)
alpha = 0.05

# Print the results
print("Observed Frequencies:")
print(observed_frequencies)
print("\nExpected Frequencies:")
print(expected_frequencies)
print("\nChi-Square Statistic:", chi2)
print("p-value:", p_value)

# Interpret the results
if p_value < alpha:
    print("\nThe observed data significantly deviates from the expected
        distribution.")
else:
    print("\nThe observed data fits the expected distribution.")
```

```
Observed Frequencies:
[58 55]

Expected Frequencies:
[56.5, 56.5]

Chi-Square Statistic: 0.07964601769911504
p-value: 0.7777776907897473

The observed data fits the expected distribution.
```

Given the large p-value, we could not reject the null hypothesis that there are significant differences between expected and observed frequencies.

Pearson/Spearman Correlation Test

We have seen how to compute Pearson/Spearman correlation coefficient. However, what can we say when we get small values? Are those supposed to be zero, but we got something different from zero due to sampling, or are they significantly different from zero?

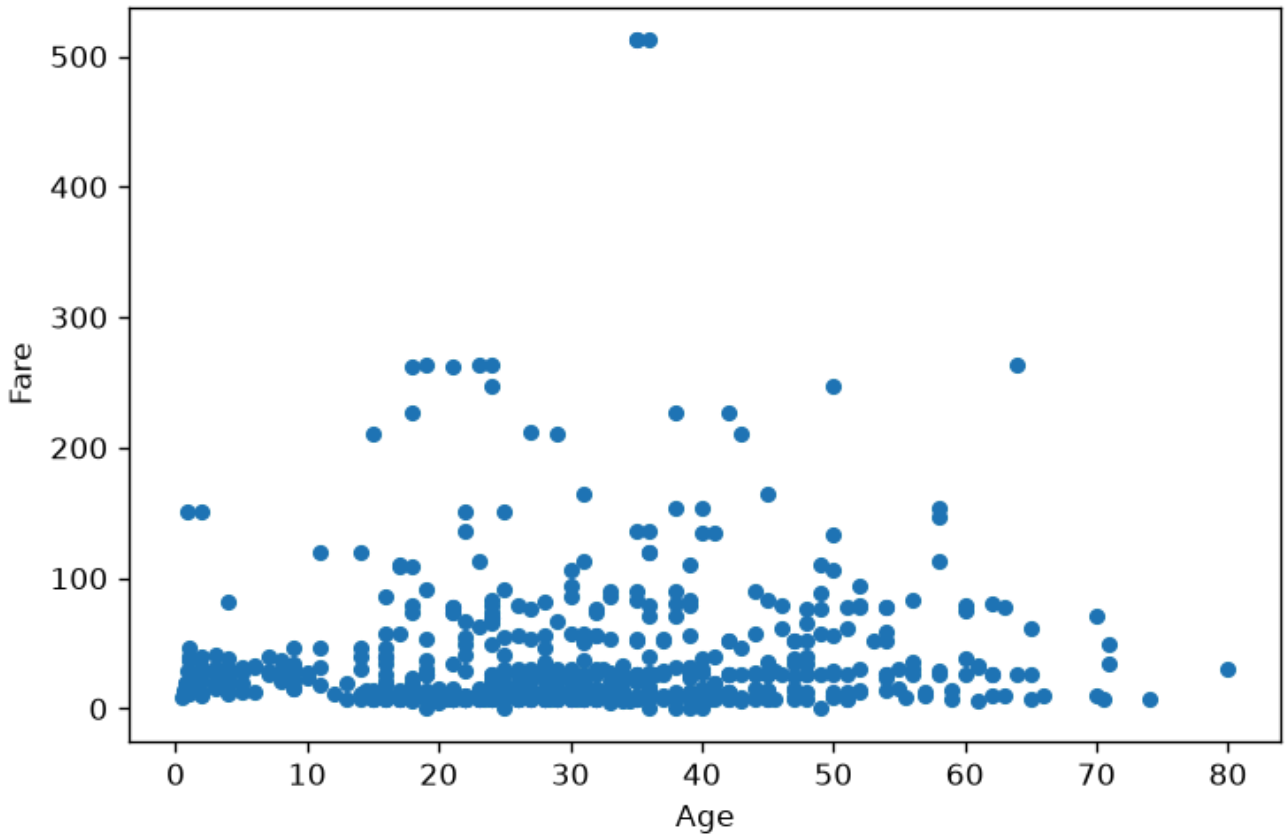
The statistical tests associated with the correlation coefficients are used to determine whether the observed correlation between two variables is statistically significant or if it might have occurred due to random chance. This test assesses whether the correlation in the sample data is likely to reflect a true correlation in the population.

The null hypothesis is that **there is no statistically significant correlation between the two variables in the population**. In other words, the true correlation coefficient in the population is zero.

Let us consider the Titanic dataset. We find the following correlation between the `Age` and `Fare` variables:

```
titanic.plot.scatter(x='Age',y='Fare')
tt=titanic.dropna()
print(f"Correlation between age and fare:
        {titanic[['Age','Fare']].dropna().corr().values[0][1]:.2f}")
```

```
Correlation between age and fare: 0.10
```



Is this small positive correlation “true” or due to chance? Let us run a statistical test:

```
import seaborn as sns
import pandas as pd
import scipy.stats as stats

tt = titanic[['Age','Fare']].dropna()

# Extract the 'age' and 'fare' columns
age = tt['Age']
fare = tt['Fare']

# Perform a Pearson correlation test
r, p_value = stats.pearsonr(age, fare)

# Define the significance level
alpha = 0.05

# Display the results
print(f"Pearson correlation coefficient (r): {r}")
print(f"P-value: {p_value}")

# Compare the p-value to alpha
if p_value ⩽ alpha:
    print("Reject the null hypothesis. There is a significant correlation between age
        and fare.")
else:
    print("Fail to reject the null hypothesis. There is no significant correlation
        between age and fare.")
```

```
Pearson correlation coefficient (r): 0.0960666917690389
P-value: 0.010216277504447095
Reject the null hypothesis. There is a significant correlation between age and
fare.
```

The p-value is small enough to reject the null hypothesis: the correlation is small but statistically significant.

Similar tests exist for Spearman coefficient:

```
import seaborn as sns
import pandas as pd
import scipy.stats as stats

tt = titanic[['Age','Fare']].dropna()

# Extract the 'age' and 'fare' columns
age = tt['Age']
fare = tt['Fare']

# Perform a Pearson correlation test
r, p_value = stats.spearmanr(age, fare)

# Define the significance level
alpha = 0.05

# Display the results
print(f"Spearman correlation coefficient (r): {r}")
print(f"P-value: {p_value}")

# Compare the p-value to alpha
if p_value ⩽ alpha:
    print("Reject the null hypothesis. There is a significant correlation between age
        and fare.")
else:
    print("Fail to reject the null hypothesis. There is no significant correlation
        between age and fare.")
```

```
Spearman correlation coefficient (r): 0.1350512177342878
P-value: 0.00029580903243060916
Reject the null hypothesis. There is a significant correlation between age and
fare.
```

Assessing whether a Sample is Normally Distributed

While the Normal distribution is pervasive, in some cases, it is useful to assess whether a given sample follows a normal distribution before assuming this is true. Let us consider the dataset of heights and weights and plot the distribution of weights:

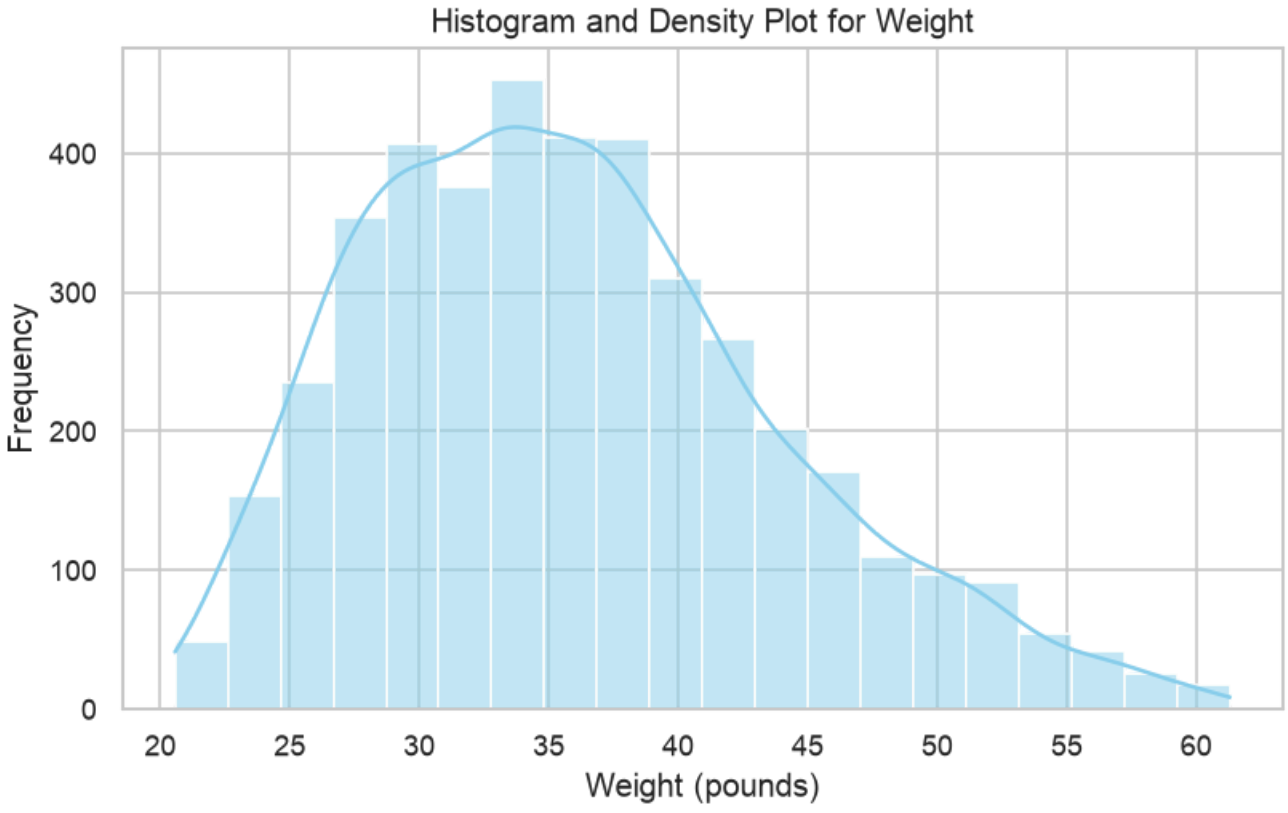
```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Load the dataset
hw = pd.read_csv(DATA + "height_weight.csv")
hw['height'] = (hw['height'] / 2.54).astype(int)  # Convert height to inches
hw['weight'] = hw['weight'] / 2.205               # Convert weight to pounds

# Set up the plotting style
# `sns.set` rewrites the rcParams, including the transparent background the site
# uses (see site/notes/matplotlibrc), so it is asked to leave that alone. The
# whitegrid grid lines are grey, not white, so they survive the change.
sns.set(style="whitegrid",
        rc={"figure.facecolor": "none", "axes.facecolor": "none",
            "savefig.facecolor": "none"})

# Plotting for weight
sns.histplot(hw['weight'], kde=True, color="skyblue", bins=20)
plt.title('Histogram and Density Plot for Weight')
plt.xlabel('Weight (pounds)')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```



Does the distribution look Gaussian? Let us compute Skewness and Kurtosis:

```python
import pandas as pd
from scipy.stats import skew, kurtosis

# Load and preprocess the data
hw = pd.read_csv(DATA + "height_weight_pounds.csv")

# Calculate skewness and kurtosis for weight
weight_skewness = skew(hw['weight'])
weight_kurtosis = kurtosis(hw['weight'])

print(f"Skewness of weight: {weight_skewness:.2f}")
print(f"Kurtosis of weight: {weight_kurtosis:.2f}")
```

```
Skewness of weight: 0.57
Kurtosis of weight: -0.06
```

We not that:

- We have a positive Skewness: this indicates that the distribution is skewed towards the right side (the right tail is longer) as compared to a Normal distribution;
- We have a Kurtosis slightly lower than zero: the distribution is slightly “flatter” than a Normal distribution.

While Skewness and Kurtosis can help characterize deviations from normality, there are tests which can be used.

Quantile-Quantile Plots (Q-Q Plots)

Quantile-Quantile plots, or Q-Q plots, are a powerful tool to **assess how well a sample matches a theoretical distribution**. The idea is simple: we compare the **quantiles of the empirical data** with the **quantiles of a reference distribution**—typically the normal distribution.

To make the comparison meaningful and remove scale effects, the empirical data is often **standardized into z-scores**. If the sample truly follows the theoretical distribution, the quantiles will align, and the points will fall along the diagonal line $y = x$.

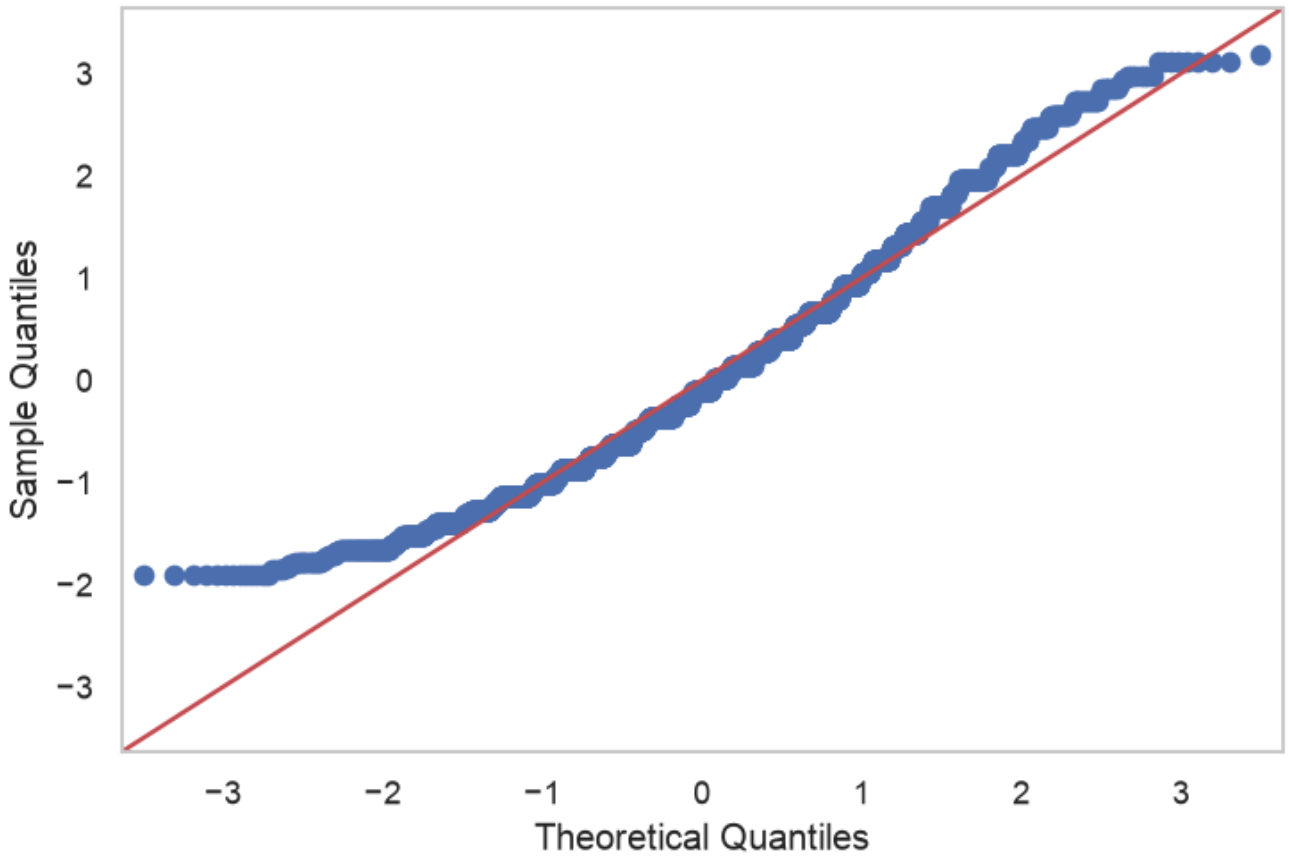
Deviations from this line reveal **systematic differences** between the empirical and theoretical distributions. For example:

- A curve that bends away from the diagonal suggests skewness.
- S-shaped patterns may indicate heavier or lighter tails than expected.

In the following graph, we show the Q-Q plot for the sample of cookie package weights. It helps us visually evaluate whether the sample distribution is approximately normal—a key assumption in many statistical methods.

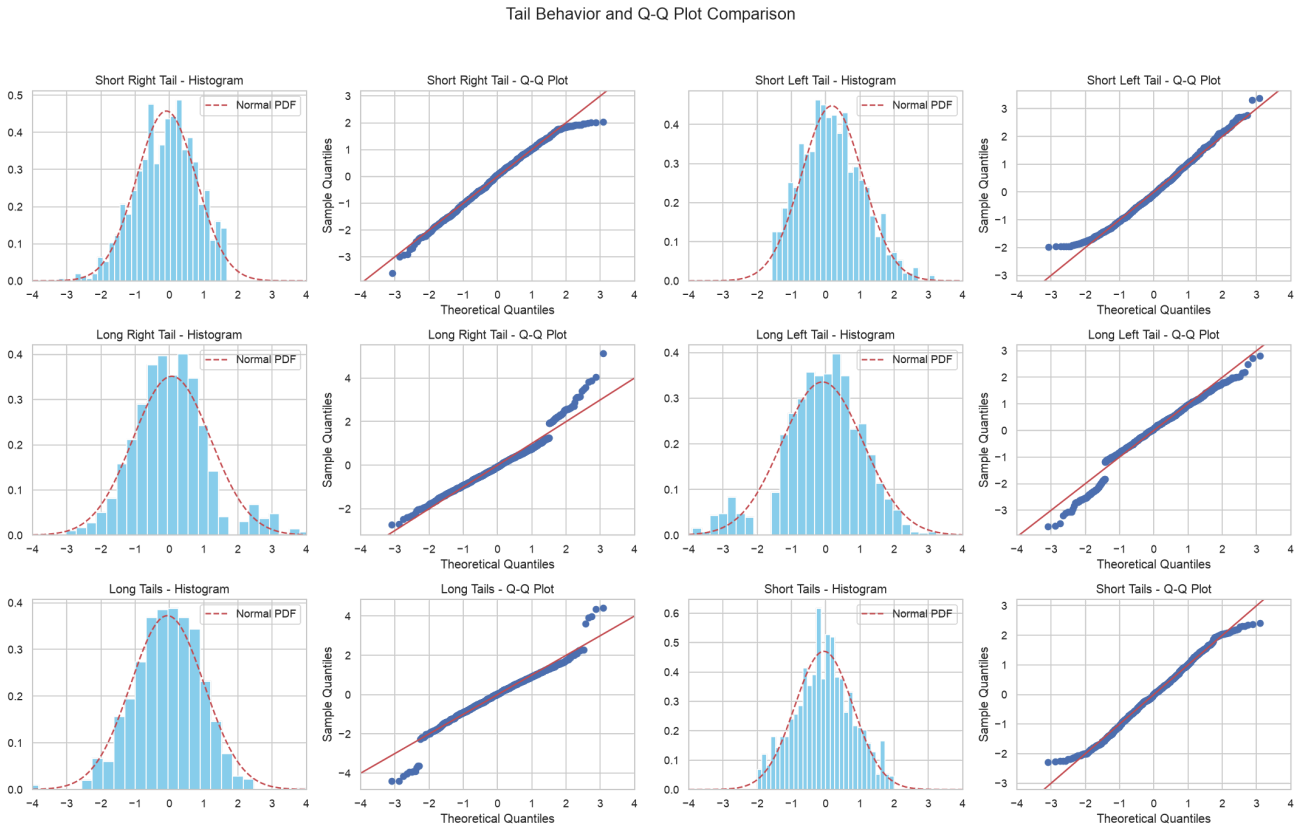
```
from statsmodels.graphics.gofplots import qqplot
import pandas as pd
from matplotlib import pyplot as plt

# fit=True means the distribution should be fitted to the data
# and the data should be standardized
# line='45' means a 45-degree reference line will be plotted
qqplot(hw['weight'], fit=True, line='45')
plt.grid()
plt.show()
```



The plot relates the “theoretical” quantiles with those of the sample. The fact that the points on the plot do not lie on the diagonal indicates that there is a discrepancy between the empirical data distribution and the Gaussian distribution.

Analyzing a Q-Q Plot can be complex. In practice, **there are some guidelines to understand how a sample deviates from a theoretical distribution**. The following figure compares the q-q plots of different distributions:



Every time we observe a Q-Q Plot, we can relate these features to characteristics of the distribution. Clearly, these characteristics can also combine to create more complex Q-Q Plots, as seen in the case of weights.

Shapiro-Wilk Normality Test

The Shapiro-Wilk test is a statistical test used to assess whether a sample follows a Gaussian (normal) distribution. It is used with small samples ($n \leq 2000$) It works by comparing the observed data to what you would expect if the data were drawn from a truly Gaussian distribution.

The null hypothesis for this test is that the population is normally distributed.

We will not see the formal details of this test, but we can use it in our analyses. Here is the result on the `weight` sample in our height-weight dataset:

```python
from scipy.stats import shapiro
import pandas as pd

# Perform the Shapiro-Wilk test
statistic, p_value = shapiro(hw['weight'])

# Set the significance level (alpha)
alpha = 0.05

print(f"Test statistic: {statistic:0.2f}")
print(f"P-value: {p_value:0.2f}")

# Check the p-value against the significance level
if p_value > alpha:
    print("Sample looks Gaussian (fail to reject H0)")
else:
    print("Sample does not look Gaussian (reject H0)")
```

```
Test statistic: 0.97
P-value: 0.00
Sample does not look Gaussian (reject H0)
```

D'Agostino's K-squared test

When samples are large ($n \geq 50$), the D'Agostino's K-squared test is more used. It is based on Skewness and Kurtosis.

The null hypothesis for this test is that the population is normally distributed.

Here is the result for our example:

```python
import pandas as pd
from scipy.stats import normaltest

# Perform D'Agostino's K-squared test
statistic, p_value = normaltest(hw['weight'])

# Set the significance level (alpha)
alpha = 0.05

print(f"Test statistic: {statistic:.2f}")
print(f"P-value: {p_value:.2f}")

# Check the p-value against the significance level
if p_value > alpha:
    print("Sample looks Gaussian (fail to reject H0)")
else:
    print("Sample does not look Gaussian (reject H0)")
```

```
Test statistic: 201.64
P-value: 0.00
Sample does not look Gaussian (reject H0)
```

Exercises

Exercise 1

Consider the sample of all passengers in the titanic dataset. Compute the mean age of all passengers. Then, compute the confidence bounds for this mean setting the confidence level to 0.95. Is the estimated mean a reliable estimate of the population mean?

Exercise 2

Consider the samples of all passengers in the titanic dataset. Compute the variance of the ages of all passengers. Then, compute the confidence bounds for this variance setting the confidence level to 0.95. Is the estimated variance a reliable estimate of the population variance?

Exercise 3

Extract the sample of values of Sex of passengers in second class in the titanic dataset. Compute the contingency table of absolute counts of the Sex column. Compute the confidence bounds for the proportions of Sex in the sample. Are the estimated bounds reliable? Repeat the analysis for class 1 and 3.

Exercise 4

The average age of women in the US population is 40. Run a statistical test to assess whether the sample of ages in the `infert` dataset has the same mean as the US population. What is the result of the test? Are women in `infert` younger or older than the average?

You can load the dataset installing the `pydataset` library with `pip install pydataset` and then:

```
from pydataset import data; infert = data("infert")
```

Exercise 5

Consider the `titanic` dataset. Follow these steps to assess that males and females **are not equally distributed**:

- Obtain a table of absolute frequencies of the values of `Sex` ;
- Show a barplot comparing the proportions of male and female passengers. Do these look equally distributed?
- Run a statistical test to assess whether the proportion of males and females are equal ($p = 0.5$). What is the result of the test?
- Run the statistical test setting $p = 1/3$. Is the result different? Can we reject the null hypothesis? Why?

References

- Chapters 6-8 of [1];
- Parts of chapter 9 of [2].

[1] Gonick, L., & Smith, W. (1993). The cartoon guide to statistics. HarperCollins Publishers, Inc.

[2] Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Linear regression

Chapter resources

Slides (PDF) Open in Colab Practice quiz

Linear regression is a fundamental and widely used statistical technique in data analysis and machine learning. It is a powerful tool for **modeling and understanding the relationships between variables** and **make predictions on unseen data**. At its core, linear regression aims to model a linear relationship between a **dependent variable** (the one you want to predict) and **one or more independent variables** (the ones used for prediction). This technique allows us to make predictions, infer associations, and gain insights into how changes in independent variables influence the target variable. Linear regression is both intuitive and versatile, making it a valuable tool for tasks ranging from simple trend analysis to more complex predictive modeling and hypothesis testing.

In this context, we will explore the concepts and applications of linear regression, its different types, and how to implement it using Python.

The Auto MPG Dataset

We will consider the Auto MPG dataset, which contains 398 measurements of 8 different properties of different cars:

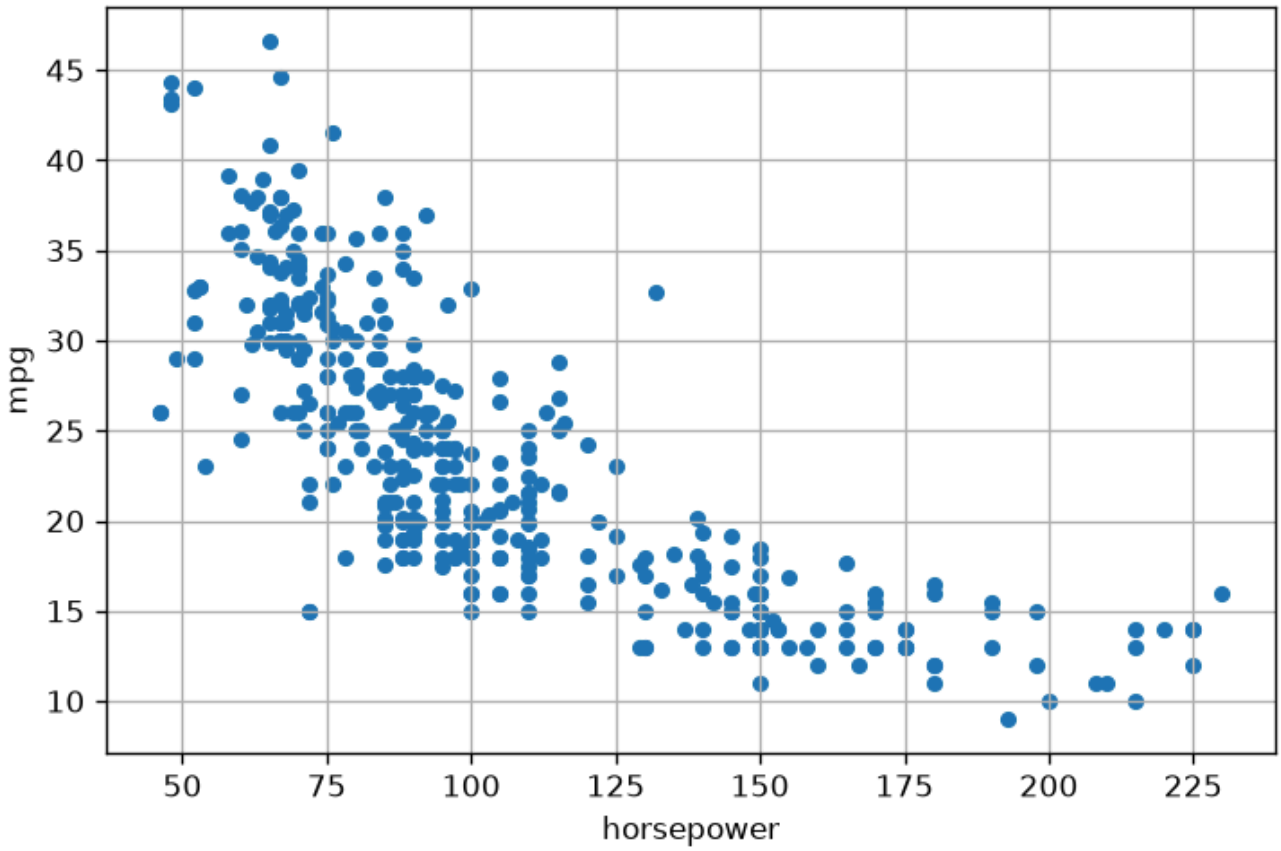
	displacement	cylinders	horsepower	weight	acceleration	model_year	origin	mpg
0	307.0	8	130.0	3504	12.0	70	1	18.0
1	350.0	8	165.0	3693	11.5	70	1	15.0
2	318.0	8	150.0	3436	11.0	70	1	18.0
3	304.0	8	150.0	3433	12.0	70	1	16.0
4	302.0	8	140.0	3449	10.5	70	1	17.0
...	...	...	...	...	...	...	...	...
393	140.0	4	86.0	2790	15.6	82	1	27.0
394	97.0	4	52.0	2130	24.6	82	2	44.0
395	135.0	4	84.0	2295	11.6	82	1	32.0
396	120.0	4	79.0	2625	18.6	82	1	28.0
397	119.0	4	82.0	2720	19.4	82	1	31.0

398 rows × 8 columns

Here is a description of the different variables:

- **Displacement**: The engine's displacement (in cubic inches), which indicates the engine's size and power.
- **Cylinders**: The number of cylinders in the engine of the car. This is a categorical variable.
- **Horsepower**: The engine's horsepower, a measure of the engine's performance.
- **Weight**: The weight of the car in pounds.
- **Acceleration**: The car's acceleration (in seconds) from 0 to 60 miles per hour.
- **Model Year**: The year the car was manufactured. This is often converted into a categorical variable representing the car's age.
- **Origin**: The car's country of origin or manufacturing. Car Name: The name or identifier of the car model.
- **MPG (Miles per Gallon)**: The fuel efficiency of the car in miles per gallon. It is the variable to be predicted in regression analysis.

We will start by exploring the relationship between the variables `horsepower` and `MPG`. Let's visualize the related scatterplot:



Regression Models

Regression models, in general, aim to **study the relationship between two variables**, X and Y, by defining a mathematical model f such that:

$$Y = f(X) + \epsilon$$

Here:

- f is a deterministic function which can be used to **predict the values of Y from the values of X**;
- ϵ is an **error term**, i.e., a variable capturing everything that is not captured by the deterministic function f. It can be due to different reasons, the main of which are:
  - f is not an accurate deterministic function of the process. Since we don't know the "true" function f and we are only estimating it, we may obtain a suboptimal f for which $Y \neq f(X)$. The error term captures the differences between our predictions and the true values.
  - Y cannot only be predicted from X, but some other variable is needed to correctly predict Y from X. For instance, X could be "years of education" and Y can be "income". While may expect that "income" is not completely predicted from "years of education". This can happen also because we don't always have observations for all relevant variables.
  - the problem has inherent stochasticity which cannot be entirely modeled within the deterministic function f. For instance, consider the problem of predicting the rate of wins in poker based on the expertise of the player. The expertise surely allows to predict the rate of wins, but wins partially depend also on random factors, such as how the deck was shuffled.

Note that, often, we model f in a way that we have its **analytical form**. This is very powerful. If we have the analytical form of the function f which **explains** how Y is influenced from X (**can be predicted from** X), then we can really understand deeply the connection between the two variables!

The function f can take different forms. The most common one is the **linear form** that we will see in the next section. While the linear form is very simple (and hence we can anticipate it will be a limited model in many cases), it has the great advantage to be **easy to interpret**.

Simple Linear Regression

Simple linear regression aims to model the **linear relationship** between two variables X and Y. In our example dataset, we will consider $X = \text{horsepower}$ and $Y = \text{mpg}$.

Since we are trying to model a linear relationship, we can imagine **a line passing through the data**. The simple linear regression model is defined as:

$$Y \approx \beta_0 + \beta_1 X$$

In our example:

$$\text{mpg} \approx \beta_0 + \beta_1 \text{horsepower}$$

It is often common to introduce a **"noise" variable** which captures the randomness due to which the expression above is approximated and write:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

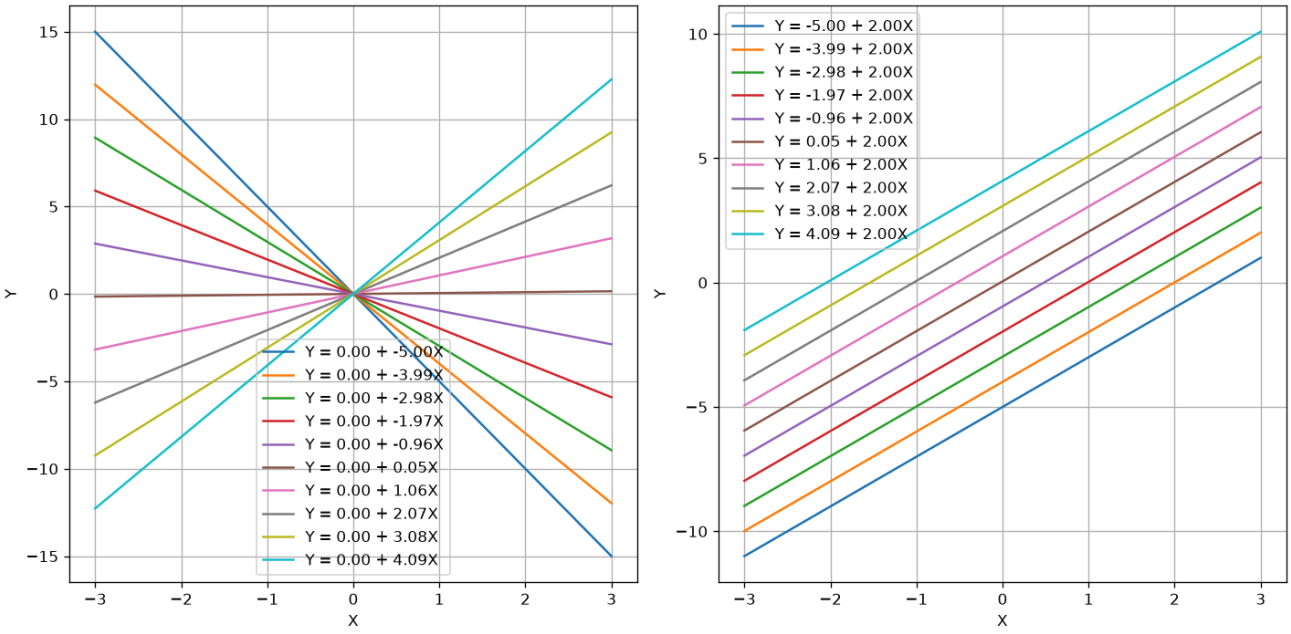
As we will see later, we expect ϵ to be **small and randomly distributed**.

Given the model above, we will call:

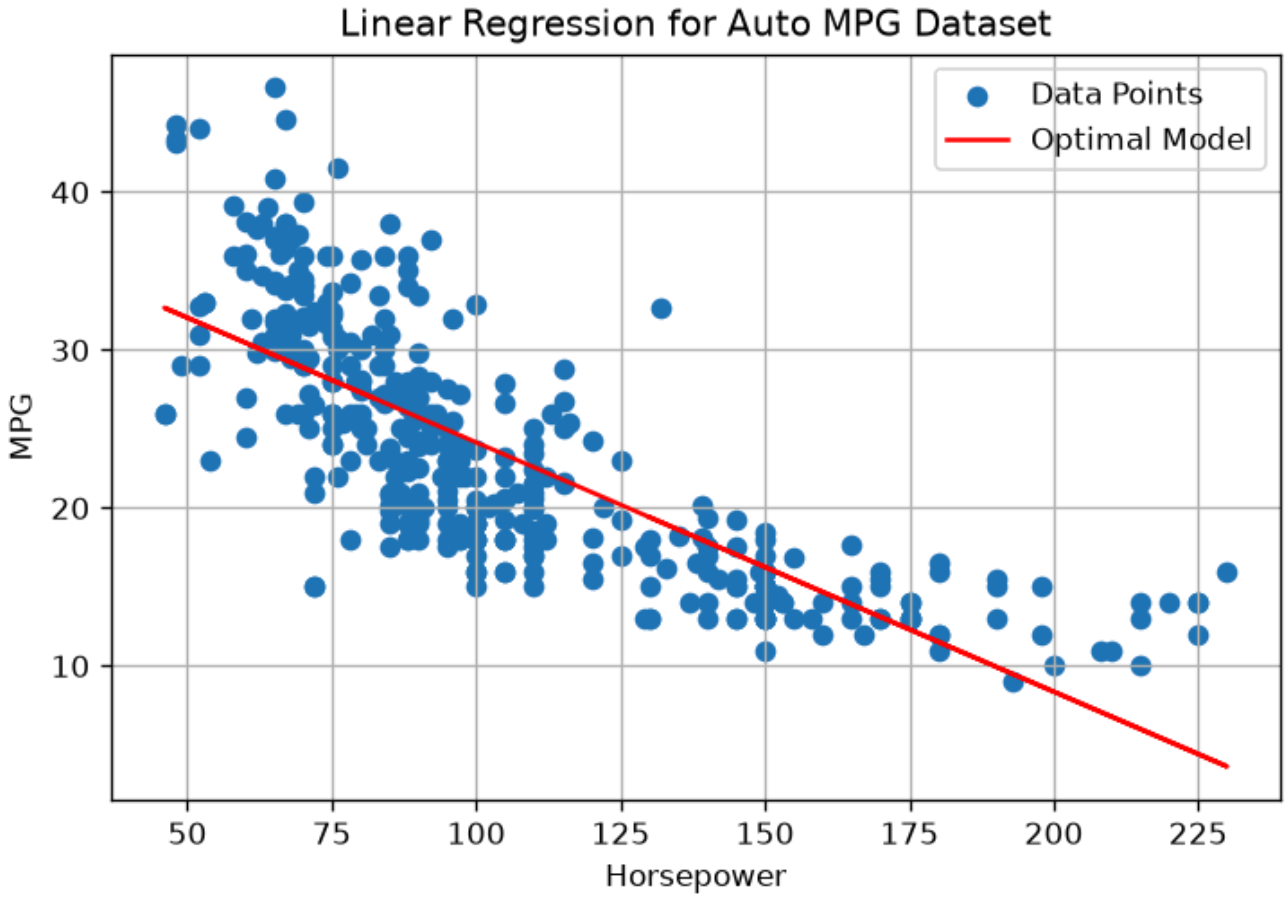
- X, the **independent variable** or **regressor**;
- Y, the **dependent variable** or **regressed variable**.

The values β_0 and β_1 are called **coefficients** or **parameters** of the model.

The mathematical model above has a geometrical interpretation. Indeed, specific values of β_0 and β_1 identify a given line in the 2D plane, as shown in the plot below:



We hence aim to estimate two appropriate values $\hat{\beta}_0$ and $\hat{\beta}_1$ from data in a way that they provide a model which represents well our data. In the case of our example, we expect the geometrical model to have this aspect:



This line will also be called the **regression line**.

Estimating the Coefficients - Ordinary Least Squares (OLS)

To estimate the coefficients of our optimal model, we should first define what is a good model. We will say that **a good model is one that predicts well the Y variable from the X one**. We already know from the example above that, since the relationship is not perfectly linear, the model will make some mistakes.

Let $\{(x_i, y_i)\}$ be our set of observations. Let

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$$

be the prediction of the model for the observation x_i. For each data point (x_i, y_i), we will define the deviation of the prediction from the y_i value as follows:

$$e_i = y_i - \hat{y}_i$$

We use the symbol e for error and also call these numbers **the residuals**, as they are the **differences (or residuals) between our prediction and the ground truth value**.

These numbers will be positive or negative based on whether we underestimate or overestimate the y_i values, while **they will be zero when the prediction is perfect**.

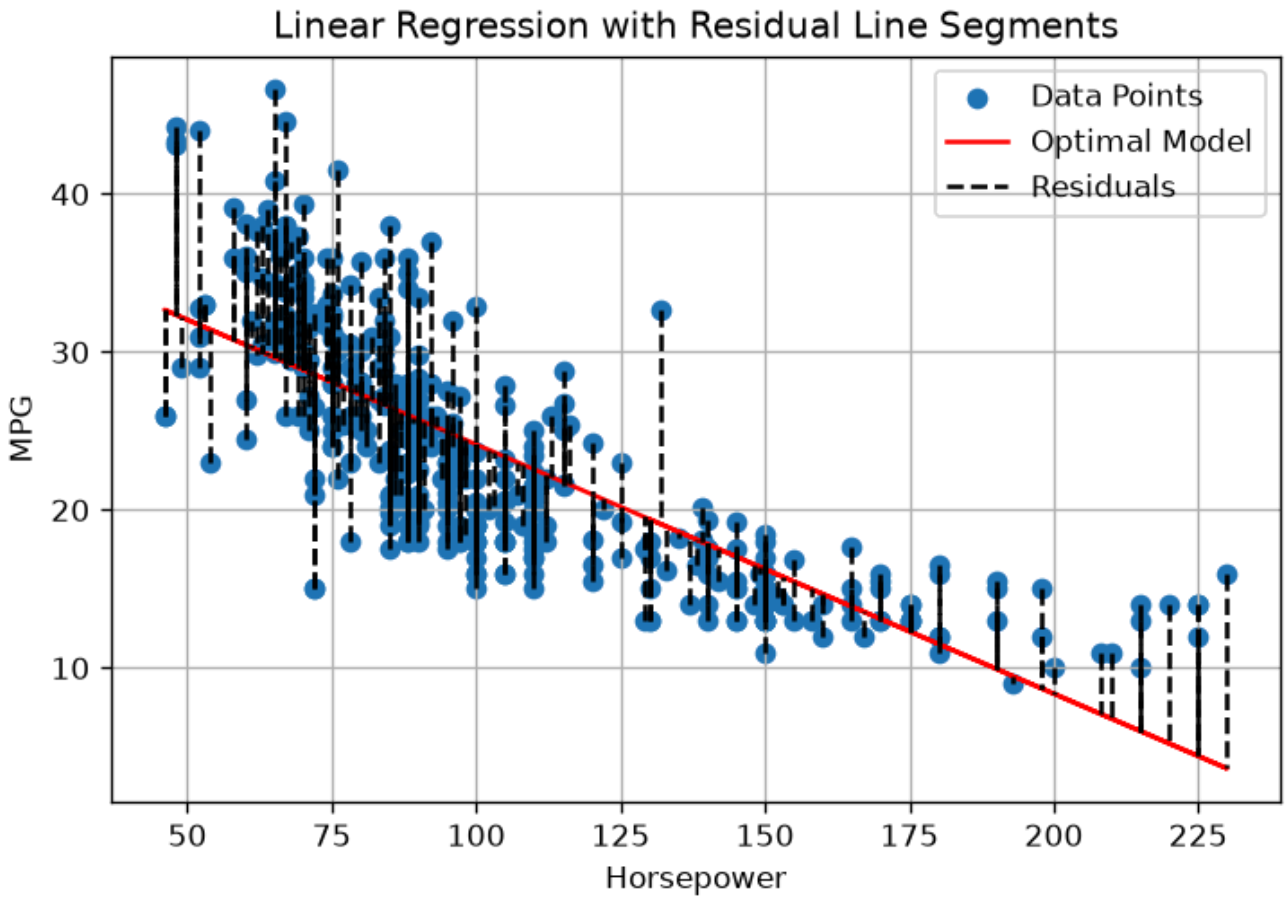
As a global error indicator for the model, given the data, we will define the **residual sum of squares (RSS)** as:

$$RSS = e_1^2 + e_2^2 + \ldots + e_n^2$$

or equivalently:

$$RSS = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + \ldots + (y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n)^2$$

This number will be the sum of the square values of the dashed segments in the plot below:



Intuitively, if we minimize these numbers, we will find the line which **best fits the data**.

We can obtain estimates for $\hat{\beta}_0$ and $\hat{\beta}_1$ by minimizing the RSS using an approach called **ordinary least squares**.

We can write the RSS as a function of the parameters to estimate:

$$RSS(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

This is also called a **cost function** or **loss function**.

We aim to find:

$$(\hat{\beta}_0, \hat{\beta}_1) = \arg\min_{\beta_0, \beta_1} RSS(\beta_0, \beta_1)$$

The minimum can be found setting:

$$\frac{\partial RSS(\beta_0, \beta_1)}{\partial \beta_0} = 0$$

$$\frac{\partial RSS(\beta_0, \beta_1)}{\partial \beta_1} = 0$$

Doing the math, it can be shown that:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})}$$

$$\hat{\beta}_0 = \overline{y} - \hat{\beta}_1 \overline{x}$$

> ⓘ **The same fit, seen as learning**
>
> Minimising the residual sum of squares is a special case of a much more general recipe: choose a loss, average it over the data, and pick the hypothesis that minimises the average. That recipe is **empirical risk minimisation**, and it is the frame the whole of Module 2 is built on — see chapter 13. For now it is enough to know that ordinary least squares is what it reduces to when the loss is the squared error.

Interpretation of the Coefficients of Linear Regression

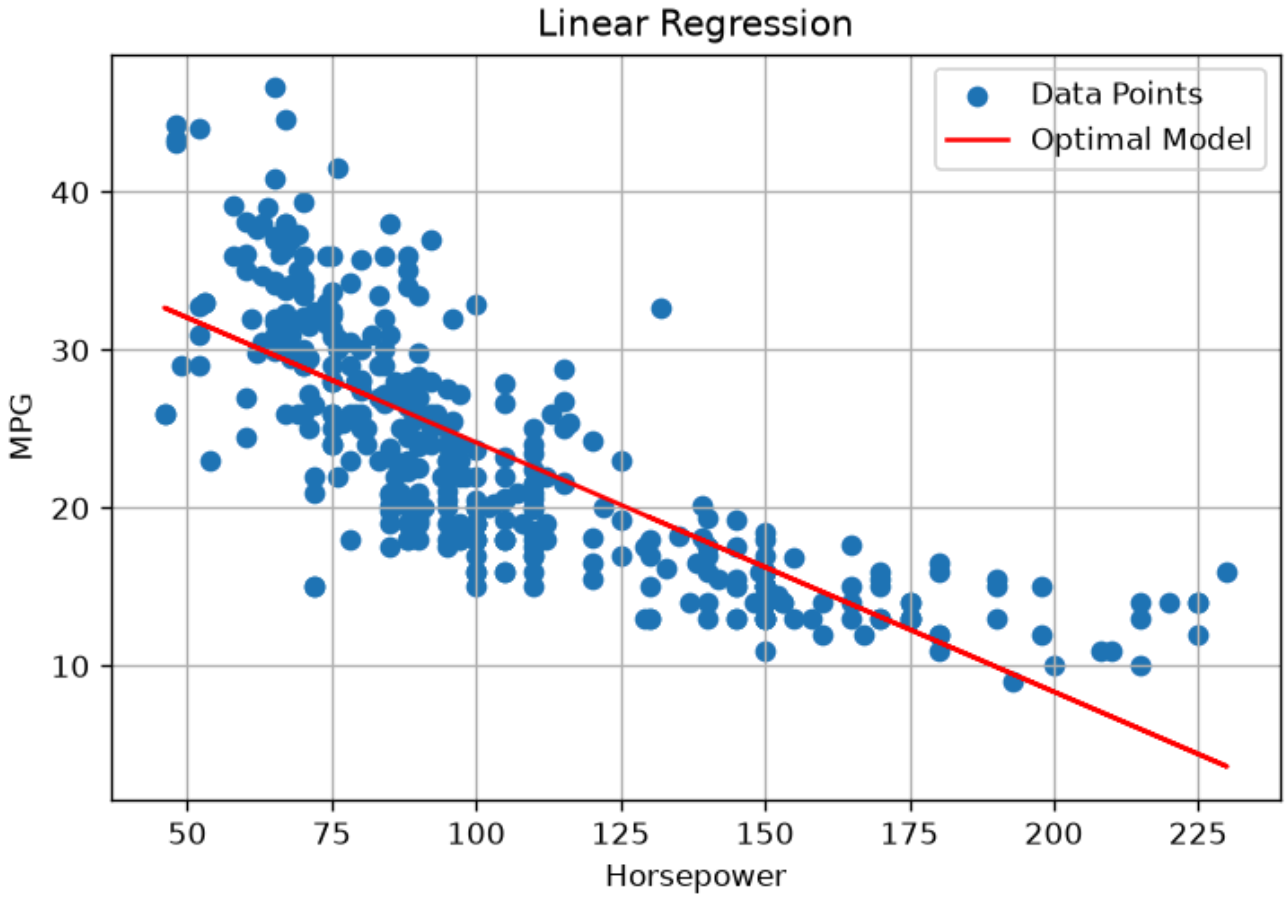
Using the formulas above, we find the following values for the example above:

```
beta_0: 39.94
beta_1: -0.16
```

These parameters identify the following line:

$$y = 39.94 - 0.16x$$

The plot below shows the line on the data:



Apart from the geometric interpretation, the coefficients of a linear regressor have an important **statistical interpretation**. In particular:

The intercept β_0 is the value of y that we get when the input value x is equal to zero $x = 0$ (i.e., $f(0)$). This value **may not always make sense**. For instance, in the example above, we have: $\beta_0 = 39.94$, which means that, **when the horsepower is 0, then the consumption in mpg is equal to 39.94**.

The coefficient β_1 indicates the steepness of the curve. If β_1 is large, then the curve is steep. This indicates that a small change in x is associated to a large change in y. In general, we can see that:

$$f(x+1) - f(x) = \beta_0 + \beta_1(x+1) - \beta_0 - \beta_1 x = \beta_1(x+1-x) = \beta_1$$

which reveals that **when we observe an increment of one unit of x, we observe an increment of β_1 units in y**. In our example, $\beta_1 = -0.16$, hence we can say that, for cars with one additional unit of horsepower, we observe a drop in mpg of -0.16 units.

Accuracy of the Coefficient Estimates

Recall that we are trying to model the relationship between two random variables X as Y with a simple linear model:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

This means that, once we find appropriate values of β_0 and β_1, we expect these to summarize the **linear relationship in the population** or the **population regression line**. Also, recall that these values are obtained using two formulas which are based on realizations of X of Y and can be hence seen as **estimators**:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

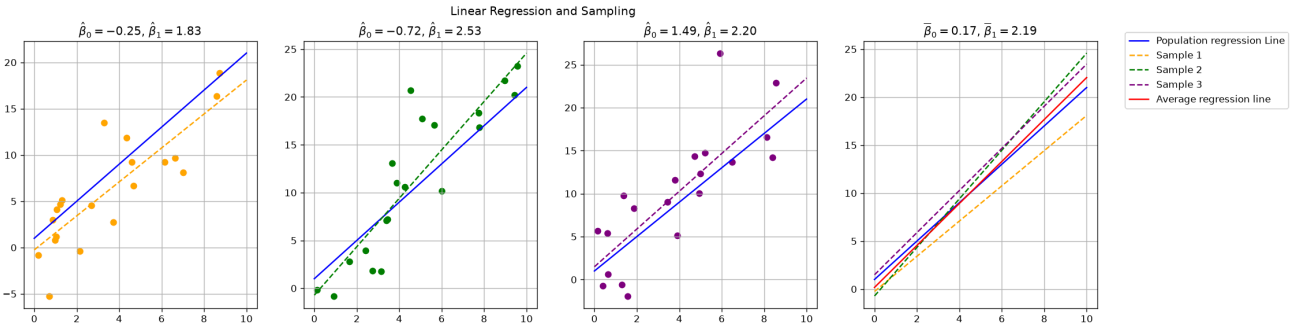
$$\hat{\beta}_0 = \overline{y} - \hat{\beta}_1 \overline{x}$$

We now recall that, being estimates, they provide values related to a given **realization of the random variables**.

Let us consider an ideal population for which:

$$Y = 2x + 1$$

Ideally, given a sample from the population, we expect to obtain $\hat{\beta}_0 \approx 1$ and $\hat{\beta}_1 \approx 2$. In practice, different samples may lead to different estimates and hence different regression lines, as shown in the plot below:



Each of the first three subplots shows a different sample drawn from the population, with its corresponding estimated regression line, along with the true population regression line. The last subplot compares the different estimated lines with the population regression line and the average regression line (in red). Coefficient estimates are shown in the subplot titles.

As can be noted, each estimate can be inaccurate, while the average regression line is very close to the population regression line. This is due to the fact that our estimators for the parameters of the regression coefficients **have non-zero variance**. In practice, it can be shown that these estimators are **unbiased** (hence the average regression line is close to the population line).

Confidence Intervals of the Regression Coefficients

Since the formulas to compute regression parameters can be seen as estimators, we can compute **standard errors** and **confidence intervals**.

For instance, for our model

$$\text{mpg} \approx \beta_0 + \text{horsepower} \cdot \beta_1$$

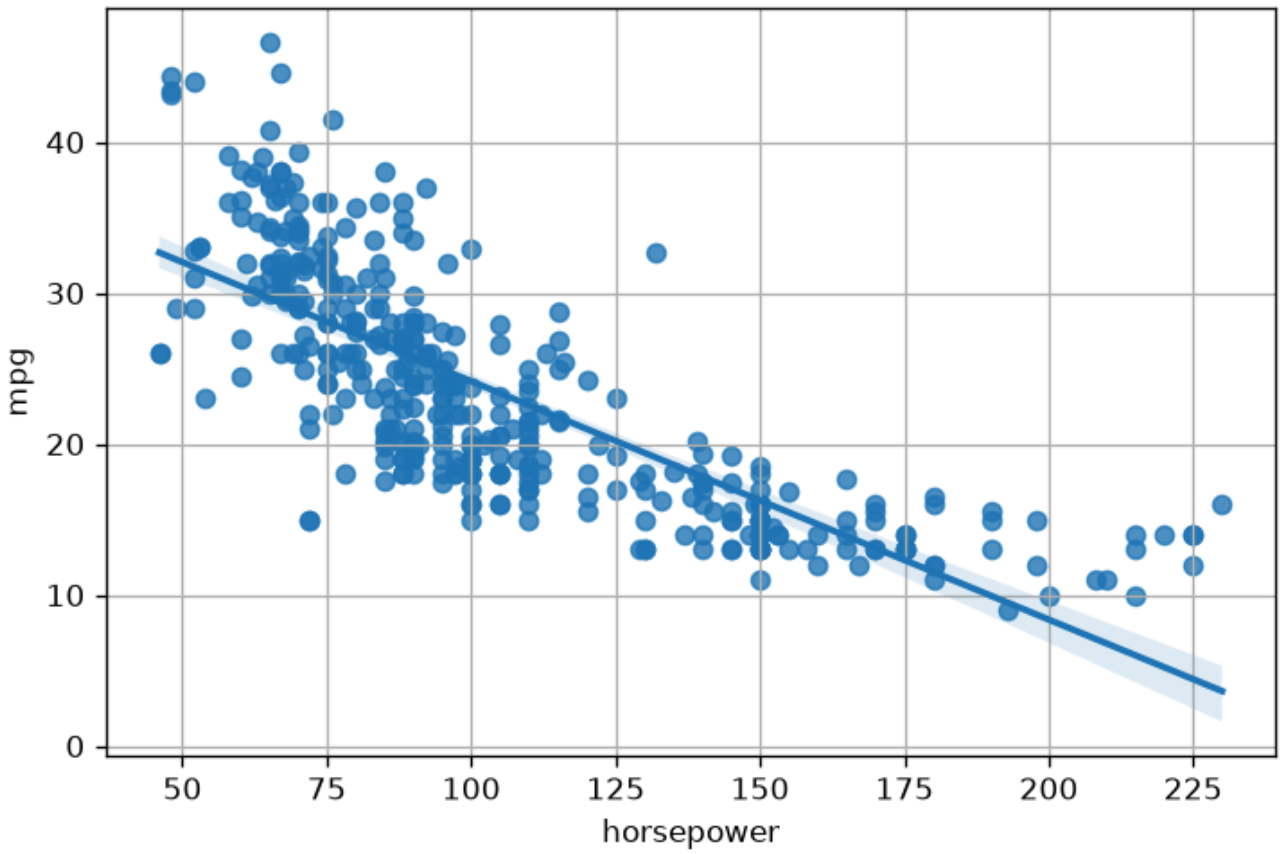
We will have the following confidence interval at a 95% confidence level (which is a common CL):

	COEFFICIENT	STD ERROR	CONFIDENCE INTERVAL
	39.94	0.717	$[38.53, 41.35]$
	-0.1578	0.006	$[-0.17, -0.15]$

From the table above, we can say that:

- The value of `mpg` for `horsepower=0` lies somewhere between 38.53 and 41.35;
- An increase of `horsepower` by one unit is associated to a decrease of `mpg` between -0.17 and -0.15.

It is also common to see plots like the following one:



In the plot above, the shading around the line illustrates the variability induced by the confidence intervals estimated for the coefficients.

Statistical Tests for the Significance of Coefficients

Seeing the aforementioned formulas as estimators also allows to perform **hypothesis tests** on the coefficients. In practice, it is common to perform a statistical test to assess whether the coefficient β_1 is significantly different from zero. It is interesting to check this because, if β_1 was equal to zero, then there would not be any correlation between the variables (and hence the linear regressor would not be useful). Indeed, if $\beta_1 = 0$:

$$Y = \beta_0 + \epsilon$$

Hence Y cannot be predicted from X and the two variables are not associated.

The **null hypothesis** of the test is as follows:

$$H_0 : \text{There is no association between } X \text{ and } Y \Leftrightarrow \beta_1 = 0$$

While the **alternative hypothesis** is formulated as follows:

$$H_1 : \text{There is some association between } X \text{ and } Y \Leftrightarrow \beta_1 \neq 0$$

We won't see the mathematical details of this test, but it works as usual: we compute some statistic t following a known distribution (a t-Student distribution in this case), then compute a **p-value** and reject the null hypothesis if this is under the significance level $p < \alpha$ (often set to $\alpha = 0.05$).

A similar test is conducted to check whether β_0 is significantly different from zero. But in this case it is not a big deal if the test fails as $\beta_0 = 0$ is a perfectly reasonable result (the regression line passes through the origin).

Let's see the updated table from the same example:

	COEFFICIENT	STD ERROR	t	P>|t|	CONFIDENCE INTERVAL
β_0	39.94	0.717	55.66	0	$[38.53, 41.35]$
β_1	-0.1578	0.006	-24.49	0	$[-0.17, -0.15]$

From the table above, we can conclude that both β_0 and β_1 are significantly different than 0 (the p-value is equal to zero). This can also be noted by the fact that the confidence intervals do not contain the zero number.

Assessing Model Accuracy

The hypothesis tests (like the t-test for β_1) tell us *if* a statistically significant relationship exists. They do not, however, tell us **how well the model fits the data** or **how accurate its predictions are**.

To measure performance, we use different metrics that generally fall into two categories, aligning with our "two goals":

1. **Metrics for Understanding (Goodness-of-Fit):** These are used in the "Statistical" approach to measure how well the model explains the data it was trained on. (e.g., RSE, R^2)
2. **Metrics for Prediction (Predictive Accuracy):** These are used in the "Machine Learning" approach to measure how well the model performs on new, unseen data. (e.g., MSE, RMSE, MAE)

The Building Block: Residuals and RSS

All regression metrics are built upon the **residuals**—the errors from the model. A residual (e_i) is the difference between the observed value (y_i) and the predicted value ($\hat{y}_i$):

$$e_i = y_i - \hat{y}_i$$

The **Residual Sum of Squares (RSS)** which we already saw is the foundational quantity that OLS (Ordinary Least Squares) minimizes. It is the sum of all squared residuals:

$$RSS = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Metrics for Understanding (Goodness-of-Fit)

When our goal is **understanding** (inference), we typically evaluate the model on the entire dataset it was trained on.

1. Residual Standard Error (RSE)

The RSE is the *statistical* estimate of the standard deviation of the irreducible error, ϵ. It measures the "typical" error of the model.

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

- **Why $n-2$?** We divide by the "degrees of freedom" ($n-2$) because we estimated *two* parameters (β_0 and β_1). This makes RSE an unbiased estimate of σ (the standard deviation of ϵ).
- **Interpretation:** RSE is an **absolute measure** of the model's "lack of fit," expressed in the **same units as Y**. A smaller RSE means the model fits the data better.

To know if an RSE value is "good," you must compare it to the scale of Y.

- **Example:** For our `mpg` data, $RSE = 4.91$. The mean value of `mpg` is $\overline{y} = 23.52$. This means our model's typical error (4.91) is about **20%** of the average value, which may be acceptable or not, depending on the context.

2. The R^2 Statistic

The RSE is an absolute measure. It's often more useful to have a *relative* measure that tells us what proportion of the data's variance our model can explain. This is the R^2 **(Coefficient of Determination)**.

To calculate R^2, we first define a "baseline" model that just predicts the average Y for every X:

$$\hat{y}_{baseline} = \overline{y}$$

The error of this "dumb" model is the **Total Sum of Squares (TSS)**:

$$TSS = \sum_{i=1}^{n} (y_i - \overline{y})^2$$

- **TSS** measures the **total variance** in Y (before any regression).
- **RSS** measures the **unexplained variance** in Y (what's left over *after* regression).

The R^2 statistic is the proportion of total variance that is "explained" by the model:

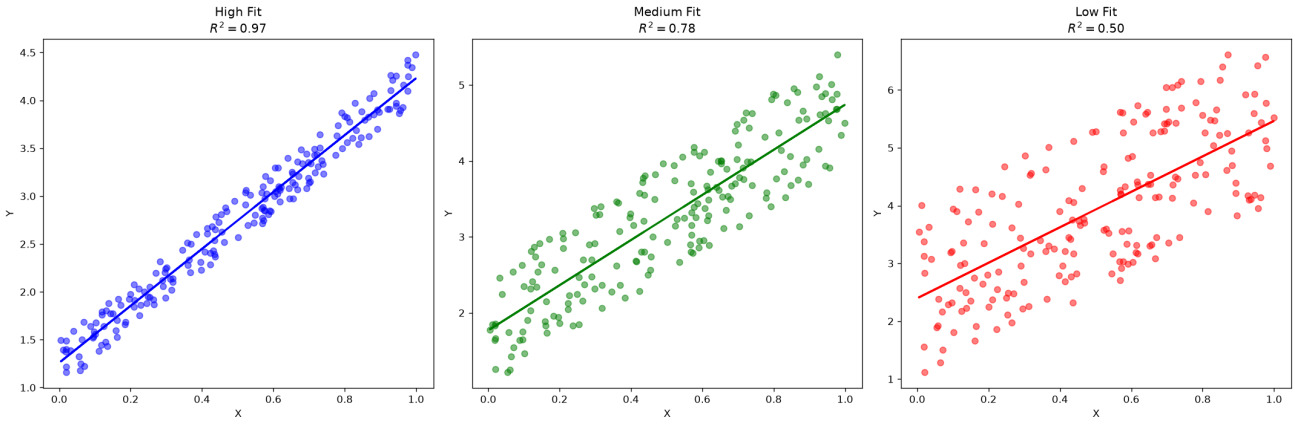
$$R^2 = \frac{TSS - RSS}{TSS} = 1 - \frac{RSS}{TSS}$$

- **Interpretation:** R^2 is a value between 0 and 1 (or 0% to 100%).
  - $R^2 = 0$: Our model is no better than just guessing the mean ($RSS = TSS$).
  - $R^2 = 1$: Our model perfectly explains all the variance ($RSS = 0$).
  - **Example:** For our `mpg` data, $R^2 = 0.61$. This means **"61% of the variance in `mpg` can be explained by `horsepower`."**

For simple linear regression (one X), R^2 is also equal to the square of the Pearson correlation coefficient: $R^2 = \rho^2$.

Visual Examples

The plot below shows examples of linear regression fits with the different evaluation measures that we saw:



Diagnostic Plots: Checking Model Validity

The metrics above (R^2, $RMSE$) are "scoring" metrics. They give you a single number to summarize *how much* error you have.

However, they don't tell you *why* you have error, or if your model is fundamentally **invalid**. A model can have a "good" R^2 but still be flawed. We use **residual plots** as *diagnostic* tools to check this.

1. The Residuals vs. Fitted Plot

This is the most important diagnostic plot. We plot the **Residuals (e_i)** on the y-axis against the **Fitted (Predicted) Values ($\hat{y}_i$)** on the x-axis.

What we *want* to see: A boring, random cloud of points with no pattern, centered around the $y = 0$ line. This indicates our model has captured the pattern and the remaining errors are just random noise.

What we *don't* want to see (Red Flags):

- **A "U-Shape" or any Clear Pattern:** This means our model is **wrong**. The relationship between X and Y was **not linear**, and our simple linear model has failed to capture the true non-linear pattern.
  - **Diagnosis:** Violates the **Linearity** assumption.
  - **Fix:** You need a more complex model (e.g., polynomial regression).
- **A "Fanning" or "Cone" Shape:** The residuals get bigger (more spread out) as the predicted value $\hat{y}$ gets bigger.
  - **Diagnosis:** This is called **Heteroscedasticity** (non-constant variance).
  - **Problem:** This violates a core assumption of linear regression. It means our p-values and confidence intervals are unreliable because the model is less accurate for high-value predictions.

The figure below shows some examples of what to expect:

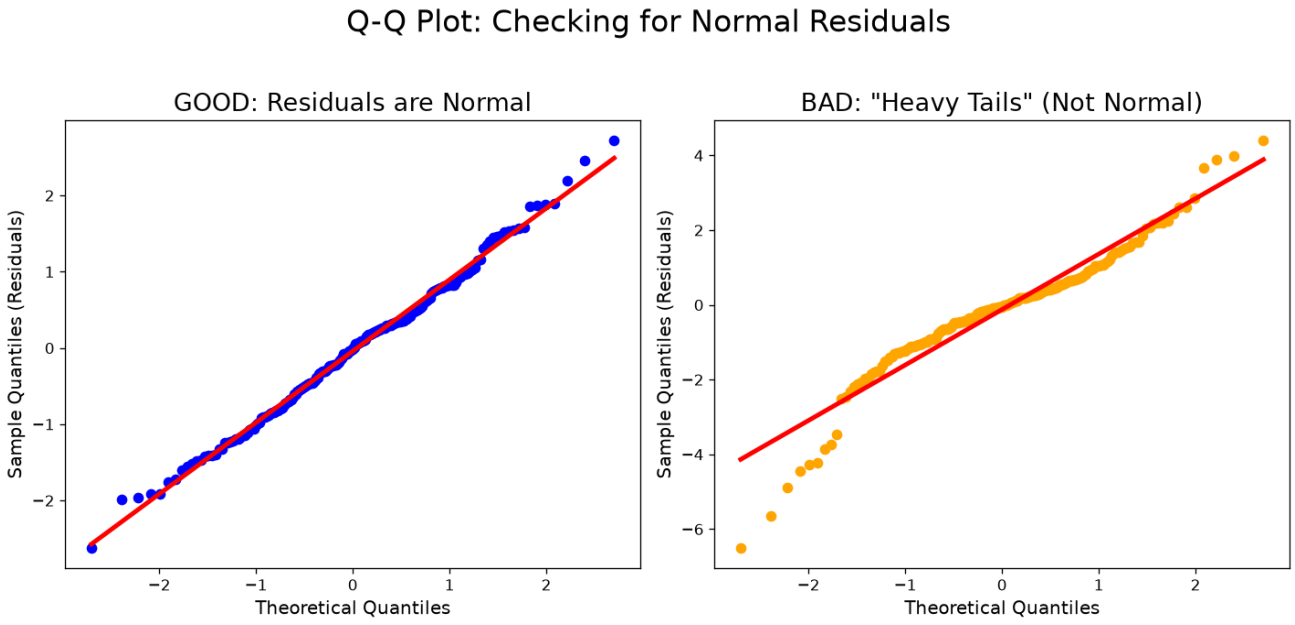


2. The Q-Q Plot (Quantile-Quantile Plot)

This is a plot used to check if the residuals are **normally distributed**. This is another key assumption for all our statistical tests (p-values, t-statistics).

- **Purpose:** To check the **Normality** assumption.
- **How to read it:** If the residuals are truly normally distributed, the points will fall perfectly along the diagonal line.
- **Problem:** If the points "S-curve" or peel away from the line, it means the residuals are not normal, and our p-values and confidence intervals may be inaccurate.

The figure below shows examples of "good" and "bad" residuals:



Multiple Linear Regression

In the **MPG example**, we saw that about 61% of the variance in Y was explained by the model. One may wonder why about 39% of the variance could not be explained. Some common reasons may be:

- There is stochasticity in the data which prevents us to learn an accurate function to predict Y from X;
- The relationship between X and Y is far from linear, so we cannot predict Y accurately;
- The prediction of Y also depends on other variables.

While in general the unexplained variance is due to a combination of the aforementioned factors, the third one is often very common and relatively easy to fix. In our case, we are trying to predict `mpg` from `horsepower`. However, we can easily imagine how other variables may contribute to the estimation of `mpg`. For instance, two cars with the same `horsepower` but different `weight` may have different values of `mpg`. We can hence try to find a model with **also uses `weight` to predict `mpg`**. This is simply done by adding an additional coefficient for the new variable:

$$mpg = \beta_0 + \beta_1 horsepower + \beta_2 weight$$

The obtained model is called **multiple linear regression**. If we fit this model (we will see how to estimate coefficients in this case), we obtain the following R^2 value:

$$R^2 = 0.71$$

An increment of $+0.1$!

In general, we can include as many variables as we think is relevant to add and define the following model:

$$Y = \beta_0 + \beta_1 X_1 + \ldots + \beta_n X_n$$

For instance, the following model:

$$mpg = \beta_0 + \beta_1 horsepower + \beta_2 weight + \beta_3 model\_year$$

Has an R^2 value of:

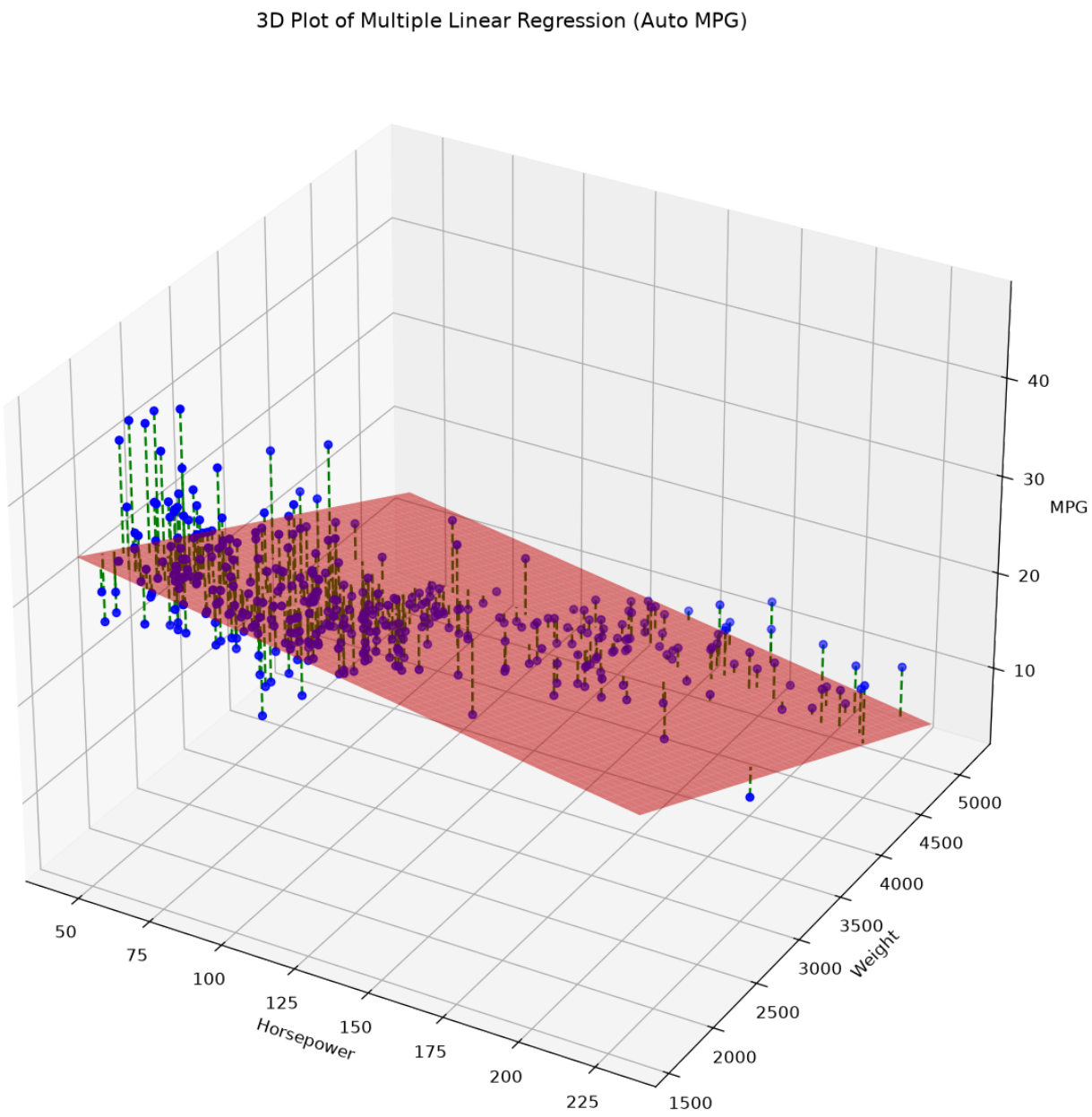
$$R^2 = 0.808$$

Geometrical Interpretation

The multiple regression model based on two variable has a geometrical interpretation. Indeed, the equation:

$$Z = \beta_0 + \beta_1 X + \beta_2 Y$$

identifies a plane in the 3D space. We can visualize the plane identified by the $mpg = \beta_0 + \beta_1 horsepower + \beta_2 weight$ model as follows:



The dashed lines indicate the residuals. The best fit of the model minimizes again the sum of squared residuals. This model makes predictions selecting the Z value which intersect the plane for given values of X and Y.

In general, when we consider n variables, the linear regressor will be **a (n-1)-dimensional hyperplane in the n-dimensional space**.

Statistical Interpretation

The statistical interpretation of a multiple linear regression model is very similar to the interpretation of a simple linear regression model. Given the general model:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_i x_i + \ldots + \beta_n x_n$$

we can interpret the coefficients as follows:

- The value of β_0 indicates the value of y when all independent variables are set to zero;
- The value of β_i indicates the increment of y that we expect to see when x_i increments by one unit, **provided that all other values $x_j | j \neq i$ are constant**.

In the considered example:

$$mpg = \beta_0 + \beta_1 horsepower + \beta_2 weight$$

we obtain the following estimates for the coefficients:

$\hat{\beta}_0$	$\hat{\beta}_1$	$\hat{\beta}_2$
45.64	-0.05	-0.01

We can interpret these estimates as follows:

- Cars with zero `horsepower` and zero `weight` will have an `mpg` of 45.64 ($\approx 19.4 Km/l$).
- An increment of one unit of `horsepower` is associated to a decrement of `mpg` of -0.05 units, provided that `weight` is constant. This makes sense: cars with more `horsepower` will probably consume more fuel.
- An increment of one unit of `weight` is associated to a decrement of `mpg` of `-0.01` units, provided that `horsepower` is constant. This makes sense: heavier cars will consume more fuel.

Let's compare the estimates above with the estimates of our previous model:

$$mpg = \beta_0 + \beta_1 horsepower$$

In that case, we obtained:

$\hat{\beta}_0$	$\hat{\beta}_1$
39.94	-0.16

We can note that the coefficients are different. This happens because, when we add more variables, **the model explains variance in a different way**. If we think more about it, this is coherent with the interpretation of the coefficients. Indeed:

- 39.94 is the expected value of `mpg` when `horsepower=0`, but all other variables have unknown values. 45.64 is the expected value of `mpg` when `horsepower=0` and `weight=0`. This is different, as in the second case we are (virtually) looking at a subset of data for which both horsepower and weight are zero, while in the first case, we are only looking at data for which `horsepower=0`, but `weight` can be any value. In some sense, we can see 39.94 as an average value for different values of `weight` (and all other unobserved variables).

- -0.16 is the expected increment of `mpg` when we observe an increment of one unit of `horsepower` and we don't know anything about the values of the other variables. -0.05 is the expected increment of `mpg` when `horsepower` and `weight` are held constant, so, again, we are (virtually) looking at a different subset of the data in which the relationship between `mpg` and `horsepower` may be a bit different.

Note that, also in the case of multiple regression, we can estimate confidence intervals and perform statistical tests. In our example, we will get this table:

	COEFFICIENT	STD ERROR	t	P>|t|	CONFIDENCE INTERVAL
/	45.64	0.793	57.54	0	$[44.08, 47.20]$
/	-0.05	0.011	-4.26	0	$[-0.07, -0.03]$
/	-0.01	0.001	-11.53	0	$[-0.007, -0.005]$

Estimating the Regression Coefficients

Given the general model:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_i x_i + \ldots + \beta_n x_n$$

We can define our **cost function** again as the residual sum of squares:

$$RSS(\beta_0, \ldots, \beta_n) = \sum_{i=1}^{m} (y_i - \beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_n x_n^{(i)})^2$$

Where m is the total number of observations and $x_j^{(i)}$ is the jth variable (x_j) of the ith observations.

The $\hat{\beta}_0, \ldots, \hat{\beta}_n$ values that minimize the loss function above are the **multiple least square coefficient estimates**.

To find these optimal values, it is convenient to use matrix notation. Given m observations, we will have m equations:

$$y^{(1)} = \beta_0 + \beta_1 x_1^{(1)} + \ldots + \beta_n x_n^{(1)} + e^{(1)}$$

$$y^{(2)} = \beta_0 + \beta_1 x_1^{(2)} + \ldots + \beta_n x_n^{(2)} + e^{(2)}$$

$$\dots$$

$$y^{(m)} = \beta_0 + \beta_1 x_1^{(m)} + \dots + \beta_n x_n^{(m)} + e^{(m)}$$

We can write the m equations in matrix form as follows:

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{e}$$

where:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}, \mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \vdots & & \\ 1 & x_1^{(m)} & x_2^{(m)} & \dots & x_n^{(m)} \end{bmatrix}, \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \mathbf{e} = \begin{bmatrix} e^{(1)} \\ e^{(2)} \\ \vdots \\ e^{(m)} \end{bmatrix}$$

The matrix $\mathbf{X}$ is s **design matrix**.

In the notation above, we want to minimize:

$$RSS(\beta) = \sum_{i=1}^{m} (e^{(i)})^2 = \mathbf{e}^T\mathbf{e}$$

It can be shown that, by the **least squares method**, the RSS is minimized by the estimate:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

The F-Test

When fitting a multiple linear regressor, it is common to perform a statistical test to check whether at least one of the regression coefficients is significantly different from zero (in the population). This test is called an $F-test$. We define the **null and alternative hypotheses** as follows:

$$H_0 : \beta_1 = \beta_2 = \dots = \beta_n = 0$$

$$H_a : \exists j \; s.t. \; \beta_j \neq 0$$

The null hypothesis (the one we want to reject with this test) is that all coefficients are zero in the population. If this is true, than the multiple regressor is not reliable and we should discard it. Note that we exclude β_0 from these coefficients because it represents the position of the regression line (intercept) and does not denote correlation. The alternative hypothesis is that at least one of the coefficients is different from zero.

We will not see the mathematical details, but in this case, we compute an **F-statistic** which follows a **F-distribution**.

The test is hence carried out as usual, finding a p-value which indicates the probability to observe a statistic larger than the observed one if all regression coefficients are zero in the population.

In our example of regressing `mpg` from `horsepower` and `weight`, we will find:

R^2	F-statistic	Prob(F-statistic)
0.706	467.9	3.06e-104

This indicates that the regressor is statistically relevant as the p-value (Prob(F-statistic)) is very small (under the significance level $\alpha = 0.05$).

Variable Selection by Backward Elimination

Let's now try to fit a multiple linear regressor on our dataset by including all variables. Our dependent variable will be `mpg`, while the set of dependent variables will be:

```
['displacement' 'cylinders' 'horsepower' 'weight' 'acceleration'
 'model_year' 'origin']
```

We obtain the following measures of fit:

R^2	F-statistic	Prob(F-statistic)
0.821	252.4	2.04e-139

The regressor has a good R^2 and the p-value of the F-test is very small. We can conclude that there is some relationship between the independent variables and the dependent one.

The estimates of the regression coefficients will be:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-17.2184	4.644	-3.707	0.000	-26.350	-8.087
horsepower	-0.0170	0.014	-1.230	0.220	-0.044	0.010
weight	-0.0065	0.001	-9.929	0.000	-0.008	-0.005
displacement	0.0199	0.008	2.647	0.008	0.005	0.035
cylinders	-0.4934	0.323	-1.526	0.128	-1.129	0.142
acceleration	0.0806	0.099	0.815	0.415	-0.114	0.275
model_year	0.7508	0.051	14.729	0.000	0.651	0.851
origin	1.4261	0.278	5.127	0.000	0.879	1.973

From the table, we can see that not all predictors have a p-value below the significance level $\alpha = 0.05$. In particular:

- `horsepower` has a large p-value of 0.22;

- `cylinders` has a large p-value of 0.128;
- `acceleration` has a large p-value of 0.415.

This means that, within the current regressor, there is no meaningful relationship between these variables and `mpg`. A legitimate question is

> How is it possible that `horsepower` is not associated to `mpg` in this regressor if it was associated to it before?!

However, we should recall that, when we consider a different set of variables, the interpretation of the coefficients changes. So, even if in the previous models, `horsepower` was correlated to `mpg`, now it is not correlated anymore. We can imagine that the relationship between these variables is now explained by the other variables which we have introduced.

Even if the model is statistically significant, it does make sense to get rid of the variables with poor relationships with `mpg`. After all, if we remove a variable, the estimates of the other coefficients may change.

A common way to remove these variables is by **backward selection** or **backward elimination**. This consists in iteratively removing the variable with the largest p-value. We remove one variable at a time and re-compute the results, iterating until all variables have a small p-value.

let's start by removing `acceleration`. This is the result:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-15.5635	4.175	-3.728	0.000	-23.773	-7.354
horsepower	-0.0239	0.011	-2.205	0.028	-0.045	-0.003
weight	-0.0062	0.001	-10.883	0.000	-0.007	-0.005
displacement	0.0193	0.007	2.579	0.010	0.005	0.034
cylinders	-0.5067	0.323	-1.570	0.117	-1.141	0.128
model_year	0.7475	0.051	14.717	0.000	0.648	0.847
origin	1.4282	0.278	5.138	0.000	0.882	1.975

Note that the coefficients have changed. We now remove `cylinders`, which has the largest p-value of 0.117:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-16.6939	4.120	-4.051	0.000	-24.795	-8.592
horsepower	-0.0219	0.011	-2.033	0.043	-0.043	-0.001
weight	-0.0063	0.001	-11.124	0.000	-0.007	-0.005
displacement	0.0114	0.006	2.054	0.041	0.000	0.022
model_year	0.7484	0.051	14.707	0.000	0.648	0.848
origin	1.3853	0.277	4.998	0.000	0.840	1.930

All variables now have an acceptable p-value ($\alpha = 0.05$). We are done. Note that, by removing the two variables, `horsepower` now has an acceptable p-value. This indicates that one of the removed variables was redundant with respect to `horsepower`.

Collinearity and the Instability of Least Squares

Let's reflect again on the issue we saw in our model: the `horsepower` p-value was significant in the simple model but became non-significant ($p > 0.05$) in the multiple regression. This isn't a contradiction; it's a classic symptom of **collinearity**.

Collinearity is a high linear correlation between two predictor variables. The more general (and common) problem is **multicollinearity**, which occurs when one predictor X_j can be accurately predicted by a linear combination of *other* predictors in the model (e.g., $X_j \approx \beta_0 + \beta_1 X_1 + \dots$).

The primary problem with multicollinearity is that it makes our coefficient estimates **unstable and unreliable**. The standard errors of the coefficients for the correlated predictors become massively inflated. This is what we saw: `horsepower` became non-significant because `weight` (which is highly correlated with it) "stole" all of its explanatory power.

This instability can be explained by **looking directly at the mathematical solution for the OLS coefficients** $\hat{\beta}$:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

The key to this entire equation is the $(X^T X)^{-1}$ term, which is the inverse of the "design matrix." The $X^T X$ matrix is, in essence, a matrix of the covariances between all of our predictor variables.

1. **Perfect Collinearity (Singularity):** If one variable is a perfect linear combination of another (e.g., $X_1 = 2 \cdot X_2$), the columns of X are not linearly independent. This causes the $X^T X$ matrix to be **singular**, meaning its determinant is zero and its inverse **does not exist**. There is no unique solution for $\hat{\beta}$.

2. **Multicollinearity (Ill-Conditioning):** In our `mpg` case, the variables aren't *perfectly* correlated, but *highly* correlated. This means $X^T X$ is not perfectly singular, but it is **"ill-conditioned"** or **"near-**

singular". Its determinant is *almost* zero.

When we compute the inverse, $(X^T X)^{-1}$, we are "dividing by almost zero." This causes the numbers *inside* the inverse matrix to explode, becoming massive.

The formula for the **standard error** of each coefficient $\hat{\beta}_j$ is directly proportional to the values in this inverse matrix. Therefore, if the values in $(X^T X)^{-1}$ are huge, the standard errors for our coefficients become **inflated**. This leads to a tiny t-statistic ($t = \hat{\beta}_j / SE(\hat{\beta}_j)$) and a massive, non-significant p-value. This is exactly what happened to `horsepower`.

When `statsmodels` (or `numpy`) encounters a *perfectly* singular $X^T X$ matrix (e.g., you forget to drop one dummy variable, creating perfect multicollinearity), it cannot compute the standard inverse.

Instead of crashing, it computes the **Moore-Penrose Pseudoinverse**, denoted as $(X^T X)^+$. This is a generalization of the inverse that finds the *one* solution $\hat{\beta}$ that has the smallest ℓ_2-norm. In practice, `statsmodels` will identify this redundancy, report that the matrix is singular, and often arbitrarily set one of the redundant variable's coefficients to zero to provide a stable (but not unique) set of coefficients.

Adjusted R^2

While in the case of simple regression we saw that $R^2 = \rho(x, y)^2$ (where ρ is the correlation coefficient), in the case of multiple regression, it turns out that:

$$R^2 = \rho(Y, \hat{Y})^2$$

In general, having more variables in the linear regressor will reduce the error term and increase the covariance between Y and $\hat{Y}$, hence increasing the R^2. However, in general having a small increase in R^2 when we add a new variable may not be good. Indeed, we could prefer a simpler model with a slightly smaller R^2 value.

This is again a problem of bias-variance tradeoff.

To express this, we can compute the adjusted R^2 as follows:

$$\overline{R}^2 = 1 - \frac{m-1}{m-n-1}(1 - R^2)$$

Where m is the number of data points and n is the number of independent variables. The $\overline{R}^2$ re-balances the R^2 accounting for the introduction of additional variables.

For instance the last model we fit has an $R^2 = 0.820$ and $\overline{R}^2 = 0.818$. In general, when using multiple regressor and especially for variable selection, it is good practice to use the adjusted R^2 rather than standard R^2.

Qualitative Predictors

So far, we have studied relationships between continuous variables. In practice, linear regression allows to also study relationship between **continuous dependent variables** and **qualitative independent variables**. We will consider another dataset similar to the **Auto MPG** dataset:

	mpg	horsepower	fuelsystem	fueltype	length	cylinders
0	21	111.0	mpfi	gas	168.8	4
1	21	111.0	mpfi	gas	168.8	4
2	19	154.0	mpfi	gas	171.2	6
3	24	102.0	mpfi	gas	176.6	4
4	18	115.0	mpfi	gas	176.6	5
...	...	...	...	...	...	...
200	23	114.0	mpfi	gas	188.8	4
201	19	160.0	mpfi	gas	188.8	4
202	18	134.0	mpfi	gas	188.8	6
203	26	106.0	idi	diesel	188.8	6
204	19	114.0	mpfi	gas	188.8	4

205 rows × 6 columns

In this case, besides having numerical variables, we also have qualitative ones such as `fuelsystem` and `fueltype`. Let's see what are their unique values:

```
fuelsystem: ['mpfi' '2bbl' 'mfi' '1bbl' 'spfi' '4bbl' 'idi' 'spdi']
fueltype: ['gas' 'diesel']
```

We will not see the meaning of all the values of `fuelsystem`, while the values of `fueltype` are self-explanatory.

Predictors with Only Two Levels

We will first see the case in which qualitative predictors only have two levels. To handle these as independent variables, we can define a new **dummy variable** which will encode 1 as one of the two levels and 0 as the other one. For instance, we can introduce a `fueltype[T.gas]` variable defined as follows:

$$fueltype[T.gas] = \begin{cases} 1 & \text{if } fueltype = gas \\ 0 & \text{otherwise} \end{cases}$$

If we fit the model:

$$mpg = \beta_0 + \beta_1 horsepower + \beta_2 fueltype[T.gas]$$

We obtain an $R^2 = 0.661$ with $Prob(F - statistic) \approx 0$ and the following estimates for the regression parameters:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	41.2379	1.039	39.705	0.000	39.190	43.286
fueltype[T.gas]	-2.7658	0.918	-3.013	0.003	-4.576	-0.956
horsepower	-0.1295	0.007	-18.758	0.000	-0.143	-0.116

How do we interpret this result?

- The value of `mpg` when `horsepower=0` `fueltype=diesel` (i.e., `fueltype[T.gas]=0`) is 41.2379 ;
- An increase of one unit of `horsepower` is associated to a decrease of 0.1295 units of `mpg` provided that `fueltype=diesel`;
- For gas vehicles we expect to see a decrease of `mpg` equal to 2.7658 with respect to diesel vehicles.

Predictors with More than Two Levels

When predictors have n levels, we need to introduce **multiple dummy variables**. Specifically, we need to introduce $n - 1$ binary variables. For instance, if the levels of the variable `income` are `low`, `medium` and `high`, we could introduce two variables `income[T.low]` and `income[T.medium]`. These are sufficient to express all possible values of `income` as shown in the table below:

`income`	`income[T.low]`	`income[T.medium]`
`low`	1	0
`medium`	0	1
`high`	0	0

Note that we could have introduced a new variable `income[T.high]` but this would have been redundant and so **correlated to the other two variables**, which is something we know we have to avoid in linear regression.

If we fit the model which predicts `mpg` from `horsepower` and `fuelsystem` we obtain $R^2 = 0.734$, $Prob(F - statistic) \approx 0$ and the following estimates for the regression coefficients:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	38.8638	1.234	31.504	0.000	36.431	41.297
fuelsystem[T.2bbl]	-1.6374	1.127	-1.453	0.148	-3.860	0.585
fuelsystem[T.4bbl]	-12.0875	2.263	-5.341	0.000	-16.551	-7.624
fuelsystem[T.idi]	-0.3894	1.300	-0.299	0.765	-2.954	2.175
fuelsystem[T.mfi]	-5.8285	3.661	-1.592	0.113	-13.049	1.392
fuelsystem[T.mpfi]	-5.4942	1.202	-4.570	0.000	-7.865	-3.123
fuelsystem[T.spdi]	-5.2446	1.612	-3.254	0.001	-8.423	-2.066
fuelsystem[T.spfi]	-6.1522	3.615	-1.702	0.090	-13.282	0.978
horsepower	-0.0968	0.009	-11.248	0.000	-0.114	-0.080

As we can see, we have added different correlation coefficients in order to deal with the different levels. Not all predictors have a low p-value, so we can remove those with backward elimination. We will see some more examples in the laboratory.

Linear Regression in Python

In this notebook, we will perform a complete statistical linear regression analysis. Our goal is **understanding** and **inference**. We want to find a model that is not only accurate but also **parsimonious** (simple) and **statistically valid**.

We will use the “Auto MPG” dataset and the `statsmodels` library.

Our process will be:

1. **Simple Linear Regression (SLR):** Fit a simple model and diagnose its problems.
2. **Multiple Linear Regression (MLR):** Fit a “full” model with all potential predictors.
3. **Variable Selection:** Use **Backward Elimination** based on p-values to simplify the model.
4. **Final Model Interpretation:** Interpret the coefficients of our final, parsimonious model.
5. **Diagnostic Plots:** Check the final model’s assumptions to ensure our conclusions are reliable.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Import statsmodels
# smf allows us to use the "formula API" (e.g., 'y ~ x')
# sm gives us access to more functions, like diagnostic plots
import statsmodels.api as sm
import statsmodels.formula.api as smf


# Load and Clean the Data
mpg = sns.load_dataset('mpg')

# Clean the data
mpg = mpg.drop('name', axis=1)
mpg['horsepower'] = pd.to_numeric(mpg['horsepower'], errors='coerce')

# Drop rows with missing values for this exercise
mpg_clean = mpg.dropna()

print(f"Shape of cleaned dataset: {mpg_clean.shape}")
mpg_clean.head()
```

```
Shape of cleaned dataset: (392, 8)
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin
0	18.0	8	307.0	130.0	3504	12.0	70	usa
1	15.0	8	350.0	165.0	3693	11.5	70	usa
2	18.0	8	318.0	150.0	3436	11.0	70	usa
3	16.0	8	304.0	150.0	3433	12.0	70	usa
4	17.0	8	302.0	140.0	3449	10.5	70	usa

Simple Linear Regression

Let's start with the simplest model: **"Can `horsepower` predict `mpg`?"**

We use the formula `mpg ~ horsepower`. `statsmodels` automatically adds the intercept (β_0).

```
# Define and fit the SLR model
# OLS stands for "Ordinary Least Squares"
model_slr = smf.ols('mpg ~ horsepower', data=mpg_clean).fit()

# Print the full summary table
print(model_slr.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.606
Model:                            OLS   Adj. R-squared:                  0.605
Method:                 Least Squares   F-statistic:                     599.7
Date:                Sat, 26 Sep 2026   Prob (F-statistic):           7.03e-81
Time:                        16:16:04   Log-Likelihood:                -1178.7
No. Observations:                 392   AIC:                             2361.
Df Residuals:                     390   BIC:                             2369.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     39.9359      0.717     55.660      0.000      38.525      41.347
horsepower    -0.1578      0.006    -24.489      0.000      -0.171      -0.145
==============================================================================
Omnibus:                       16.432   Durbin-Watson:                   0.920
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               17.305
Skew:                           0.492   Prob(JB):                     0.000175
Kurtosis:                       3.299   Cond. No.                         322.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

The `.summary()` output is dense, but it's the heart of statistical analysis. Let's break it into three parts:

Part 1: The Top Table (Model Metadata)

Field	Meaning
`R-squared`	**0.606** (or 60.6%) - As we saw, this is the "proportion of variance explained." It tells us `horsepower` alone explains 60.6% of the variability in `mpg`.
`Adj. R-squared`	**0.605** - A "corrected" version of R^2 that penalizes for adding useless variables. In SLR, it's almost identical to R^2.
`Method:`	`Least Squares` - The method we used (OLS).
`No. Observations:`	**392** - The number of rows (n) used to build the model.
`F-statistic:`	**599.7** - A score that tests the *overall* significance of the model.
`Prob (F-statistic):`	**7.03e-81** - A p-value associated with the F-statistic. This is a tiny number ($0.000\ldots$ with 80 zeros). Since it is < 0.05, it tells us our model as a whole is statistically significant (i.e., it's far better than a null model that just guesses the mean).

Part 2: The Coefficients Table (The Core)

Field	`Intercept` (β_0)	`horsepower` (β_1)
`coef`	**39.9359**	**-0.1578**
`std err`	0.717	0.006
`t`	55.660	-24.489
`P>|t|`	**< 0.001**	**< 0.001**
`[0.025 ... 0.975]`	[38.525, 41.347]	[-0.170, -0.146]

Interpreting the Coefficients

- `Intercept (coef)`: 39.93. This is our β_0. Theoretically, it's the predicted value of `mpg` when `horsepower` is 0. (In this context, it has no real practical interpretation, but it's mathematically necessary).
- `horsepower (coef)`: -0.1578. This is our β_1, the slope. It's the most important part.
  - **Interpretation:** "For every **1-unit increase** in `horsepower`, we expect a **0.1578 decrease** in `mpg`, holding all else constant."

Interpreting the Significance (p-value)

- `horsepower (P>|t|)`: This is `< 0.001`. This is the p-value for the null hypothesis $H_0 : \beta_1 = 0$.
  - **Conclusion:** Because the p-value is much smaller than $\alpha = 0.05$, we **reject the null hypothesis**. We can conclude that **there is a statistically significant relationship** between `horsepower` and `mpg`.

Interpreting the Confidence Interval (CI):

- `horsepower ([0.025 ... 0.975])`: [-0.170, -0.146]
  - **Interpretation:** "We are 95% confident that the *true* value of the slope (β_1) in the population lies between -0.170 and -0.146."
  - **Note:** The interval **does not include 0**, which confirms our conclusion from the p-value.

Part 3: The Bottom Table (Residual Diagnostics)

This table provides tests to check if the residuals (ϵ) meet the regression assumptions (like normality).

Residual Diagnostics

The coefficients table (p-values, CIs, etc.) is **only valid if** our model meets the **assumptions of linear regression**. The two main checks we must perform are:

1. **Linearity and Homoscedasticity:** Residuals should be randomly scattered around 0 with no clear pattern (e.g., no "U-shape" or "funnel").
2. **Normality:** The residuals should follow a normal distribution.

We use diagnostic plots to check these.

Interpreting the SLR Summary

- `Adj. R-squared`: 0.605. `horsepower` alone explains about 60.5% of the variance in `mpg`.
- `Prob (F-statistic)`: 7.03e-81. This is nearly zero, so the model is statistically significant (better than just guessing the mean).
- `horsepower coef` (β_1): -0.1578.

- `horsepower P>|t|`: < 0.001. The p-value is tiny, so we are confident there is a real, statistically significant relationship between `horsepower` and `mpg`.

This looks good, but is the model *valid*? Let's check the diagnostic plots.

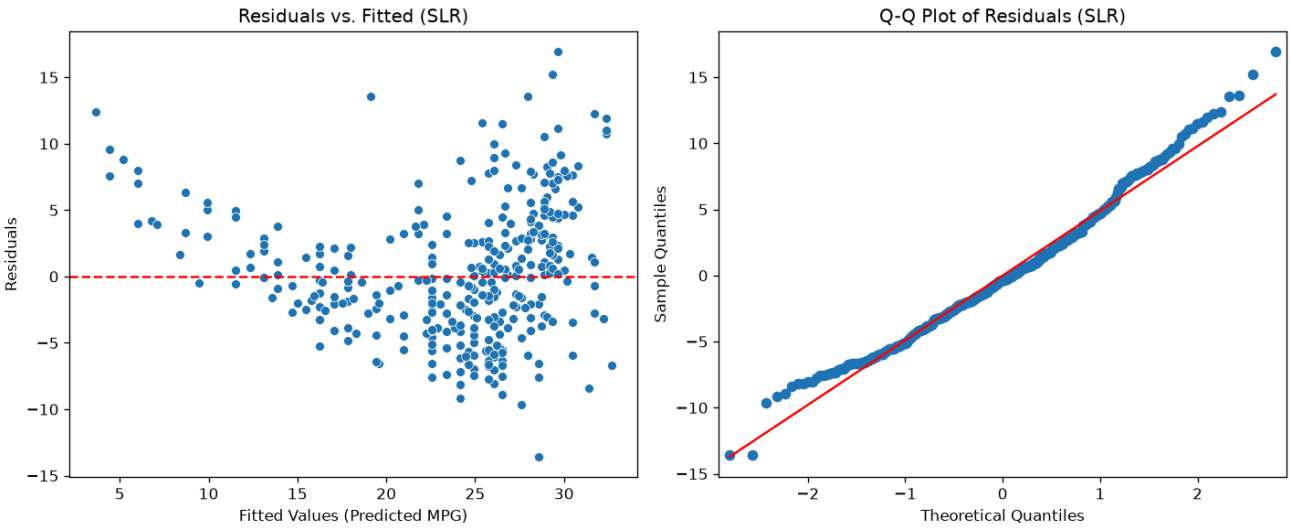
```
# --- SLR Diagnostic Plots ---
fitted_values_slr = model_slr.fittedvalues
residuals_slr = model_slr.resid

plt.figure(figsize=(12, 5))

# 1. Residuals vs. Fitted Plot
ax1 = plt.subplot(1, 2, 1)
sns.scatterplot(x=fitted_values_slr, y=residuals_slr, ax=ax1)
ax1.axhline(0, color='red', linestyle='--')
ax1.set_title('Residuals vs. Fitted (SLR)')
ax1.set_xlabel('Fitted Values (Predicted MPG)')
ax1.set_ylabel('Residuals')

# 2. Q-Q Plot
ax2 = plt.subplot(1, 2, 2)
sm.qqplot(residuals_slr, line='s', ax=ax2) # 's' = standardized line
ax2.set_title('Q-Q Plot of Residuals (SLR)')

plt.tight_layout()
plt.show()
```



SLR Diagnostic Analysis

1. **Residuals vs. Fitted (Left):** This is a **problem**. The plot shows a clear **"U-shape"** (a non-linear pattern). This means our assumption of a *linear* relationship is wrong. Our model is **underfitting**.
2. **Q-Q Plot (Right):** The residuals also peel away from the line at the tails, suggesting the errors are not perfectly normally distributed.

Conclusion: The SLR model is simple and significant, but it's **fundamentally flawed** because it fails the linearity assumption. We need a more complex model.

Multiple Linear Regression - The "Full" Model

Let's build a "full" model that includes all potential predictors. This will be our starting point for backward elimination.

Our formula will be: `mpg ~ cylinders + displacement + horsepower + weight + acceleration + model_year + C(origin)`

Handling Categorical Variables: `C(origin)`

The `origin` column is categorical (USA, Europe, Japan). If we just add `origin` to the model, `statsmodels` will treat it as a number, which is wrong.

By wrapping it in `C()`, we tell `statsmodels` to treat it as a **C**ategorical variable. It will automatically:

1. Choose a "reference" level (by default, the lowest one, which in our case is Europe).
2. Create "dummy variables" for the other levels.

The output will show coefficients for `C(origin)[T.japan]` and `C(origin)[T.usa]`. These coefficients represent the *difference in mpg* compared to the Europe, holding all other variables constant.

```
# Define and fit the "Full" MLR model
all_features = 'mpg ~ cylinders + displacement + horsepower + weight + acceleration +
        model_year + C(origin)'
model_full = smf.ols(all_features, data=mpg_clean).fit()

# Print the full summary table
print(model_full.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.824
Model:                            OLS   Adj. R-squared:                  0.821
Method:                 Least Squares   F-statistic:                     224.5
Date:                Sat, 26 Sep 2026   Prob (F-statistic):          1.79e-139
Time:                        16:16:04   Log-Likelihood:                -1020.5
No. Observations:                 392   AIC:                             2059.
Df Residuals:                     383   BIC:                             2095.
Df Model:                           8
Covariance Type:            nonrobust
===============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept     -15.3246      4.602     -3.330      0.001     -24.374      -6.276
C(origin)[T.japan]     0.2232      0.566      0.394      0.694      -0.890      1.336
C(origin)[T.usa]      -2.6300      0.566     -4.643      0.000      -3.744      -1.516
cylinders             -0.4897      0.321     -1.524      0.128      -1.121      0.142
displacement           0.0240      0.008      3.133      0.002       0.009      0.039
horsepower            -0.0182      0.014     -1.326      0.185      -0.045      0.009
weight                -0.0067      0.001    -10.243      0.000      -0.008      -0.005
acceleration           0.0791      0.098      0.805      0.421      -0.114      0.272
model_year             0.7770      0.052     15.005      0.000       0.675      0.879
==============================================================================
Omnibus:                       23.395   Durbin-Watson:                   1.291
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               34.452
Skew:                           0.444   Prob(JB):                     3.30e-08
Kurtosis:                       4.150   Cond. No.                     8.56e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 8.56e+04. This might indicate that there are strong multicollinearity or other numerical problems.
```

Interpreting the Full Model

- `Adj. R-squared`: 0.818. A massive improvement! Our model now explains 81.8% of the variance.
- **Multicollinearity Warning:** `statsmodels` warns us: `The condition number is large... This might indicate that there is strong multicollinearity`.
- **P-Values:** Look at the `P>|t|` column. We see several variables with high p-values (>0.05):
  - `C(origin)[T.japan]` (p=0.694)
  - `cylinders` (p=0.128)
  - `horsepower` (p=0.185)
  - `acceleration` (p=0.421)

These variables are not statistically significant *in this specific model*. This is due to **multicollinearity**. `cylinders`, `horsepower`, and `acceleration` are all measuring the same thing (engine power) and are redundant.

Backward Elimination

Our full model is bloated and unreliable. We will use **Backward Elimination** to simplify it, creating a "parsimonious" model that is both accurate and interpretable.

The Process:

1. Fit the full model.
2. Find the predictor with the **highest p-value** > 0.05.
3. Remove that predictor.
4. Refit the model and go back to step 2.
5. Stop when all remaining predictors have $p < 0.05$.

As a first step, we should remove `C(origin)[T.japan]` (p=0.694). However, this is one of the levels of `origin`, so we cannot just remove it. Instead, we can specify a different reference level than the default one to see if this improves things:

```
# Define and fit the "Full" MLR model
all_features = 'mpg ~ cylinders + displacement + horsepower + weight + acceleration +
        model_year + C(origin, Treatment(reference="usa"))'
model_full = smf.ols(all_features, data=mpg_clean).fit()

# Print the full summary table
print(model_full.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.824
Model:                            OLS   Adj. R-squared:                  0.821
Method:                 Least Squares   F-statistic:                     224.5
Date:                Sat, 26 Sep 2026   Prob (F-statistic):          1.79e-139
Time:                        16:16:04   Log-Likelihood:                -1020.5
No. Observations:                 392   AIC:                             2059.
Df Residuals:                     383   BIC:                             2095.
Df Model:                           8
Covariance Type:            nonrobust
================================================================================
================================
                                                   coef    std err          t
P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
--------------------------------
Intercept                                       -17.9546      4.677     -3.839
0.000     -27.150      -8.759
C(origin, Treatment(reference="usa"))[T.europe]     2.6300      0.566      4.643
0.000       1.516       3.744
C(origin, Treatment(reference="usa"))[T.japan]      2.8532      0.553      5.162
0.000       1.766       3.940
cylinders                                       -0.4897      0.321     -1.524
0.128      -1.121       0.142
displacement                                     0.0240      0.008      3.133
0.002       0.009       0.039
horsepower                                      -0.0182      0.014     -1.326
0.185      -0.045       0.009
weight                                          -0.0067      0.001    -10.243
0.000      -0.008      -0.005
acceleration                                     0.0791      0.098      0.805
0.421      -0.114       0.272
model_year                                       0.7770      0.052     15.005
0.000       0.675       0.879
==============================================================================
Omnibus:                       23.395   Durbin-Watson:                   1.291
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               34.452
Skew:                           0.444   Prob(JB):                     3.30e-08
Kurtosis:                       4.150   Cond. No.                     8.70e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 8.7e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Great, this solved the problem for `origin`! We note that we have the same adjusted R-squared value, so we did not harm or improve the regressor.

We now have the following high p-values:

- `cylinders` (p=0.128)
- `horsepower` (p=0.185)
- `acceleration` (p=0.421)

Let's remove `acceleration` next:

```
# Define and fit the "Full" MLR model
all_features = 'mpg ~ cylinders + displacement + horsepower + weight + model_year +
        C(origin, Treatment(reference="usa"))'
model_full = smf.ols(all_features, data=mpg_clean).fit()

# Print the full summary table
print(model_full.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.824
Model:                            OLS   Adj. R-squared:                  0.821
Method:                 Least Squares   F-statistic:                     256.7
Date:                Sat, 26 Sep 2026   Prob (F-statistic):          1.49e-140
Time:                        16:16:04   Log-Likelihood:                -1020.8
No. Observations:                 392   AIC:                             2058.
Df Residuals:                     384   BIC:                             2089.
Df Model:                           7
Covariance Type:            nonrobust
================================================================================
================================
                                                   coef    std err          t
P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
--------------------------------
Intercept                                       -16.3323      4.219     -3.871
0.000     -24.628      -8.037
C(origin, Treatment(reference="usa"))[T.europe]     2.6345      0.566      4.654
0.000       1.521       3.748
C(origin, Treatment(reference="usa"))[T.japan]      2.8574      0.552      5.172
0.000       1.771       3.944
cylinders                                       -0.5028      0.321     -1.568
0.118      -1.133       0.128
displacement                                     0.0234      0.008      3.070
0.002       0.008       0.038
horsepower                                      -0.0250      0.011     -2.320
0.021      -0.046      -0.004
weight                                          -0.0065      0.001    -11.209
0.000      -0.008      -0.005
model_year                                       0.7739      0.052     14.994
0.000       0.672       0.875
==============================================================================
Omnibus:                       25.943   Durbin-Watson:                   1.290
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               39.879
Skew:                           0.468   Prob(JB):                     2.19e-09
Kurtosis:                       4.251   Cond. No.                     7.86e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 7.86e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The adjusted R-squared is the same, but we now have the following high p-values:

- `cylinders` (p=0.118)

Let's remove `cylinders`:

```
# Define and fit the "Full" MLR model
all_features = 'mpg ~  displacement + horsepower + weight + model_year + C(origin,
        Treatment(reference="usa"))'
model_full = smf.ols(all_features, data=mpg_clean).fit()

# Print the full summary table
print(model_full.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.823
Model:                            OLS   Adj. R-squared:                  0.820
Method:                 Least Squares   F-statistic:                     297.9
Date:                Sat, 26 Sep 2026   Prob (F-statistic):          2.80e-141
Time:                        16:16:04   Log-Likelihood:                -1022.0
No. Observations:                 392   AIC:                             2058.
Df Residuals:                     385   BIC:                             2086.
Df Model:                           6
Covariance Type:            nonrobust
================================================================================
================================
                                                 coef    std err          t
P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
--------------------------------
Intercept                                    -17.5036      4.160     -4.207
0.000     -25.683      -9.324
C(origin, Treatment(reference="usa"))[T.europe]     2.5958      0.567      4.581
0.000       1.482       3.710
C(origin, Treatment(reference="usa"))[T.japan]      2.7722      0.551      5.033
0.000       1.689       3.855
displacement                                 0.0155      0.006      2.699
0.007       0.004       0.027
horsepower                                  -0.0230      0.011     -2.149
0.032      -0.044      -0.002
weight                                      -0.0066      0.001    -11.449
0.000      -0.008      -0.005
model_year                                   0.7749      0.052     14.986
0.000       0.673       0.877
==============================================================================
Omnibus:                       24.729   Durbin-Watson:                   1.275
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               37.452
Skew:                           0.455   Prob(JB):                     7.37e-09
Kurtosis:                       4.211   Cond. No.                     7.73e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 7.73e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The adjusted R-squared is a bit lower, but all p-values are below the threshold now!

Final Model Interpretation

Now we have a **final, parsimonious model**.

- `Adj. R-squared`: 0.820. We removed two variables and our model's explanatory power is *still* just as high.
- **All p-values are < 0.05**. We can confidently interpret every coefficient.

Final Coefficient Analysis:

- `weight`: -0.0066. "For every 1lb increase in `weight`, `mpg` decreases by 0.0066, holding all other variables constant. (This is the net effect of increasing weight).
- `model_year`: 0.7749. "For each newer `model_year`, `mpg` increases by 0.78, holding all other variables constant. (Cars get more efficient over time).
- `C(origin)[T.europe]`: 2.5958. "A car from `Europe` gets **2.6 more mpg** than a car from the `USA`, holding all other variables constant".
- `C(origin)[T.japan]`: 2.7722. "A car from `Japan` gets **2.77 more mpg** than a car from the `USA`, holding all other variables constant".

Final Diagnostic Check

Our p-values and coefficients are only reliable if this *final* model is valid. Let's check the residuals.

```
# --- SLR Diagnostic Plots ---
fitted_values_slr = model_slr.fittedvalues
residuals_slr = model_slr.resid

plt.figure(figsize=(12, 5))

# 1. Residuals vs. Fitted Plot
ax1 = plt.subplot(1, 2, 1)
sns.scatterplot(x=fitted_values_slr, y=residuals_slr, ax=ax1)
ax1.axhline(0, color='red', linestyle='--')
ax1.set_title('Residuals vs. Fitted (Simple Linear Regression)')
ax1.set_xlabel('Fitted Values (Predicted MPG)')
ax1.set_ylabel('Residuals')
ax1.set_ylim(-20, 20)

# 2. Q-Q Plot
ax2 = plt.subplot(1, 2, 2)
sm.qqplot(residuals_slr, line='s', ax=ax2) # 's' = standardized line
ax2.set_title('Q-Q Plot of Residuals (Simple Linear Regression)')

plt.tight_layout()
plt.show()

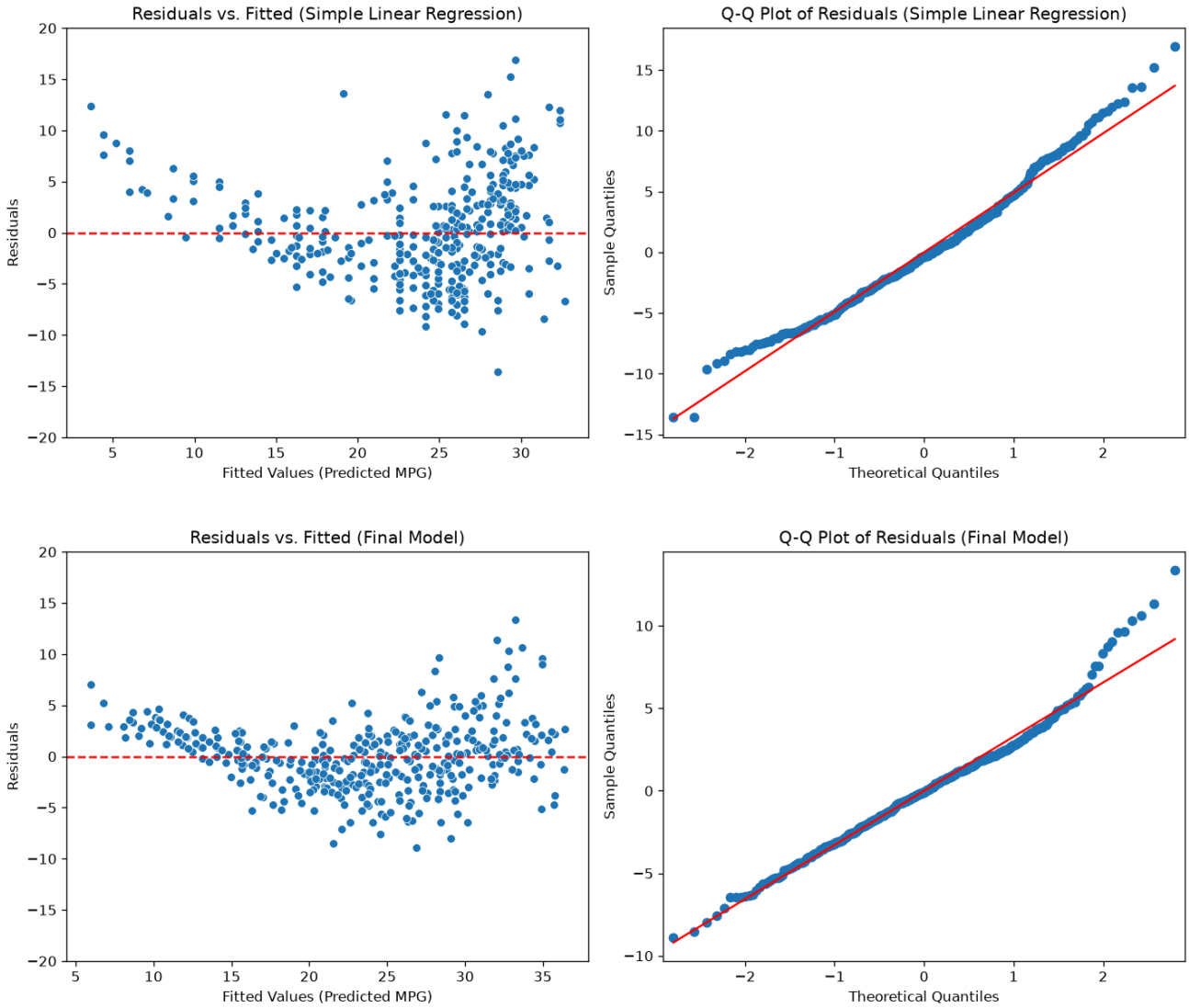
# --- Final Model Diagnostic Plots ---
fitted_values_final = model_full.fittedvalues
residuals_final = model_full.resid

plt.figure(figsize=(12, 5))

# 1. Residuals vs. Fitted Plot
ax1 = plt.subplot(1, 2, 1)
sns.scatterplot(x=fitted_values_final, y=residuals_final, ax=ax1)
ax1.axhline(0, color='red', linestyle='--')
ax1.set_title('Residuals vs. Fitted (Final Model)')
ax1.set_xlabel('Fitted Values (Predicted MPG)')
ax1.set_ylabel('Residuals')
ax1.set_ylim(-20, 20)

# 2. Q-Q Plot
ax2 = plt.subplot(1, 2, 2)
sm.qqplot(residuals_final, line='s', ax=ax2)
ax2.set_title('Q-Q Plot of Residuals (Final Model)')


plt.tight_layout()
plt.show()
```



1. **Residuals vs. Fitted (Left):** This is a bit better. The problematic "U-shape" is still there but less pronounced.
2. **Q-Q Plot (Right):** This also looks a bit improved, with the points falling closer to the normal-distribution line in the left part of the plot.

Exercises

Exercise 1

Consider the Titanic dataset. Calculate a linear regressor that predicts `Fare` values from `Survived`, `Pclass`, `Sex`, and `Age` values. Insert dummy variables where appropriate. Is the obtained regressor a good regressor? Which variables contribute significantly to the regression? Are there irrelevant variables? Eliminate these variables using the backward elimination technique. Discuss the meaning of the identified coefficients.

Exercise 2

Consider the Titanic dataset. Calculate a linear regressor that predicts `Age` values from `Survived`, `Pclass`, `Sex`, and `Fare` values. Insert dummy variables where appropriate. Is the obtained regressor a good regressor? Is it a better or worse regressor than the one calculated in the previous exercise? Which variables contribute significantly to the regression? Are there irrelevant variables? Eliminate such variables using the backward elimination technique. Discuss the meaning of the identified coefficients.

References

- Chapter 3 of [1]
- Parts of chapter 11 of [2]

[1] Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.

[2] James, Gareth Gareth Michael. An introduction to statistical learning: with applications in Python, 2023.https://www.statlearning.com

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Logistic regression

Chapter resources

Slides (PDF) Open in Colab Practice quiz

We have seen how K-NN allows to perform classification in a **non-parametric way**. However, it has some short-comings:

- It doesn't work with many features;
- It requires memorization of the training set;
- Each classification requires a search of the K-nearest neighbors in the training set.

Compare this to a parametric predictive model such as the linear regression:

$$f(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$$

Parametric models are fast and compact - once we train them, we don't need to save the training data and they can be used to classify new examples in a very natural way.

While linear regression is a powerful parametric models, it allows to model relationships between **continuous independent and dependent variables** and **between qualitative independent variables and continuous variables**. However, it does not allow to model relationships between continuous or qualitative independent variables and **qualitative dependent variables**.

Example Data

Establishing such relationships is useful in different contexts. For instance, let us consider the Breast Cancer Wisconsin dataset:

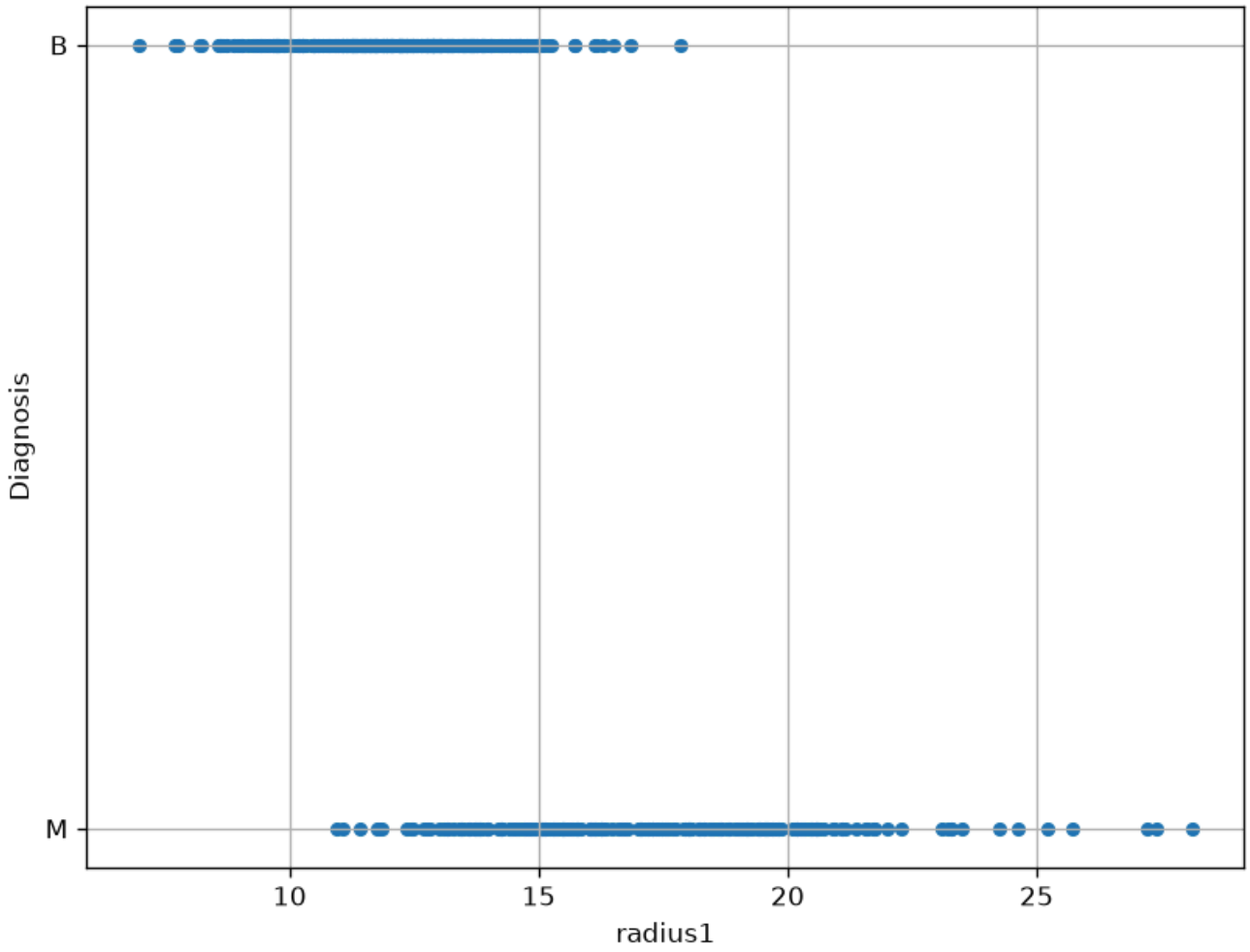
	radius1	texture1	perimeter1	area1	smoothness1	compactness1	concavity1	concave_points1	symmetr
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809
...	...	...	...	...	...	...	...	...	...
564	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	0.1726
565	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	0.1752
566	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	0.1590
567	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	0.2397
568	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	0.1587

569 rows × 31 columns

The dataset contains several measurements of given quantities measured from digitized image of a fine needle aspirate (FNA) of a breast mass, together with a categorical variable `Diagnosis` with two levels: `M` (malignant) and `B` (benign).

In this case, it would be good to be able to study whether a relationship exists between some of the considered independent variables and the dependent variable or simply to classify an example in one of the two classes from new observation.

We will consider the `radius1` variable for the moment. Let us plot this variable with respect to `Diagnosis`:



From the plot above, we can note that there is some form of relationship between the two variables. Indeed:

- For low values of `radius1`, we tend to have more benign cases;
- For large values of `radius1`, we tend to have more malignant cases.

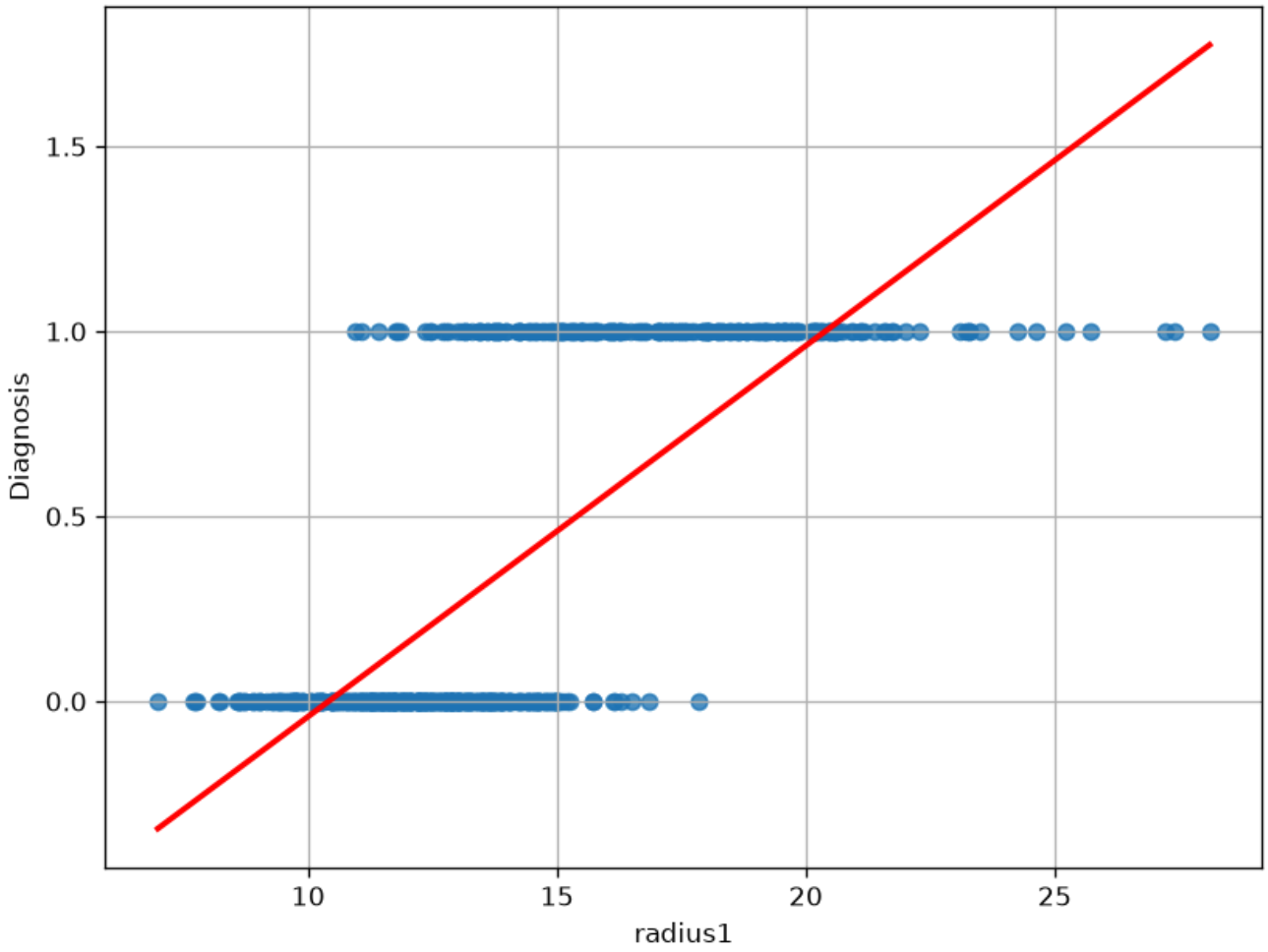
Limits of Linear Regression

Of course, we would like to quantify this relationship in a more formal way. **As in the case of a linear regressor, we want to define a model which can predict the independent variable y from the dependent variables x_i. If such model gives good predictions, than we can trust its interpretation as a means of studying the relationship between the variables.**

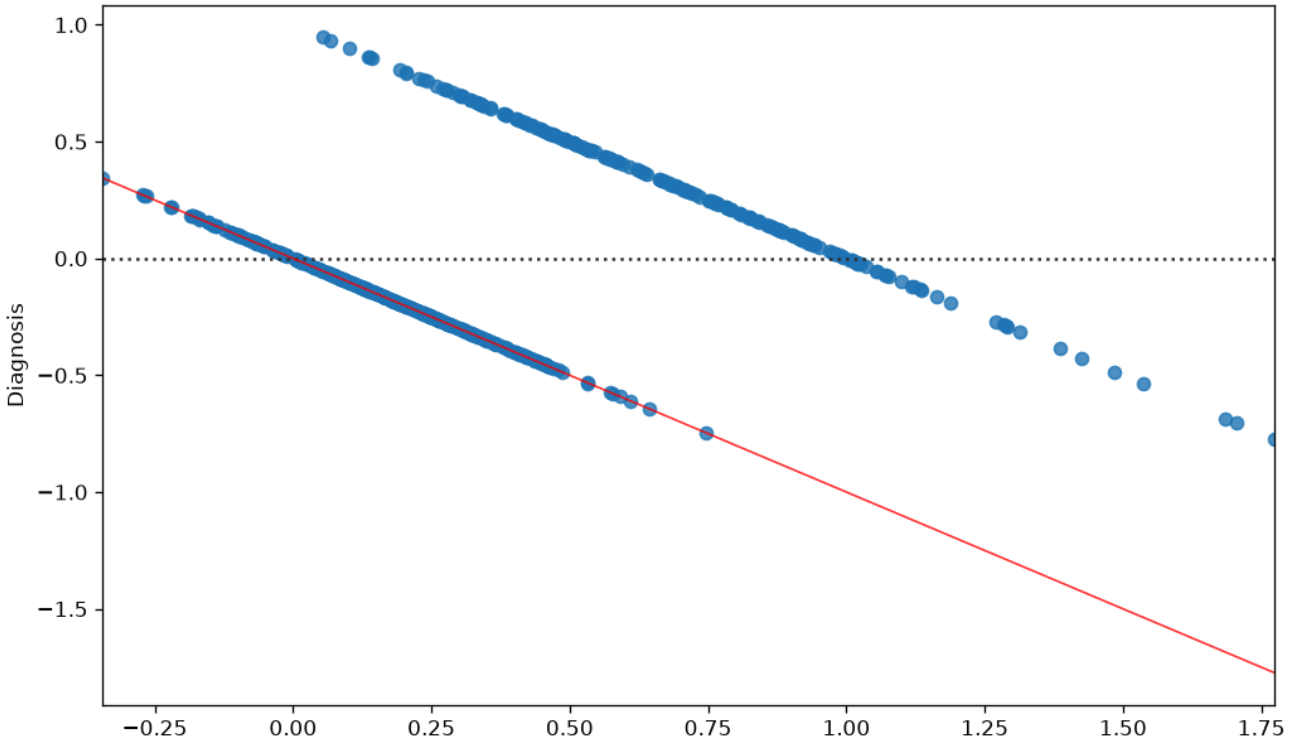
We can think of converting `B ⇒ 1` and `M ⇒ 0`, and then compute a linear regressor:

$$Diagnosis = \beta_0 + \beta_1 radius1$$

This would be the result:



We can immediately see that this function does not model the relationship between the two variables very well. While we obtain a statistically relevant regressor with $R^2 = 0.533$ and statistically relevant coefficients, the residual plot will look like this:



The correlation between the residuals and the independent variable is a strong indication that the true relationship between the two variables is not correctly modeled. After all, from a purely predictive point of view, we are using a linear regressor which takes the form:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

while the values `Diagnosis` variable belong to the set $\{0, 1\}$ and we would need instead a function with the following form:

$$f : \mathbb{R} \rightarrow \{0, 1\}$$

However, the linear regressor cannot directly predict **discrete values**.

In practice, while with a linear regressor we wanted to predict continuous values, now we want to assign observations $\mathbf{x}$ to discrete bins (in this case only two possible ones). As we will better study later in the course, this problem is known as classification.

From Binary Values to Probabilities

If we want to model some form of continuous value, we could think to transition from $\{0, 1\}$ to $[0, 1]$ using probabilities, which is a way to turn discretized values to "soft" values indicating our belief in the fact that `Diagnosis` will take either a 0 or 1 value. We could hence think to model the following probability, rather than modeling `Diagnosis` directly:

$$P(Diagnosis = 1 | radius1)$$

Note that modeling that probability directly is what **discriminative models do**.

However, even in this case, a model of the form:

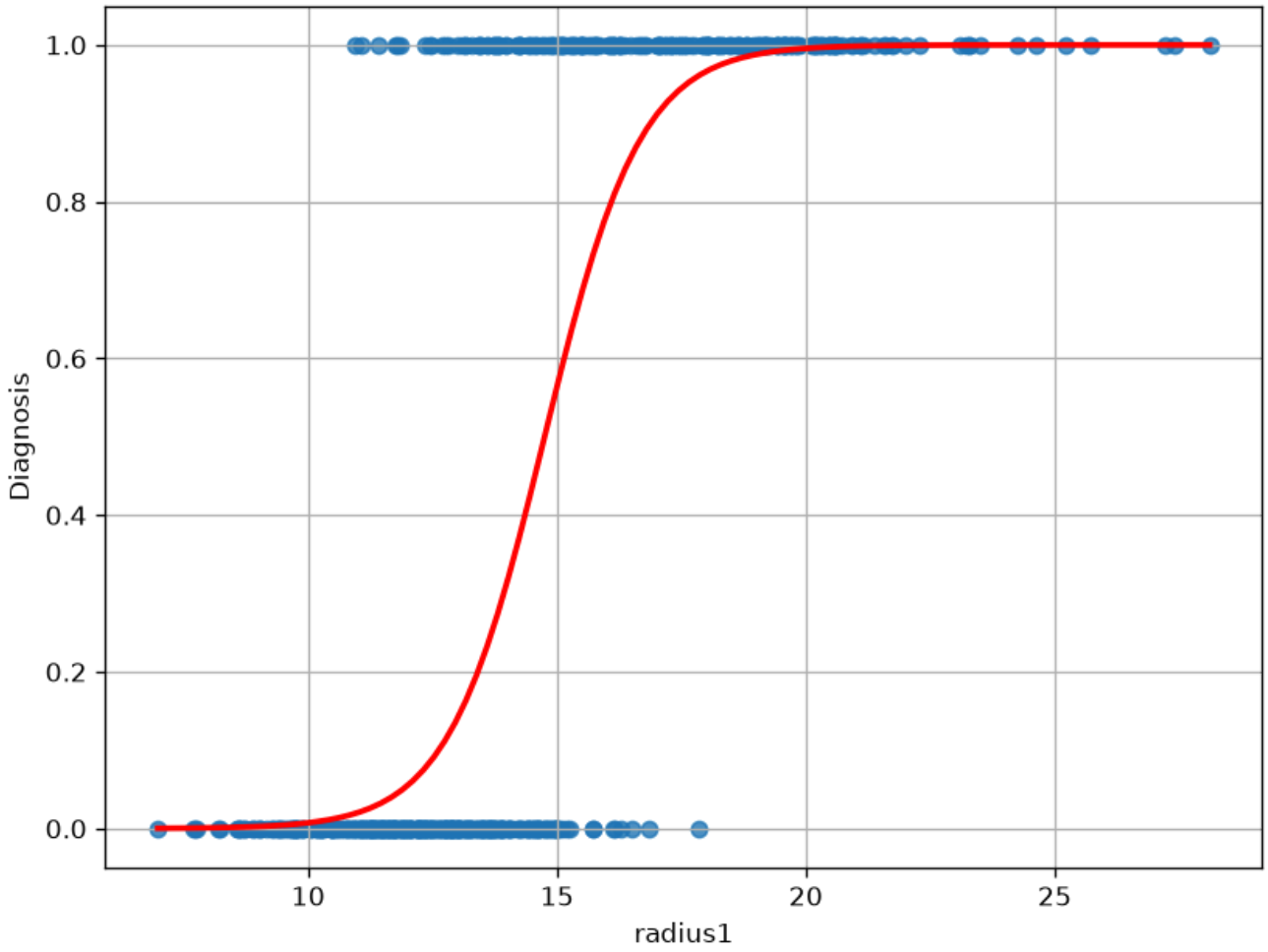
$$P(Diagnosis = 1 | radius1) = \beta_0 + \beta_1 radius1$$

Would not be appropriate. Indeed, while $P(Diagnosis = 1 | radius1)$ needs to be in the $[0, 1]$ range, the linear combination $\beta_0 + \beta_1 radius1$ will naturally output values **smaller than** 0 and **larger than** 1. How should we interpret such values?

Intuitively, we would expect to $P(Diagnosis = 1 | radius1)$ to assume values in the $[0, 1]$ range for intermediate values (say `radius` $\in [10, 20]$), while for extremely low values of (say `radius` < 10) the probability **should saturate to** 0 and for extremely large values (say `radius` > 20) the probability should saturate to 1.

When `radius1` takes large values (say larger than 20), we expect **probability to saturate to** 1.

In practice, we would expect a result similar to the following:



As can be noted, the function above is not linear, and hence it cannot be fit with a linear regressor. However, we have seen that a linear regressor can be tweaked to also represent nonlinear functions.

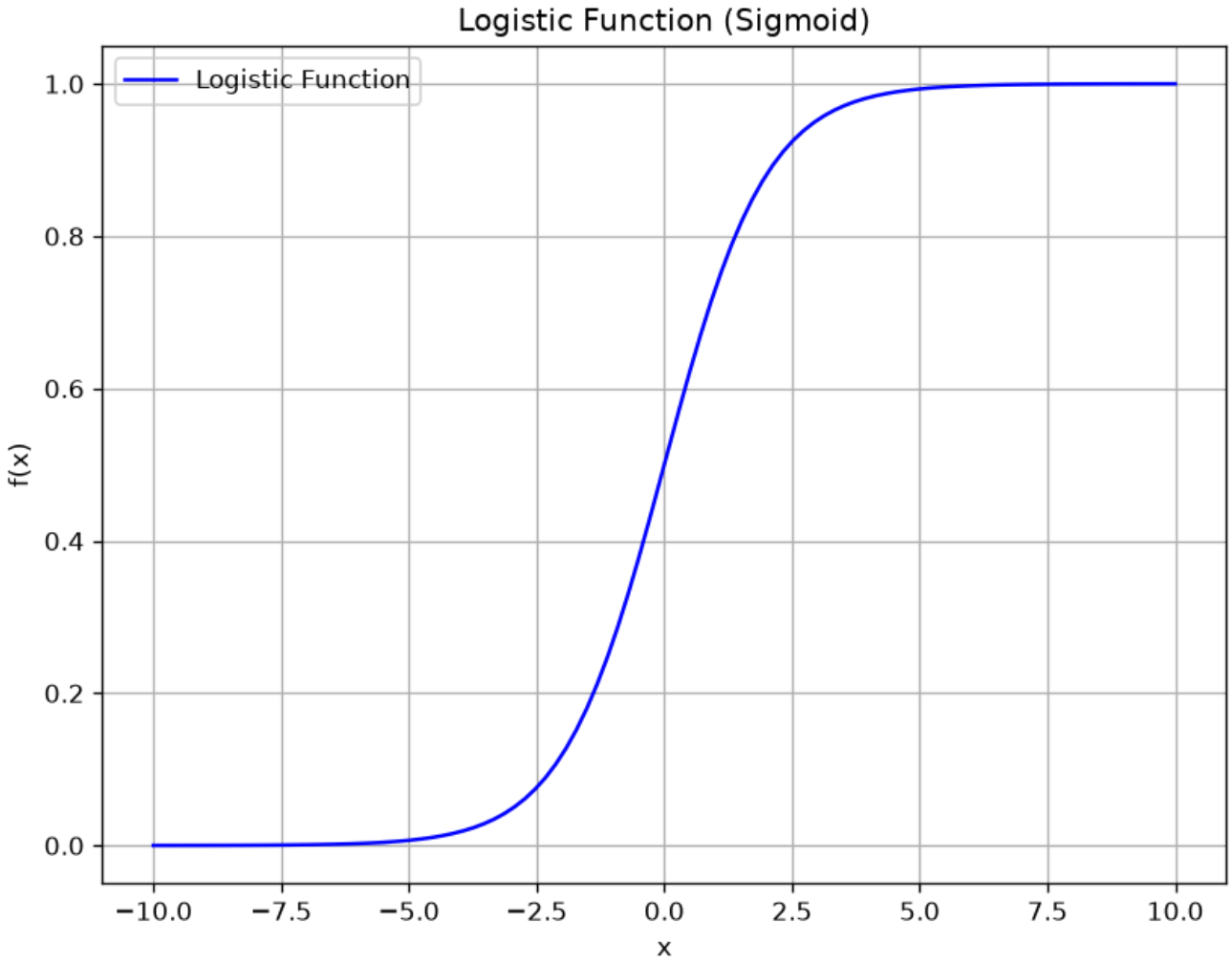
The Logistic Function

We need to find a **transformation of the formulation of the linear regressor to transform its output into a nonlinear function of the independent variables**. Of course, we do not want *any* transformation, but one that has the previously highlighted properties.

In practice the **logistic function has some nice properties that, as we will se in a moment, allow to easily interpret the resulting model in a probabilistic way**. The logistic function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

and has the following shape:



As we can see, the function has the properties we need:

- Its values are comprised between 0 and 1;
- It saturates to 0 and 1 for extreme values of x.

Additionally, it is differentiable, which will be useful for optimization later.

The Logistic Regression Model

In practice, we define our model, **the logistic regressor model** as follows (**simple logistic regression**):

$$P(Diagnosis = 1|X) = f(\beta_0 + \beta_1 X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

Or, more in general (**multiple logistic regression**):

$$P(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n)}}$$

It is easy to see that:

$$p = \frac{1}{1+e^{-x}} \Rightarrow p + pe^{-x} = 1 \Rightarrow pe^{-x} = 1 - p \Rightarrow e^{-x} = \frac{1-p}{p} \Rightarrow e^{x} = \frac{p}{1-p}$$

Hence:

$$e^{\beta_0+\beta_1 x_1+\ldots+\beta_n x_n} = \frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})}$$

The term on the right is called the odd of $P(y=1|\mathbf{x})$.

The odd of $P(y=1|\mathbf{x})$ is the number of times we believe the example will be positive (observed $\mathbf{x}$) over the number of times we believe the example will be negative.

For instance, if we believe that the example will be positive 3 times out of 10, then the odd will be $\frac{3}{7}$.

By taking the logarithm of both terms, we obtain:

$$\log\left(\frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$$

The expression:

$$\log\left(\frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})}\right)$$

Is the logarithm of the odd (log odd), and it is called **logit**, hence the **logistic regression** is sometimes called **logit regression**.

The expression above shows how a logistic regressor can be seen as **a linear regressor (the expression on the right side of the equation) on the logit (the log odd)**. This paves the way to useful interpretations of the model, as shown in the next section.

Estimation of the Parameters of a Logistic Regressor

To fit the model and find suitable values for the $\beta_\mathbf{i}$ parameters, we will define a **cost function**, similarly to what we have done in the case of linear regression.

Even if we can see the logistic regression problem as the linear regression problem of fitting the $logit(p) = \beta^T\mathbf{x}$ function, differently from linear regression, **we should note that we do not have the ground truth probabilities p**. Indeed, our observations only provide input examples $\mathbf{x}^{(i)}$ and the corresponding labels $y^{(i)}$.

Starting from the definition:

$$P\left(y=1|\mathbf{x};\beta\right)=f_{\beta}\left(\mathbf{x}\right)=\frac{1}{1+e^{-\beta^{T}\mathbf{x}}}=\sigma(\beta^{T}\mathbf{x})$$

We can write:

$$P\left(y=1|\mathbf{x};\beta\right)=f_{\beta}(\mathbf{x})$$

$$P\left(y=0|\mathbf{x};\beta\right)=1-f_{\beta}(\mathbf{x})$$

Since y can only take values 0 and 1, this can also be written as follows in a more compact form:

$$P\left(y|\mathbf{x};\beta\right)=\left(f_{\beta}\left(\mathbf{x}\right)\right)^{y}\left(1-f_{\beta}\left(\mathbf{x}\right)\right)^{1-y}$$

Indeed, when $y=1$, the second factor is equal to 1 and the expression reduces to $P\left(y=1|\mathbf{x};\beta\right)=f_{\beta}(\mathbf{x})$. Similarly, if $y=0$, the first factor is equal to 1 and the expression reduces to $1-f_{\beta}(x)$.

We can estimate the parameters by maximum likelihood, i.e., choosing the values of the parameters which maximize the probability of the data under the model identified by the parameters β:

$$L\left(\beta\right)=P(Y|X;\beta)$$

If we assume that the training examples are all independent, the likelihood can be expressed as:

$$L\left(\beta\right)=\prod_{i=1}^{N}P(y^{(i)}|\mathbf{x}^{(i)};\beta)=\prod_{i=1}^{N}f_{\beta}\left(\mathbf{x}^{(i)}\right)^{y^{(i)}}\left(1-f_{\beta}\left(\mathbf{x}^{(i)}\right)\right)^{1-y^{(i)}}$$

Maximizing this expression is equivalent to minimizing the negative logarithm of $L(\beta)$ (negative log-likelihood - nll):

$$nll\left(\beta\right)=-\log L\left(\beta\right)=-\sum_{i=1}^{N}\log\left[f_{\beta}\left(\mathbf{x}^{(i)}\right)^{y^{(i)}}\left(1-f_{\beta}\left(\mathbf{x}^{(i)}\right)\right)^{1-y^{(i)}}\right]=$$

$$=-\sum_{i=1}^{N}\left[y^{(i)}\log f_{\beta}\left(\mathbf{x}^{(i)}\right)+\left(1-y^{(i)}\right)\log\left(1-f_{\beta}\left(\mathbf{x}^{(i)}\right)\right)\right]$$

Hence, we will define our cost function as:

$$J\left(\beta\right)=-\sum_{i=1}^{N}\left[y^{(i)}\log f_{\beta}\left(\mathbf{x}^{(i)}\right)+\left(1-y^{(i)}\right)\log\left(1-f_{\beta}\left(\mathbf{x}^{(i)}\right)\right)\right]$$

This can be rewritten more explicitly in terms of the β parameters as follows:

$$J(\beta) = -\sum_{i=1}^{N} [y^{(i)} \log \sigma\left(\beta^T \mathbf{x}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(\beta^T \mathbf{x}^{(i)}\right)\right)]$$

Similarly to linear regression, we now have a cost function to minimize in order to find the values of the β_i parameters. Unfortunately, in this case, $J(\beta)$ assumes a nonlinear form **which prevents us to use the least square principles** and **there is no closed form solution for the parameter estimation**. In these cases, parameters can be estimated using some form of **iterative solver**, which begins with an initial guess for the parameters and iteratively refine them to find the final solution. Luckily, the logistic regression cost function **is convex, and hence only a single solution is admitted, independently from the initial guess**.

Different iterative solvers can be used in practice. The most commonly used is the **gradient descent algorithm**, which requires the cost function to be differentiable.

Visualizing the Cost Function (The “Loss Landscape”)

We have just derived the cost function $J(\beta)$, also known as **Log-Loss** or **Binary Cross-Entropy**.

$$J(\beta) = -\sum_{i=1}^{N} [y^{(i)} \log \sigma\left(\beta^T \mathbf{x}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(\beta^T \mathbf{x}^{(i)}\right)\right)]$$

You were told that this function is **convex**, which guarantees a single global minimum. But what does that actually *look like*?

Let’s visualize it. We will build a simple model `Diagnosis ~ radius1`, which has two parameters: β_0 (intercept) and β_1 (slope). We will then plot the Cost J for every possible combination of β_0 and β_1. We are looking for a “bowl” shape.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from mpl_toolkits.mplot3d import Axes3D # Moved import to top

# --- 1. Load and Prepare Data ---
data = pd.read_csv(DATA + "breast_cancer_wisconsin.csv")

# Map Diagnosis to 0 (Benign) and 1 (Malignant)
data['Diagnosis'] = data['Diagnosis'].replace({'B':0,'M':1})

# Extract our 1D feature and target
X = data[['radius1']].values
y = data['Diagnosis'].values

# --- 2. CRITICAL STEP: Scale the Feature ---
# This is vital for visualization. Without it, the "bowl"
# will be extremely stretched and impossible to see.
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

def compute_cost(X, y, beta_0, beta_1):
    """
    Calculates the total log-loss for a given beta_0 and beta_1.
    """
    # 1. Calculate the linear part (z)
    # --- THIS IS THE FIX ---
    # We .flatten() X to make it (n,) to match y's shape (n,)
    z = beta_0 + (X.flatten() * beta_1)
    # -----------------------

    # 2. Apply the sigmoid function to get probabilities (p_hat)
    p_hat = 1 / (1 + np.exp(-z)) # p_hat is now shape (n,)

    # --- Numerical Stability ---
    # We must "clip" values to prevent log(0) or log(1-1),
    # which would result in -inf.
    epsilon = 1e-10
    p_hat = np.clip(p_hat, epsilon, 1 - epsilon)
    # -------------------------

    # 3. Calculate the log-loss for each sample
    # y (n,) and p_hat (n,) will now multiply element-wise
    cost_per_sample = -(y * np.log(p_hat) + (1 - y) * np.log(1 - p_hat))

    # 4. Return the *total* cost for the whole dataset
```

```
    return np.sum(cost_per_sample)

# 1. Create ranges for Beta 0 (intercept) and Beta 1 (slope)
beta_0_range = np.linspace(-5, 5, 100)
# --- UPDATED as requested ---
beta_1_range = np.linspace(-2, 14, 100)
# ----------------------------

# 2. Create the 2D grid
B0, B1 = np.meshgrid(beta_0_range, beta_1_range)
costs = np.zeros_like(B0)

# 3. Calculate the cost for every point on the grid
for i in range(B0.shape[0]):
    for j in range(B0.shape[1]):
        costs[i, j] = compute_cost(X_scaled, y, B0[i, j], B1[i, j])

# Find the minimum for plotting
min_cost_idx = np.unravel_index(np.argmin(costs), costs.shape)
best_b0 = B0[min_cost_idx]
best_b1 = B1[min_cost_idx]
min_cost = costs[min_cost_idx]

# --- Create the Plot ---
fig = plt.figure(figsize=(18, 7))

# --- Subplot 1: 3D Surface Plot ---
ax1 = fig.add_subplot(1, 2, 1, projection='3d')
ax1.plot_surface(B0, B1, costs, cmap='viridis', alpha=0.8)
# --- UPDATED to show both optimal parameters ---
ax1.scatter(best_b0, best_b1, min_cost, color='red', s=200,
            edgecolor='black', label=f'Global Minimum (β0={best_b0:.2f}, β1=
        {best_b1:.2f})')
# -----------------------------------------------
ax1.set_xlabel('Intercept (β0)')
ax1.set_ylabel('Slope (β1)')
ax1.set_zlabel('Cost J(β) (Log-Loss)')
ax1.set_title('3D Loss Landscape')
ax1.view_init(elev=20, azim=135) # Adjust angle
ax1.legend()

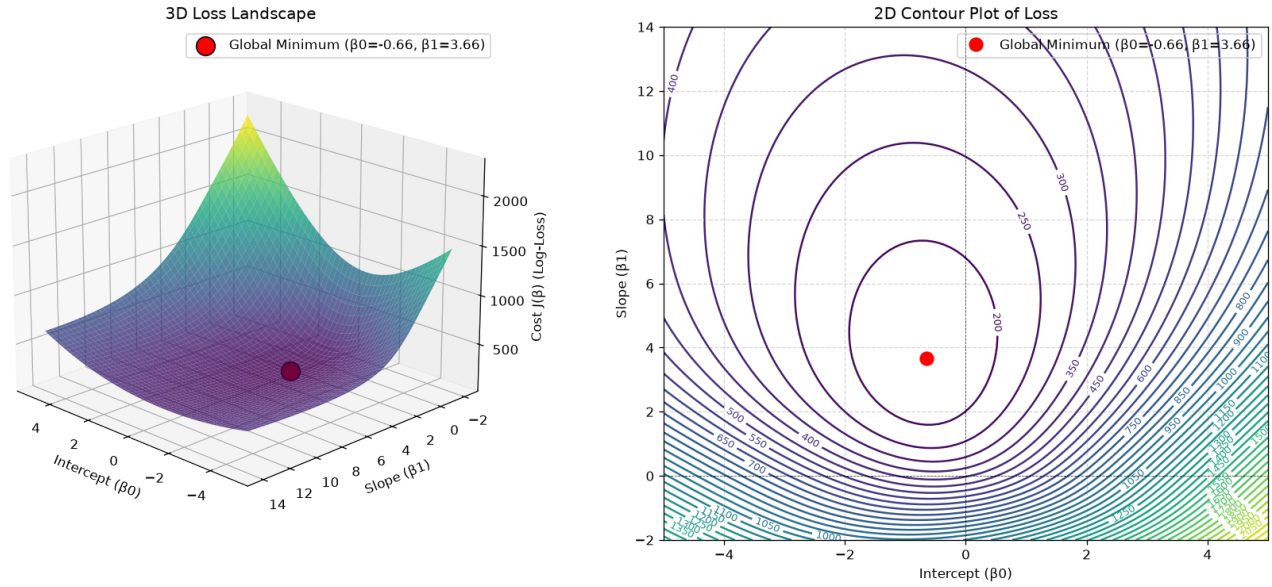
# --- Subplot 2: 2D Contour Plot ---
ax2 = fig.add_subplot(1, 2, 2)
contours = ax2.contour(B0, B1, costs, levels=50, cmap='viridis')
plt.clabel(contours, inline=True, fontsize=8)

# --- UPDATED to show both optimal parameters ---
ax2.plot(best_b0, best_b1, 'ro', markersize=10,
         label=f'Global Minimum (β0={best_b0:.2f}, β1={best_b1:.2f})')
```

```
# ------------------------------------------
ax2.set_xlabel('Intercept (β0)')
ax2.set_ylabel('Slope (β1)')
ax2.set_title('2D Contour Plot of Loss')
ax2.axhline(0, color='k', ls='--', lw=0.5)
ax2.axvline(0, color='k', ls='--', lw=0.5)
ax2.legend()
ax2.grid(True, linestyle='--', alpha=0.5)

plt.show()

print(f"Optimal parameters:")
print(f"  β0 (Intercept): {best_b0:.4f}")
print(f"  β1 (Slope):       {best_b1:.4f}")
print(f"  Minimum Cost (Log-Loss): {min_cost:.4f}")
```




```
Optimal parameters:
  β0 (Intercept): -0.6566
  β1 (Slope):       3.6566
  Minimum Cost (Log-Loss): 165.0107
```

The plots clearly show a single, smooth, **convex "bowl"**.

- There are no "local minima" or "bumpy" areas for an optimizer to get stuck in.
- This is why we were told the logistic regression cost function **is convex**.
- The **red dot** at the very bottom of this bowl represents the *one* optimal set of parameters (β_0, β_1) that minimizes the cost.

An iterative solver like **Gradient Descent** is guaranteed to find this minimum, just like a marble released from anywhere on the bowl's rim is guaranteed to roll to the bottom.

Statistical Interpretation of the Coefficients of a Logistic Regressor

Let's now see how to interpret the coefficients of a logistic regressor.

Interpretation of the intercept β_0

Remember that the regression model (in the case of simple logistic regression) is as follows:

$$\log(\frac{p}{1-p}) = \beta_0 + \beta_1 x$$

Applying what we know about logistic regressors, we can write:

$$x = 0 \Rightarrow \log \frac{p}{1-p} = \beta_0$$

To have a clearer picture, we can exponentiate both sides of the equation and write:

$$x = 0 \Rightarrow \frac{p}{1-p} = e^{\beta_0}$$

Remember that $\frac{p}{1-p}$ is the odd that the dependent variable is equal to 1 when observing x and, as such, it has a clear interpretation. For example, if the odds of an event are $\frac{3}{1}$, then it is 3 times more likely to occur than not to occur. So, **for $x = 0$, it is e^{β_0} times more likely that the dependent variable is equal to 1, rather than being equal to 0**.

Interpretation of variable coefficients β_i

We know that:

$$odds(p|x) = \frac{P(y=1|x)}{1-P(y=1|x)}$$

We can write:

$$\log odds(p|x) = \beta_0 + \beta_1 x$$

Hence:

$$\log odds(p|x+1) - \log odds(p|x) = \beta_0 + \beta_1(x+1) - \beta_0 - \beta_1 x = \beta_1(x+1) - \beta_1 x = \beta_1$$

Exponentiating both sides, we get:

$$e^{\log odds(p|x+1)-\log odds(p|x)} = e^{\beta_1} \Rightarrow \frac{e^{\log odds(p|x+1)}}{e^{\log odds(p|x)}} = e^{\beta_1} \Rightarrow \frac{odds(p|x+1)}{odds(p|x)} = e^{\beta_1} \Rightarrow odds(p|x +$$

We can thus say that **increasing the variable x by one unit corresponds to a multiplicative increase in odds by e^{β_1}**.

This analysis can be easily extended to the case of a multiple logistic regressor. Hence in general, given the model:

$$P(y=1|\mathbf{x}) = \sigma(\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n)$$

We can say that:

- e^{β_0} is the odd of y being equal to 1 rather than 0 when $x_i = 0,\ \forall i$;
- An increment of one unit in the independent variable x_i corresponds to a multiplicative increment of e^{β_i} in the odds of $y = 1$. So if $e^{\beta_i} = 0.05$, then $y = 1$ is 5% more likely for a one-unit increment of x.

Example of Logistic Regression

Let us now apply logistic regression to a larger set of variables in our regression problem. We will consider the following independent variables:

- `radius1`
- `texture1`
- `perimeter1`
- `area1`
- `smoothness1`
- `compactness1`
- `concavity1`
- `symmetry1`

The dependent variable is again `Diagnosis`.

Once fit to the data, we will obtain the following parameters:

R^2	Adj. R^2	F-statistic	Prob(F-statistic)	Log-Likelihood
0.670	0.666	142.4	1.16e-129	-78.055

All values have interpretations similar to the ones obtained in the case of linear regression. The Log-Likelihood reports the value of the logarithm of the likelihood which was used to train the data.

The estimates for the coefficients are as follows:

```
ols("Diagnosis ~ radius1 + texture1 + perimeter1 + area1 + smoothness1 + compactness1
        + concavity1 + symmetry1",data).fit().summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-2.6591	0.224	-11.896	0.000	-3.098	-2.220
radius1	0.4688	0.133	3.532	0.000	0.208	0.730
texture1	0.0219	0.003	7.376	0.000	0.016	0.028
perimeter1	-0.0473	0.021	-2.272	0.023	-0.088	-0.006
area1	-0.0009	0.000	-3.985	0.000	-0.001	-0.000
smoothness1	5.1389	1.221	4.208	0.000	2.740	7.538
compactness1	0.3080	0.854	0.360	0.719	-1.370	1.986
concavity1	2.0973	0.414	5.065	0.000	1.284	2.911
symmetry1	1.2739	0.568	2.244	0.025	0.159	2.389

We notice that not all variables have a statistically relevant relationship with the dependent variable. Applying backward elimination, we remove `compactness1` and obtain the following estimates:

```
ols("Diagnosis ~ radius1 + texture1 + perimeter1 + area1 + smoothness1 + concavity1
        symmetry1",data).fit().summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-2.6708	0.221	-12.086	0.000	-3.105	-2.237
radius1	0.4360	0.097	4.517	0.000	0.246	0.626
texture1	0.0219	0.003	7.405	0.000	0.016	0.028
perimeter1	-0.0419	0.014	-2.915	0.004	-0.070	-0.014
area1	-0.0010	0.000	-4.477	0.000	-0.001	-0.001
smoothness1	5.3093	1.125	4.719	0.000	3.099	7.519
concavity1	2.1479	0.389	5.517	0.000	1.383	2.913
symmetry1	1.3132	0.557	2.359	0.019	0.220	2.407

These are now all statistically relevant. For instance, we can see that:

- When all variables are set to zero, the odds of the benign tumor are $e^{-2.6708} \approx 0.07$, or $\frac{7}{100}$. This is a base value.
- An increment in one unit of `texture1` increments the odds of a benign tumor multiplicatively by a factor of $e^{0.0219} \approx 1.02$ (a $+2\%$), when all other variables are constant.
- An increment of one unit of `perimeter1` decrements the odds of benign tumor multiplicatively by a factor of $e^{-0.0419} \approx 0.96$ (a -4%), when all other variables are constant.

Evaluating the Logistic Regression Model

The `statsmodels` summary table gives us a lot of information, including a value for `R-squared`. It's critical to understand that this is **NOT** the same R^2 we used in linear regression.

The Problem: We Can't Use R^2

In OLS (Linear Regression), $R^2 = 1 - \frac{RSS}{TSS}$. This metric is based on minimizing the *sum of squared residuals* (RSS).

In Logistic Regression, our model does not predict a continuous value. It predicts a *probability*. We don't minimize squared residuals; we minimize the **negative log-likelihood** (also called **Cross-Entropy Loss**).

Because the underlying cost function is different, the R^2 metric is not mathematically valid. A model that predicts a probability of `0.9` for an event that happens (`y=1`) is a *fantastic* prediction, but it would have a "residual" of $(1 - 0.9) = 0.1$, which OLS would want to square and "punish." This doesn't make sense.

The Solution: "Pseudo R^2"

To get a similar "0-to-1" metric for *goodness-of-fit*, statisticians have developed several **"Pseudo R^2"** measures. The one `statsmodels` reports by default is **McFadden's R^2**.

The logic is the same as OLS R^2: "How much better is our model than a 'dumb' baseline?"

- **In OLS:** We compared our model's error (RSS) to the baseline's error (TSS).
- **In Logistic Regression:** We compare our model's **Log-Likelihood** to a baseline model's Log-Likelihood.

1. **Baseline Model (Log-Likelihood Null, LL_{Null}):** A "dumb" model that only knows the *prior* (e.g., $P(y=1) = 0.35$). This is an "intercept-only" model. `statsmodels` calculates this for you.
2. **Our "Full" Model (Log-Likelihood, LL_{Model}):** This is the log-likelihood of our *fitted* model. It will be a "less negative" (better) number than LL_{Null}.

McFadden's R^2 is then defined as:

$$R^2_{McFadden} = 1 - \frac{LL_{Model}}{LL_{Null}}$$

Interpretation:

- LL_{Model} is the value reported as "Log-Likelihood" in your summary table.

- LL_{Null} is reported as "LL-Null".
- `Pseudo R-sq.`: This is the $R^2_{McFadden}$ score.

Consider the following example:

- `Log-Likelihood: -105.19` (LL_{Model})
- `LL-Null: -452.39` (LL_{Null})
- `Pseudo R-sq.: 0.77`

This `0.77` was calculated as $1 - (\frac{-105.19}{-452.39}) \approx 0.77$.

It is interpreted similarly to R^2: "Our model's features explain about **77%** of the 'deviance' (uncertainty) in the outcome, compared to a model that just guesses the average."

Important: You cannot compare a `Pseudo R-sq.` from logistic regression to an R^2 from OLS. They are not equivalent. It is only useful for comparing *nested* logistic regression models (like we do in backward elimination).

Predictive Evaluation

When we use the model in a purely predictive way, we can also use all evaluation measures we have seen for predictive analysis, such as accuracy, precision, recall, F1, confusion matrix etc.

Logistic Regression in Python

Let's look at an example of logistic regression in Python. We will use the R dataset `biopsy`. We can load it using `statsmodels`:

```
from statsmodels.datasets import get_rdataset
biopsy = get_rdataset('biopsy',package='MASS')
print(biopsy.__doc__)
```

```
====== ===============
biopsy R Documentation
====== ===============

Biopsy Data on Breast Cancer Patients
-------------------------------------

Description
~~~~~~~~~~~

This breast cancer database was obtained from the University of
Wisconsin Hospitals, Madison from Dr. William H. Wolberg. He assessed
biopsies of breast tumours for 699 patients up to 15 July 1992; each of
nine attributes has been scored on a scale of 1 to 10, and the outcome
is also known. There are 699 rows and 11 columns.

Usage
~~~~~

.. code:: R

   biopsy

Format
~~~~~~

This data frame contains the following columns:

``ID``
   sample code number (not unique).

``V1``
   clump thickness.

``V2``
   uniformity of cell size.

``V3``
   uniformity of cell shape.

``V4``
   marginal adhesion.

``V5``
   single epithelial cell size.

``V6``
   bare nuclei (16 values are missing).
```

```
``V7``
   bland chromatin.

``V8``
   normal nucleoli.

``V9``
   mitoses.

``class``
   ``"benign"`` or ``"malignant"``.

Source
~~~~~~

P. M. Murphy and D. W. Aha (1992). UCI Repository of machine learning
databases. [Machine-readable data repository]. Irvine, CA: University of
California, Department of Information and Computer Science.

O. L. Mangasarian and W. H. Wolberg (1990) Cancer diagnosis via linear
programming. *SIAM News* **23**, pp 1 & 18.

William H. Wolberg and O.L. Mangasarian (1990) Multisurface method of
pattern separation for medical diagnosis applied to breast cytology.
*Proceedings of the National Academy of Sciences, U.S.A.* **87**, pp.
9193-9196.

O. L. Mangasarian, R. Setiono and W.H. Wolberg (1990) Pattern
recognition via linear programming: Theory and application to medical
diagnosis. In *Large-scale Numerical Optimization* eds Thomas F. Coleman
and Yuying Li, SIAM Publications, Philadelphia, pp 22-30.

K. P. Bennett and O. L. Mangasarian (1992) Robust linear programming
discrimination of two linearly inseparable sets. *Optimization Methods
and Software* **1**, pp. 23-34 (Gordon & Breach Science Publishers).

References
~~~~~~~~~~

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with
S-PLUS.* Fourth Edition. Springer.
```

The dataset contains 699 observations and 11 columns. Each observation contains measurements of 9 quantities relating to tissue samples that can be “benign” or “malignant” tumors. Let’s start by manipulating the data a bit. We visualize the values of the `class` variable:

```
biopsy.data['class'].unique()
```

```
array(['benign', 'malignant'], dtype=object)
```

To calculate the logistic regression model using statsmodels, it is necessary to convert these values into integers (0 or 1). Furthermore, it is advisable to avoid calling the column `class` as this is a reserved word for statsmodels. We build a new column `cl` that contains the modified `class` values:

```
biopsy.data['cl'] = biopsy.data['class'].replace({'benign':0, 'malignant':1})
```

We will use the `logit` object form statsmodels:

```
from statsmodels.formula.api import logit
model = logit('cl ~ V1 + V2 + V3 + V4 + V5 + V6 + V7 + V8 + V9',biopsy.data).fit()
model.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.075321
         Iterations 10
```

Logit Regression Results

Dep. Variable:	cl	No. Observations:	683
Model:	Logit	Df Residuals:	673
Method:	MLE	Df Model:	9
Date:	Tue, 15 Sep 2026	Pseudo R-squ.:	0.8837
Time:	19:40:30	Log-Likelihood:	-51.444
converged:	True	LL-Null:	-442.18
Covariance Type:	nonrobust	LLR p-value:	2.077e-162

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-10.1039	1.175	-8.600	0.000	-12.407	-7.801
V1	0.5350	0.142	3.767	0.000	0.257	0.813
V2	-0.0063	0.209	-0.030	0.976	-0.416	0.404
V3	0.3227	0.231	1.399	0.162	-0.129	0.775
V4	0.3306	0.123	2.678	0.007	0.089	0.573
V5	0.0966	0.157	0.617	0.537	-0.210	0.404
V6	0.3830	0.094	4.082	0.000	0.199	0.567
V7	0.4472	0.171	2.609	0.009	0.111	0.783
V8	0.2130	0.113	1.887	0.059	-0.008	0.434
V9	0.5348	0.329	1.627	0.104	-0.110	1.179

The logistic regressor explains the relationship between variables well (R^2 high) and is significant (p-value almost zero). Some coefficients have a high p-value. Let's start by eliminating the variable `V2`, which has the highest p-value:

```
model = logit('cl ~ V1 + V3 + V4 + V5 + V6 + V7 + V8 + V9',biopsy.data).fit()
model.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.075321
         Iterations 10
```

Logit Regression Results

Dep. Variable:	cl	No. Observations:	683
Model:	Logit	Df Residuals:	674
Method:	MLE	Df Model:	8
Date:	Tue, 15 Sep 2026	Pseudo R-squ.:	0.8837
Time:	19:40:30	Log-Likelihood:	-51.445
converged:	True	LL-Null:	-442.18
Covariance Type:	nonrobust	LLR p-value:	2.036e-163

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-10.0976	1.155	-8.739	0.000	-12.362	-7.833
V1	0.5346	0.141	3.784	0.000	0.258	0.811
V3	0.3182	0.174	1.826	0.068	-0.023	0.660
V4	0.3299	0.121	2.723	0.006	0.092	0.567
V5	0.0961	0.156	0.618	0.537	-0.209	0.401
V6	0.3831	0.094	4.082	0.000	0.199	0.567
V7	0.4465	0.170	2.628	0.009	0.114	0.779
V8	0.2125	0.112	1.902	0.057	-0.006	0.432
V9	0.5341	0.328	1.630	0.103	-0.108	1.176

We proceed by removing `V5`, which has a p-value of 0.537:

```
model = logit('cl ~ V1 + V3 + V4 + V6 + V7 + V8 + V9',biopsy.data).fit()
model.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.075598
         Iterations 10
```

Logit Regression Results

Dep. Variable:	cl	No. Observations:	683
Model:	Logit	Df Residuals:	675
Method:	MLE	Df Model:	7
Date:	Tue, 15 Sep 2026	Pseudo R-squ.:	0.8832
Time:	19:40:30	Log-Likelihood:	-51.633
converged:	True	LL-Null:	-442.18
Covariance Type:	nonrobust	LLR p-value:	2.240e-164

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-9.9828	1.126	-8.865	0.000	-12.190	-7.776
V1	0.5340	0.141	3.793	0.000	0.258	0.810
V3	0.3453	0.172	2.012	0.044	0.009	0.682
V4	0.3425	0.119	2.873	0.004	0.109	0.576
V6	0.3883	0.094	4.150	0.000	0.205	0.572
V7	0.4619	0.168	2.746	0.006	0.132	0.792
V8	0.2261	0.111	2.037	0.042	0.009	0.444
V9	0.5312	0.324	1.637	0.102	-0.105	1.167

We remove `V9`, which has a p-value greater than 0.05:

```
model = logit('cl ~ V1 + V3 + V4 + V6 + V7 + V8',biopsy.data).fit()
model.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.078436
         Iterations 9
```

Logit Regression Results

Dep. Variable:	cl	No. Observations:	683
Model:	Logit	Df Residuals:	676
Method:	MLE	Df Model:	6
Date:	Tue, 15 Sep 2026	Pseudo R-squ.:	0.8788
Time:	19:40:30	Log-Likelihood:	-53.572
converged:	True	LL-Null:	-442.18
Covariance Type:	nonrobust	LLR p-value:	1.294e-164

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-9.7671	1.085	-9.001	0.000	-11.894	-7.640
V1	0.6225	0.137	4.540	0.000	0.354	0.891
V3	0.3495	0.165	2.118	0.034	0.026	0.673
V4	0.3375	0.116	2.920	0.004	0.111	0.564
V6	0.3786	0.094	4.035	0.000	0.195	0.562
V7	0.4713	0.166	2.837	0.005	0.146	0.797
V8	0.2432	0.109	2.240	0.025	0.030	0.456

All coefficients now have an acceptable p-value. Let's proceed to the analysis of the coefficients. We calculate the exponentials:

```
np.exp(model.params)
```

```
Intercept    0.000057
V1           1.863641
V3           1.418374
V4           1.401487
V6           1.460166
V7           1.602133
V8           1.275287
dtype: float64
```

- The near-zero value of the exponential of the intercept indicates that, when all variables take zero values, the odds are very low. This suggests that p is low, while $1 - p$ is very high. The probability of having a malignant tumor is therefore very low if all variables take zero values;
- An increase of one unit in the value of `V1` corresponds to an increase of approximately 86% in the odds, making the possibility of a malignant tumor higher;
- An increase of one unit in the value of `V3` corresponds to an increase of approximately 41% in the odds;
- An increase of one unit in the value of `V4` corresponds to an increase of approximately 40% in the odds;

- An increase of one unit in the value of `V6` corresponds to an increase of approximately 46% in the odds;
- An increase of one unit in the value of `V7` corresponds to an increase of approximately 60% in the odds;
- An increase of one unit in the value of `V8` corresponds to an increase of approximately 27% in the odds;

The increase of variables generally causes an increase in the odds. Therefore, we expect the values of the variables to be small in the presence of benign tumors.

Exercises

1. **Read the coefficients.** Fit a logistic regression of `Survived` on `Sex` and `Pclass`. Convert each coefficient into an odds ratio, and write one sentence per coefficient in the language of the problem — no statistical vocabulary.
2. **Odds, log-odds, probability.** For a female passenger in second class, compute the predicted log-odds, the odds and the probability by hand from the coefficients, then check against `predict`. Which of the three is linear in the predictors?
3. **A continuous predictor.** Add `Age` to the model. Is its coefficient significant? Compute how much the predicted probability changes between a 20-year-old and a 40-year-old, all else equal — and note that the answer depends on what "all else" is.
4. **Confounding, again.** Fit `Survived ~ Age` alone, then `Survived ~ Age + Pclass`. Does the age coefficient change? Relate what you see to chapter 11.
5. **Where the linear model shows.** Plot the predicted probability of survival against age for each class. Why are the three curves parallel on the log-odds scale but not on the probability scale?

References

- Chapter 4 of [1]

[1] James, Gareth Gareth Michael. An introduction to statistical learning: with applications in Python, 2023.https://www.statlearning.com

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Extending regression models: interactions, polynomial and multinomial terms

Chapter resources

Slides (PDF) Open in Colab Practice quiz

Chapter 8 and chapter 9 introduced the two workhorse regression models in their simplest form: one straight line, one logistic curve, each with its coefficients read as a statement about the phenomenon. This chapter keeps that statistical reading and relaxes three of its restrictions.

Interaction terms let the effect of one variable depend on another — the honest way to say "it depends". **Polynomial terms** let a relationship bend without abandoning least squares. **Multinomial logistic regression** extends the logistic model from two outcomes to many.

All three are still about *explaining*: we fit with `statsmodels`, read coefficients, and check whether they say something defensible. Chapter 14 takes the same models and asks a different question — how well do they predict data they have never seen?

Extending the Linear Model with Interaction Terms

A simple linear model assumes the effect of each predictor on Y is **additive**. The formula

$$mpg = \beta_0 + \beta_1 \text{horsepower} + \beta_2 \text{weight}$$

assumes that the effect of 1 unit of `horsepower` (which is β_1) is **constant**, regardless of the car's `weight`.

Is that true? We can test this by adding an **interaction term**:

$$mpg = \beta_0 + \beta_1 \text{horsepower} + \beta_2 \text{weight} + \beta_3 (\text{horsepower} \times \text{weight})$$

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.api as sm
import statsmodels.formula.api as smf

# Load and clean the data (same as last lecture)
mpg = sns.load_dataset('mpg')
mpg = mpg.drop('name', axis=1)
mpg['horsepower'] = pd.to_numeric(mpg['horsepower'], errors='coerce')
mpg = mpg.dropna()

# Fit the simple additive model
model_additive = smf.ols('mpg ~ horsepower + weight', data=mpg).fit()

# Fit the new interaction model
model_interaction = smf.ols('mpg ~ horsepower + weight + horsepower * weight',
        data=mpg).fit()

print("--- Additive Model (R-squared) ---")
print(f"Adj. R-squared: {model_additive.rsquared_adj:.4f}")

print("\n--- Interaction Model (R-squared) ---")
print(f"Adj. R-squared: {model_interaction.rsquared_adj:.4f}")

print("\n--- Interaction Model Coefficients ---")
print(model_interaction.summary().tables[1])
```

```
--- Additive Model (R-squared) ---
Adj. R-squared: 0.7049

--- Interaction Model (R-squared) ---
Adj. R-squared: 0.7465

--- Interaction Model Coefficients ---
==============================================================================
                        coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
Intercept            63.5579      2.343     27.127      0.000      58.951
68.164
horsepower           -0.2508      0.027     -9.195      0.000      -0.304
-0.197
weight               -0.0108      0.001    -13.921      0.000      -0.012
-0.009
horsepower:weight  5.355e-05   6.65e-06      8.054      0.000    4.05e-05    6.66e-
05
==============================================================================
```

Interpreting the Interaction

Notice two things:

1. The **Adj. R-squared** increased! This indicates the interaction model is a better fit.
2. The p-value for the `horsepower:weight` term is tiny (< 0.001). This confirms the interaction is statistically significant.

But how do we interpret the coefficient for `horsepower`? We can rewrite the formula:

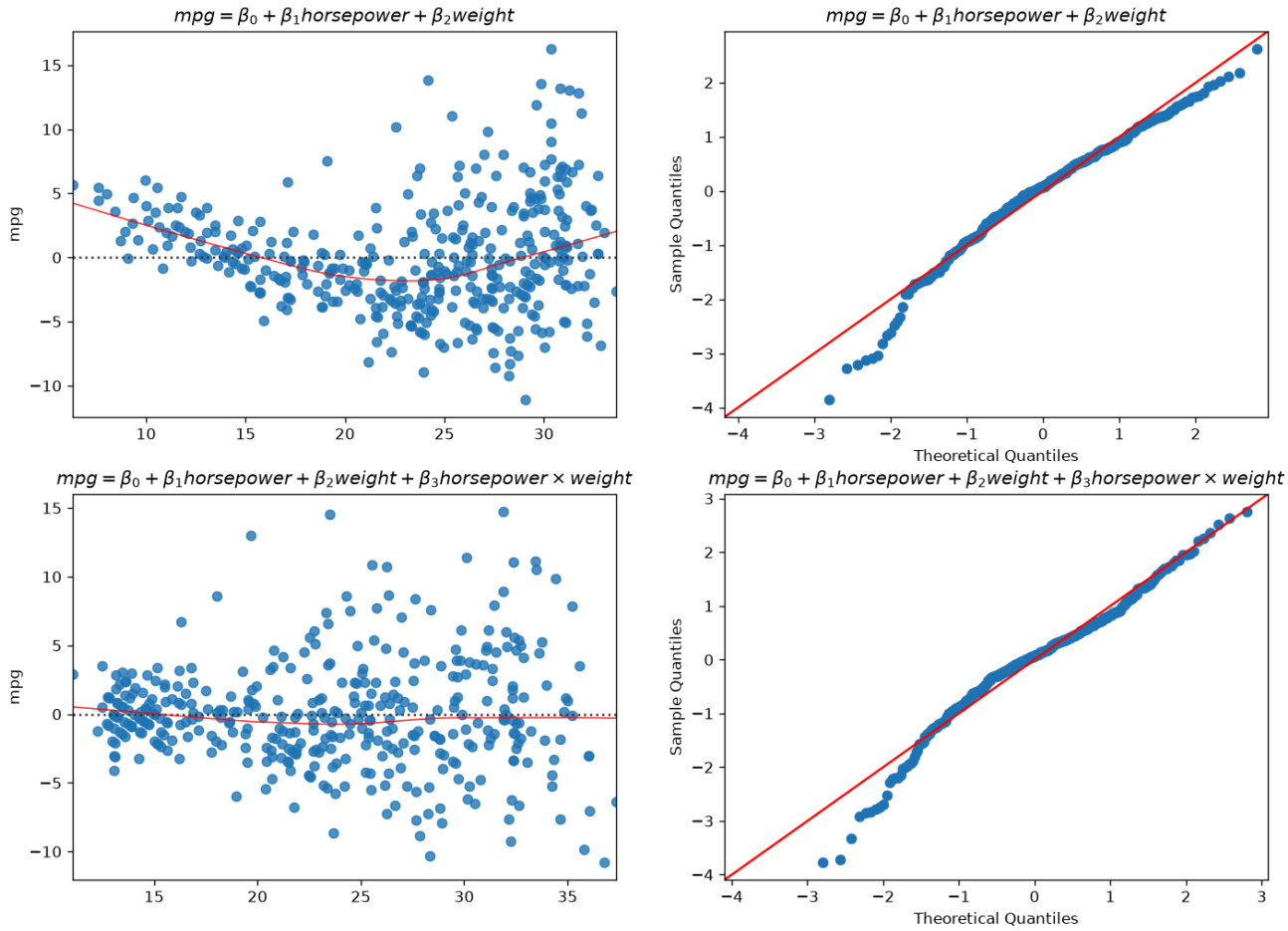
$$mpg = \beta_0 + (\beta_1 + \beta_3 \times \text{weight}) \times \text{horsepower} + \beta_2 \text{weight}$$

The "effect" of `horsepower` is no longer a single number, β_1. It is now $(\beta_1 + \beta_3 \times \text{weight})$. This means the effect of `horsepower` *depends on* the `weight` of the car.

In practice, **when we increase `weight` by one unit, we increase the effect of `horsepower` on `mpg` by β_3 units**. This means that for heavier cars, `horsepower` has a less negative effect on `mpg`.

This makes sense mechanically: in a very heavy car, horsepower is needed just to move it, so the efficiency penalty is smaller compared to a light car where extra horsepower is "overkill.

Let's diagnose our residuals to see if we gained something:

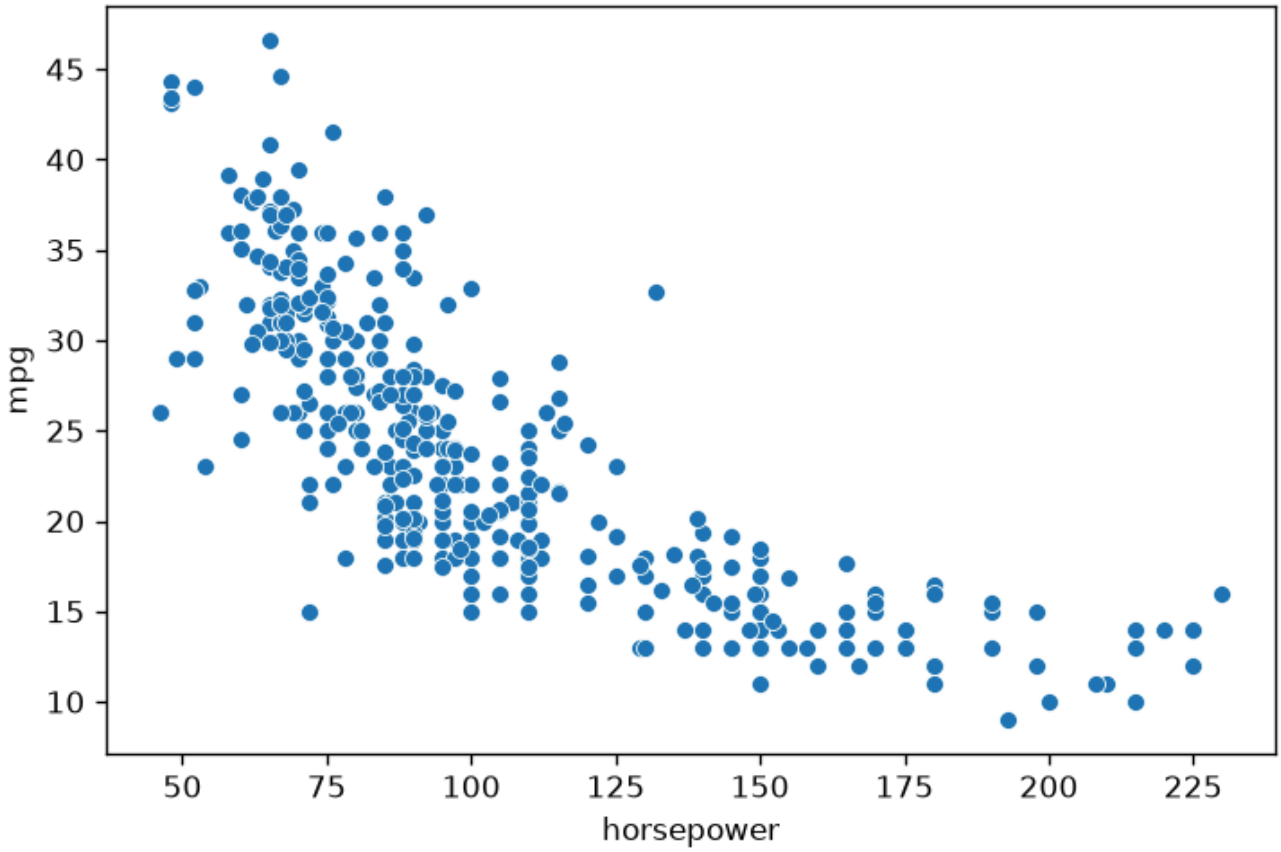


The situation improved! We now have a better model, even if not perfect.

Quadratic and Polynomial Regression

We will now take another route to address our nonlinear problem. Let's have a look at the relationship between `mpg` and `horsepower`:

```
sns.scatterplot(x='horsepower',y='mpg',data=data)
plt.show()
```



This relationship does not look linear. It looks instead as a quadratic, which would be fit by the following model:

$$mpg = \beta_0 + \beta_1 horsepower + \beta_2 horsepower^2$$

Again, the model above is nonlinear in `horsepower`, but we can still fit it with a linear regressor if we add a new variable $z = horsepower^2$.

The fit model will obtain $R^2 = 0.688$, larger than the $R^2 = 0.606$ obtained by the base model ($mpg = \beta_0 + \beta_1 horsepower$). Both have a large F-statistic.

The estimated coefficients are:

```
ols("mpg ~ horsepower + I(horsepower**2)", data).fit().summary().tables[1]
```

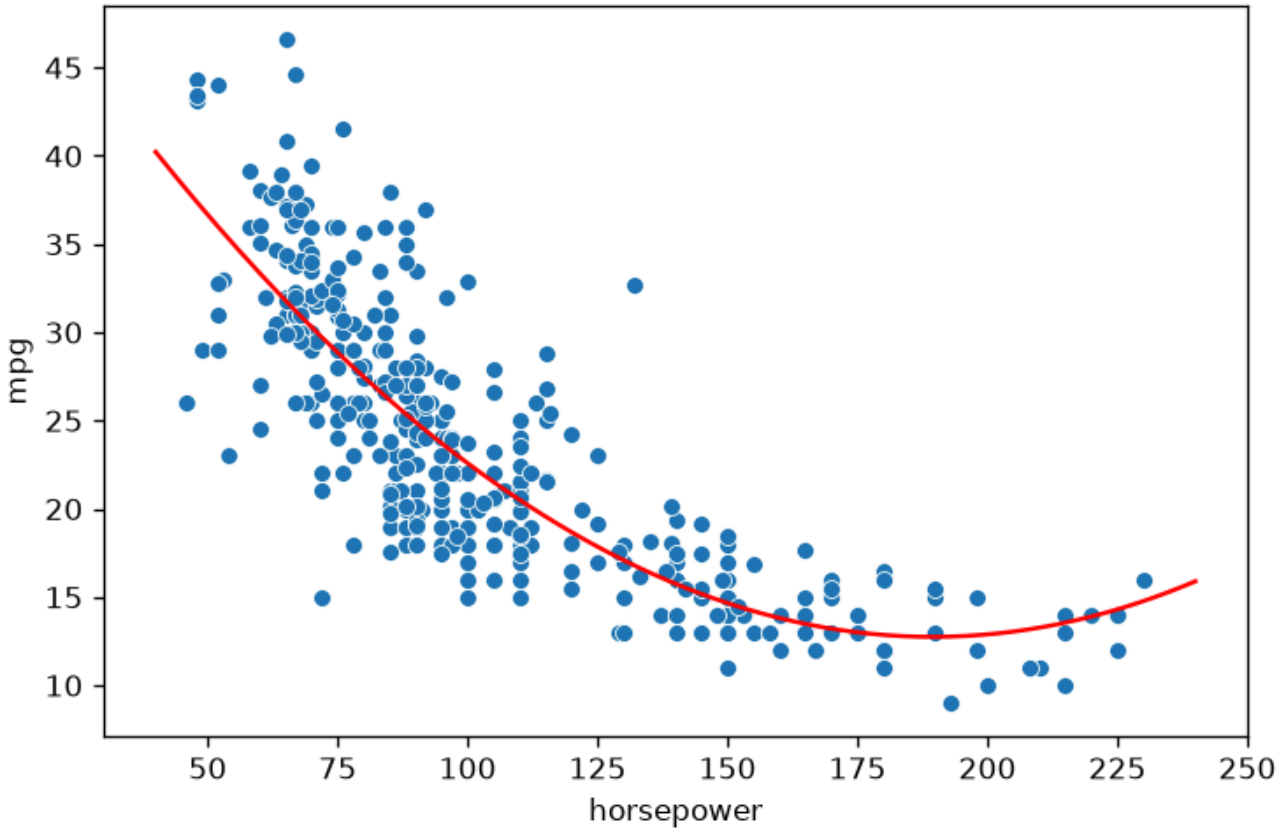
	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	56.9001	1.800	31.604	0.000	53.360	60.440
horsepower	-0.4662	0.031	-14.978	0.000	-0.527	-0.405
I(horsepower ** 2)	0.0012	0.000	10.080	0.000	0.001	0.001

The coefficients now describe the quadratic:

$$y = 56.9001 - 0.4662x + 0.0012x^2$$

If we plot it on the data, we obtain the following:

```
model=ols("mpg ~ horsepower + I(horsepower**2)", data).fit()
b0, b1, b2 = model.params.values
sns.scatterplot(x='horsepower',y='mpg',data=data)
x = np.linspace(40,240,200)
y = b0+b1*x+b2*x**2
plt.plot(x,y,'r')
plt.show()
```



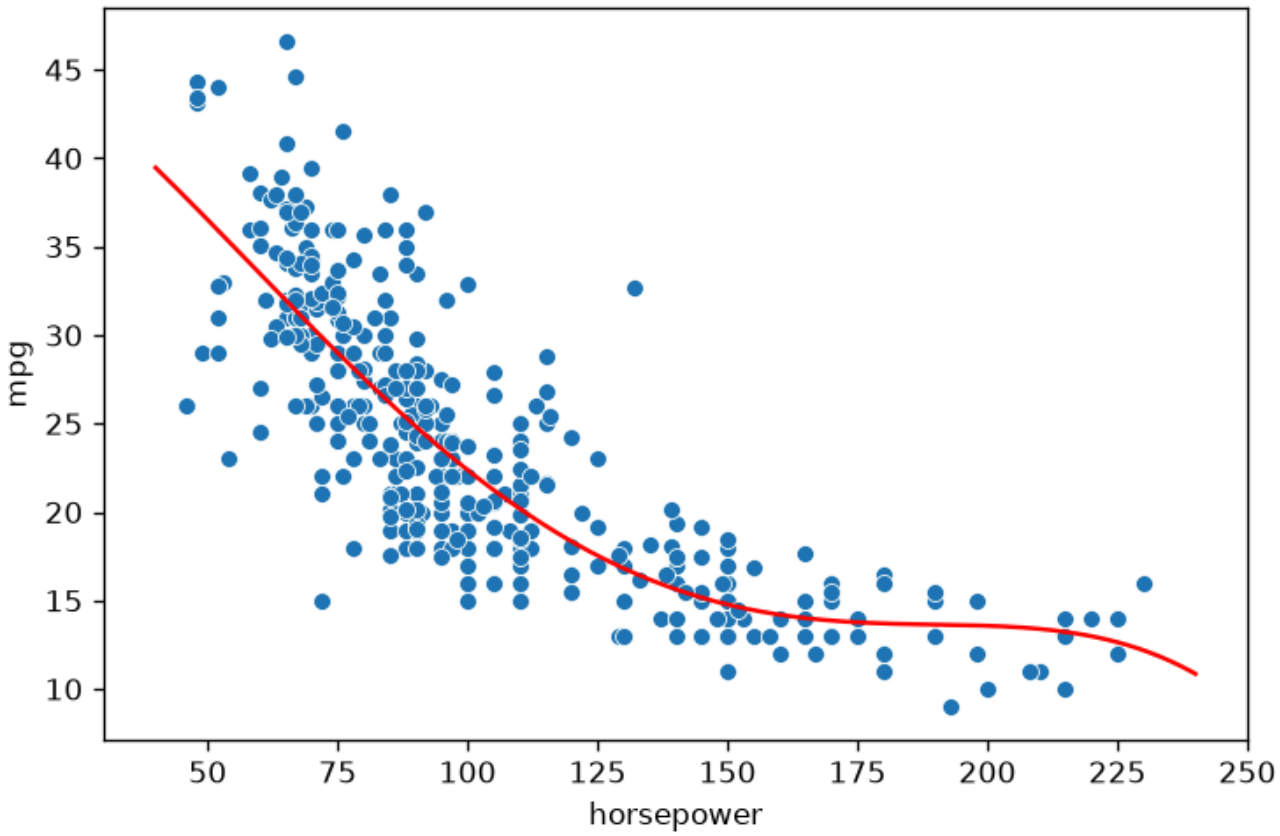
Polynomial Regression

In general, we can fit a polynomial model to the data, choosing a suitable degree d. For instance, for $d = 4$ we have:

$$mpg = \beta_0 + \beta_1 horsepower + \beta_2 horsepower^2 + \beta_3 horsepower^3 + \beta_4 horsepower^4$$

which identifies the following fit:

```
model=ols("mpg ~ horsepower + I(horsepower**2) + I(horsepower**3) +
        I(horsepower**4)", data).fit()
b0, b1, b2, b3,b4 = model.params.values
sns.scatterplot(x='horsepower',y='mpg',data=data)
x = np.linspace(40,240,200)
y = b0+b1*x+b2*x**2+b3*x**3+b4*x**4
plt.plot(x,y,'r')
plt.show()
```



This approach is known as **polynomial regression** and allows to turn the linear regression into a nonlinear model. Note that, when we have more variables, polynomial regression also includes interaction terms. For instance, the linear model:

$$y = \beta_0 + \beta_1 x + \beta_2 y$$

becomes the following polynomial model of degree 2:

$$y = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 x^2 + \beta_4 y^2 + \beta_5 xy$$

As usual, we only have to add new variables for the squared and interaction term and solve the problem as a linear regression one. This is easily handled by libraries. Note that, as the number of variables increases, the number of terms to add for a given degree also increases.

The key observation here is that, while the polynomial regression function is **nonlinear in the variable**, it is still **linear in its coefficients**, which makes it still solvable with ordinary

least squares.

Multinomial Logistic Regression

In many cases, we want to study the relationship between a set of **continuous or categorical independent variable and a non-binary categorical dependent variable**.

Let us consider the `diabetes` dataset:

```
import statsmodels.api as sm

# Try loading the dataset
diabetes_data = sm.datasets.get_rdataset("Diabetes", "heplots").data
diabetes_data.head()
```

	relwt	glufast	glutest	instest	sspg	group
0	0.81	80	356	124	55	Normal
1	0.95	97	289	117	76	Normal
2	0.94	105	319	143	105	Normal
3	1.04	90	356	199	108	Normal
4	1.00	90	323	240	143	Normal

This dataset contains clinical measurements from 145 subjects, used to study glucose metabolism and insulin resistance.

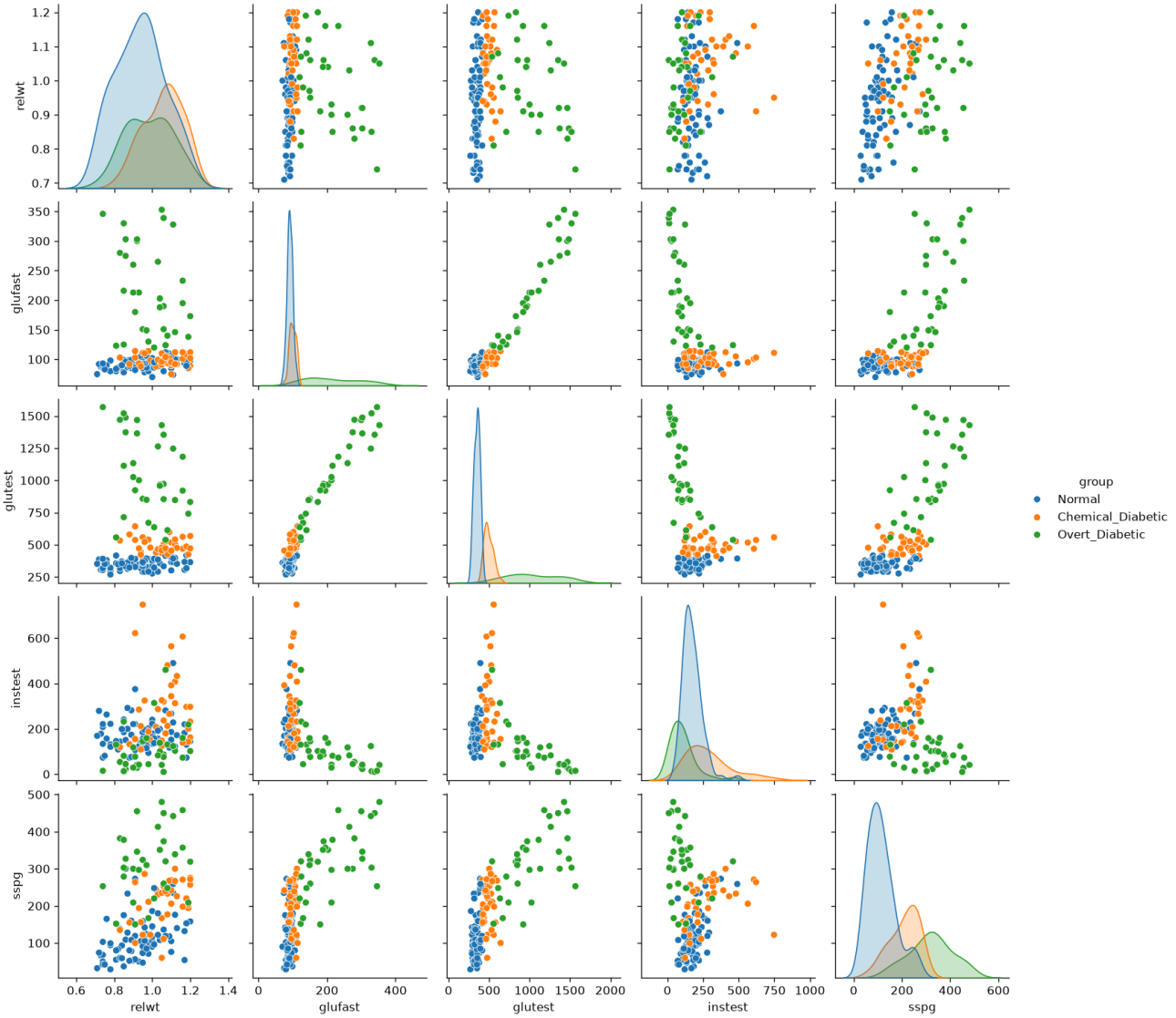
It was originally presented in *Data: A Collection of Problems from Many Fields* (Andrews & Herzberg, 1985) and is included in the R package **heplots**.

Variables

- **relwt**: Relative weight (ratio of actual to expected weight)
- **glufast**: Fasting plasma glucose level
- **glutest**: Glucose intolerance (oral glucose tolerance test area)
- **instest**: Insulin response (insulin area)
- **sspg**: Steady state plasma glucose (measure of insulin resistance)
- **group**: Diagnostic group
  - *Normal*
  - *Chemical_Diabetic*
  - *Overt_Diabetic*

Let's visualize the data:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.pairplot(diabetes_data, hue='group')
plt.show()
```



Differently from previous examples, our categorical variable contains three levels. Also, we can easily identify an interesting **base class**, i.e., the "Normal" group.

We will define an extension of the logistic regression model, the multinomial logistic regression model, based on this concept of "base" class.

The Multinomial Logistic Regression Model

When the dependent variable can assume more than two values, **we can define the multinomial logistic regression model**. In this case, we select one of the values of the dependent variable Y as **a baseline class** (e.g., the "Normal" group). Without loss of generality, let K be the number of classes and let $Y = 1$ be the baseline class. Recall that in the case of the logistic regressor, we modeled the logarithm of the odd as our linear function:

$$\log\left(\frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$$

Since we have more than one possible outcomes for the dependent variable, rather than modeling the odds, a multinomial logistic regressor models the logarithm of the ratio between a given class k and the baseline class 1 as follows:

$$\log\left(\frac{P(Y=k|X=\mathbf{x})}{P(Y=1|X=\mathbf{x})}\right) = \beta_{k0} + \beta_{k1} x_1 + \ldots + \beta_{kn} x_n$$

Note that, in practice, we need to define a different linear combination for each class $k = 1 \ldots K$, hence we need $(n+1) \times (k-1)$ parameters.

Doing the math, it can be shown that:

$$P(Y=k|X=\mathbf{x}) = \frac{e^{\beta_\mathbf{k}{}^T \mathbf{X}}}{1 + \sum_{l=2}^{K} \mathbf{e}^{\beta_1^\mathbf{T} \mathbf{X}}}$$

$$P(Y=1|X=\mathbf{x}) = \frac{1}{1 + \sum_{l=2}^{K} \mathbf{e}^{\beta_1^\mathbf{T} \mathbf{X}}}$$

Where $\beta_\mathbf{k} = (\beta_0, \beta_1, \ldots, \beta_k)$ and $\mathbf{X} = (1, x_1, \ldots, x_n)$.

These two expressions can be used to compute the probabilities of the classes once that the parameters have been estimated and x is observed.

ⓘ A note on the softmax parametrisation

Machine learning usually writes this same model without a baseline class, giving every class its own coefficient vector:

$$P(Y=k|X=\mathbf{x}) = \frac{e^{\beta_k^T \mathbf{x}}}{\sum_{l=1}^{K} e^{\beta_l^T \mathbf{x}}}$$

This is the **softmax** form. It is the same model — a *reparametrisation*, and an **overparametrised** one: adding a constant vector to every β_k leaves all the probabilities unchanged, so the coefficients are identified only up to that shift. Choosing a baseline, as above, is exactly what removes the ambiguity, which is why statistics does it and why single coefficients can be read as log-odds ratios here and not there.

The distinction matters when you read a library's output — `scikit-learn` fits the softmax form, `statsmodels` the baseline form — but nothing in this course depends on it, and **the exam does not ask about softmax**.

Let's fit the following model:

$$\text{group} = \beta_0 + \beta_1 \text{relwt} + \beta_1 \text{glutest}$$

We obtain the following results:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
import pandas as pd

# Assume you already loaded the Diabetes dataset into diabetes_data
# Columns: relwt, glufast, glutest, instest, sspg, group

# Recode group into integers (baseline = Normal)
diabetes_data2 = diabetes_data.copy()
diabetes_data2['group'] = diabetes_data2['group'].map({
    'Normal': 0,
    'Chemical_Diabetic': 1,
    'Overt_Diabetic': 2
}).astype(int)

# Fit multinomial logistic regression
model = smf.mnlogit("group ~ glutest", data=diabetes_data2)
result = model.fit()

# Summary with coefficients, standard errors, z-scores, p-values
print(result.summary())

# Likelihood ratio test for overall model fit
print("\nModel Likelihood Ratio Test:")
print(result.llr, result.llr_pvalue)
```

```
Optimization terminated successfully.
         Current function value: 0.100843
         Iterations 14
                          MNLogit Regression Results
==============================================================================
Dep. Variable:                  group   No. Observations:                  145
Model:                        MNLogit   Df Residuals:                      141
Method:                           MLE   Df Model:                            2
Date:                Sat, 26 Sep 2026   Pseudo R-squ.:                  0.9013
Time:                        16:16:25   Log-Likelihood:                -14.622
converged:                       True   LL-Null:                       -148.10
Covariance Type:            nonrobust   LLR p-value:                 1.076e-58
==============================================================================
   group=1       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -90.3738     42.527     -2.125      0.034    -173.726      -7.022
glutest        0.2153      0.101      2.128      0.033       0.017       0.413
------------------------------------------------------------------------------
   group=2       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept   -109.8266     43.021     -2.553      0.011    -194.147     -25.506
glutest        0.2471      0.102      2.428      0.015       0.048       0.447
==============================================================================

Model Likelihood Ratio Test:
266.9538631863749 1.0757346252010753e-58
```

We can see that the model estimates two sets of coefficients: one for `group=1` (Chemical Diabetic) and one for `group=2` (Overt Diabetic). No coefficients are estimated for `group=0` (Normal), which serves as the baseline category. Let us interpret the results:

Model Overview

- **Dependent variable**: `group`, with three diagnostic categories
- **Predictors included**: `glutest` (glucose intolerance)
- **Estimation method**: Maximum Likelihood Estimation (MLE)
- **Convergence**: Successful after 14 iterations
- **Log-likelihood**: −14.622 (final value of our cost function)
- **Pseudo R-squared**: 0.9013 (good explanatory power)
- **Likelihood Ratio Test**:
  - Statistic: 266.95
  - p-value: almost zero → highly significant improvement over the null model

Coefficient Interpretation

- The intercept for `group=1` is -90.3738. This means that the odds of being *Chemical Diabetic* versus *Normal* are $e^{-90.3738} \approx 0$ when `glutest` is zero. This is an extremely low value, indicating

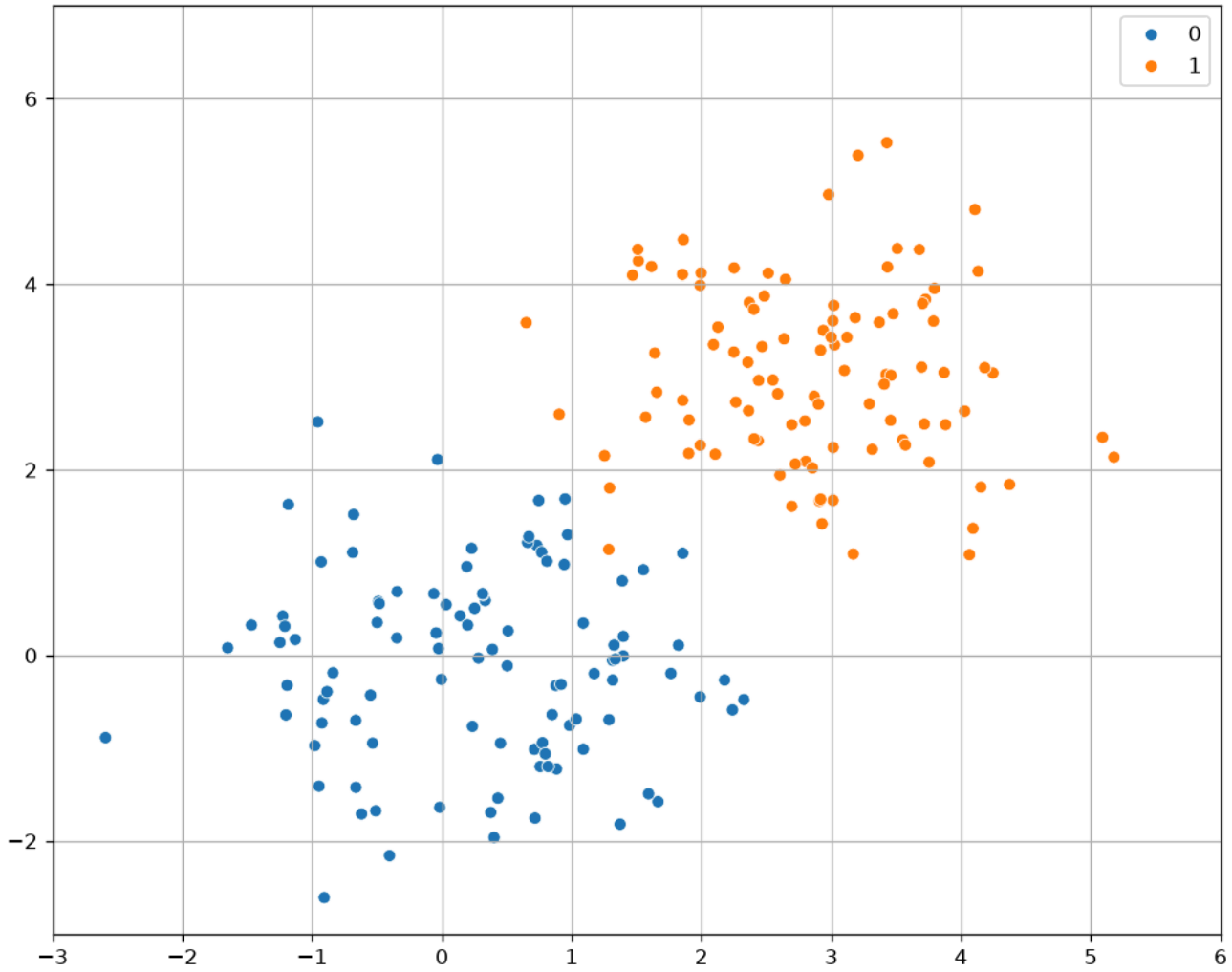
that when `glutest` is zero, it is much more probable that the subject is *Normal* rather than *Chemical Diabetic*.

- The intercept for `group=2` is -109.8266. This means that the odds of being *Overt Diabetic* versus *Normal* are $e^{-109.8266} \approx 0$ when `glutest` is zero. This indicates that when everything is zero, it is much more probable that the subject is *Normal* rather than *Overt Diabetic*. Hence, when `glutest` is zero, the subject is almost certainly *Normal*.
- The coefficient 0.2153 for `glutest` in `group=1` suggests that a one-unit increase in `glutest` increases the odds of being *Chemical Diabetic* versus *Normal* by $e^{0.2153} \approx 1.24$, an increase of about **24%**.
- The coefficient 0.2471 for `glutest` in `group=2` implies a similar increase in odds for *Overt Diabetic* versus *Normal*: $e^{0.2471} \approx 1.28$, an increase of about **28%**.

Geometrical Interpretation of the Coefficients of a Logistic Regressor

Similar to linear regression, also the coefficients of logistic regression have a geometrical interpretation, which is particularly interesting when treating the classifier from a purely predictive point of view. We will see that, while linear regression finds a «curve» that fits the data, logistic regression finds a hyperplane that separates the data.

Let us consider a simple example with bi-dimensional data $\mathbf{x} \in \mathfrak{R}^2$ as the one shown in the following:



Let us assume that we fit a logistic regressor model to this data

$$P\left(y=1|\mathbf{x}\right)=\frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2)}}$$

and find the following values for the parameters:

$$\begin{cases}\beta_0=-3.47\\ \beta_1=1.17\\ \beta_2=1.43\end{cases}$$

We know that these parameters allow to find a probability value according to the formula above. We can use these values to **classify the observations $\mathbf{x}$**. In practice, a reasonable criterion to classify observations would be:

$$\hat{y}=\begin{cases}1 & \text{if } P(y=1|\mathbf{x})\geq 0.5\\ 0 & \text{otherwise}\end{cases}$$

This makes sense as we are assigning the observations to the group for which the posterior probability $P(y|\mathbf{x})$ is higher.

To understand how the data is classified, we can look at those points in which the classifier is uncertain, which is often called **the decision boundary**, i.e., those points in which $P\left(y=1|\mathbf{x}\right)=0.5$.

We note that:

$$P\left(y=1|\mathbf{x}\right)=0.5 \Leftrightarrow e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2)}=1 \Leftrightarrow 0=\beta_0+\beta_1 x_1+\beta_2 x_2$$

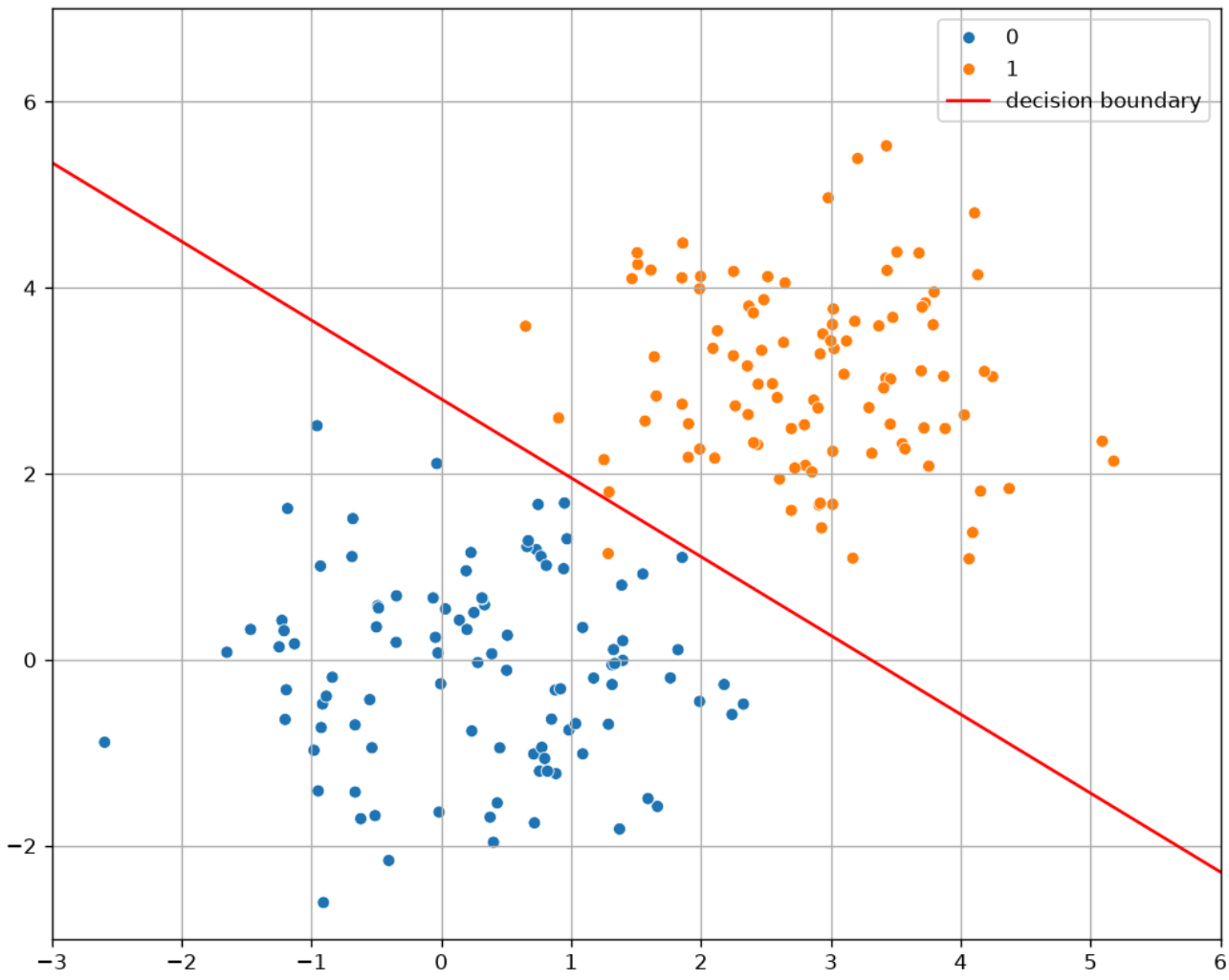
This last equation is the equation of a line (in the form $ax+by+c=0$). We can see it in explicit form:

$$x_2=-\frac{\beta_1}{\beta_2}x_1-\frac{\beta_0}{\beta_2}$$

So, we have found a line which has a

- Angular coefficient equal to $-\frac{\beta_1}{\beta_2}$;
- Intercept equal to $-\frac{\beta_0}{\beta_2}$;

If we plot this line, we obtain the **decision boundary** which separates the elements from the two classes:



As can be seen, the decision boundary found by a logistic regressor is a line. This is because **a logistic regressor is a linear classifier**, despite the logistic function is not linear!

Exercises

1. **Interaction, read aloud.** Fit `Fare ~ Age * C(Pclass)` on the Titanic data. Write one sentence saying what the interaction term means, without using the word “interaction”.
2. **When an interaction is not needed.** Fit the same model without the interaction and compare. Does the interaction earn its parameters? Justify the answer with something other than the p-value.
3. **Polynomial degree.** Fit polynomials of degree 1 to 6 to a relationship of your choice. Plot each one. Which degree would you defend to a colleague, and on what grounds — given that this chapter has no test set?
4. **Multinomial coefficients.** Fit a multinomial logistic regression of `Pclass` on `Fare` and `Age`. Interpret one coefficient with respect to the baseline category, then refit with a different baseline and confirm that the conclusions do not change.

References

- James, Witten, Hastie, Tibshirani, Taylor. *An Introduction to Statistical Learning with Applications in Python*, sections 3.3 (interactions and polynomial terms) and 4.3 (multinomial logistic regression).
- Heumann, Schomaker, Shalabh. *Introduction to Statistics and Data Analysis*, chapter 11.
- `statsmodels` formula API documentation, for the `*` and `:` operators.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Causal analysis

Chapter resources

Slides (PDF) Open in Colab Practice quiz

We have seen how to characterize the relationship between two or more variables with correlation and linear regression. We also said that **correlation is not causation**, meaning that, if we observe a correlation between two variables, that does not necessarily imply a cause-effect relationship between them. However, in many cases, it is still useful (or required!) to establish a cause-effect relationship (or lack thereof) between two variables. Consider the following examples:

- We assess that **a given disease is more frequent among smokers**. Is this just a correlation or is there a cause-effect relationship? If a smoker stops smoking, will their risk to get the disease be reduced?
- **We develop a new drug** and assess that people taking this drug are less likely to get disease X. Is this just a correlation or is there a cause-effect relationship? This is important, as people should take the drug only if benefits are clear.
- **A large company founds that product sites located in the geographical areas close to the sea are more productive than the others**. Is this just a correlation or is it a cause-effect relationship? Shall we move all sites closer to the sea?

Luckily, the field of **causal analysis** has, over the years, developed a set of tools which allow to answer those questions.

The goal of this lecture is to provide a primer of causal analysis, introducing the main concepts and discussing some techniques which can be applied in some simple cases.

Tutoring vs Performance

Let us consider an example:

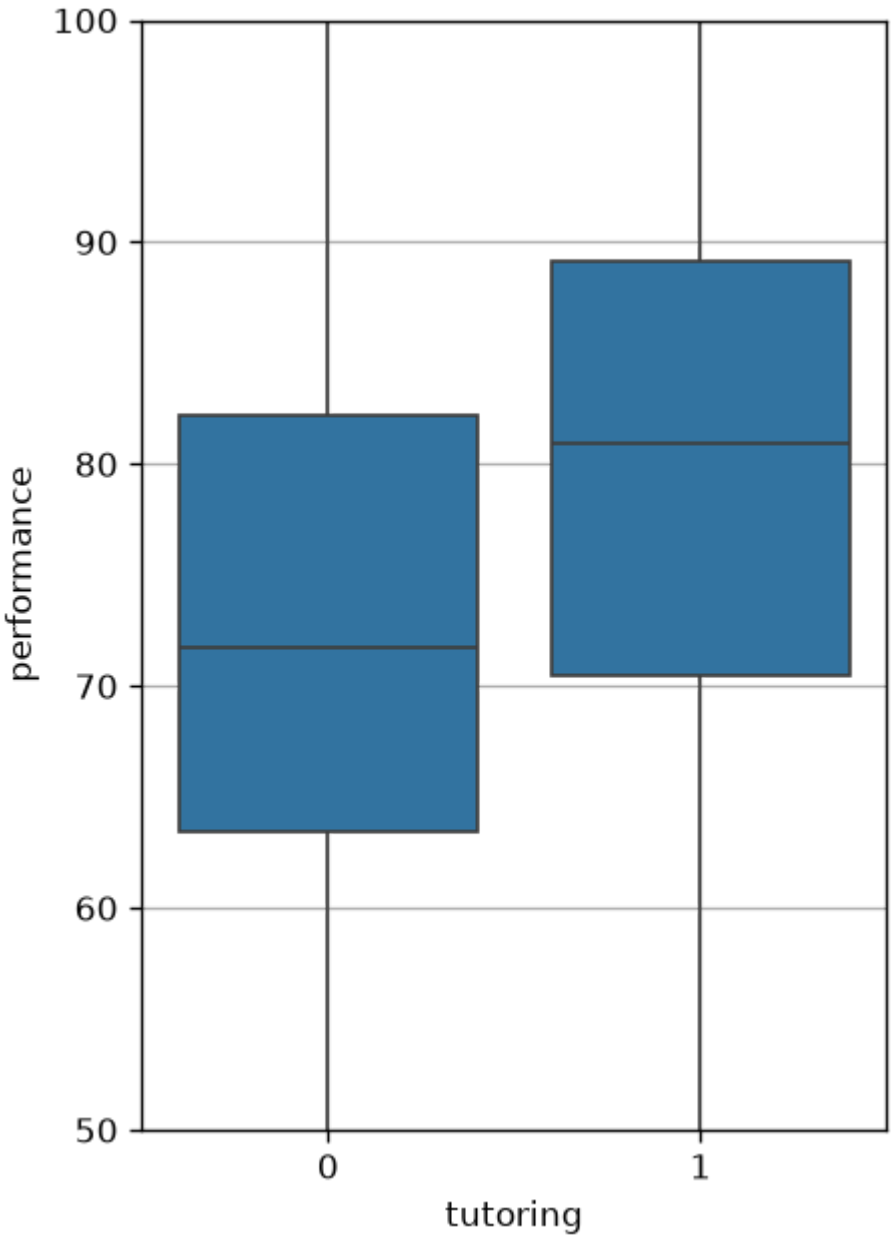
> A school wants to see if a new teaching method is effective. To do so, they ask 50 students if they want to **voluntarily enroll a tutoring program**. After one year, they measure the average performance of students who participated in the programme and compare it to those of students who did not participate in the programme.

We obtain the following data:

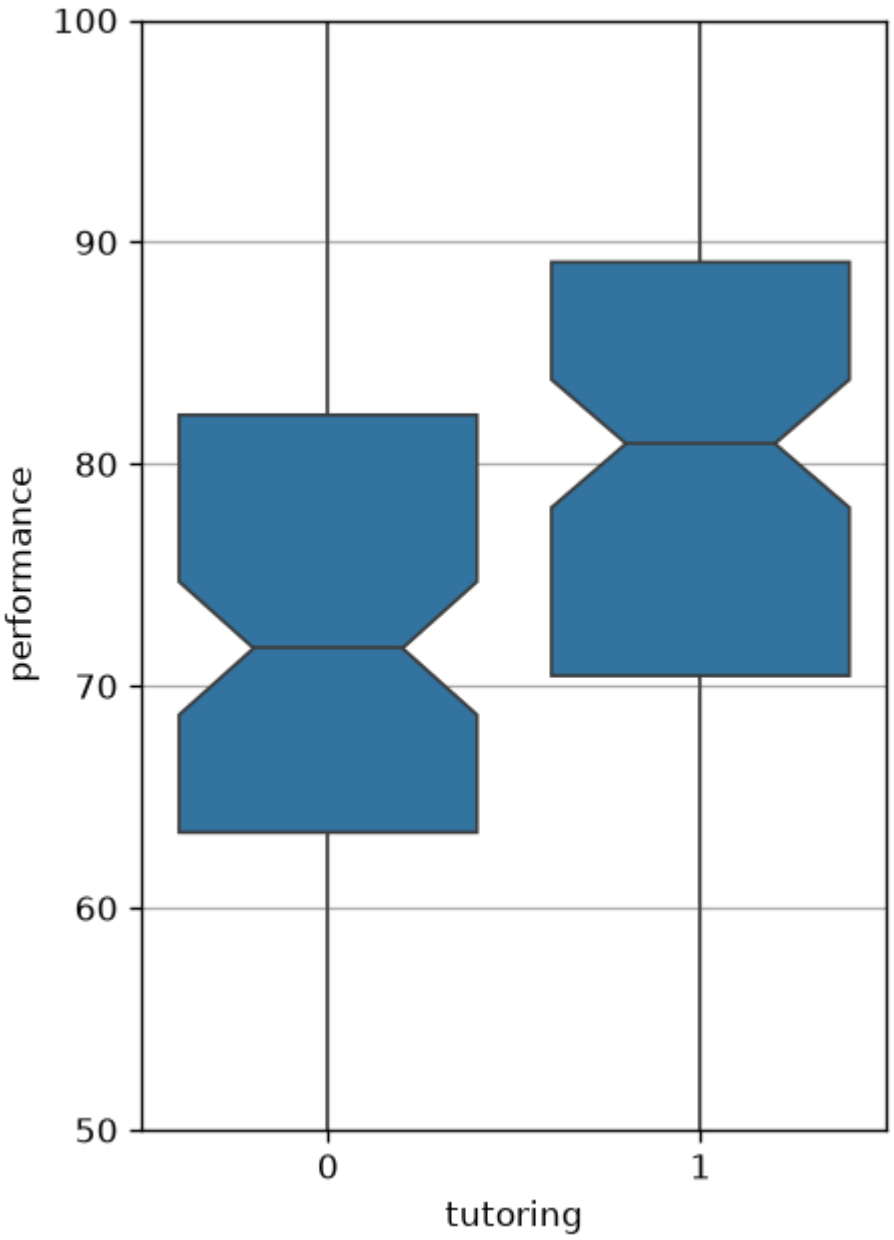
	tutoring	performance
0	0	76.432447
1	1	91.550461
2	1	85.107378
3	0	69.981030
4	1	70.111151
...	...	...
195	0	68.477160
196	0	87.783594
197	1	88.220111
198	1	93.399423
199	1	84.082421

200 rows × 2 columns

We want to understand if `tutoring` has an effect on `performance`. Let us show some boxplots:



The comparison tell us `tutoring` seems to significantly improve performance! However, we took a statistics class, so we want to check if everything is statistically significant. We add a **notch** to each plot. The notch will highlight the confidence intervals for the estimation of the median value:



Since the notches do not intersect, there is no overlap between the confidence intervals for a C.L. of 95%, hence it is very unlikely that the two median values are very close to each other. The difference seem to be statistically relevant. If we want to be very sure, we can **compute a two-sample t-Test** to ensure that the mean values of the two samples actually are distinct.

The null hypothesis will be that the two samples have the same mean. If we reject this hypothesis, then the two means are distinct. We obtain the following test statistic and p-value:

```
T-test results - T-statistic: 4.086568160195115, p-value: 6.358186591194334e-05
```

It looks like the tutoring program is a great success!

Measuring Causation - Potential

Outcomes (Counterfactuals)

The comparison above seems to be clear: we should enroll everybody in the tutoring program! But we also know that **correlation is not (necessarily) causation!**. We are seeing a correlation here - is it also causation? Of course, we want to know whether it is worth to extend the tutoring program or not!

To be able to truly assess a causal relationship between the two variables, ideally, we would like to **take the same student and let them attend and not attend the tutoring program at the same time, so that then we can compare, for the same student, the effect of tutoring alone.** This is of course impossible to do and even if we could it would be unethical. Can we ask a student not to follow a program which we suspect will harm their performance? Even if we ask the same student to first follow the tutoring, then stop, the student "before" and "after" is not the same student after all, so the two values will never be fully comparable.

We will now **enter the realm of potential outcomes**. In practice, we ask ourselves questions such as **"what if this student did not take the tutoring?"**. Let's see some notation first. We will define two variables:

- T_i: a "treatment" value indicating whether the observation i (student i) took the treatment (i.e., attended the tutoring) or not. Here we use the standard terminology of causal inference which uses the word "treatment" as in health-related sciences. We will assume this variable to be binary: $T_i = 1$ means "took the treatment", while $T_i = 0$ means "didn't take the treatment". We will call the group of observations for which $T_i(1)$ the **treatment group** and the group of observations for which $T_i(0)$ the **control group**.
- Y_i: the "outcome" of observation i. In our case, this is the average performance of student i.

If for a unit i, the treatment value is $T_i = t$, we will denote the related observation Y_i as $Y_i(t)$. We note that, for a given unit i, we will **either observe $Y_i(0)$ or $Y_i(1)$, but never both!**.

Nevertheless, we will introduce the concept of **potential outcome** as the **value we would observe if we were able to intervene on the treatment variable T_i and change it**. If for a unit i we have $T_i = 0$, we will directly observe the outcome $Y_i(0)$, but we will also **theoretically define** the potential outcome $Y_i(1)$. This **is not real and cannot be observed** but it will serve as a useful theoretical concept.

Potential outcomes are also known as **counterfactuals** because they answer to the question "what if...".

If we had potential outcomes, for each unit i we could compute the **individual treatment effect** (the effect of taking the treatment) as:

$$Y_i(1) - Y_i(0)$$

This is the effect of attending the tutoring. If we average this over the distribution of observations, we obtain the **average treatment effect**:

$$ATE = E[Y(1) - Y(0)]$$

This value will tell us **how the exam performance varies in average if the same subjects takes the tutoring or not**. As such, it would indicate **the causal effect of tutoring on examination performance**.

Alternatively, we can compute the **average treatment effect on the treated**:

$$ATT = E[Y(1) - Y(0)|T = 1]$$

This simply restricts the computation to the treated subjects, and **also establishes a causal effect**.

The Bias Arising from Correlation

Without potential outcomes, we cannot compute the ATE or ATT scores. Indeed, we have **two distinct groups**, the **treated** (i.e., those observations i such that $T_i = 1$) and the **untreated**, also known as **control group** (i.e., those observations i such that $T_i = 0$).

We could think to replace that calculation with the only data we can observe:

$$E[Y(1)|T = 1] - E[Y(0)|T = 0]$$

This is what **a correlation can measure**: the change in average value of Y in two distinct groups with different characteristics.

Note that there are no potential outcomes in the expression above as we directly observe $Y_i(1)$ when $T_i = 1$ and we directly observe $Y_i(0)$ when $T_i = 0$. However, the expression above is now **comparing different groups**. Indeed, the subjects who took the treatment are different from the ones who did not take the treatment, and we do not know if their exam performance scores are really comparable. **Imagine that, for some reason, students in group 1 are, by chance, better students, then the difference we measure is going to be biased and will not reveal a cause-effect relationship**.

We can show this in formulas. Let's add and subtract the counterfactual term $E[Y(0)|T = 1]$ to the expression above:

$$E[Y(1)|T = 1] - E[Y(0)|T = 0] = E[Y(1)|T = 1] - E[Y(0)|T = 0] - E[Y(0)|T = 1] + E$$

We note that:

$$ATT = E[Y(1) - Y(0)|T = 1] = E[Y(1)|T = 1] - E[Y(0)|T = 1]$$

Hence, we can see the expression above as follows:

$$E[Y(1)|T=1] - E[Y(0)|T=0] = E[Y(1)|T=1] - E[Y(0)|T=1] + E[Y(0)|T=1] - E$$

$$= ATT + E[Y(0)|T=1] - E[Y(0)|T=0]$$

We will call the last term the bias:

$$BIAS = E[Y(0)|T=1] - E[Y(0)|T=0]$$

So the whole expression can be seen as:

$$E[Y(1)|T=1] - E[Y(0)|T=0] = ATT + BIAS$$

which shows that the effect given by the correlation (left term) is **biased**. Note that the bias cannot be computed (it contains a counterfactual term), but we can interpret it as follows:

> The bias measures how the treated ($T=1$) and control ($T=0$) groups would differ if no one had taken the treatment.

In practice:

- If the groups are similar to each other (e.g., treated and control group are mixed students with similar abilities), then the bias will be small or zero. Indeed, in this case, the difference in exam performance among the two groups if everybody sleeps less than 8 hours should be same, as students have similar abilities.
- If the groups are dissimilar (e.g., control students are all better students), then the bias will be different from zero. Indeed, if students in the control group are "better students", if all sleep less than 8 hours, students in the control group will perform better in average, but this has nothing to do with sleeping less or more or with cause-effect relationships among the observed variables.

Note that, given the observations above, we now know **when association allows to establish a cause-effect mechanism**. This happens when the bias is zero:

$$BIAS = E[Y(0)|T=1] - E[Y(0)|T=0] = 0$$

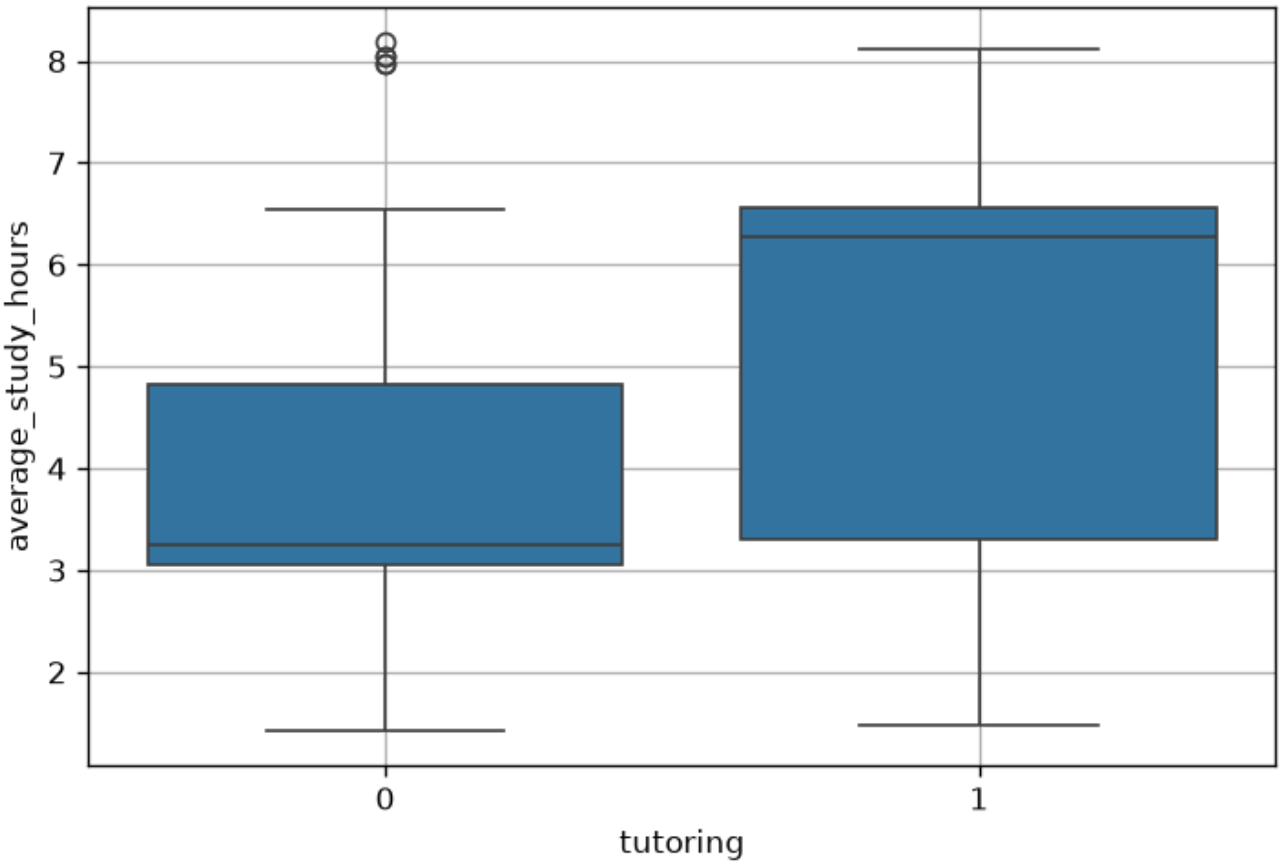
That is to say, when the difference between treated and untreated is zero (or at least very small).

Given this intuition, we now try to estimate the students' abilities by asking how many hours they study per day. We get the following data:

	tutoring	performance	average_study_hours
0	0	76.432447	3.316778
1	1	91.550461	8.025442
2	1	85.107378	6.433760
3	0	69.981030	4.758812
4	1	70.111151	3.151239
...	...	...	...
195	0	68.477160	3.124462
196	0	87.783594	6.338848
197	1	88.220111	7.859334
198	1	93.399423	7.907677
199	1	84.082421	6.264832

200 rows × 3 columns

We cannot measure bias from this data because we do not have access to counterfactual observations, but, now that we have access to a proxy for students' abilities, we can try to see if, in the two groups, abilities are equally distributed. Let's see this with a boxplot:

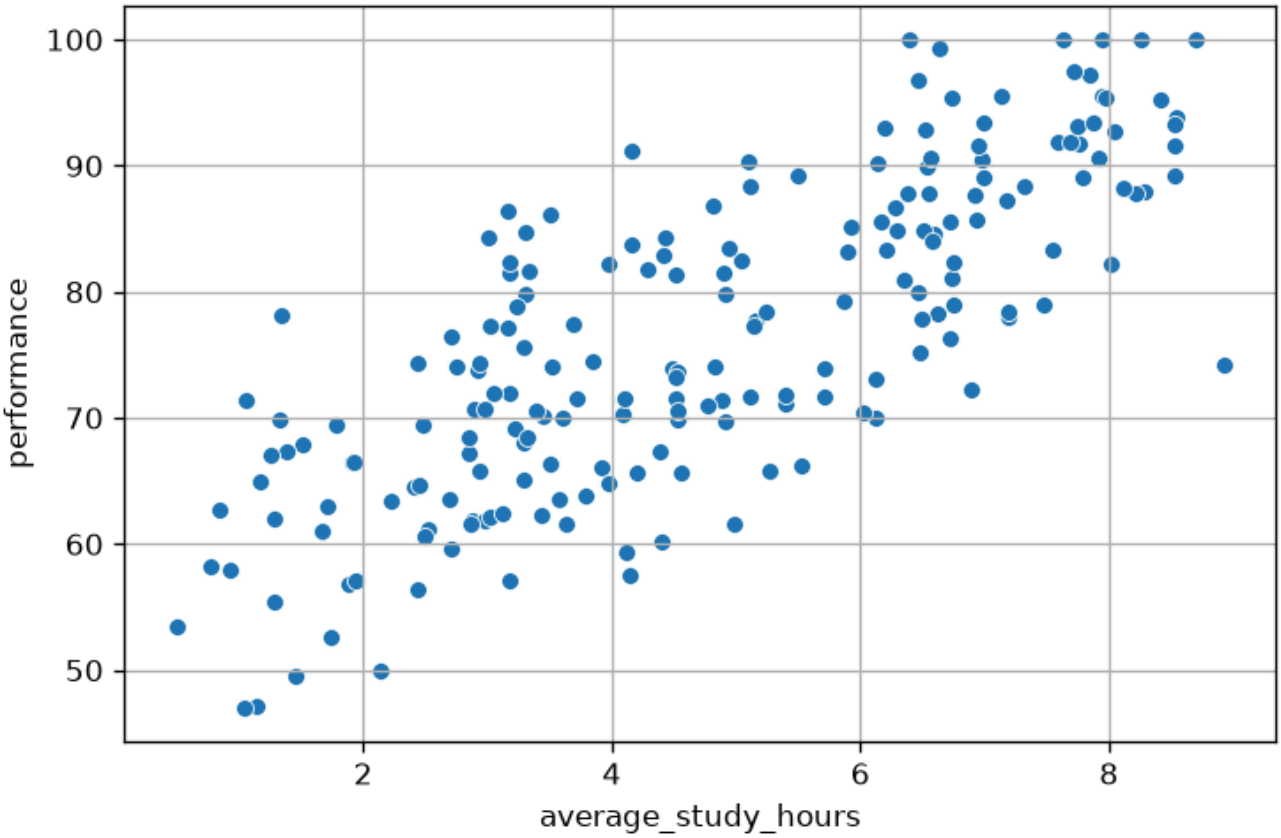


From the picture above, we see that students' skills are not equally distributed in the two groups. **This is not enough to say that the bias is different from zero, as we are not observing counterfactuals,**

but we now see that the difference in outcomes could be due to two different causes:

- Tutoring improves exam performance.
- Some of the difference may be due to the fact that students following the tutoring program are also students who study more in average (**maybe they are more motivated, so they study more and also choose the tutoring program**) and this may affect the final performance.

We can check the hypothesis that average study hours affects performance with a scatterplot:



From the picture above there seem to be a correlation between the two!

Now that we have an intuition into what may make correlation be not indicative of causation, let's see how to reduce bias to establish cause-effect relationships.

Randomized Controlled Trials (RCT)

We have seen that correlation is causation when the treated and control groups are comparable for everything except the treatment. The most robust method to remove this bias is via **randomized experiments**, or **Randomized Controlled Trials** (RCT).

A randomized experiment **randomly assigns individuals in a population to a treatment or a control group**. By performing a random assignment, we wish to make the two groups indistinguishable. Of course **one of the two groups will take the treatment and this will likely make the two groups distinguishable**, but we wish to make sure that, **apart from the treatment, the two groups are indeed indistinguishable**.

It can be shown that random assignment makes sure that the potential outcomes are conditionally independent given the treatment:

$$Y(0) \perp Y(1)|T$$

So, if the treatment T is fixed, the potential outcomes are independent. For instance, if we fix $T = 1$ (so we are taking subjects who attended the tutoring), then knowing the observed value $Y_i(1)$ (i.e., the observed exam performance of student i), does not tell me anything about the potential outcome $Y_i(0)$ (the exam performance that the same student would obtain if they did not attend the tutoring).

> Recall our example on conditional independence: if height and vocabulary are conditionally independent given age, this means that height and vocabulary are independent in the same age groups, while they are not across age groups, so **age is the only thing that make height and vocabulary dependent**. Similarly, if $Y(0) \perp Y(1)|T$, then **the treatment is the only thing generating a difference between the outcome in the treated and in the control group**.

If this conditional independence is valid, then:

$$E[Y(0)|T = 0] = E[Y(0)|T = 1] = E[Y(0)]$$

and:

$$E[Y(1)|T = 0] = E[Y(1)|T = 1] = E[Y(1)]$$

Which leads to:

$$E[Y(1)|T = 1] - E[Y(0)|T = 0] = E[Y(1)] - E[Y(0)] = E[Y(1) - Y(0)] = ATE$$

Hence, in this case, **correlation would be causation**.

Randomizing tutoring

We now have a tool to remove the bias probably arising from our prior experiment. Let's say we could repeat the experiment, now with random assignment.

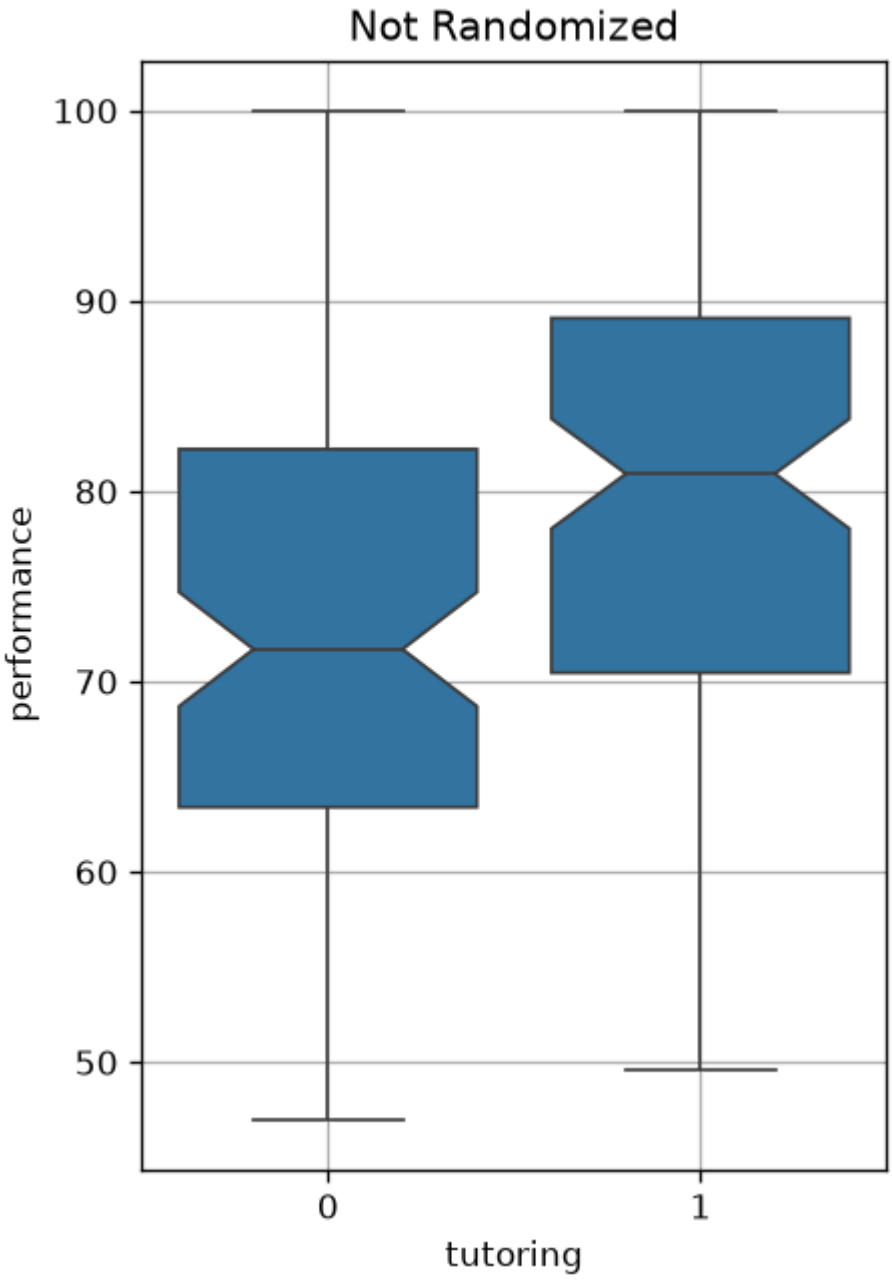
Word of caution: this is not always possible. For instance, in this case, it would not be very polite to "force" students to enroll programs, so we may not be able to carry out this experiment, but let us assume for a moment that we can.

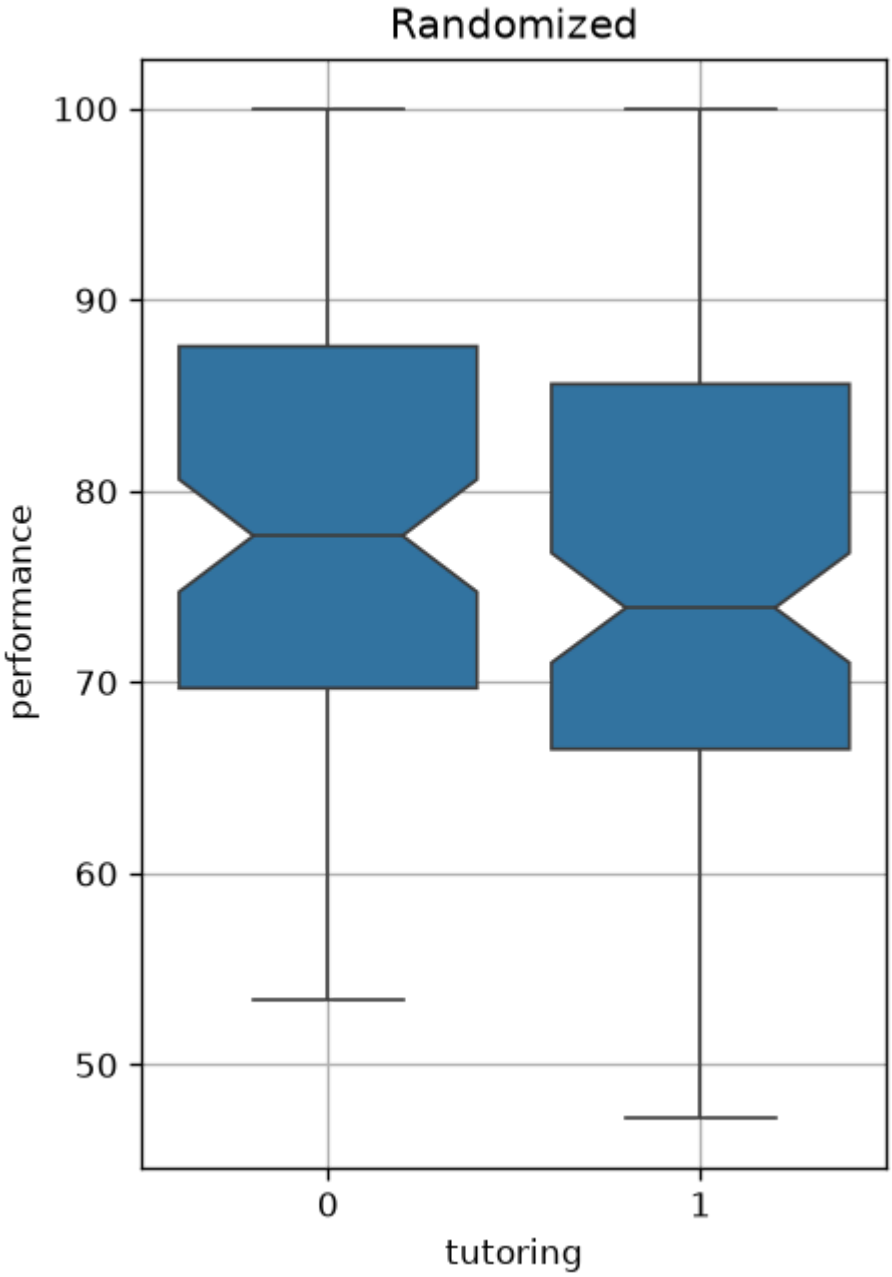
We repeat the experiment and obtain these results:

	tutoring	performance
0	0	76.432447
1	1	91.550461
2	1	85.107378
3	0	69.981030
4	1	70.111151
...	...	...
195	0	68.477160
196	0	87.783594
197	0	87.970111
198	1	93.399423
199	1	84.082421

200 rows × 2 columns

Let's visualize the boxplots:


Not Randomized
100
90
80
70
60
50
performance
0
1
tutoring



We now see a less dramatic (and actually negative!) improvement. What happened?

By performing a random assignment, we got rid of selection bias. Students voluntarily choosing to enroll in the programme were probably more motivated and keen to study. We now see that the impact of tutoring is very small. We can perform a t-test to assess if this is statistically significant:

```
T-test results - T-statistic: -1.1327518742165315, p-value: 0.2586882978200425
```

We got a large p-value, the difference is not statistically significant.

Causal Effect and Observational Studies

We have seen how to interpret an observed association as a cause-effect relationship we need to reduce bias. The gold standard is to use randomized controlled trials, however **this is not always**

feasible or ethical. Consider for instance **the problem of estimating the effect of smoking on the development of a given disease**. To perform a randomized experiment, we should select subjects randomly in two groups and ask people in one of the two groups to smoke. **This is of course unethical, considering that we suspect that smoking has a bad effect on health**.

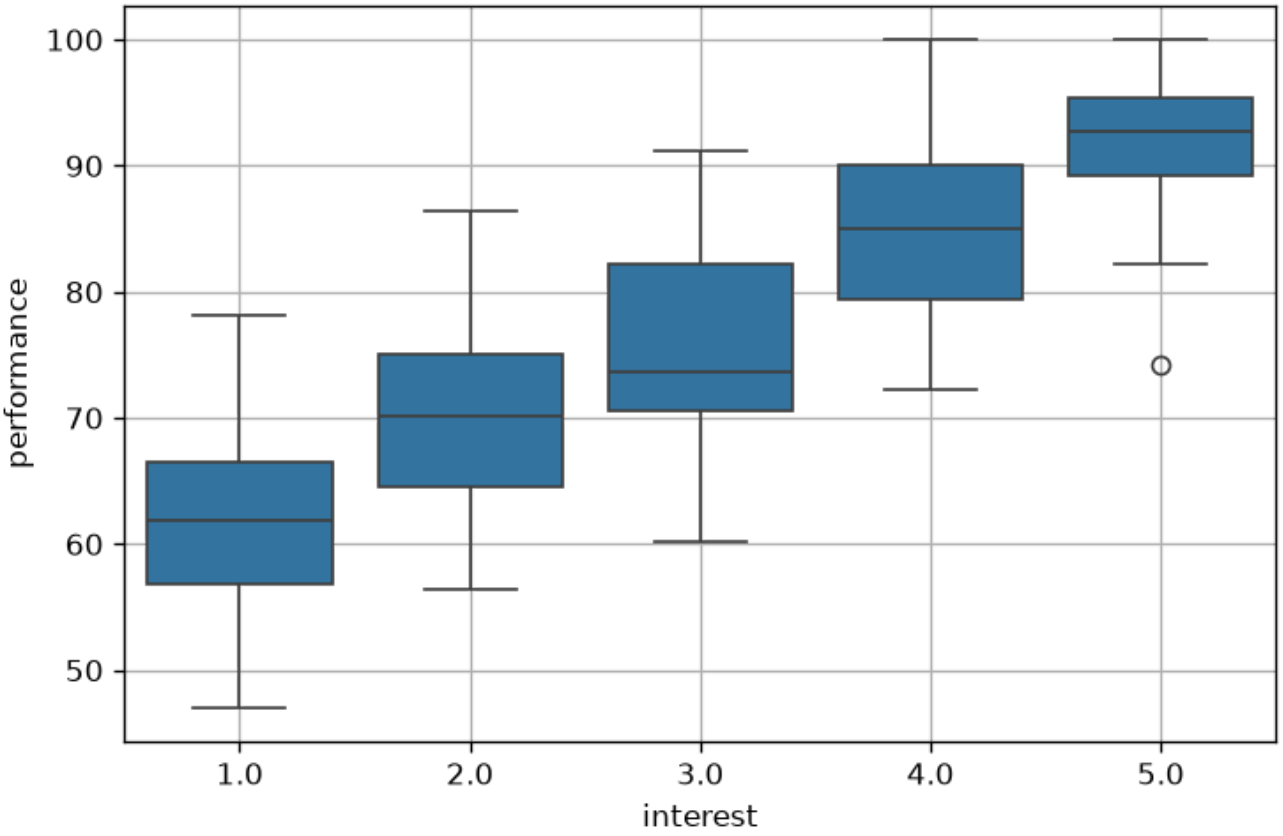
In these cases, we can only resort to **observational data**: we collect data of smokers and non-smokers and observe association in the data. However, we know that this process is **subject to bias**. How do we deal with it? In this part of the lecture, we will see that **graphical causal models** give us a framework to **set our believes on where bias come from**. If we can accurately model the source of bias, we can remove it by **controlling** on the variable "causing" the bias.

Let us consider our past example: students and tutoring. Let us assume that we cannot perform the random assignment (again, it's unethical!). Since we imagine that **the interest of students towards study can be a source of bias**, we now ask students to also tell us how interested they are in studying in general in a scale from 1 to 5. We obtain the following observations:

	tutoring	performance	interest
0	0	76.432447	2.0
1	1	91.550461	5.0
2	1	85.107378	4.0
3	0	69.981030	3.0
4	1	70.111151	2.0
...	...	...	...
195	0	68.477160	2.0
196	0	87.783594	4.0
197	1	88.220111	5.0
198	1	93.399423	5.0
199	1	84.082421	4.0

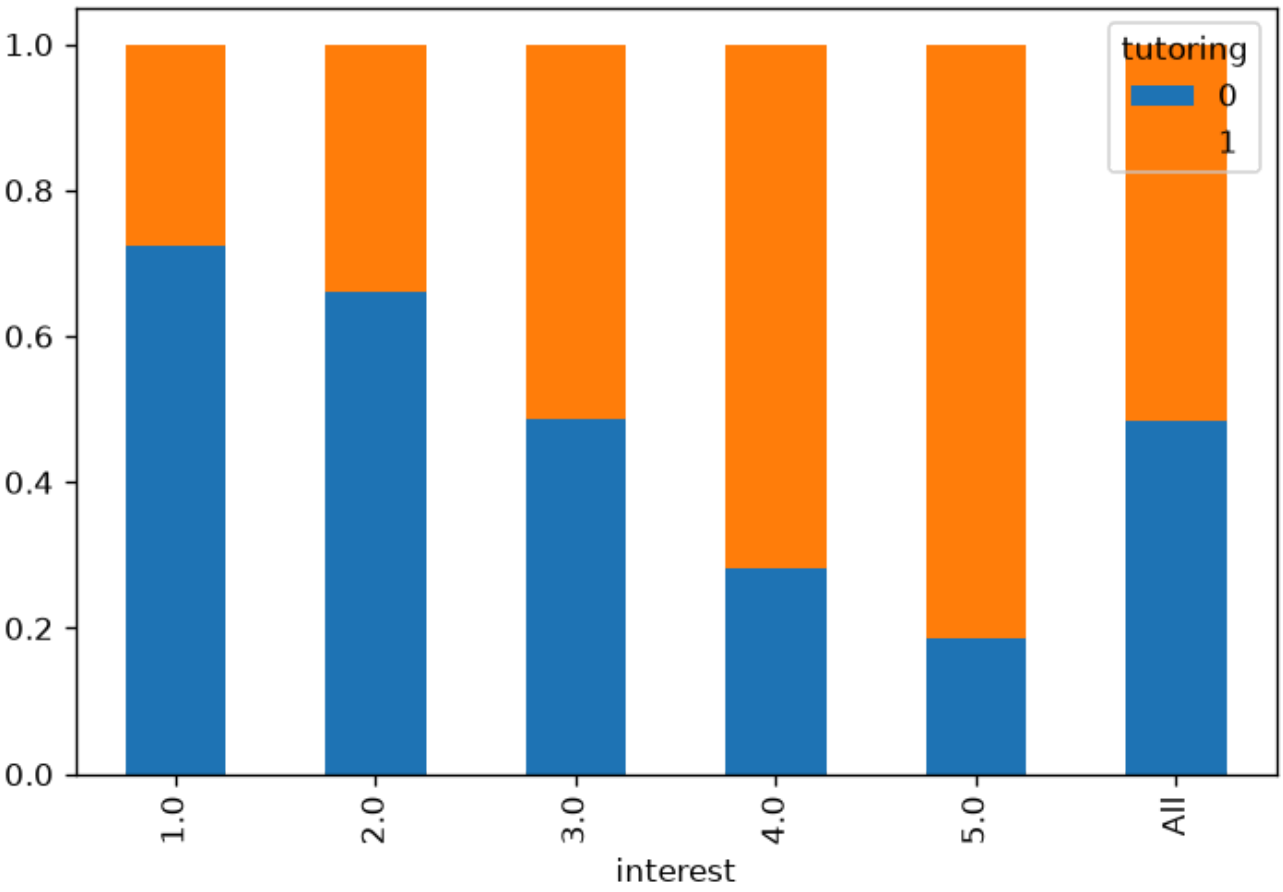
200 rows × 3 columns

These are the same observations as the first case (before random assignment), but with an additional variable. We suspected that **interested students have an average higher performance and that interested students are more likely to enroll in the tutoring program**. Let us see graphically if this is true. If we split the data by interest and compute the boxplots of performance, we get the following graph:

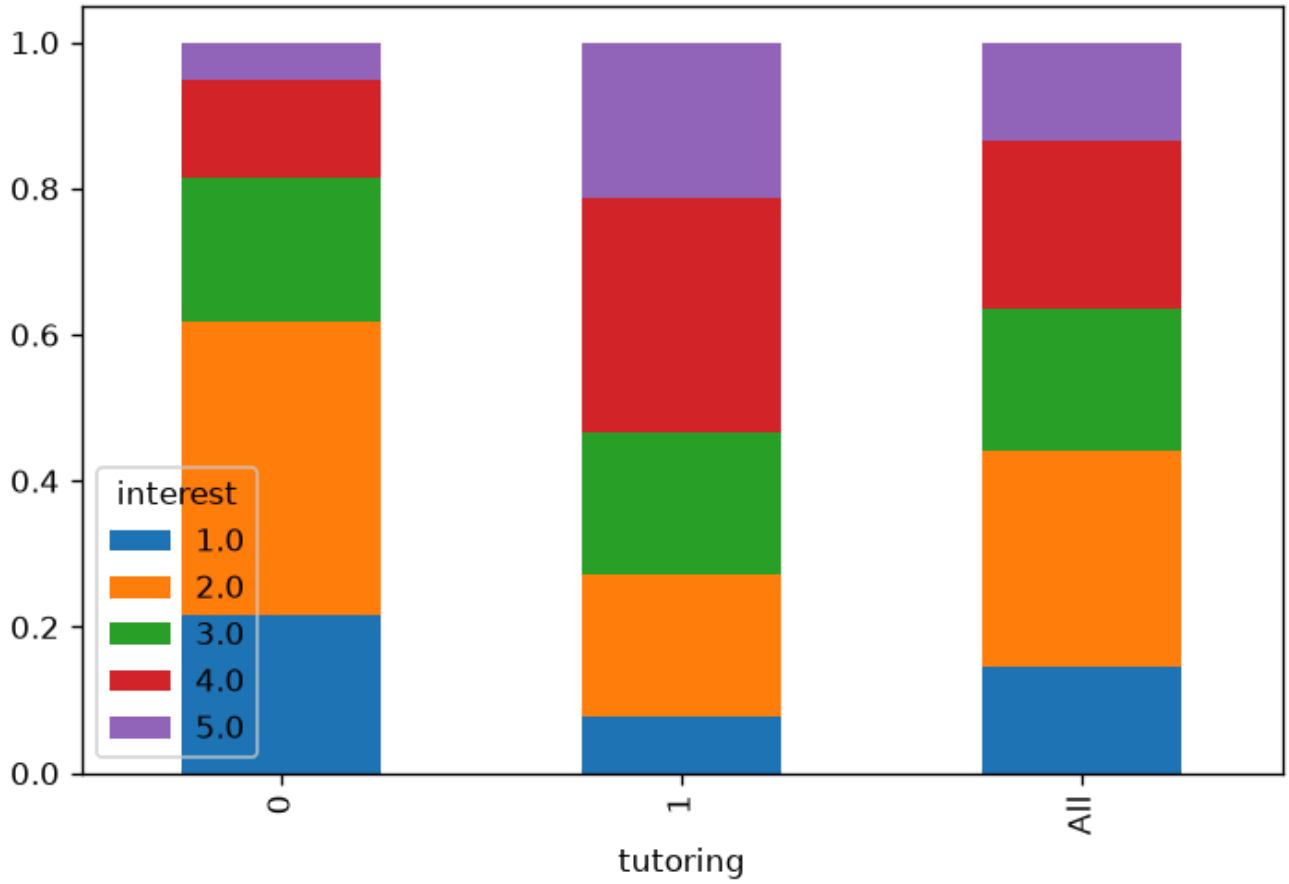


We were right! (ok, this is an exaggerated picture, but this example is made up...)

Let us check how correlated are `interest` and `tutoring` with a barplot:



We can also check the distribution of interest in the two groups:



We can go ahead and compute a χ^2 test of independence to check if the association between the two variables is statistically significant:

```
Chi-square statistic: 31.22
Cramer V statistic: 0.40
Chi-square p-value: 2.761663e-06
```

We were right also in this case! Interest affects tutoring, hence interest is our source of bias!

The correlation between the `interest` variable and `tutoring` is making the "treated" (`tutoring=1`) and the "control" (`tutoring=0`) groups not comparable. Indeed, if people in the treated group have more chances of getting a higher response value (performance), then no wonder that we observe such dramatic differences among the two groups! In other words, **the effect we observe may be due to the bias introduced by `interest`, the effect of `tutoring`, or a combination of both**.

Controlling for `interest`

How do we make the groups more comparable? Recall that we want:

$$Y(0) \perp Y(1)|T$$

We have seen that this is not true because of `interest`. Indeed, if we pick a person in the group $T = 0$ with a high $Y(0)$, we can imagine this is due to a large value of `interest`, so we can easily

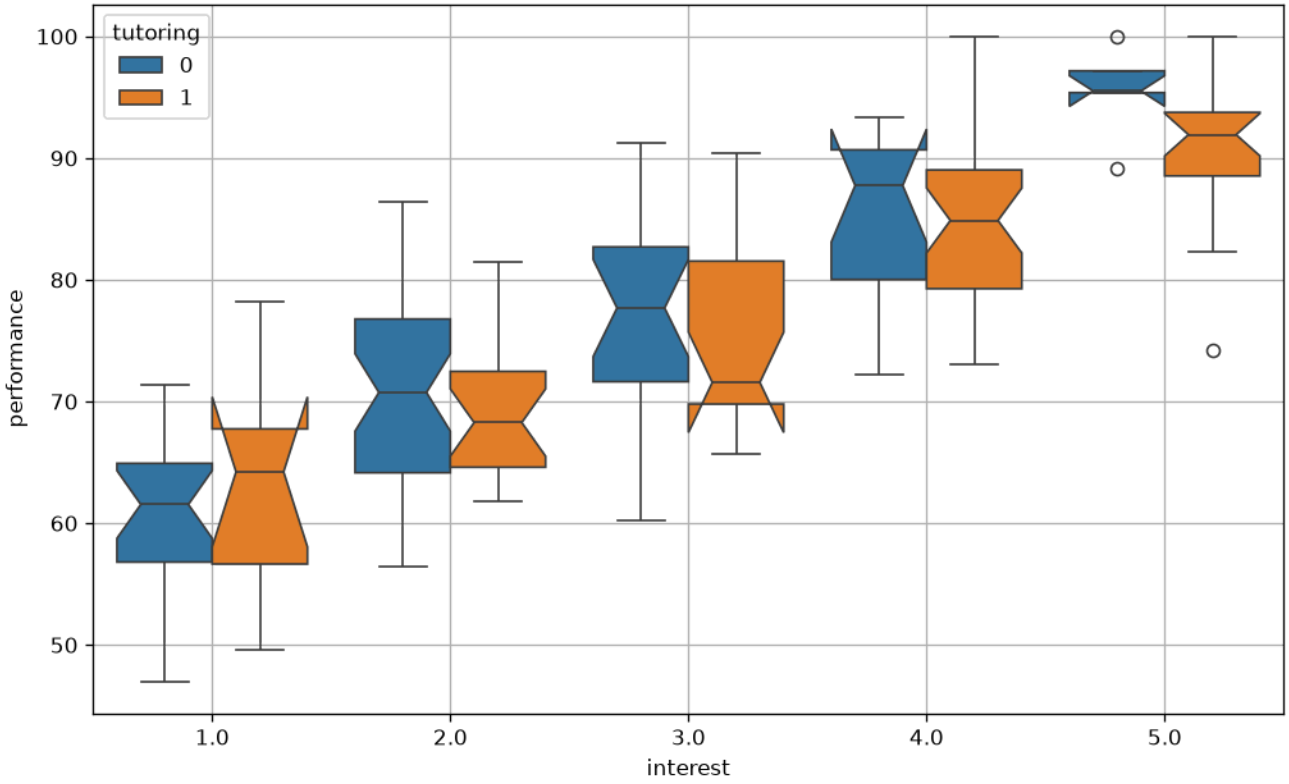
predict that $Y(1)$ will also be large. **If we can predict $Y(1)$ from $Y(0)$, then they are not independent (conditioned on $T = 1$ in this case)**.

To obtain the statement above, we should **perform a random assignment**, which we cannot do. However, we know that `interest` is the source of bias. Let us assume that it is the **only source of bias**. Then, if X represents `interest`, we can say that:

$$Y(0) \perp Y(1)|T, X$$

Indeed, if we pick a person from a group $T = 0$ and with a given interest value $X = 4$, then, given a large $Y(0)$ we cannot really say anything about $Y(1)$. Indeed, if $X = 4$ and $Y(0) = 80$, we have a student with great interest and high performance. What can we say about $Y(1)$? We could say that it will be high (the student has high interest), but any other student in this group will probably have a high score, so we cannot predict anything which will be significantly different from random (i.e., from the score of another random student within this group).

But, how do we condition on X? Well, the easiest way to do it, is to **divide the data into different groups based on the values of X and assess any association within the groups**. Let us do it:



In the plot above, we compare the effect of tutoring in the different `interest` groups. As we can see, the association between `tutoring` and `performance` tends to vanish within the groups. This is shown by the fact that the notches (confidence intervals) tend to intersect. We can get a clearer picture by performing 5 different two-sample statistical tests assessing whether the means of `performance` in the two groups differ in a statistically significant way:

	t_statistic	p_value
interest		
1.0	0.934780	0.358187
2.0	-0.781177	0.437931
3.0	-0.867406	0.391310
4.0	0.164684	0.869947
5.0	-1.383074	0.178871

As we can see, there is no significant association between `performance` and `tutoring`, which suggests the lack of a causal effect.

The act of conditioning on X (e.g., by dividing into groups) is called "controlling for X".

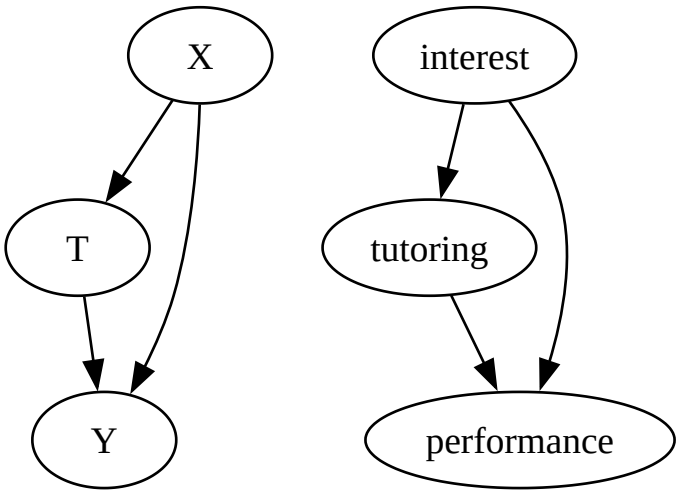
Graphical Causal Models

We have seen a simple case in which we could identify the main source of bias which was leading to an observed association not implying a cause-effect mechanism. However, not all cases are so simple and **more than one variable may be responsible for bias**. The question now is

> How do we identify the variables we should control for?

To help us in this quest, we can use graphical causal models. A graphical causal model is a **directed acyclic graph** formalizing cause-effect relationships between variables. **Each node is a random variable, while a directed edge $A \rightarrow B$ indicates that A causes B.**

The causal graph describing the hypotheses we made in this example will look like this:



Indeed, we expect `interest` (X) to influence both `performance` (Y) and whether a subject belong to the treated or control group, i.e., the `tutoring` variable (T). From the model above, we can identify

`interest` as a source of bias, but **as we will see shortly, we can use a set of rules to identify which variables we should control for or not control for from a graphical model**.

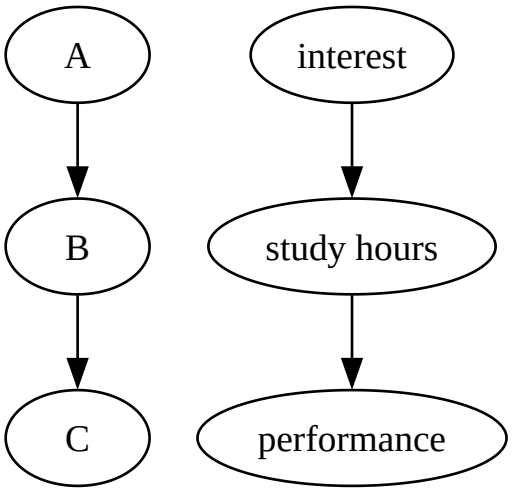
Before proceeding, a short note on **how graphical models are created**: these are generally designed by the data analyst who encodes their own **beliefs** on how the world works. These come from a "Bayesian" view of the world in which we first establish our beliefs and then use a theoretical framework to draw conclusions from them.

Main Structures of Causal Graphical Models

Once we form a graphical model, we can identify a set of standard structures which will allow us to choose which variables to control for. We will see the main structures, but others beside the ones discussed here may exist.

Chains

The most straightforward structure is a chain of this kind:



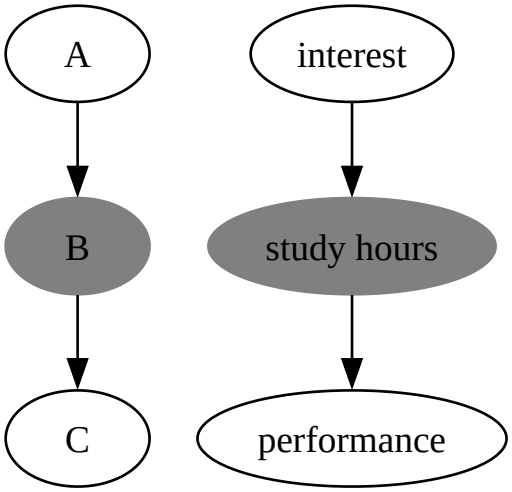
The example above, `interest` in studying causes the number of `study hours`, which in turn causes `performance`. Let us imagine no other variable is involved (i.e., `interest` is the only variable causing `study hours` and `performance` is only caused by `study hours`). Then we will easily find that:

- `interest` and `study hours` will be associated;
- `study hours` and `performance` will be associated.
- `interest` and `performance` will be associated.

Indeed, for instance:

- If a person has a great interest, then they will study a lot and have good performance;
- If a person has good performance, we can infer that it studies a lot and hence has great interest;
- If a person studies a lot, we infer that it has great interest and good performance;

Let say we **control for** B, `study hours`. We will denote the graph as follows (variable we are controlling for is in gray):



If we **control for B, `study hours`, we will find that `interest` and `performance` are not associated anymore**. Indeed:

> If we consider a group of people who study the same amount of hours and I take a person with good performance, then I don't know if they have great interest or not. Or better, I cannot distinguish a person with great interest from a person with little interest as they all study the same amount of hours.

In mathematical terms, we know that in general:

$$A \not\perp C$$

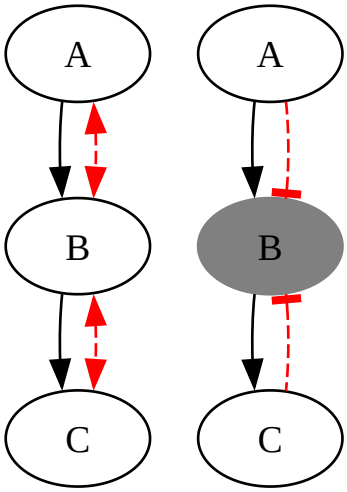
but, by controlling for B, we obtain:

$$A \perp C | B$$

Hence, in groups with the same value of B, we will see an independence between A and C.

In general, we say that **conditioning on B blocks the dependency between A and C**, hence **while there is a causal effect between A and C, we cannot see it anymore when we condition on B**.

The graph below shows two identical graphs, except that we are conditioning on B in the version on the right. Besides solid black arrows denoting cause-effect relationships, we also show **in red how dependency flows**.



- In the graph on the left, dependency flows in a bidirectional way **through variable** B. This means that A depends on B and C and similarly C depends on B and A;
- In the graph on the right, conditioning on B "stops" the dependency. Now:
  - C and A are independent of B. This makes sense because conditioning on B means fixing a value for B, so A and C will be independent of the constant;
  - A and C are independent (we are conditioning on B, so we have $A \perp C|B$).

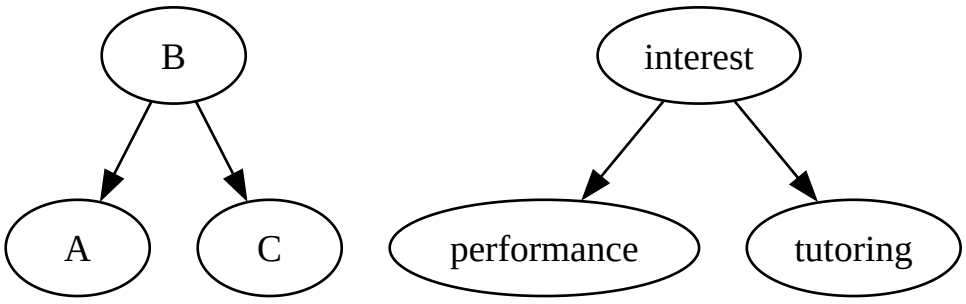
As a result, if we want to study the relation between A and C, we **should not condition on B**, as this conditioning will make the association between A and C vanish, even if there is a causal effect between these variables.

We are encountering an important concept:

> Conditioning on **any** variable is not a good way to remove bias. If we condition on the "wrong" variable, we may introduce some bias which was not there in the first place and estimate the lack of a causal effect even in cases in which the causal effect is there.

Forks

We have a fork structure when a variable causes two other variables. Let us see an example:



In this case B (`interest`) is causing both A (`performance`) and C (`tutoring`). This is the example we have seen before. In this case, as we have seen, we may see a correlation between `tutoring` and

`performance`. Indeed, a large value of `performance` will probably mean that `interest` is high, hence `tutoring` will probably be equal to 1.

> We will call B (or `interest`) the **common cause**.

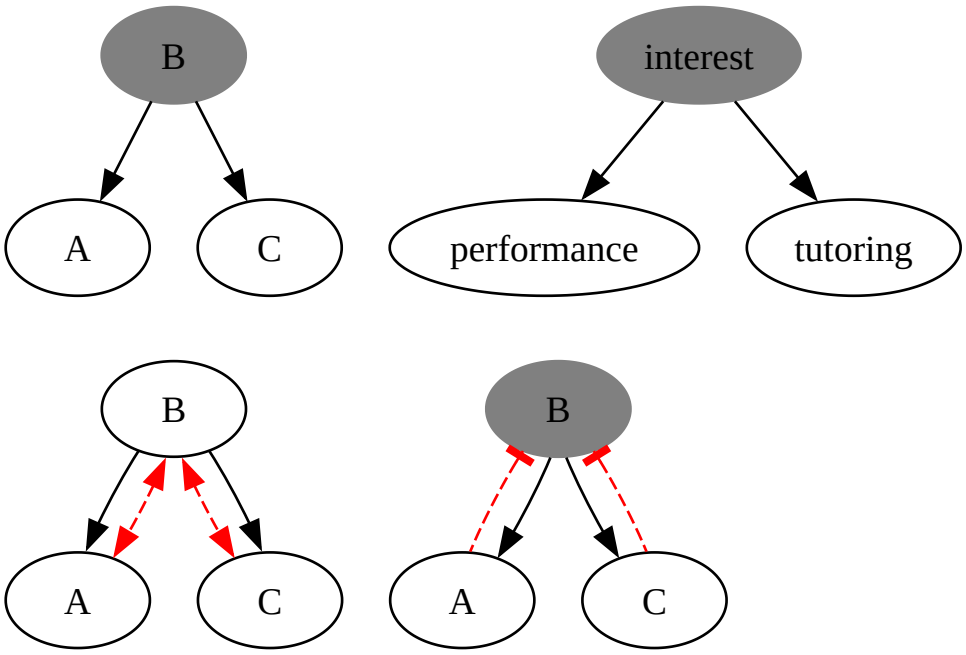
Although we will likely observe correlations between any pair of the A, B, and C variables, we know from the graph that there is no cause-effect relationship between A and C.

We say that the fork structure **creates a backdoor path such that dependence (or association) flows through B (the backdoor) to go from A to C**.

As we noted in the previous example, conditioning on B (`interest`) makes the association between A and C vanish:

$$A \perp C | B$$

Hence in the following graph, we will observe no association between `performance` and `tutoring`, hence confirming the lack of a causal effect:

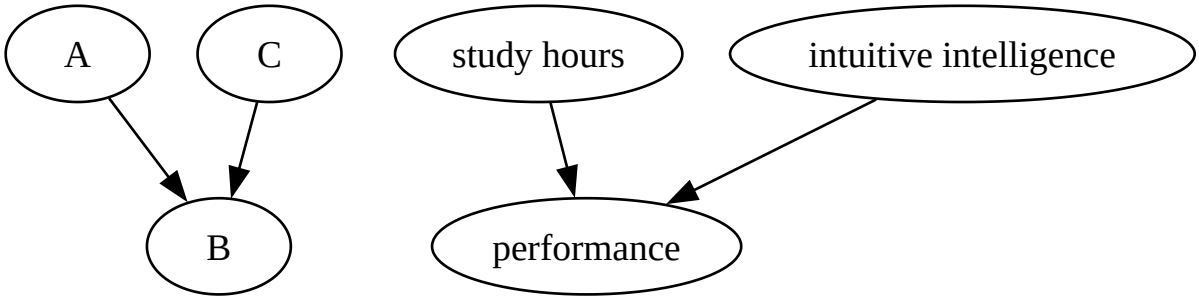


When we do not condition on B, dependency (hence association) flows from A to C through B. Hence, **we observe an association which is not due to a causal effect**.

When we condition on B, dependency (hence association) does not flow through B, so A and C will be independent and we will measure no association, as it should be.

Colliders

The last, important structure, is a collider. This is in some sense the opposite of a fork: a variable is caused by two different variables. Let us see this with an example:



The graph above tells us that performance is caused by `study hours` and `intuitive intelligence`. We are making the assumption here that `intuitive intelligence` does not affect `study hours` in any way. This might be inaccurate, but we will anyway assume so for the sake of a simple example.

This structure is called a **collider**, because two arrows (from A and C) collide on a single node (B).

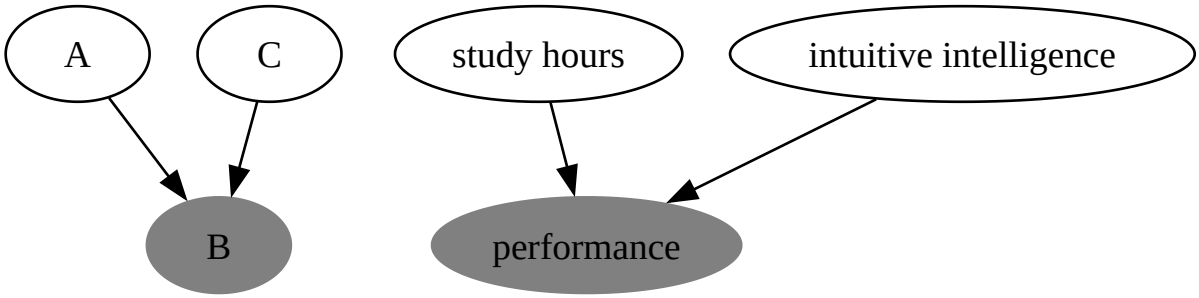
A collider **blocks dependency**. This means that by default, if we do not condition on `performance`, we will observe no association between `study hours` and `intuitive intelligence`. Indeed:

- If we know that a person studies a lot, we cannot really say anything about their intuitive intelligence;
- If we know that a person has intuitive intelligence, we cannot say how much they study.

So, in the graph above:

$$A \perp C$$

Let us not consider a case in which we condition on B (`performance`), as shown in the following graph:



Conditioning on `performance` changes things. Indeed:

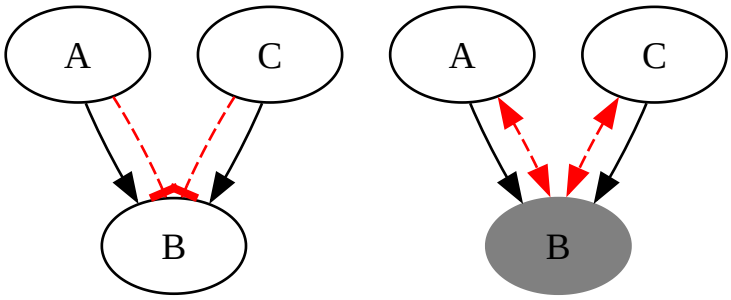
- If we know that a student has a **good performance** (i.e., we consider the group of good performing students when controlling), then **knowing that they do not study a lot, will tell us that they probably have good intuitive intelligence**. Indeed, if the fixed performance cannot be explained by the large number of study hours, it must be due to good intuitive intelligence;
- Conversely, if we know that a good performing student has bad intuitive intelligence, then their performance is probably due to a large number of study hours.

In practice, conditioning on `performance` makes `study hours` and `intuitive intelligence` dependent:

$$A \not\perp C|B$$

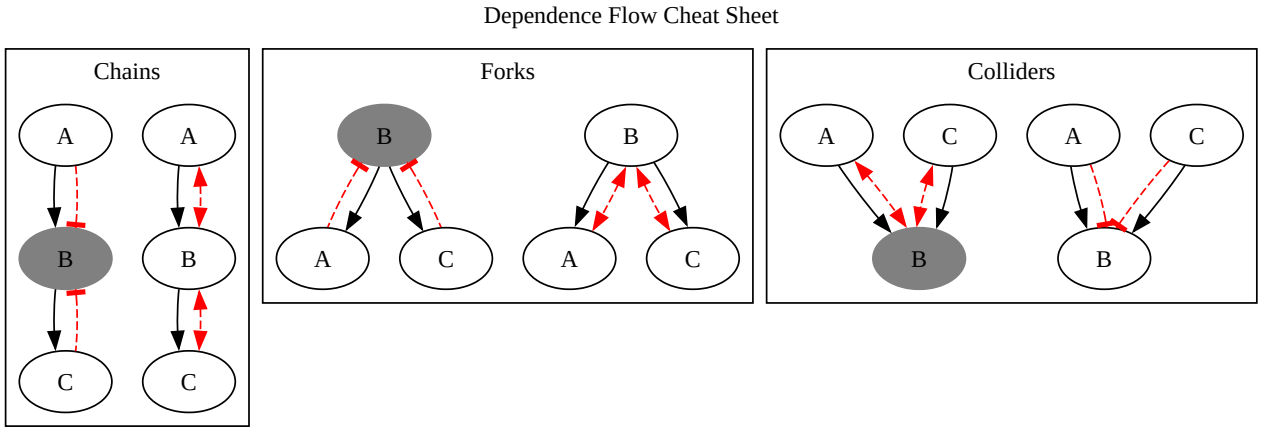
This phenomenon is called **explaining away**, because, one we fix B (`performance`), knowing A (`study hours`) already explains B, hence making C less likely.

In general, **a collider blocks dependency, while conditioning on it unblocks a dependency path**. We can use the same conventions as before to illustrate this effect:



Dependence Flow Rules

Using these structures, we can determine if two nodes A and B are dependent by checking **if an unblocked path exists between A and B**. This can be done by combining the rules we have seen above, which are also summarized in the following figure:



The cheat sheet above is replicated from here and here. You can find more information at those URLs.

We will say that a path between A and B is blocked if:

- It contains a non-collider that **has been conditioned on**;
- It contains a collider that **has not been conditioned on** and has no descendants that have been conditioned on.

Let us see an example from here and let us try to answer some questions:

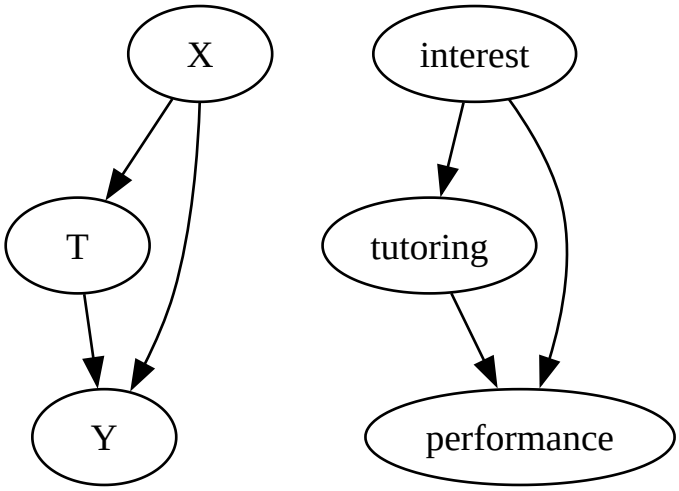
> ⓘ **Working through d-separation by hand**
>
> Deciding whether two nodes are independent given a set of others is a mechanical procedure — trace every path, block it or not according to the three rules above — and it is worth doing on paper a few times. The exercises at the end of the chapter ask you to. In practice you will rarely do it by hand: what matters for an analysis is recognising confounding and selection bias, which is what the next section is about.

Using Graphical Causal Models to Diagnose Bias

We will now see how using graphical causal models allows to diagnose bias and understand which variables we should condition on and which we should not when trying to measure a causal effect through correlation. We will see that we have two main sources of bias: **confounding bias** and **selection bias**.

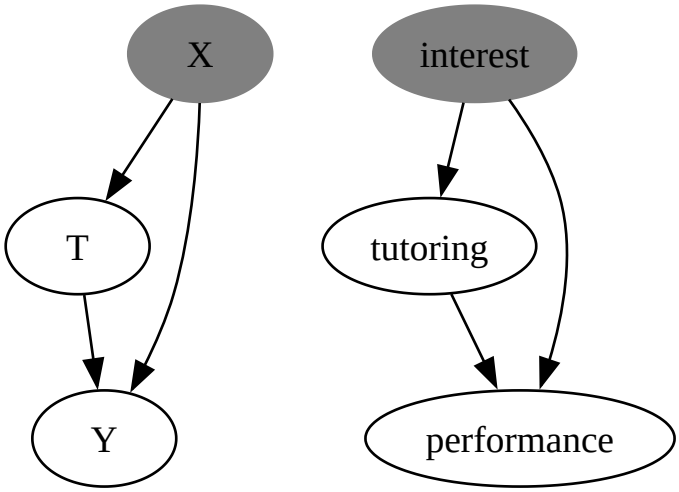
Confounding Bias

Confounding bias happens when **the treatment and the effect have a common cause**. Let us consider the following example, which we already saw previously:



Here, the outcome Y is `performance`, which is caused by both the treatment T (`tutoring`) and another common cause X (`interest`). The common cause X is called **a confounder** because it mixes ups the effects of `tutoring` on `performance` and `interest` on `performance`. Indeed, if we measure the correlation between `tutoring` and `performance`, we obtain a large number, but we do not know if this is due to the common cause `interest`. Maybe, people choose tutoring because they are interested and obtain good performance because they are interested as well.

If we want to be able to measure the **direct effect between the treatment and the outcome**, we need to **close the backdoor path between Y and T**. This is done by **conditioning on the common cause**:



Note that this corresponds to comparing subjects with the same level of interest. In this case, we will observe the direct effect of `tutoring` on `performance`, since all students in the considered group have the same level of `interest` (so any association between `tutoring` and `performance` is given by the causal effect). By conditioning on X, we obtain:

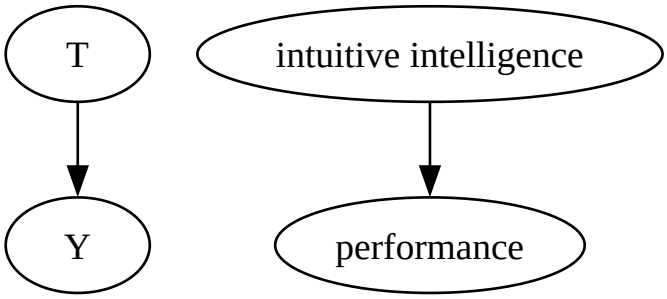
$$Y(0) \perp Y(1)|T, X$$

Selection Bias

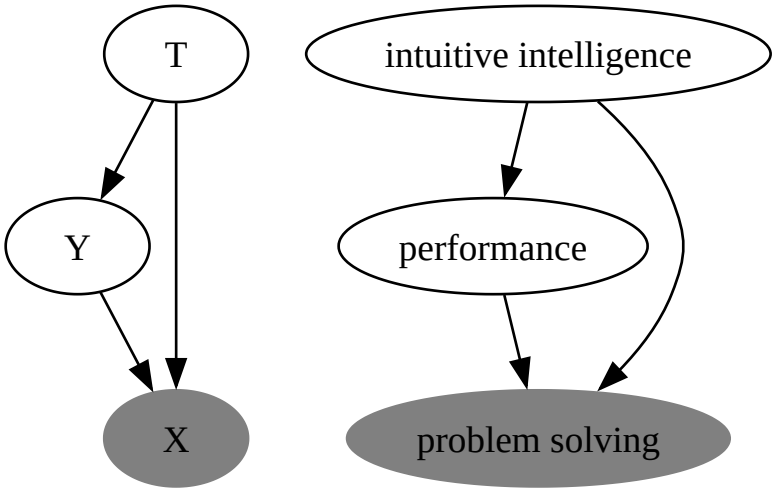
While confounding bias happens when we do not condition on a common cause, selection bias happens when **we control for too many variables**. While one may think that it is a good idea to control on any variable "just in case", this is not true in general and can lead to **selection bias**.

Controlling For a Common Effect

Let us consider the following example in which we want to measure the causal effect of `intuitive intelligence` on `performance`:

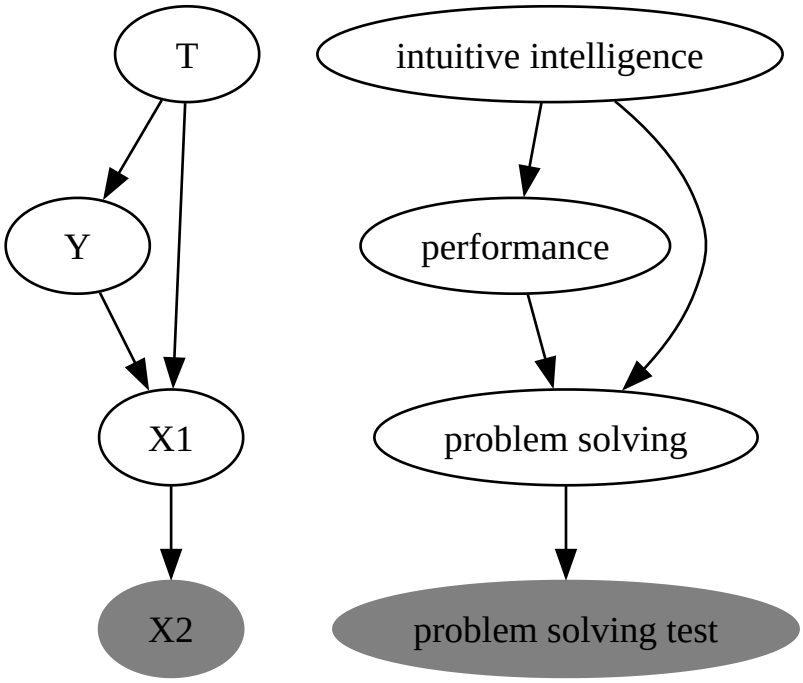


In our data, we also have a score of a test on problem solving. **"Just to be sure", we condition on** `problem solving`:



The problem is that `problem solving` is a common effect of `performance` and `intuitive intelligence`. By conditioning on the common effect, we are introducing a form of bias. Indeed, **if we look at people with good problem solving abilities, we are probably looking at people with either good performance or good intuitive intelligence**. Indeed, knowing that the person has good performance **explains away whether they have intuitive intelligence**, hence showing a correlation which does not represent a causal effect. **By conditioning on the common cause, we opened a backdoor path between Y and T which affects the perceived association between treatment and outcome**.

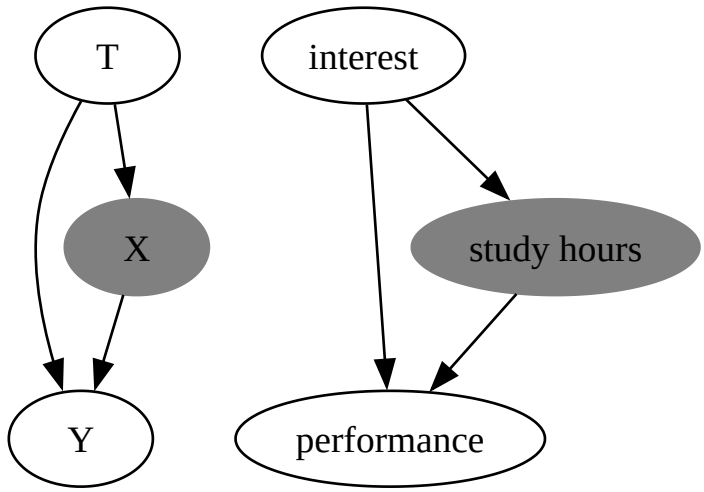
Note that the same is true if we condition on a descendant of the common cause. For instance:



Controlling on a Mediator

Another case of selection bias, arising from cases in which we control for variables we should not control on, is given by the case in which we control on a mediator of the treatment and the outcome.

Let us consider this example:



We want to assess the effect of `interest` on performance. Let's say we could randomize interest (for instance, we selected students with different levels of interest randomly across different schools). **"Just to be sure", we condition on** `study hours`. Conditioning on the mediator closes a path between `interest` and `performance`, hence reducing the measured effect between `interest` and `performance`.

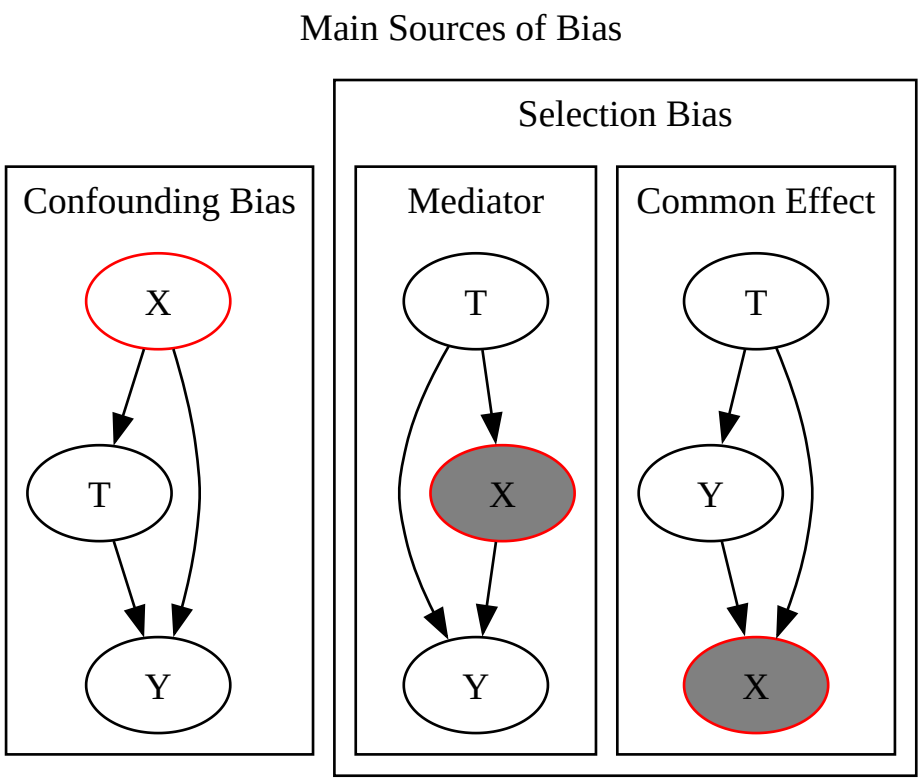
Indeed, while we expect that interested students will study more and obtain better performance, by observing only people who study a lot, `interest` **and** `performance` **may seem to be little correlated. Indeed, even people with little interest will obtain good performance if they study for long hours.**

Main Sources of Bias

To summarize, the main sources of bias are three:

- **Confounders**: we should control on the common cause;
- **Selection Bias - Common Effect**: we should not control on the common effect;
- **Selection Bias - Mediator**: we should not control on the mediator.

These are shown in the graph below:

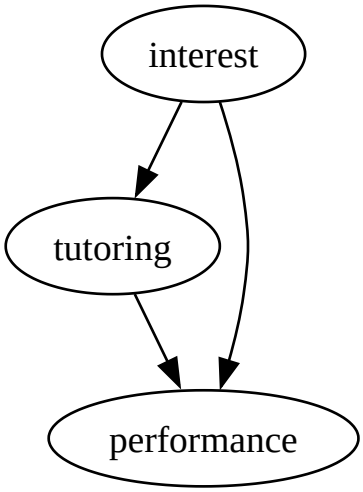


In the figure above, we mark in red the nodes that are not conditioned on, but should be conditioned on, and the nodes that are conditioned on, but should not be conditioned on.

Linear Regression for Causal Analysis

We have seen how to choose a suitable set of variables to control for when we aim to establish a cause-effect relationship. When we said "we control for X", we implied that we can divide the data in groups and measure association within each group, to then aggregate results and obtain an average effect.

In this section, we will see how we can automate this process with **linear regression**. Let us consider our previous example in which we wanted to measure the causal effect of `tutoring` on `performance`, when we know that `interest` can be a confounder:



This is the data:

	tutoring	performance	interest
0	0	76.432447	2.0
1	1	91.550461	5.0
2	1	85.107378	4.0
3	0	69.981030	3.0
4	1	70.111151	2.0
...	...	...	...
195	0	68.477160	2.0
196	0	87.783594	4.0
197	1	88.220111	5.0
198	1	93.399423	5.0
199	1	84.082421	4.0

200 rows × 3 columns

Recall that in this case the assignment is not random, so we expect `interest` to influence both `tutoring` and `performance`. If we ignore this bias, we can compute the average difference in the effect of the treated and untreated:

$$E[performance|tutoring = 1] - E[performance|tutoring = 0]$$

We can do this in computational terms as follows:

```
students[tutoring==1]['performance'].mean()-students[tutoring==0]
        ['performance'].mean()
```

```
np.float64(-2.043673682064565)
```

Another relevant value to compute is the average performance of the untreated:

```
students[students['tutoring']==0]['performance'].mean()
```

```
np.float64(72.92912990981884)
```

We can see how these values are **automatically computed within a linear regressor** of the kind:

$$performance = \beta_0 + \beta_1 tutoring$$

These are the coefficients after fitting:

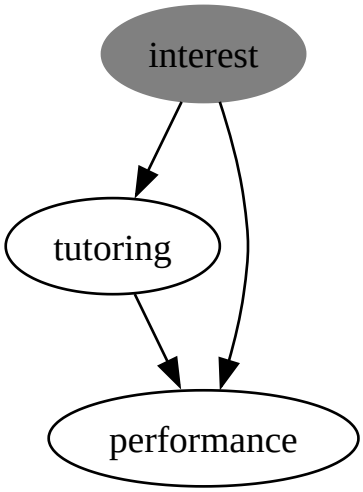
	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	72.9291	1.199	60.832	0.000	70.565	75.293
tutoring	6.8269	1.671	4.087	0.000	3.533	10.121

Coherently with the interpretation we previously gave of linear regression, we obtain estimates for those two values:

- `intercept`: is the average performance of the untreated, i.e., the value of `performance` we expect to see in average when `tutoring=0`;
- `tutoring`: is the increment of `performance` we expect to see in average when we observe an increment of one unit in `tutoring` (i.e., when passing from `tutoring=0` to `tutoring=1`). This is the average difference of `performance` when comparing the treated `tutoring=1` and the untreated `tutoring=0`.

Also, when we use a linear regressor, we get confidence intervals and the results of statistical tests "for free".

If we look at the result above, **there is non-zero correlation between `performance` and `tutoring`**. Indeed, the coefficient of `tutoring` allows us to observe that **students who enroll in the `tutoring` programme, increase their average `performance` by 6.8269 points** (remember that performance lies in the $[0, 100]$ range). However, we also know that this result is biased due to the confounder `interest`. To remove this bias, we should condition on `interest`:



We note that one way to **control for a variable is to include it among the independent variables in the linear regressor**, i.e., fitting a model like this:

$$performance = \beta_0 + \beta_1 tutoring + \beta_2 interest$$

The result is as follows:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	54.1697	1.296	41.804	0.000	51.614	56.725
tutoring	-0.9516	1.129	-0.843	0.400	-3.178	1.275
interest	7.8097	0.441	17.711	0.000	6.940	8.679

We note that, after adding this variable, the values of the coefficients changed substantially. **We are in particular interested in the coefficient of `tutoring`, which is measuring the correlation between `tutoring` and `performance`**. We note that we can now interpret this value as follows:

> β_1 (coefficient of `tutoring`) is the increment we expect to observe in `performance` when `tutoring` passes from 0 to 1 **and** `interest` **is held constant**.

This last bit is the fundamental one: **by including the `interest` variable, we are effectively controlling for `interest`**. Indeed, the coefficient of `tutoring` now virtually looks at a subset of the data in which `interest` is constant. This is in practice the average effect of `tutoring` when comparing groups of data with constant `interest`.

Hence, **even if we cannot physically perform a random assignment, linear regression allows us to control for variables, hence enabling the analysis of observational data**. This is fundamental, as it is not in always feasible to perform random tests.

Getting back to our result above, we note that the coefficient of `tutoring` has a p-value of 0.4. This means that, while a value different from zero has been computed for it, the hypothesis test is telling us

that **we do not have enough elements to say that this value actually is different from zero**. Hence, we can conclude that `tutoring` does not have a statistically significant impact on `performance`.

Note that, **if we compute a linear regressor on the randomized set (i.e., the one in which we pretended we could perform random assignment)**

$$performance = \beta_0 + \beta_1 tutoring$$

we obtain these results:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	72.9291	1.199	60.832	0.000	70.565	75.293
tutoring	6.8269	1.671	4.087	0.000	3.533	10.121

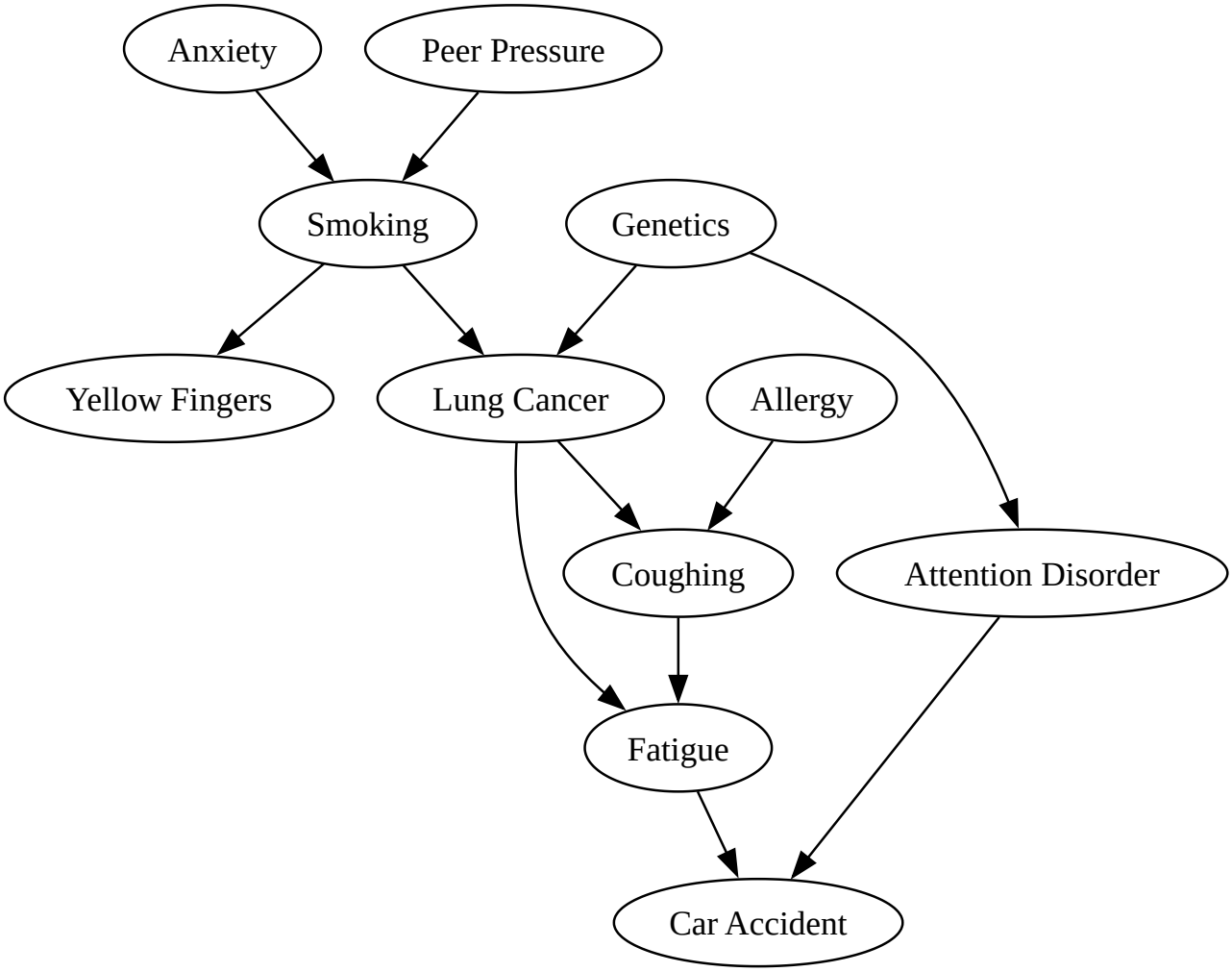
Coherently with the conclusions made with observational data, there is not a significant relationship between `tutoring` and `performance`.

Example on the LUCAS Dataset

We will see some examples on the LUCAS dataset, which can be found here:

https://www.causality.inf.ethz.ch/data/LUCAS.html

LUCAS is an **artificial dataset** in which data has been generated drawing values from known distribution to emulate a causal phenomenon. Besides the data, the authors propose the **following true causal graph** (which reflects the way data has been generated):



Let us have a look at the data:

	Smoking	Yellow_Fingers	Anxiety	Peer_Pressure	Genetics	Attention_Disorder	Born_an_Even_Day	Car_
0	0	0	1	0	0	1	0	1
1	0	1	0	0	0	0	1	0
2	1	1	1	0	1	1	1	1
3	0	0	0	1	0	0	1	0
4	1	1	1	0	0	1	1	1
...	...	...	...	...	...	...	...	...
1995	0	0	1	0	0	0	0	1
1996	1	1	1	0	0	0	0	0
1997	1	1	0	1	0	0	1	0
1998	1	1	1	0	1	1	0	1
1999	1	1	0	1	0	0	0	0

2000 rows × 12 columns

Does `Smoking` causes `Car Accident`?

We will try to answer the question as to whether there is a causal effect between `Smoking` and `Car Accident`. We can start by fitting a logistic regressor:

$$\log \frac{P(Car\,Accident|Smoking)}{P(\neg Car\,Accident|Smoking)} = \sigma(\beta_0 + \beta_1 Smoking)$$

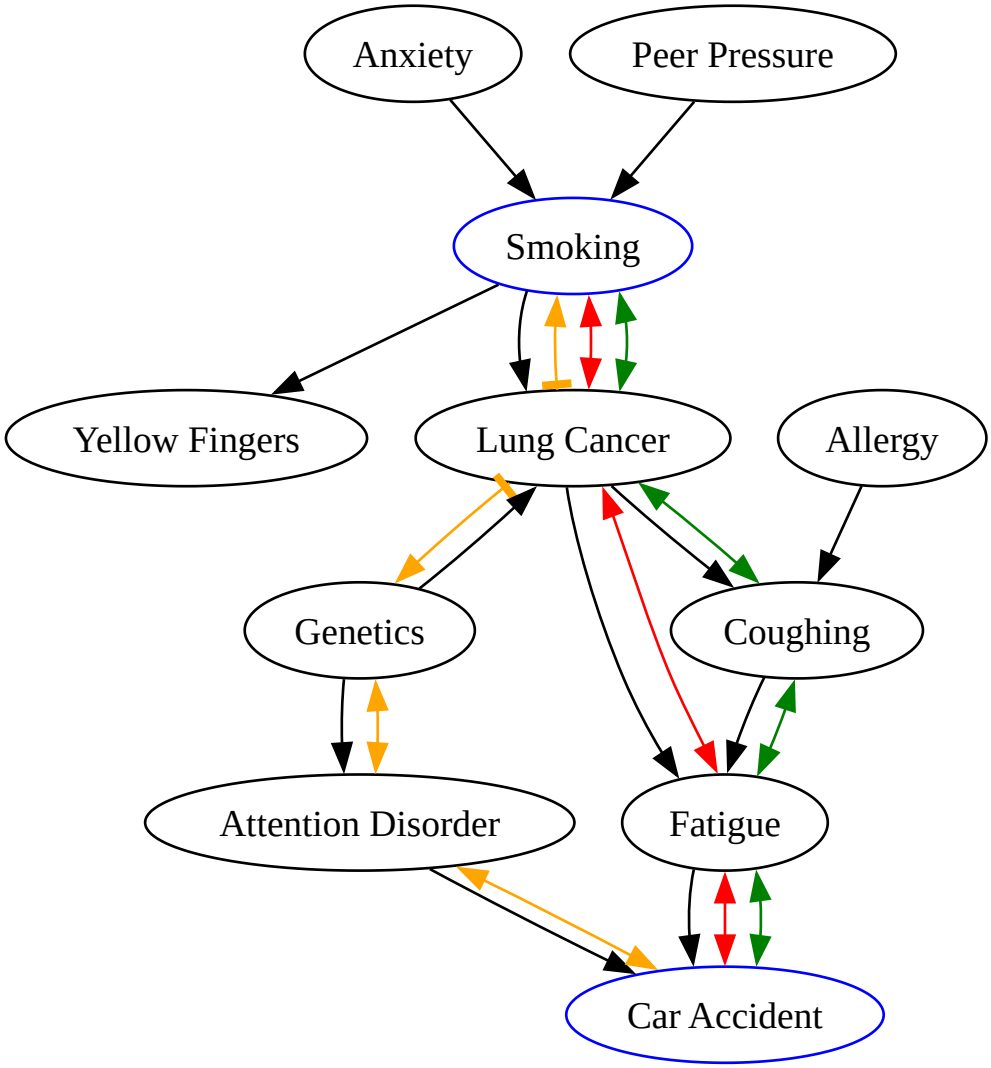
We need to fit a logistic regressor because the dependent variable is binary (all variables are binary). This is the result:

```
Optimization terminated successfully.
         Current function value: 0.588796
         Iterations 5
```

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	0.7668	0.097	7.939	0.000	0.578	0.956
Smoking	0.2596	0.113	2.299	0.022	0.038	0.481

From the observation above, it seems that `Smoking` is positively associated with `Car Accident`. Since $e^{0.2596} \approx 1.3$, we can say that smoking is associated by an increase of $+30$ in the odds of having a car accident. But, can we say that this is a causal effect? I.e., can we say that **smoking causes an increment in the odd of car accident**?

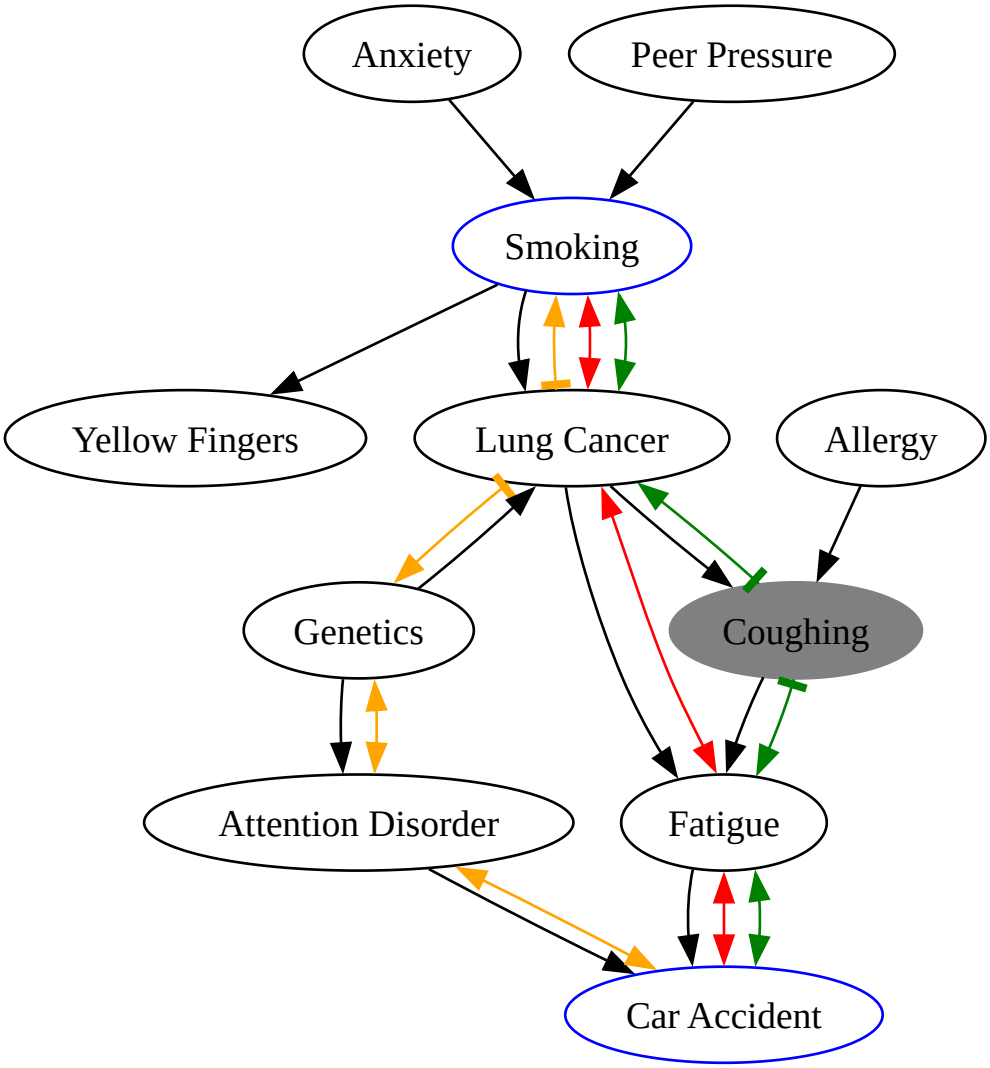
We know that there might be some bias. To check for that, let us consider the causal graph again, and let us highlight all dependence paths from the `Smoking` node to the `Car Accident` node. These represent all the different ways in which the two variables can be correlated.



We have three possible paths, highlighted in yellow, red, and green. Arrows indicate whether dependency flows or not. We can see that:

- The yellow path is blocked because there is a collider (`Lung Cancer`) that has not been conditioned on. The collider prevents `Smoking` and `Genetics` from being associated. This blocked path is ok for us. We are trying to see if smoking causes car accident, and, since, `Lung Cancer` does not cause `Genetics`, that path would not be a viable one to check for a causal effect;
- The red and green paths are not blocked. This is not ideal, because now there are two ways `Smoking` could influence `Car Accident`, i.e., directly through `Fatigue` or via `Coughing`. This may lead to a combined effect which is stronger than it is supposed to be.

If we want to estimate a causal effect, we need to block one of the two paths. The most sensible thing to do is to **condition on** `Coughing`, so that now smoking can affect Car Accident only through Lung Cancer and then Fatigue. This is illustrated as follows:



To condition on `Coughing`, we can fit the following logistic regressor:

$$\log \frac{P(Car\,Accident|Smoking, Coughing)}{P(\neg Car\,Accident|Smoking, Coughing)} = \sigma(\beta_0 + \beta_1 Smoking + \beta_2 Coughing)$$

We obtain the following coefficients:

```
Optimization terminated successfully.
         Current function value: 0.569895
         Iterations 5
```

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	0.3411	0.108	3.153	0.002	0.129	0.553
Smoking	-0.0130	0.120	-0.108	0.914	-0.248	0.222
Coughing	0.9529	0.109	8.717	0.000	0.739	1.167

We now get a different picture: the p-value of `Smoking` is very large. This means that, once we removed the influence of `Coughing`, the effect of `Smoking` on `Car Accident` is negligible (its

coefficient is not significantly different from zero). This suggests that there is no significant causal effect between the two variables.

Exercises

1. **Draw the graph.** A study finds that students who use the library more get better marks. Draw a causal graph including at least two plausible confounders. Which arrow is the causal effect of interest, and which paths are backdoor paths?
2. **d-separation by hand.** In the graph $A \to C \leftarrow B$, $C \to D$: are A and B independent? Are they independent given C? Given D? Justify each answer with the rules for chains, forks and colliders.
3. **Adjust, and over-adjust.** On the LUCAS data, estimate the effect of smoking on lung cancer (a) unadjusted, (b) adjusting for the confounders you identify from the graph,
   c. adjusting additionally for a *descendant* of the outcome. Explain why (c) is worse than (a).
4. **Selection bias in the wild.** Among hospital patients, smoking appears protective against a certain disease. Draw the graph that produces this and name the variable being conditioned on.

References

[1] Most of the content of this lecture is based on part of the awesome lecture series "Causal Inference for The Brave and True" by Matheus Facure Alves.

[2] Judea Pearl and Dana Mackenzie. "The Book of Why: The New Science of Cause and Effect". Penguin.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Storytelling with data

Chapter resources

Practice quiz

ⓘ Materials for this lecture

The slides for this lecture reproduce figures from Cole Nussbaumer Knaflic's *Storytelling with Data*, which cannot be redistributed, so they are **shared directly by the teacher** rather than published here. The book itself is available in the department library and is the reference for the whole lecture.

An analysis that nobody understands has not been completed. This lecture is about the last step: turning a set of findings into something an audience can follow, act on, and challenge.

It sits deliberately at the end of Block 1, right after the causal chapter. By this point you can establish what the data supports; the risk from here on is that a confident chart says more than the analysis does. Most misleading data graphics are not produced by liars — they are produced by people who wanted to make a point clearly.

What the lecture covers

The material follows the structure of Knaflic's book:

1. **Understand the context.** Who is the audience, what do they already know, what decision are they going to make, and what exactly do you need them to do? A chart for yourself and a chart for a stakeholder are different objects.
2. **Choose an effective visual.** Match the display to the question — the comparison you want the reader to make should be the easiest one to make on the page. Chapter 2 of these notes covers the mechanics; this is about the choice.
3. **Eliminate clutter.** Every element that is not carrying information is competing with the ones that are: gridlines, borders, legends that could be direct labels, decoration, the third dimension of a bar chart.
4. **Focus attention.** Colour, size and position are the tools that say *look here*. They work only if they are used sparingly: if everything is highlighted, nothing is.
5. **Think like a designer.** Affordance, accessibility and aesthetics — make the chart look like what it does, make it readable to a colour-blind reader and on a projector, and accept that a tidy chart is trusted more than an untidy one.
6. **Tell a story.** Narrative structure — situation, complication, resolution — applied to an analysis, so the audience remembers the point rather than the plot.

The part that belongs to this course

Two obligations run alongside the craft, and they are what we will assess.

A chart must not claim more than the analysis supports. The most common failure in project presentations is a slide showing a clean association with a title written in causal language. If chapter 11 could not close the backdoor paths, the title cannot either. This is why the project's slide deck must include a "What could confound this?" slide: it forces the question onto the same page as the claim.

Uncertainty is part of the result, not a caveat to be dropped for clarity. A point estimate shown without its interval is a cleaner chart and a worse one. Error bars, a confidence band, or a sentence stating the interval — one of the three has to be there.

For the project

Your presentation is the oral part of the exam. It is assessed on whether the audience can follow the argument, whether the charts are readable, whether the limitations are stated, and whether each member can answer questions about the work. Fifteen slides at most. See the project page.

References

- Knaflic, Cole Nussbaumer. *Storytelling with Data: A Data Visualization Guide for Business Professionals*. Wiley, 2015. **Available in the department library.**
- Tufte, Edward. *The Visual Display of Quantitative Information*. 2nd ed., 2001.
- Cairo, Alberto. *How Charts Lie*. Norton, 2019.
- Wilke, Claus O. *Fundamentals of Data Visualization*. O'Reilly, 2019. Free online: https://clauswilke.com/dataviz/

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Introduction to predictive analysis

Chapter resources

Slides (PDF) Open in Colab Practice quiz

The algorithms and techniques seen so far have been focusing on **understanding the data** in different ways. We have seen methods to **summarize the data** (like mean, median, and variance) and to **study the relationship between variables** (like covariance and correlation), and check whether **our conclusions could be generalized to the population** (statistical inference).

However, so far, we have been focusing on **describing the data and its relationships** in a somewhat **passive way**: we **observed** variables or pairs of variables, **computed scores** and **performed tests**.

We will now look at data analysis from a new perspective: that of **predictive modeling**. This is a broad field of data analysis, which is very overlapped with **machine learning**, with two main goals:

- **understanding relationships between variables** (inference)
- **building models which can be used to make predictions on unseen data** (prediction)

In both cases, the first step is to create a **predictive model of the data**. In this lesson, we will formally define what we mean by a "model" and analyze the fundamental principles that guide the construction of effective predictive systems.

Before to begin, let's see a practical example. We will consider the `Diabetes` dataset. Let's load and visualize it:

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset served by this site. It is stored compressed; pandas works that
# out from the file name.
url = DATA + "cdc_diabetes_health_indicators.csv.gz"
df = pd.read_csv(url)

# Display the first few rows
df.head()
```

	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack	PhysActivity	Fruits	Veggies	...	I
0	1	1	1	40	1	0	0	0	0	1	...	(
1	0	0	0	25	1	0	0	1	0	0	...	1
2	1	1	1	28	0	0	0	0	1	0	...	1
3	1	0	1	27	0	0	0	1	1	1	...	(
4	1	1	1	24	0	0	0	1	1	1	...	(

5 rows × 22 columns

The dataset comes from the **Behavioral Risk Factor Surveillance System (BRFSS)**, the annual health survey the US Centers for Disease Control runs by telephone. These are the 2015 responses: 253 680 people, each answering about their health, their habits and their diagnoses. Two things follow from that and matter for everything below — the answers are **self-reported**, so they carry whatever people misremember or would rather not say, and the respondents are the people a telephone survey reaches, which is not the same as the population.

The columns we will use:

- **BMI**: body mass index (weight in kg divided by height in m squared)
- **GenHlth**: self-rated general health, from 1 (excellent) to 5 (poor)
- **PhysHlth**, **MentHlth**: days of poor physical or mental health in the last 30
- **HighBP**, **HighChol**, **Smoker**, **Stroke**, **HeartDiseaseorAttack**: yes/no
- **PhysActivity**, **Fruits**, **Veggies**, **HvyAlcoholConsump**: yes/no habits
- **Age**, **Education**, **Income**: banded, not raw values — `Age` is a 1–13 bracket
- **Diabetes_binary**: the outcome, 1 if the respondent has been told they have diabetes

The outcome is positive in **13.9%** of rows. That imbalance is not a nuisance to be fixed: it is the single most useful thing about this dataset for what comes next, because a model that answers “no” every time is right 86% of the time and has learned nothing.

What is a Model?

Before we go further, let’s define what we mean by a “model”.

In the context of data analysis, a **model is a simplified representation of reality**. It is not reality itself, but a formal abstraction of a real-world process, designed to serve a specific purpose.

A classic analogy is a **geographical map**:

- A map is a **simplification**: it is flat, while the Earth is round. It omits countless details—trees, rocks, buildings.
- In that sense, a map is technically “wrong”.

This leads us to a famous aphorism from statistician George Box:

> **“All models are wrong, but some are useful.”**

This is one of the most important ideas in modeling. A **data model**—whether statistical or machine learning—is like a map:

- It is a **simplification** of a complex real-world process (e.g., the relationship between blood test results and diabetes).
- It is “wrong” because it cannot capture every nuance of that process.
- But it becomes **useful** if it helps us achieve our goal—whether that goal is to **understand** the process or to **predict** its outcomes.

The “right” model is not the most complex one, but the simplest one that is most useful for our goal.

Example

Let’s consider a concrete example using the **CDC BRFSS survey**.

Suppose we want to understand the distribution of **BMI (Body Mass Index)** among patients. We can summarize this data by fitting a **Gaussian (Normal) distribution** to the BMI values.

This Gaussian distribution *is a model*. It’s a mathematical function defined by two parameters: the mean (μ) and the standard deviation (σ).

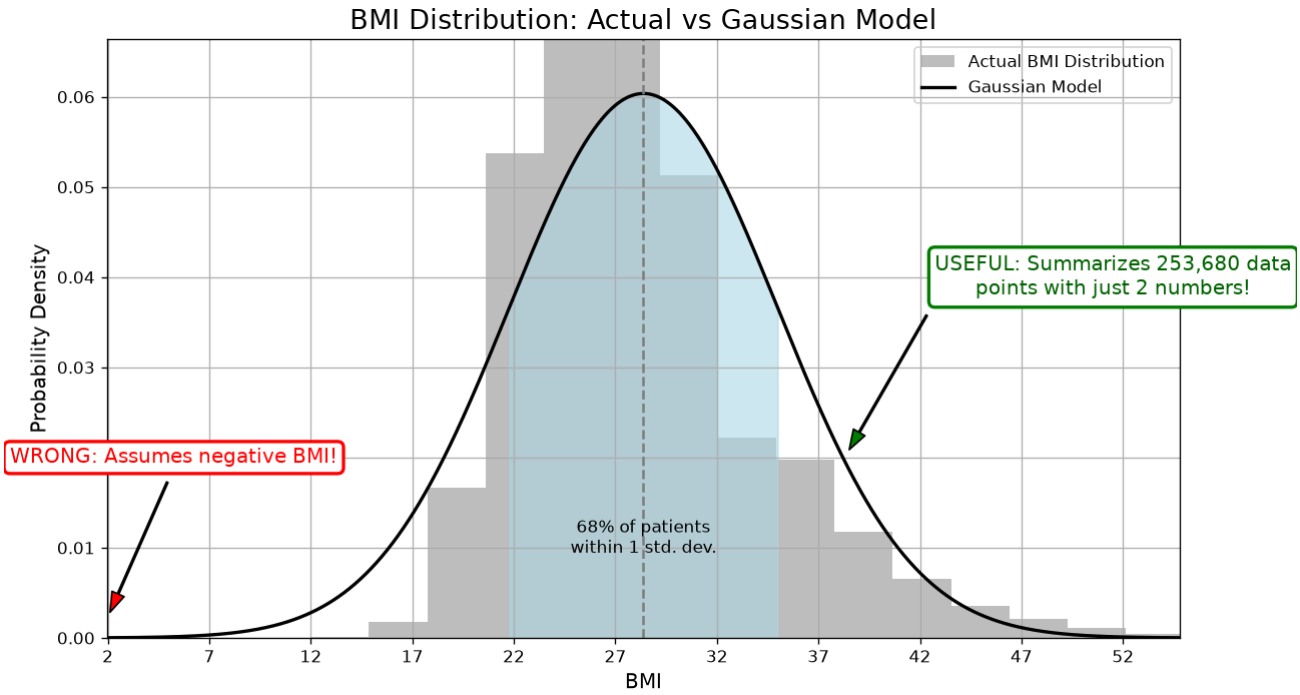
Why is this model “wrong”?

1. The actual data consists of a finite set of BMI measurements from real patients. Our model is a smooth, continuous curve that approximates this distribution.
2. The Gaussian model extends from negative infinity to positive infinity. This implies a non-zero probability of a patient having a negative BMI or an extremely high BMI, which are physiologically implausible.
3. The real-world data will never perfectly match the idealized bell curve. There may be skewness, outliers, or multiple peaks that the Gaussian model cannot capture.

Why is this model “useful”?

1. **It’s a powerful proxy for the data.** Instead of analyzing hundreds of individual BMI values, we can summarize the entire distribution with just two numbers: μ and σ.
2. **It helps us understand the data.** We gain insight into the “typical” BMI and how much variation exists across the population.
3. **It allows us to make reasonable inferences.** For example, we can estimate that approximately 95% of patients have a BMI between $\mu - 2\sigma$ and $\mu + 2\sigma$, which helps in identifying outliers or high-risk individuals.

This simple model, while imperfect, is incredibly useful for understanding the structure of our data.



We can also verify this claim numerically:

```
Mean BMI: 28.38
Standard Deviation: 6.61
Patients within 1 std dev: 196003 out of 253680
Percentage within 1 std dev: 77.26%
```

Predictive Models

Beyond the Gaussian example above, when we perform predictive analysis, we assume in general to observe:

- A variable Y, which we usually call the **dependent variable** or **response variable** - this is the variable we want to predict;
- A set of variables $X = (X_1, X_2, \ldots, X_n)$, which we usually call our **predictors** - this are variables we want to use to predict Y.

We can express a predictive model with this general form:

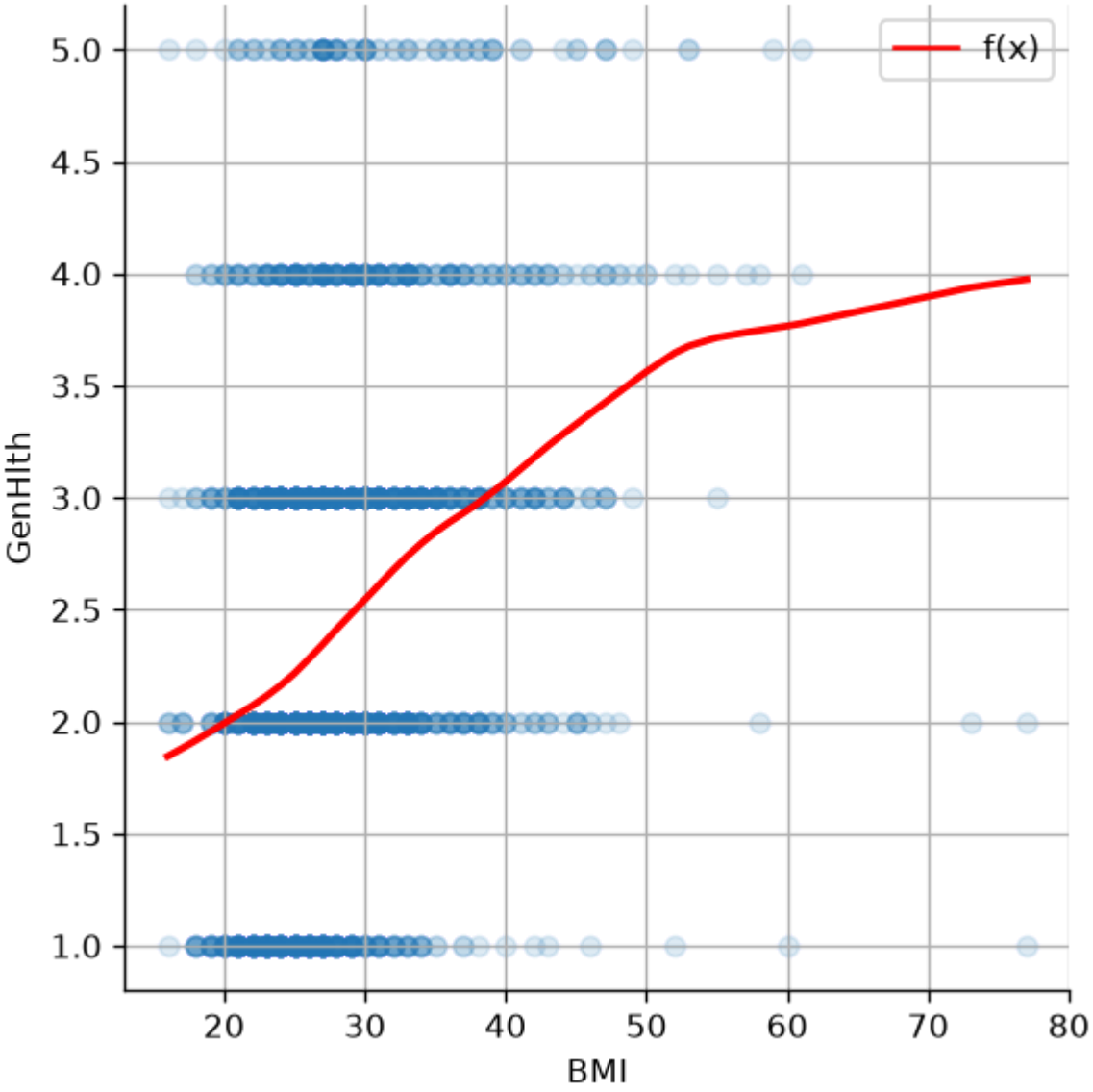
$$Y = f(X) + \epsilon$$

where:

- f is our predictive model - we want to **find this function** (more on this later);
- ϵ is the **error term** - since it's very hard to have perfect predictions (e.g., due to randomness in the data), we introduce this term to capture the errors that the model will make.

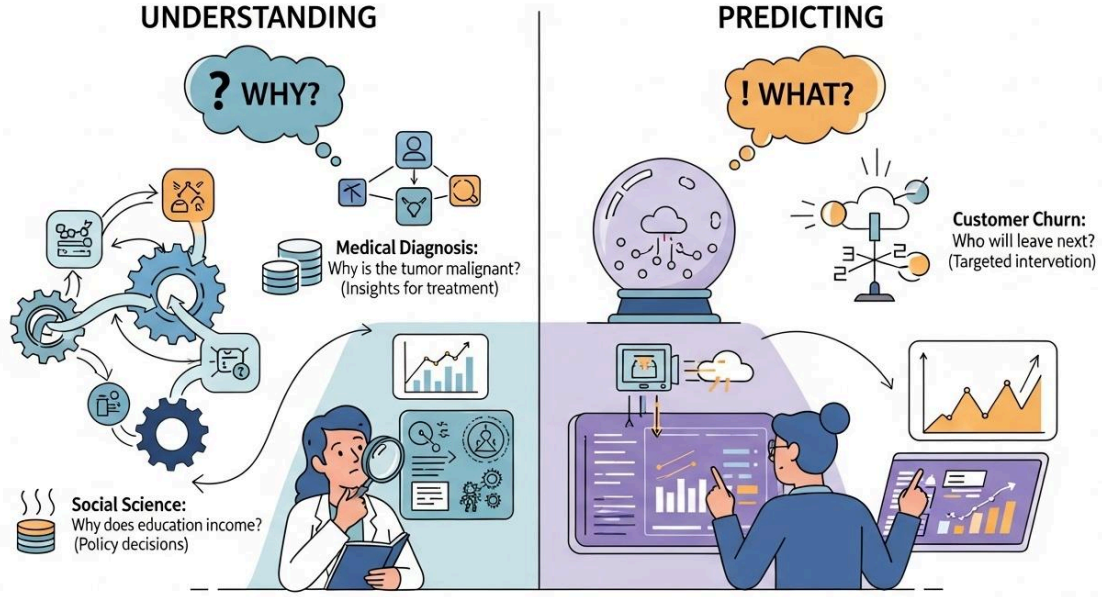
Example

In our example, we may be interested in predicting how a person rates their own health (`GenHlth`, 1 for excellent to 5 for poor) from their body mass index. If we look at the data, we may imagine an f function like the one drawn in red below. The survey has 253 680 rows, far more than a scatter plot can show, so this draws a random sample of 2 000 of them — the trend line is what carries the point, not the individual dots:



The Main Goal: Understanding vs. Predicting

Now that we have a definition of a model, we must address the primary reason for building one. The "usefulness" of a model, as George Box famously said, depends entirely on our goal. In data analysis, we typically pursue two main—sometimes competing—objectives: **Understanding** and **Predicting**.



Understanding (Inference)

The first objective is **Explanation**, or **Statistical Inference**. The central question is **"Why?"**

When our goal is understanding, we use models to open up the "black box" of a real-world process. We aim to quantify relationships between variables and interpret their effects.

- **Primary Goal:** To interpret the model's parameters (e.g., coefficients, p-values) to gain insights.
- **Key Question:** "What is the relationship between variable `X` and variable `Y`?" or "How much does `Y` change if I change `X`?"

Example (from the CDC survey):

Suppose we want to understand how `BMI` and `HighBP` influence diabetes. We may build a model which predicts the values of `Diabetes_binary` from `BMI` and `HighBP`. Interpreting the model, we may find that:

- A one-unit increase in `BMI` is associated with a 6% increase in the odds of having diabetes.
- `BMI` and `HighBP` can explain the values of `Diabetes_binary` only in a limited way (e.g., they explain only a limited amount of variance).

Here, our goal is not to predict whether a specific patient has diabetes, but to **interpret** the relationships and assess which factors are most influential.

Predicting (Prediction)

The second objective is **Prediction**. The central question is **"What will happen next?"**

When our goal is prediction, we care less about *why* the model works and more about *how well* it performs. The model's value lies in its ability to accurately forecast outcomes for new or unseen data.

- **Primary Goal:** To generate the most accurate predictions possible on new data.
- **Key Question:** "Given a new observation `X`, what is the most likely value of `Y`?"

Example (from the CDC survey):

Imagine we're building a screening tool. A new respondent arrives with known values for `BMI`, `HighBP`, `Age`, and other features. Our model predicts a 78% probability that they have diabetes.

In this case, we're not focused on interpreting coefficients—we're focused on **accuracy** and **reliability** of the prediction to inform medical decisions.

The Best of Both Worlds: Where Goals Overlap

These two goals are not mutually exclusive. The most powerful models often support both inference and prediction.

Example: Diabetes Risk Management

A healthcare provider may ask:

1. **Prediction:** Which patients are most likely to develop diabetes in the next year? (We need a high-performing model to flag at-risk individuals.)
2. **Understanding:** What are the key drivers of diabetes risk in our population? (We need an interpretable model to guide public health interventions—e.g., targeting obesity or glucose control.)

Example: Feature Importance in Diagnosis

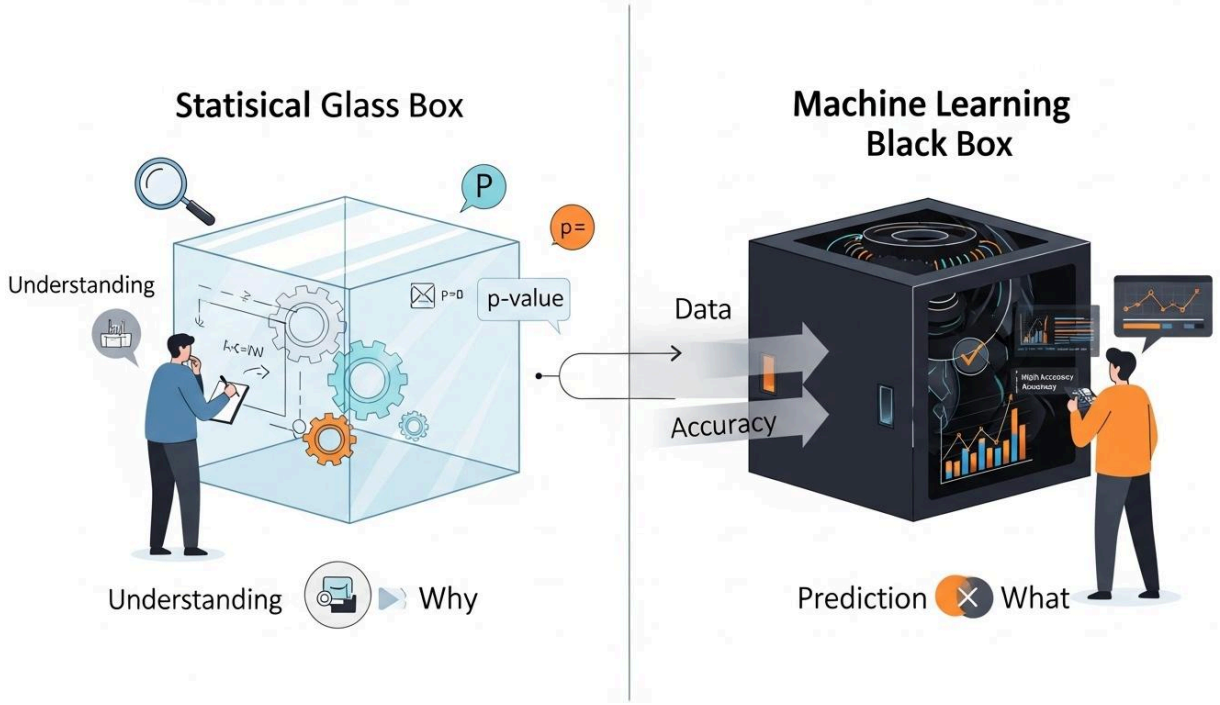
A machine learning model might predict diabetes with high accuracy, but clinicians also want to know:

- Which features (e.g., `BMI`, `HighBP`, `Age`) are most influential?
- Can we explain the model's decisions to patients and other stakeholders?

As we'll see, some techniques are designed for **Understanding** (e.g., `linear regression`, `logistic regression`), while others excel at **Prediction** (e.g., `naive Bayes`, `K-Nearest Neighbor`). Many models, like `regularized regression`, can serve both purposes if used thoughtfully.

Ultimately, our job as data scientists is to be clear about which goal we're pursuing and to choose tools that best support that goal.

The Two Approaches: Statistics vs. Machine Learning



The two goals we just discussed—Understanding and Predicting—have led to the development of two major approaches, or "cultures," in data modeling. While they share many of the same mathematical tools (like regression), their end-goals and methods for success are very different.

The Statistical Approach (The "Glass Box")

This approach is primarily focused on **Understanding** and **Inference**. It treats the model as a "glass box," where our primary goal is to look inside and understand its mechanisms.

- **Primary Goal:** To **understand** the relationship between variables and draw conclusions about the real-world process.
- **Key Question:** "Is the relationship I see in my sample data *real* (statistically significant), or is it just due to random chance?"
- **Methodology:**
  - Relies heavily on **assumptions** about the data (e.g., variables are normally distributed, observations are independent).
  - Focuses on **hypothesis testing**, **p-values**, and **confidence intervals**.
  - The entire dataset is typically used to fit the model and test its significance.
  - **Interpretability is a requirement.** A model that cannot be explained is not considered useful.

The Machine Learning Approach (The "Black Box")

This approach is primarily focused on **Prediction** and **Accuracy**. It often treats the model as a "black box": if it produces the right answer, we don't always need to know *how* it got there.

- **Primary Goal:** To build a model that makes the most **accurate** predictions possible on new, unseen data.

- **Key Question:** "When I feed new data into my model, how often does it give the correct answer?"
- **Methodology:**
  - Is **results-driven** and makes fewer assumptions about the underlying data.
  - Relies on **splitting the data** into training, validation, and test sets to simulate a "future" scenario.
  - The model's success is not measured by p-values, but by its **performance on the test set** (e.g., accuracy, error rate).
  - **Interpretability is a bonus, not a requirement.** A very complex but highly accurate model is often preferred over a simple but less accurate one.

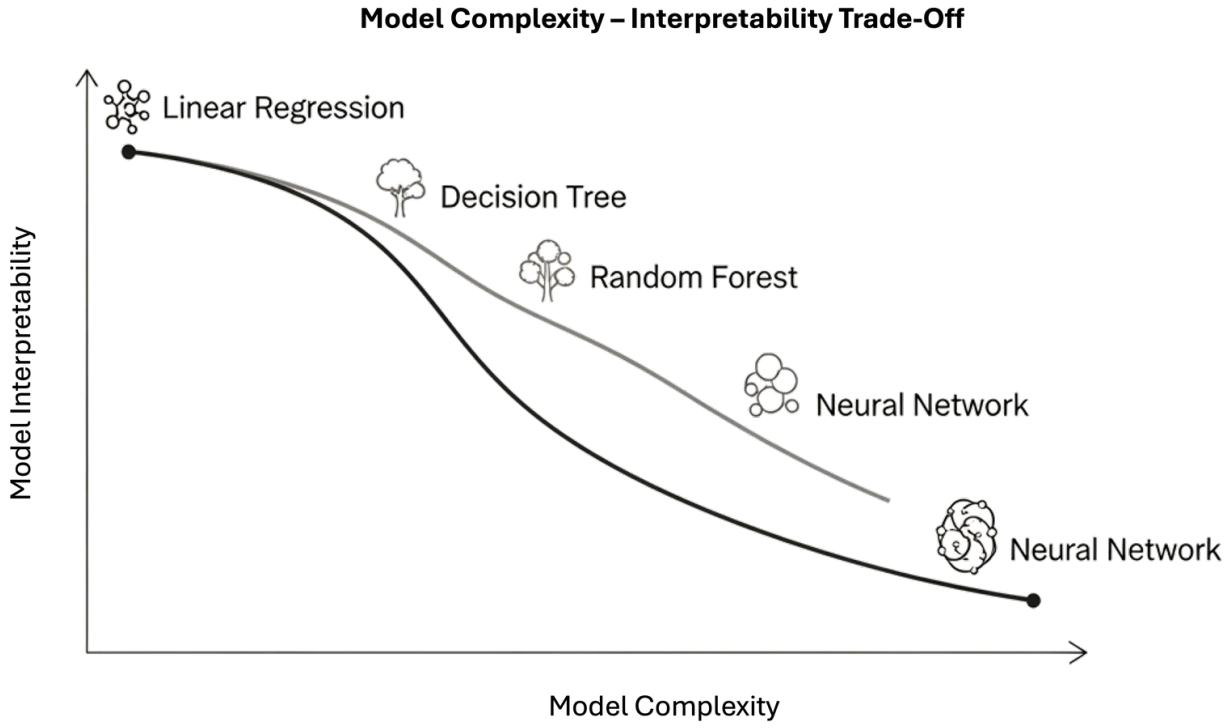
Example: the CDC BRFSS survey

This survey is a good example of where *both* approaches are valuable.

- **A Statistical (Glass Box) Goal:** A medical researcher wants to **understand the risk factors** for diabetes in this population. They would build an interpretable model (like Logistic Regression) using the *entire dataset*. Their goal is not to predict for a new patient, but to publish a paper stating, "We found that BMI and high blood pressure are the most significant risk factors, with p-values < 0.01." They care about the **coefficients** and **p-values**.

- **An ML (Black Box) Goal:** A health tech company wants to build a mobile app that **predicts a user's diabetes risk**. They don't care about p-values; they just want the highest **accuracy**. They will use a *train/test split* and try many complex models. If a "black box" neural network gets 80% accuracy on the test set, while the simple logistic regression only gets 75%, they will use the neural network for their app.

The Model Complexity - Interpretability Trade-off



The contrast between these two approaches highlights one of the most important practical challenges in data modeling: the **Model Complexity - Interpretability Trade-off**.

Generally, as the **complexity** of a model (its capacity, or the "number of knobs" it can turn) **increases**, its **interpretability** (our ability to understand *why* it makes a certain decision) **decreases**.

- **Simple Models (Low Complexity):**
  - **Examples:** Simple Linear Regression, **Logistic Regression**.
  - These models are on the **low-complexity, high-interpretability** end of the spectrum. When applied to this survey, a Logistic Regression model gives you a clean formula. We can easily look at the coefficients to say, "A one-unit increase in BMI increases the log-odds of diabetes by X."
  - They are "glass boxes," perfect for the **Statistical (Understanding)** goal. The risk is that they may be "too simple" (high bias) and fail to capture the complex, non-linear interactions between variables like 'GenHlth' and 'BMI', leading to lower predictive accuracy.
- **Complex Models (High Complexity):**
  - **Examples:** Naive Bayes, K-Nearest Neighbor, other advanced models (e.g., Neural Networks, Decision Trees, etc.)
  - These models are on the **high-complexity, low-interpretability** end. When applied to this survey, a Deep Neural Network might learn from millions of parameters interacting in ways that are not human-readable.
  - Their high capacity makes them excellent for the **Machine Learning (Prediction)** goal, as they can capture those subtle patterns (low bias) and may achieve higher accuracy. The risk is that they are "black boxes"; they can't easily explain to a patient *why* they were flagged as high-risk.

Choosing a model, therefore, is not just about finding the most accurate one. It's about **making a conscious decision about this trade-off** based on your primary goal.

The Problem Landscape

Now that we have established the *goals* of modeling (Understanding vs. Predicting), let's map out the *types* of problems we can solve. The most fundamental distinction in machine learning is between **Supervised** and **Unsupervised** learning.

Supervised Learning: Learning with an Answer Key

In **supervised learning**, our dataset contains both the **input features (X)** and the **correct answer (y)**.

We are "supervising" the model by showing it thousands of examples and their corresponding outcomes. The goal is to learn a mapping function, h, such that

$$\hat{y} = h(X)$$

is a good predictor of y.

We often use $\hat{y}$ since we are predicting a value for y and to distinguish it from the correct value y. We will also call the correct value y **ground truth** value.

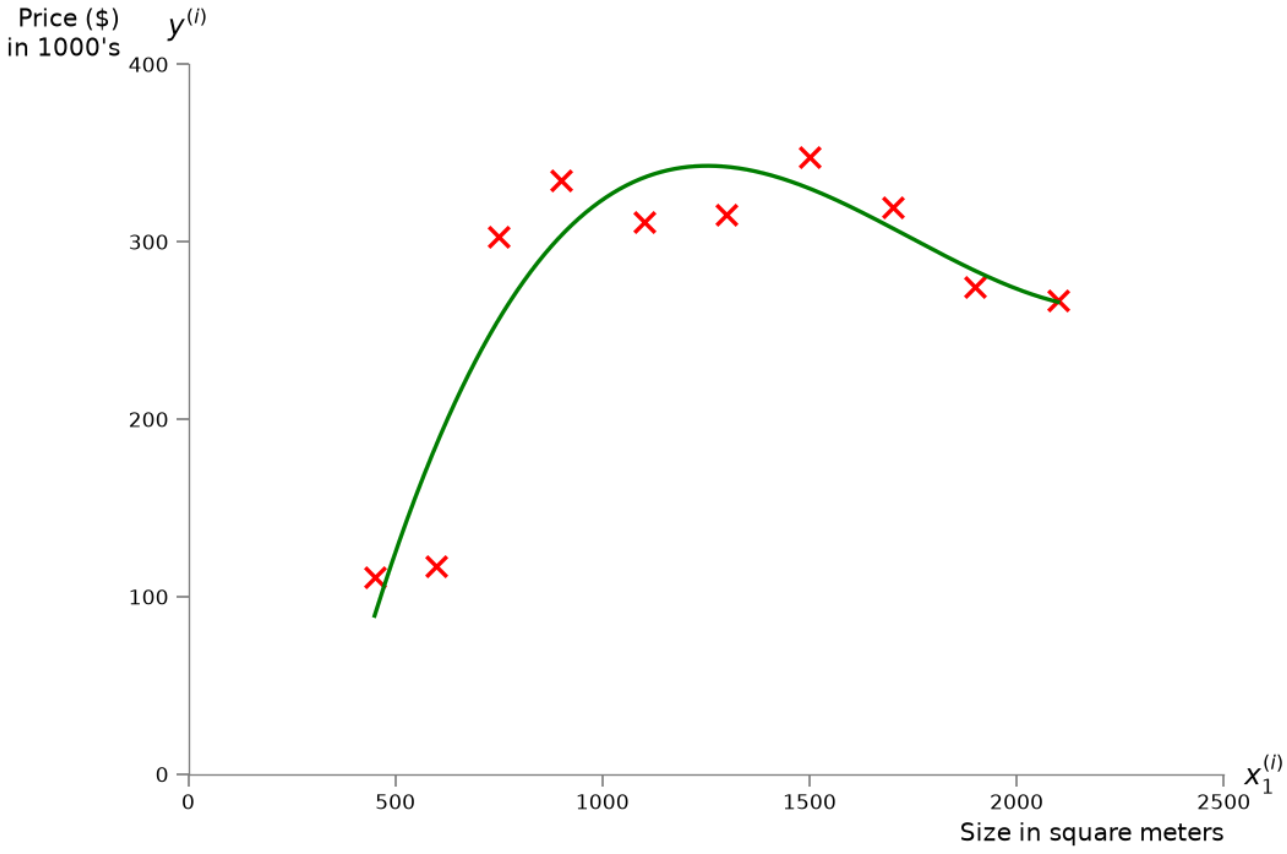
Supervised learning is primarily divided into two types of problems: Regression and Classification.

1. Regression (Predicting a Number)

In a **regression** problem, the output variable y that we want to predict is a **continuous numerical value**. Visually, the goal of regression is to find a function (like a line or a curve) that best fits the continuous data points.

> **Key Question:** "How much?" or "How many?"

For example, consider the problem of predicting the price of houses from the size in square meters:



Other Examples:

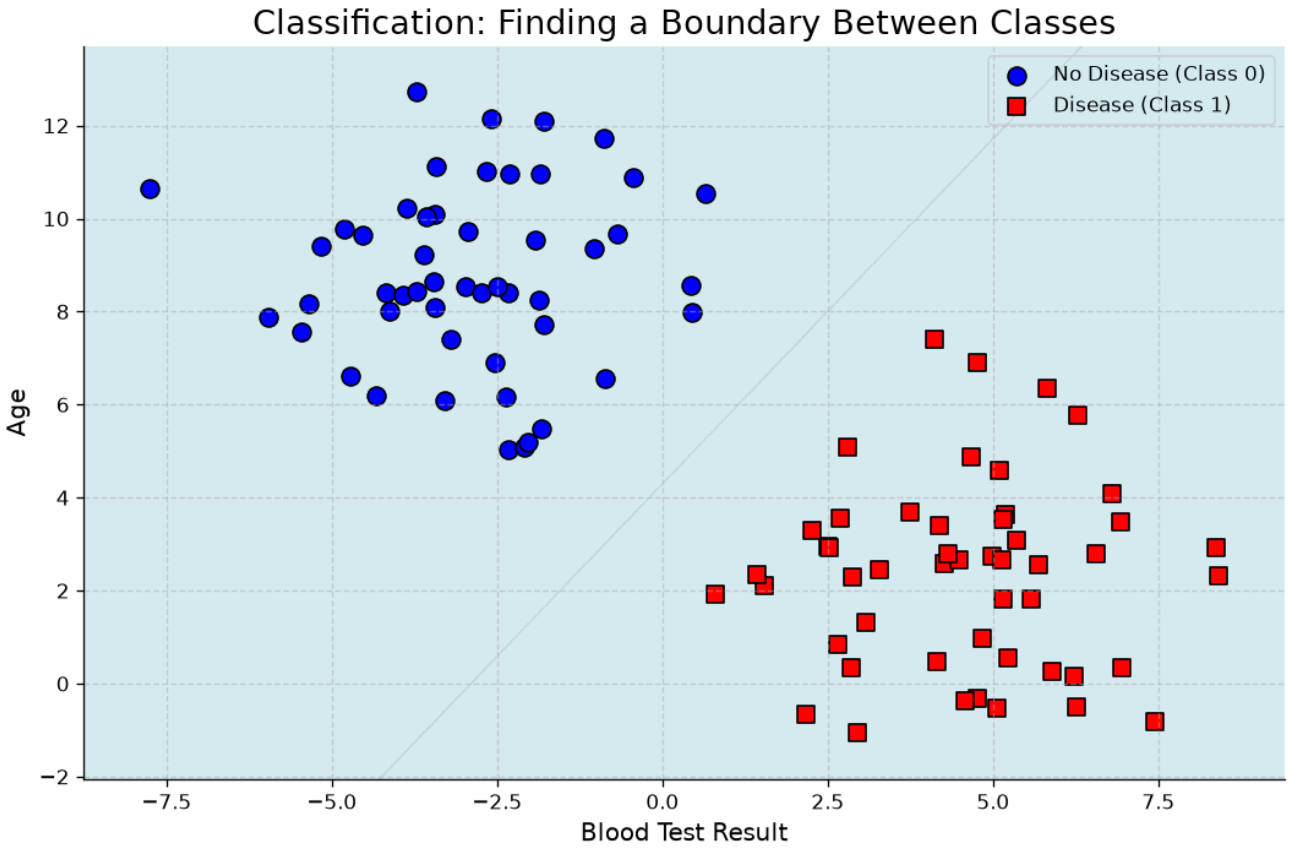
- **Market Risk or Stock Price Prediction.** Given a company's historical data (X), what will its stock price (y) be tomorrow? The output is a number (e.g., $150.25).
- **Predicting a Student's Score.** Given a student's study hours, past grades, and attendance (X), what will their final exam score (y) be? The output is a number (e.g., 87.5).

2. Classification (Predicting a Label)

In a **classification** problem, the output variable y that we want to predict is a **discrete category or label**. Visually, the goal of classification is to find a "boundary" that best separates the different labeled groups.

> **Key Question:** "Which one?" or "Is this A or B?"

For example, the plot below illustrates how a classifier learns a "boundary" to separate two different groups of patients, one with a disease (Class 1) and one without (Class 0), based on their age and the result of a blood test.



Other examples:

- **Spam Filter (Binary Classification).** Given an email's content (X), is it `spam` or `not-spam`? The output y is one of two labels.
- **Medical Diagnosis (Binary Classification).** Given a patient's symptoms and test results (X), do they have the `disease` or `no-disease`?
- **Customer Churn (Binary Classification).** Given a customer's usage and support history (X), will they `churn` (leave) or `stay` next month?
- **Image Recognition (Multiclass Classification).** Given an image (X), is it a `cat`, `dog`, or `bird`? The output y is one of several labels.

Unsupervised Learning: Learning without an Answer Key

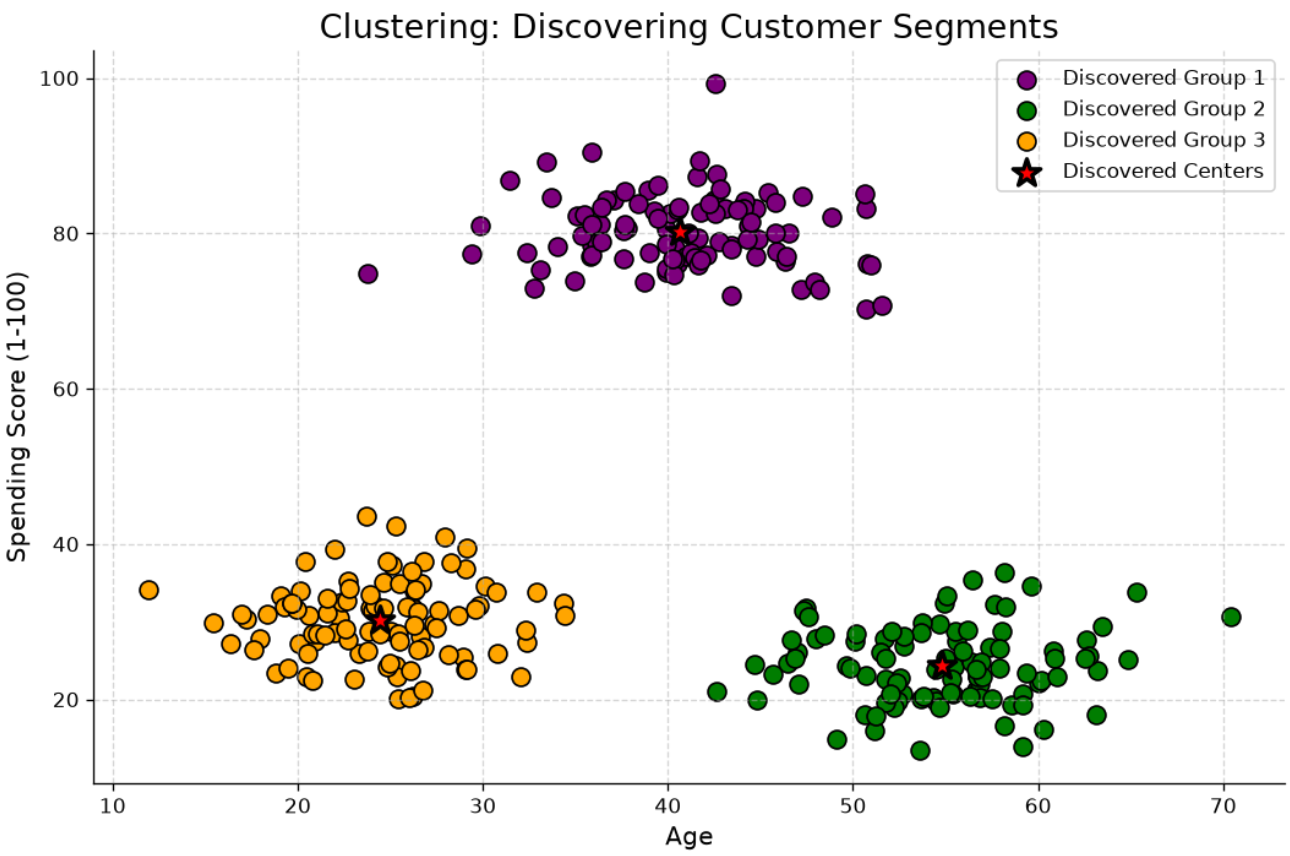
In **unsupervised learning**, our dataset only contains the **input features (X)**. We do *not* have a correct answer y to learn from. The goal is not to predict an outcome, but to find **hidden structure, patterns, or groupings** within the data itself.

1. Clustering (Finding Groups)

Clustering is the most common unsupervised task. The goal is to automatically group similar data points together into "clusters" based on their features. Visually, the goal of clustering is to discover these "hidden" groups in data that does not have any pre-existing labels.

> **Key Question:** "What are the natural groups in my data?"

For example, let's look at the "Customer Segmentation" problem. A company might have data on its customers, such as their `Age` and their `Spending Score`, but no pre-existing "groups" or labels. The data is **unsupervised**. A clustering algorithm's job is to analyze this unlabeled data and discover the hidden structure. As the plot below illustrates, the algorithm might automatically find three distinct clusters, which a business analyst could then identify as, for instance, **"Young Savers", "Prime Spenders", and "Older Savers"**. The company can then create targeted marketing campaigns for each discovered group.



2. Dimensionality Reduction

Dimensionality Reduction is the process of reducing the number of features (dimensions) in a dataset, while trying to retain as much of the original information as possible.

Instead of 100 features, we might find 2 or 3 "summary" features (called principal components) that capture most of the original data's variance.

- **Key Question:** "Can I represent my data with fewer features without losing its essential structure?"
- **Why it's useful:**
  1. **Visualization:** It allows us to plot complex, high-dimensional data (e.g., 50 features) on a 2D or 3D graph.
  2. **Efficiency:** It makes other machine learning models (like regression or classification) faster and more stable by removing redundant, noisy features.
- **Example:** `Principal Component Analysis (PCA)`.

3. Anomaly Detection

Anomaly Detection (or Outlier Detection) is the task of identifying data points that are rare and significantly different from the majority of the data.

This is an unsupervised task because we don't have a pre-labeled list of "anomalies." We just have a large set of mostly "normal" data, and our goal is to find the "odd ones out."

- **Key Question:** "Does this new data point look 'normal' compared to everything else?"
- **Example:** `Fraud Detection`. Given 1,000,000 normal credit card transactions, can we identify the 5 transactions that are fraudulent? Other examples include monitoring for a failing server in a data center or finding defective products on an assembly line.

Parametric vs. Non-Parametric Models

There are different types of models. A fundamental way to categorize these models is by how they handle their own complexity—specifically, whether their number of parameters is fixed or flexible.

Parametric Models: The "Fixed" Approach

A **parametric model** is one that makes strong assumptions about the form of the function $f(X)$. It "locks in" its structure *before* it even sees the data.

The defining feature of a parametric model is that it has a **fixed number of parameters**, regardless of the size of the training dataset.

- **How it works:** The model "learns" by finding the best values for its finite set of parameters (e.g., by minimizing empirical risk).
- **Examples:**

- **Linear Regression:** The model is pre-defined by the equation $h(x) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$. The *only* things it can learn are the coefficients ($\beta_0, \ldots, \beta_n$). No matter if you have 100 or 100 million data points, the model is still just this set of β parameters.
- **Logistic Regression:** Same as linear, but squashes the output with a logistic function. It still only learns a fixed set of β parameters.
- **Naive Bayes:** Learns a fixed set of probabilities for each feature.

The Trade-off:

- **Pros:**
  - **Simpler:** They are easier to understand and interpret ("glass box").
  - **Faster:** They are fast to train.
  - **Less Data:** They don't require as much training data.
- **Cons:**
  - **Constrained:** They are "biased" by their initial assumption (e.g., that the data is linear). If the true relationship is highly complex, a parametric model will fail to capture it.
  - **Risk:** They are prone to **underfitting**.

Non-Parametric Models: The "Flexible" Approach

A **non-parametric model** is one that makes very few or no assumptions about the form of the true function $f(X)$. It lets the data "speak for itself."

The defining feature of a non-parametric model is that the number of parameters is **not fixed**. The model's structure and complexity **grow as the amount of training data increases**.

- **How it works:** The model "learns" by essentially fitting itself to the training data. The model *is* the data.
- **Examples:**
  - **k-Nearest Neighbors (KNN):** To make a prediction for a new point, the model *literally* has to look at the "k" closest points from the *entire* training set. If your training set doubles, the "model" (the stored data it refers to) doubles.

The Trade-off:

- **Pros:**
  - **Flexible:** They can fit a much wider range of functional forms.
  - **Less Bias:** They are not limited by a wrong assumption and can capture the true, complex pattern in the data.
- **Cons:**
  - **More Data:** They require *a lot* more data to learn effectively.
  - **Slower:** They are often much slower to train.
  - **Less Interpretable:** They are often "black boxes."

- **Risk:** They are highly prone to **overfitting** (memorizing the noise in the data).

Why This Matters: The Bias-Variance Connection

This choice is the practical embodiment of the **Bias-Variance Tradeoff** that we will discuss next.

- When you choose a **Parametric Model**, you are placing a big bet on your initial assumption (e.g., "the data is linear"). You are deliberately choosing a **high-bias, low-variance** approach.
- When you choose a **Non-Parametric Model**, you are assuming you have enough data for the true pattern to reveal itself. You are choosing a **low-bias, high-variance** approach.

The next challenge, which we will cover, is how to *control* the complexity of these models to find the perfect "sweet spot" between underfitting and overfitting.

The "Learning" Principle

Now that we have distinguished between Supervised (learning with an answer key) and Unsupervised (learning without one) problems, let's build a formal framework.

For the rest of this discussion, we will focus **exclusively on the Supervised Learning problem**, as the concepts of "learning" and "prediction error" are most formally defined there.

Formal Problem Formulation

Let $\mathcal{X}$ and $\mathcal{Y}$ be two spaces of objects. We can see them as the sample spaces of two random variables X and Y, with $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ being two realizations of the random variables: $X = x$ and $Y = y$.

As defined in the previous section, the goal of predictive modeling is to find a function (or **hypothesis**) h:

$$h : \mathcal{X} \rightarrow \mathcal{Y}$$

which outputs an object $\hat{y} \in \mathcal{Y}$, given an object $x \in \mathcal{X}$. We use the symbol y to refer to the **true value** (or **ground truth**) associated with x, while $\hat{y}$ **will represent the prediction** obtained using our function h:

$$\hat{y} = h(x)$$

Example 1: Medical Diagnosis

We can imagine $\mathcal{X} = \Re^m$ and $\mathcal{Y} = \{0, 1\}$, with $\mathbf{x} \in \Re^m$ being a numerical vector representing the results of different blood tests, while $y \in \{0, 1\}$ is a response variable indicating whether the subject has (1) or does not have (0) a given disease. Finding an appropriate h will allow us to predict $\hat{y} = h(\mathbf{x})$.

Example 2: Simple Spam Filter

We want to build a **spam filter** where $\mathcal{X}$ is the set of all possible emails and $\mathcal{Y} = \{0, 1\}$. Let $f(x)$ be a function that counts the number of orthographical mistakes in an email x. We can define our hypothesis h as:

$$h(x) = \begin{cases} 1 & \text{if } f(x) > \theta \\ 0 & \text{otherwise} \end{cases}$$

Parameters: What the Model Learns

The function h is rarely defined explicitly. Instead, it is a **parametric model**, and the "learning" process consists of finding the best values for its **parameters**.

The threshold θ in Example 2 is a perfect, simple example of a **parameter**. It is a "knob" that the learning algorithm must tune. A different value for θ creates a different hypothesis function h.

A more complex model, like a linear regressor (which we will see soon), will have different parameters (e.g., coefficients $\beta_0, \beta_1, \ldots, \beta_n$). The set of all possible functions h that we can create by changing the parameters is called the **class of functions (or hypothesis space)** $\mathcal{H}$.

The question is: how do we find the *best* parameters?

The Learning Process: Statistical Learning and Risk

To find the "best" h from our hypothesis space $\mathcal{H}$, we need a way to measure how "good" it is. We do this by defining a **loss (or cost) function**, $L(\hat{y}, y)$, which measures the penalty for making a prediction $\hat{y}$ when the true value is y.

$$L(\hat{y}, y)$$

Ideally, we want to find the function h^* that has the lowest possible loss *on average* over *all possible data*, not just the data we have. This "true" average loss is called the **Risk** $R(h)$, defined as the **expected loss** under the true (but unknown) joint probability distribution $P(X, Y)$:

$$R(h) = E_{(x,y)\sim P(X,Y)}[L(h(x), y)]$$

The goal of **statistical learning** is to solve this optimization problem:

$$h^* = \arg\min_{h \in \mathcal{H}} R(h)$$

Empirical Risk Minimization (ERM): The Practical Method

There is a major problem: we can almost never compute the true Risk $R(h)$ because we do not know the true distribution $P(X, Y)$.

However, we *do* have a dataset, often called a **Training Set (TR)**, of N examples, which we assume is a representative sample from $P(X, Y)$:

$$\text{TR} = \{(x_i, y_i)\}_{i=1}^{N}$$

By the **law of large numbers**, we can get a good *estimate* of the true Risk by simply calculating the *average* loss over our training set. This is called the **Empirical Risk** $R_{emp}(h)$:

$$R_{emp}(h) = \frac{1}{N} \sum_{i=1}^{N} L(h(x_i), y_i)$$

This leads to the central "learning" principle of machine learning: the **Empirical Risk Minimization (ERM)** principle. It states that since we cannot find the h that minimizes the *true* Risk, the best we can do is choose the hypothesis $\hat{h}$ that minimizes the *empirical* risk:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} R_{emp}(h)$$

A **learning algorithm** is simply an optimization algorithm that solves this problem: it takes the training set TR and finds the parameters for $\hat{h}$ that minimize the empirical risk.

You Have Already Done This: Least Squares Is ERM

This is not a new idea, only a new name for one you met in chapter 8. Take the squared error as the loss:

$$L(h(x_i), y_i) = (h(x_i) - y_i)^2$$

and the empirical risk becomes the mean residual sum of squares:

$$R_{emp}(h) = \frac{1}{N} \sum_{i=1}^{N} (h(x_i) - y_i)^2 = \frac{1}{N} \text{RSS}$$

Minimising it over h is the same as minimising RSS, since $1/N$ is a constant:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} R_{emp}(h) = \arg\min_{h \in \mathcal{H}} \text{RSS}$$

so **ordinary least squares is empirical risk minimisation with a squared loss**. What changes in Block 2 is not the machinery but the question: in chapter 8 we minimised RSS on all the data in order to *describe* it; from here on we minimise it on a training set in order to *predict* data we have not seen, and the gap between those two numbers is the whole subject.

Changing the loss changes the algorithm. The absolute error $|h(x_i) - y_i|$ gives robust regression; the logistic loss gives logistic regression; the hinge loss gives support vector machines. The principle stays the same.

Model Capacity

Before moving on, we need to define the crucial concept of model capacity. In simple terms, a model's capacity is its **flexibility** or **complexity**.

Formally, capacity is related to the **size and richness of the hypothesis space** $\mathcal{H}$ that the model can choose from.

- **Low Capacity Model:**
  - Has a **small, constrained** hypothesis space $\mathcal{H}$.
  - It can only represent a limited set of simple functions.
  - **Example:** Modeling our hypothesis function with a simple line. It can *only* learn linear functions ($h(x) = \beta_0 + \beta_1 x$), no matter what the data looks like. It is "biased" toward simplicity.
  - **Risk:** This model may be too simple to capture the true underlying pattern.
- **High Capacity Model:**
  - Has a **large, rich** hypothesis space $\mathcal{H}$.
  - It can represent a huge set of very complex functions.
  - **Example:** A 10th-degree polynomial. It can learn extremely "wiggly" functions that can fit almost any set of points perfectly.
  - **Risk:** This model is so flexible that it can easily fit the random *noise* in the training data, not just the signal. It might be very accurate for the data it has been trained on, but not for the general distribution of the data.

Assessing Model Accuracy

Once we have used Empirical Risk Minimization (ERM) to "learn" a model $\hat{h}$, we need to know how good it is. A model that perfectly fits our training data (i.e., has a low R_{emp}) is not necessarily a good model. Our ultimate goal is a model that **generalizes** well—that is, one that makes accurate predictions on new, unseen data.

To do this, we must first define *how* to measure performance.

Measuring the quality of fit

To measure the "quality of fit" or performance of a model, we need a concrete **evaluation metric**. This metric is sometimes the same as the **loss function** $L(\hat{y}, y)$ we used in the ERM principle. It quantifies the cost or error of our predictions. The specific metric we choose depends on the problem (e.g., regression vs. classification).

For **regression problems**, the most common metric is the **Mean Squared Error (MSE)**.

If we choose our loss function $L(y, \hat{y}) = (y - \hat{y})^2$, then the **Mean Squared Error (MSE)** is precisely the **Empirical Risk** we are trying to minimize:

$$R_{emp}(h) = \frac{1}{N}\sum_{i=1}^{N}(y_i - h(x_i))^2 = \text{MSE}$$

This leads to a critical question: is a model with a very low MSE on our *training data* a good model? Not necessarily.

Overfitting and Underfitting

This brings us to the central challenge of predictive modeling. The ERM principle tells us to find the model $\hat{h}$ that *best* fits the training data. But what is the risk of doing this?

Overfitting

This is the single biggest "pitfall" in machine learning. It happens when a model becomes *too* complex and becomes *too good* at minimizing the empirical risk. It ends up "memorizing" the training data, including its random noise and quirks.

As an example, consider the following model:

$$\hat{h}(x) = y \text{ s.t. } (x, y) \in \text{TR}$$

The hypothesis $\hat{h}$ defined above will lead to an empirical risk equal to 0 as long as the loss function is defined in such a way that $L(y, y) = 0$. Indeed:

$$R_{emp}(\hat{h}(x)) = \frac{1}{N}\sum_{i=1}^{N} L(y, y) = 0$$

However, the function above will not be defined for any value:

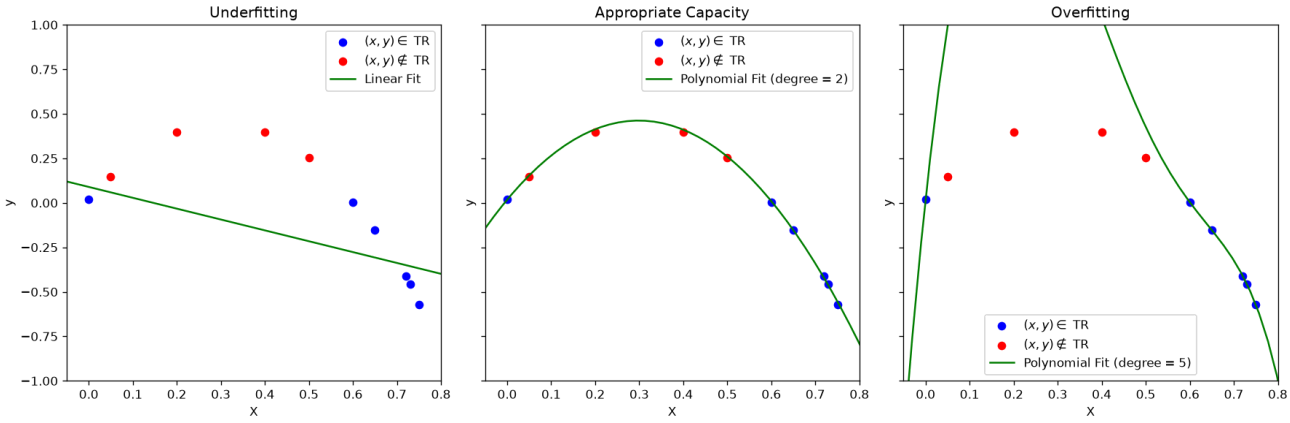
$$x' \in \mathcal{X} \text{ s.t. } (x', y) \notin \text{ TR } \forall y \in \mathcal{Y}$$

Since **we expect the training set to be a sample from the distribution $P(X, Y)$, we expect such values x' to exist** (otherwise the training *is* the population, which is in general not true).

Underfitting

The phenomenon of overfitting is often related to the **capacity of the model**. A model with a **large capacity** can represent very complex functions h and result in overfitting, while a model with a **small capacity** can represent relatively simple functions h, making overfitting harder, but possibly resulting in an **underfitting model**, i.e., a model which is too simple, and as a result cannot reach a low empirical risk.

This is best seen with the example below:



The example shows the fit of three functions to the data. As can be noted:

- A line will lead to **underfitting**. In this case, the model is too simple to model the data well, so the empirical risk will be large;
- A polynomial of high degree 5 (right) will minimize the empirical risk but find a complex model which will not describe well data which has not been seen at training time. This is a case of **overfitting**. The model is trying to model the idiosyncrasies of the training set (which can be noisy) finding a solution which will work only on the data at hand;
- An appropriate degree (center) will lead to a model with an appropriate capacity. The empirical risk is minimized and the solution also works with unseen data.

Note that this is another instance of the **bias-variance** tradeoff. Complex models (right) have a large variance and a large bias: small variations of the training set are modeled and can lead to wrong solutions. Models that are too simple (left) have a low variance, but can still have a large bias (the model is too simple and the solution is not good). Choosing an appropriate capacity (in this case by choosing an appropriate polynomial degree) leads to a good trade-off between variance and bias.

The Bias-Variance Tradeoff

This choice of capacity of a model introduces one of the most important theoretical concepts in machine learning: the **Bias-Variance Tradeoff**.

To understand this, let's recall our goal. We are trying to build a model, $\hat{f}$, that best estimates the true (but unknown) function f from our general formula, $Y = f(X) + \epsilon$.

We just defined our metric for success in regression: the **Mean Squared Error (MSE)**. We want to find a model $\hat{f}$ that has the lowest possible MSE, *especially on new, unseen test data*.

It can be mathematically proven that the **expected test MSE** at a given point x_0 can be decomposed into three fundamental quantities:

$$E[(y_0 - \hat{f}(x_0))^2] = [\text{Bias}(\hat{f}(x_0))]^2 + \text{Var}(\hat{f}(x_0)) + \text{Var}(\epsilon)$$

This formula is the key to understanding model performance. It tells us that the total expected error of our model is the sum of three parts:

1. **The Squared Bias ($[\text{Bias}(\hat{f}(x_0))]^2$)**
   - This is the error from **"Underfitting"**.
   - **Formally**, Bias is the error introduced by approximating a real-life, complex function f with a much simpler model $\hat{f}$. It's the difference between our model's *average* prediction and the *true* value.
   - **Conceptually**, a high-bias model is "stubborn" and "stuck" in its simple assumptions. **Low-capacity** models (like simple linear regression) have **High Bias**. They miss the true pattern and have a high error on *both* the training and test sets.
2. **The Variance ($\text{Var}(\hat{f}(x_0))$)**
   - This is the error from **"Overfitting"**.
   - **Formally**, Variance refers to the amount by which our model $\hat{f}$ would change if we estimated it using a *different* training dataset.
   - **Conceptually**, a high-variance model is "nervous" or "unstable." It is *too* sensitive to the specific training data it saw. **High-capacity** models (like a high-degree polynomial) have **High Variance**. They fit the training data's noise perfectly (low training error) but fail to generalize to the test set (high test error).
3. **The Irreducible Error ($\text{Var}(\epsilon)$)**
   - This is the variance of the error term ϵ from our original formula $Y = f(X) + \epsilon$.
   - **Conceptually**, this is the noise that exists in the real-world data itself. It's the part of the error that cannot be reduced by *any* model, no matter how good it is. It represents the upper limit on our model's accuracy.

The Tradeoff

This decomposition shows the fundamental tradeoff:

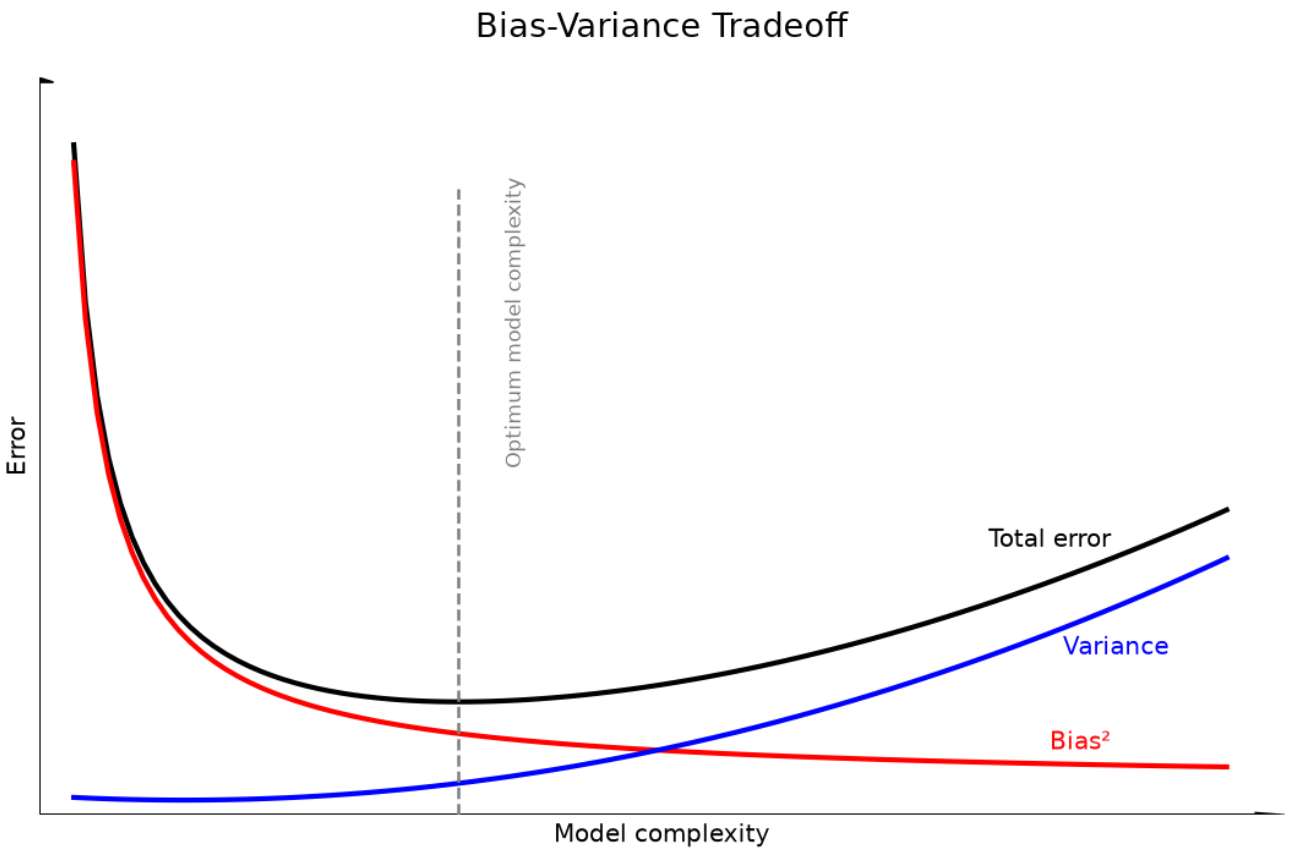
- As we **increase** model capacity (e.g., add more polynomial terms):
  - The model becomes more flexible, so **Bias decreases** (good!).
  - The model becomes more sensitive to the training data, so **Variance increases** (bad!).
- As we **decrease** model capacity (e.g., simplify the model):
  - The model's assumptions become too strong, so **Bias increases** (bad!).
  - The model becomes more stable and less sensitive to noise, so **Variance decreases** (good!).

Our goal as data scientists is not to minimize bias or variance alone, but to find the **"sweet spot"**—the optimal model capacity that minimizes the **total error**.

This leads to the next logical question: How do we *find* this sweet spot? In practice, there are different ways to find this trade-off. In the example above, the degree of the polynomial can be used to reduce

the capacity of the model. Another approach is to use **regularization techniques**, which we will see later in the course.

This is shown in the figure below:



Parameters vs. Hyperparameters

To control model capacity and prevent overfitting, we must distinguish between two types of parameters:

1. **Model Parameters:** These are the values the model *learns* from the data during the training process (i.e., through Empirical Risk Minimization). They are *internal* to the model and are the direct output of the learning algorithm.
   - **Examples:** The coefficients $\beta_0, \beta_1, \ldots$ in a linear regression; the weights in a neural network.
2. **Hyperparameters:** These are the configuration settings for the learning algorithm itself. They are *external* to the model and must be set *before* the training process begins. They act as "knobs" that we use to control the model's capacity and the learning process.
   - **Examples:** The degree `k` in a polynomial regression (which directly controls capacity); the λ regularization parameter in Ridge or Lasso regression (which we will see later); the number of clusters `k` in K-Means clustering.

The central challenge of model selection is this: The learning algorithm (ERM) can find the best *model parameters* for a *given* set of hyperparameters, but it cannot choose the best *hyperparameters*

for you.

For example, ERM can find the best 10th-degree polynomial, and it can find the best 2nd-degree polynomial. But it cannot tell you whether the 10th-degree or the 2nd-degree polynomial is the better *choice*. Choosing this `k` (the hyperparameter) is our job, and it's how we find the sweet spot in the bias-variance tradeoff.

How Do We Find the Best Model? (Model Selection)

We have established that our central challenge is the bias-variance tradeoff. We must find a model with the "sweet spot" of capacity to avoid both underfitting (high bias) and overfitting (high variance).

The process of finding this optimal model is called **model selection**. The method we use depends directly on our primary goal: **Understanding** or **Predicting**.

Approach 1: Statistical Selection (for Understanding)

When our main goal is **inference** (the "glass box" approach), we select a model based on its ability to best explain the data we have, often favoring simplicity.

This approach typically uses the **entire dataset** at once. We are not trying to predict the "future," but to find the most trustworthy explanation for the data we've observed.

We use statistical measures that balance model fit with model complexity:

- **p-values:** We test the significance of each variable and may remove those with high p-values (i.e., those that are likely just noise).
- **Adjusted R^2:** This metric measures goodness-of-fit but penalizes the model for having extra variables that don't add real explanatory power.

Approach 2: Predictive Selection (for Predicting)

When our main goal is **prediction** (the "black box" approach), our selection criteria change entirely. We no longer care about p-values. We only care about one thing: performance on unseen data.

> **The Golden Rule of Predictive Modeling:** The performance of a model on the data it was trained on is *irrelevant*. The *only* measure that matters is its performance on new, unseen data.

We must find a way to *measure* a model's performance on data it has never seen. Since we don't have a time machine, we simulate the future by splitting our existing data.

This is often needed when models are so complex that it is hard to even compute p-values or perform statistical tests.

There are a series of empirical approaches in this category, which we explore in the next sections.

However, before to proceed, we should point out that, while we could evaluate the performance of a model with the **empirical risk**, in practice, it is common to use **performance measures** (the higher the better) or **error measures** (the lower the better) which may be different from the **loss function chosen to train the algorithm**. This is due to the fact that loss functions often need to have some properties to facilitate learning, so they can represent an **approximation of or a deviation from our true objective measure of performance**. We will see more in details the main performance measures later, but for the moment we will note that we will evaluate models using a given performance measure:

$$p : \mathcal{Y}^N \times \mathcal{Y}^N \rightarrow \Re$$

where $Y = \{y_i | (x_i, y_i) \in S\}_{i=1}^N$ is a set of ground truth values from a set of data S and $\hat{Y} = \{h(x_i) | (x_i, y_i) \in S\}_{i=1}^N$ is the set of corresponding predictions. Note that, given the definition above, the empirical risk can be seen as a performance measure, but we need not restrict to the empirical risk to evaluate predictive models.

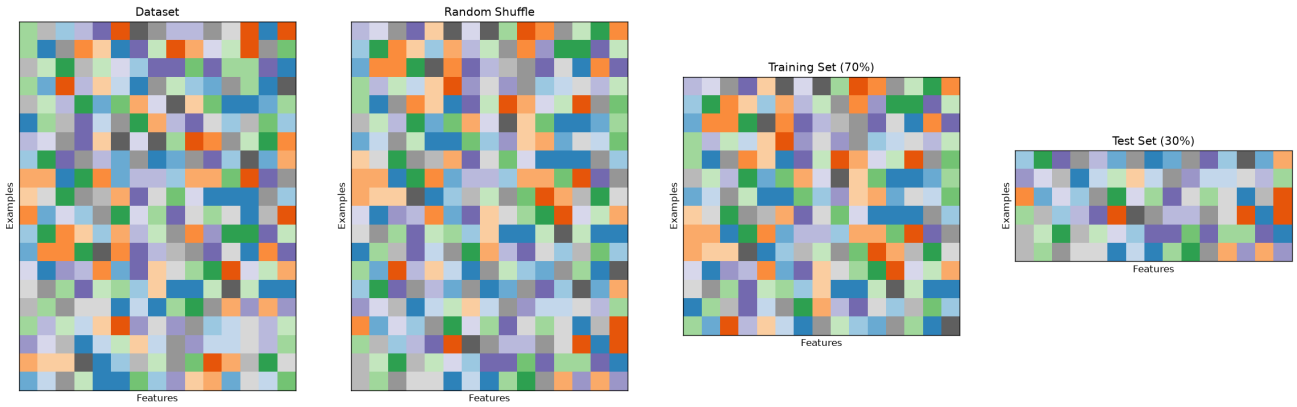
Holdout Validation or Single Split

The simplest form of cross-validation is the holdout test. In this case, the initial set of data is split into two different sets: a training set, which will be used to optimize the model (e.g., with empirical risk minimization) and a test set, which is used to evaluate the performance of the model. The act of optimizing the model on the training set is often called **training**, while the act of evaluating the performance of the model on the test set is called **testing**. This approach **pretends that the test data is not available at training time and only uses it to evaluate performance**.

The **rule number one** when using this technique is **to avoid in any way to choose any characteristic of the model based on the training data**. Indeed, if we did so, we could end up in some form of unmeasurable overfitting. To make sure that both the training and test set are i.i.d., before the split, the data is **randomly shuffled**.

Also, since training data is usually precious and datasets are often not large enough, it is common to split the data asymmetrically, choosing $70 - 80\%$ of the data for training and $20 - 30\%$ of the data for testing.

The figure below illustrates the splitting process:

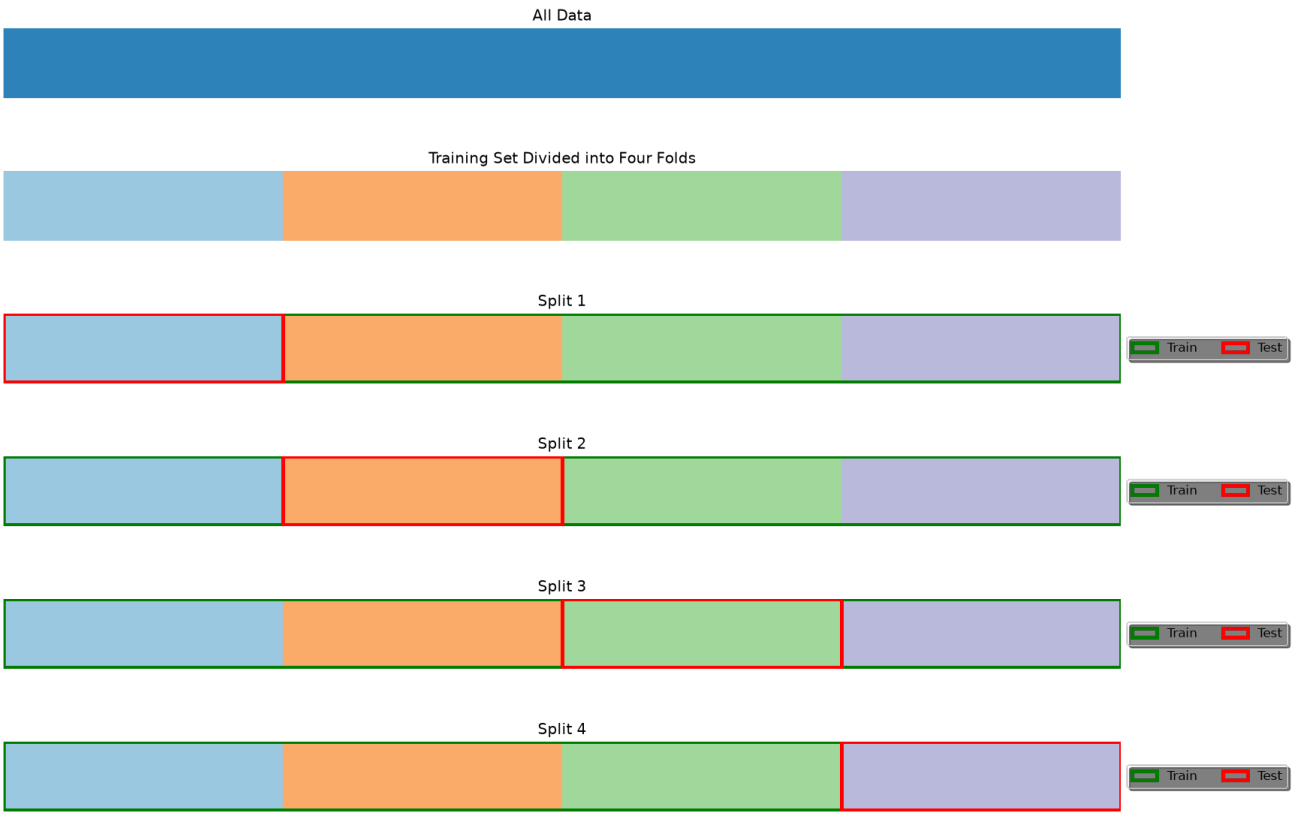


K-Fold Cross-Validation

When the dataset is very small, we may not want to sacrifice a part of it for testing only as this may lead to a biased model. Also, we should note that a small test set could lead to a biased estimation of the model performance. In these cases, rather than randomly splitting the data into two parts, we randomly split it into K different parts, which will be called **folds**.

We then perform training and testing K times, each time using fold i as the test set, and the remaining folds as the training set. The final model performance is obtained by averaging the performance scores computed in each iteration. Note that the obtained performance is unbiased, as each number in the final average is computed on data which has not been seen during training.

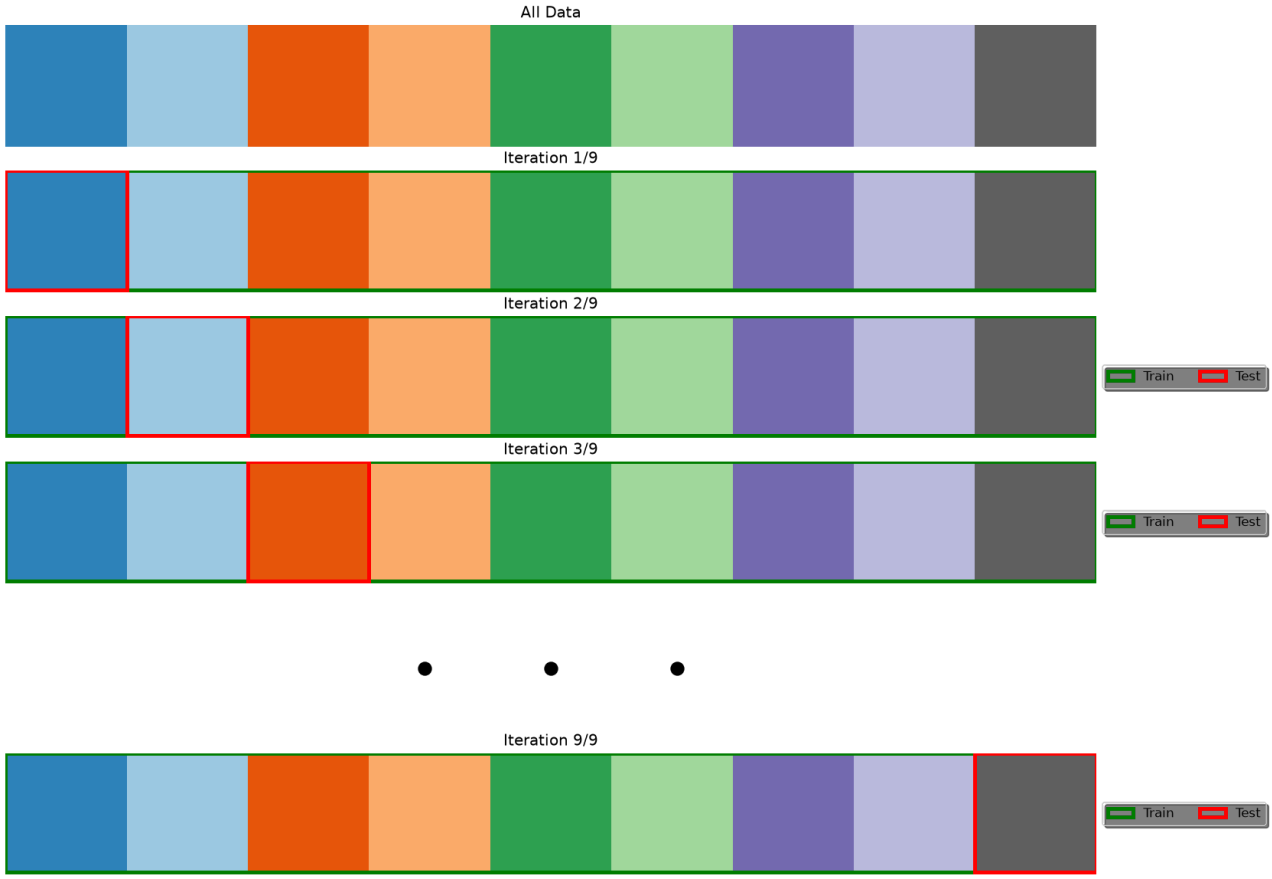
The figure below illustrates the case of a 4-fold cross-validation.



The K-Fold cross validation has the advantage of allowing to validate the model without throwing away a significant part of the data, but it is practically feasible only when the training procedure is not too computationally expensive. Indeed, if training a model takes one week, a K-Fold validation will typically require four weeks.

Leave-One-Out Cross-Validation

In leave-one-out cross-validation, the validation stage is performed in N iterations, where N is the number of elements in the dataset. At the ith iteration, the model is trained on all data points except (x_i, y_i) and tested on (x_i, y_i). The final performance is obtained by averaging the performance scores obtained at each iterations. Note that this is the same as K-Fold cross-validation with $K = N$. The figure below illustrates the process:



This approach is useful when the dataset is extremely small, but, similarly to K-Fold cross-validation, it can increase the computation time by a large margin.

Model Selection and Hyperparameter Optimization

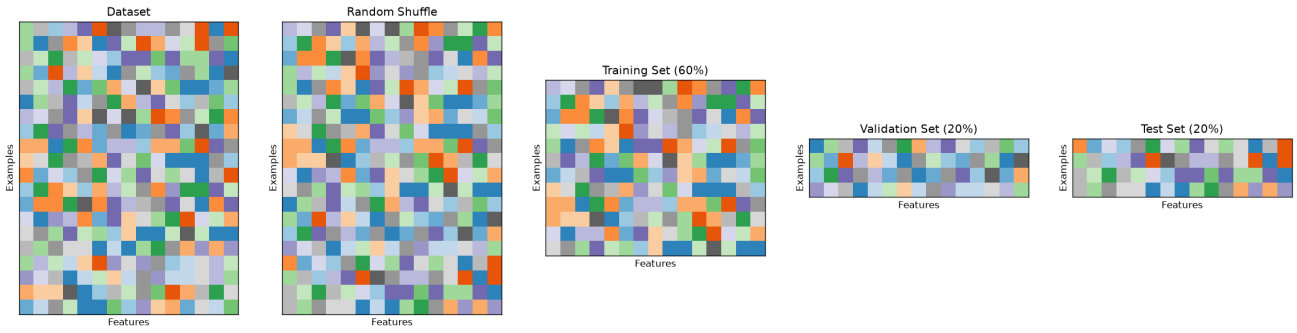
Many algorithms have some parameters which are not explicitly part of the final model h, but they need to be set in order to solve the optimization problem. **For example, in a ridge or lasso regressor, the parameter λ is used to control the amount of regularization during the learning process, however, the final parameter is not part of the model and is not automatically found during the optimization process**.

Hyperparameters are usually found using grid searches: we train models using different values of the hyperparameters and choose the model which performs best. However, **caution should be taken when hyperparameters are selected with grid search**.

Recall that **we are not allowed to make any choice on the final model using the test set**. Indeed, if we did so, we may incur in a form of overfitting which we would not be able to measure. For instance, we can choose a given parameter which works well only for that specific test set. How can we be sure that performance will be good when new data is analyzed?

To avoid this problem, we should work with three different sets of data: a **training set**, a **validation set**, and a **test set**. We will use the training set to train the model, the validation set to choose the hyperparameters and the test set to test the final performance. This is done by training different models on the test set, choosing the hyperparameter values leading to best values on the validation set, and then re-training the model on the training set (or on the union of training and validation set) to final test on the test set.

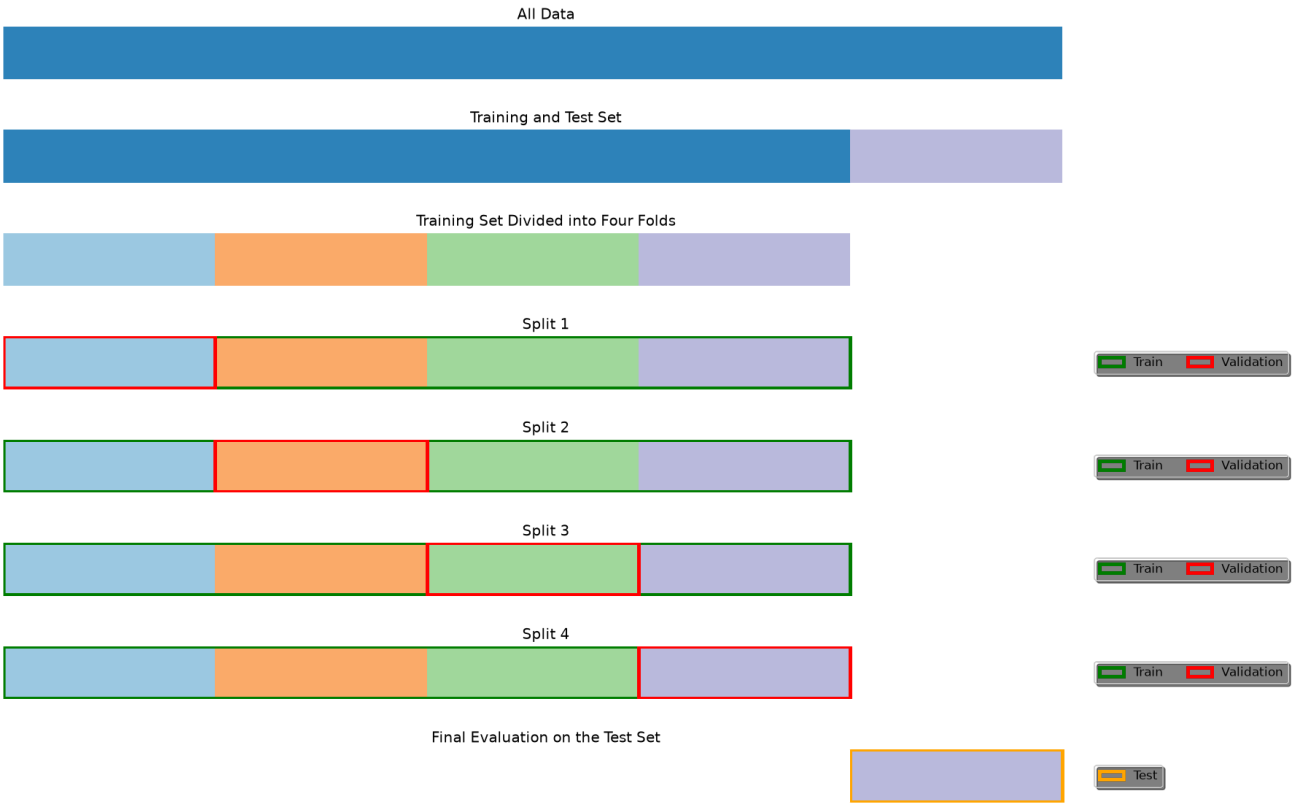
The most common scheme is to use a fixed split with a $60:20:20$ ratio, as shown in the following:



However, other combinations are possible. An approach which often used when the dataset is small is a follows:

- The dataset is split into a training and test set;
- Hyperparameters are optimized using cross-validation on the training set - this consists in executing different cross-validations with different hyperparameter values - then the parameters achieving the best performance are chosen;
- Once the best hyperparameters are found, the model is re-trained on the full training set and tested on the test set.

This is illustrated in the following figure:



Again, if the dataset is large enough and the training procedure is computationally expensive, it is common to use a fixed split as illustrated above.

The `scikit-learn` library allows to easily perform hyperparameter search using cross-validation for different algorithms.

More in general, these techniques can be used to compare different models and select the best performing one.

Exercises

1. **Which goal?** For each, say whether the aim is understanding or prediction, and what changes in how you would fit the model: (a) does this drug work? (b) which patients should be screened first? (c) why are customers leaving? (d) which customers will leave next month?
2. **A split is a random variable.** Fit a logistic regression on this data with ten different `random_state` values for the train/test split. Report the range of test accuracies. What does this say about a paper reporting a single number?
3. **Cross-validation.** Compare 5-fold and 10-fold cross-validation on the same model. Which gives the higher mean, which the higher variance, and why?
4. **Capacity.** Fit polynomial regressions of increasing degree and plot training and test error against degree on the same axes. Mark the point where overfitting begins, and explain the shape of each curve.
5. **Leakage.** Scale the whole dataset *before* splitting, then fit and evaluate. Compare with scaling inside a pipeline after the split. How large is the difference here, and why is the first version wrong even when the difference is small?

References

- Chapter 2 of James, Gareth Gareth Michael. An introduction to statistical learning: with applications in Python, 2023. https://www.statlearning.com
- https://en.wikipedia.org/wiki/Empirical_risk_minimization
- https://scikit-learn.org/stable/modules/cross_validation.html

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Regression for prediction: overfitting, regularization and the scikit-learn workflow

Chapter resources

Slides (PDF) Open in Colab Practice quiz

Up to now a regression model has been a way of *explaining*: we fitted it to all the data we had, read its coefficients, and asked whether they told a defensible story. This chapter asks the other question — **how well does the model predict data it has never seen?** — and the answer changes almost everything about how we fit, evaluate and choose a model.

Chapter 13 set out the protocol: a train/test split and cross-validation. Here we apply it to regression, meet the failure mode that makes the protocol necessary — **overfitting** — and the standard remedy, **regularization**. The chapter ends with the `scikit-learn` workflow that every practical predictive model is built from.

From Understanding to Prediction: The Machine Learning Perspective

The examples above show how quadratic and polynomial regressions allow to obtain **excellent fits** to the data, with improvements in terms of `Adj. R-squared` and significant p-vlaues.

Let's consider the quadratic fit in particular. The `Adj. R-squared` (0.686) is much higher than the simple linear model's (0.605). And the p-values for both `horsepower` and `I(horsepower**2)` are < 0.001.

This is a clear win, but at what cost?

We have lost all simple interpretability.

What is the "effect" of `horsepower` on `mpg`? It's not β_1. It *depends on the value of horsepower*. We can't just look at the table and give a simple explanation.

This is the pivot. If we are willing to sacrifice simple, at-a-glance interpretability in exchange for a model that *fits better*... why stop here?

If our main goal is no longer *understanding* (p-values) but **predictive accuracy**, we should embrace this. We have just crossed the line from statistical inference into the realm of **Machine Learning**.

Metrics for Prediction (Test Set Error)

When our goal is **prediction** (the ML approach), we do not care about R^2 or RSE on the training data. We *only* care about the model's performance on **new, unseen data** (the **Test Set**).

The following metrics are used to evaluate the model on the test set. (We will use m to denote the number of samples in the test set).

Mean Squared Error (MSE)

The MSE is the average of the squared errors on the test set. It is the "predictive" version of the R_{emp} (Empirical Risk) we saw earlier.

$$MSE_{test} = \frac{1}{m}\sum_{j=1}^{m}(y_j - \hat{y}_j)^2$$

- **Interpretation:** This is an **error measure**, so a good model has a **small MSE**.
- **Problem:** The units are **squared** (e.g., if y is in meters, MSE is in square-meters). This is not intuitive.

Root Mean Squared Error (RMSE)

The RMSE is the fix for the MSE's unit problem and is the most common metric for predictive regression. It is simply the square root of the MSE.

$$RMSE_{test} = \sqrt{MSE_{test}} = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(y_j - \hat{y}_j)^2}$$

- **Interpretation:** RMSE is in the **same units as** Y (e.g., meters). It can be read as "On average, our model's predictions on the test set are wrong by about [RMSE value]."
- **Key Property:** Because it squares errors before averaging, the RMSE **penalizes large errors** (outliers) more heavily than small errors.

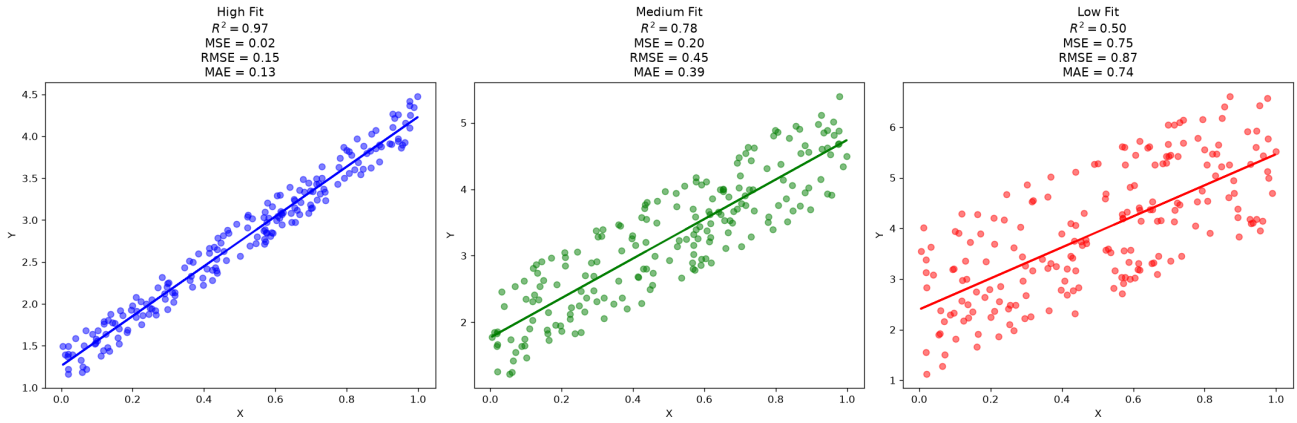
Mean Absolute Error (MAE)

The MAE is an alternative to RMSE that is also highly interpretable.

$$MAE_{test} = \frac{1}{m}\sum_{j=1}^{m}|y_j - \hat{y}_j|$$

- **Interpretation:** MAE is also in the **same units as** Y. It measures the "average absolute error" of the predictions.
- **Key Property:** Unlike RMSE, MAE does *not* disproportionately penalize large errors. It is **more robust to outliers**.

Let's revisit a plot we saw in our last lecture, adding prediction error measures to the picture:



The Problem with Power: Overfitting

We've solved our underfitting problem! By adding a $horsepower^2$ term in **our quadratic regressione example**, we created a model with a **higher capacity** (more flexibility) that fits the data better.

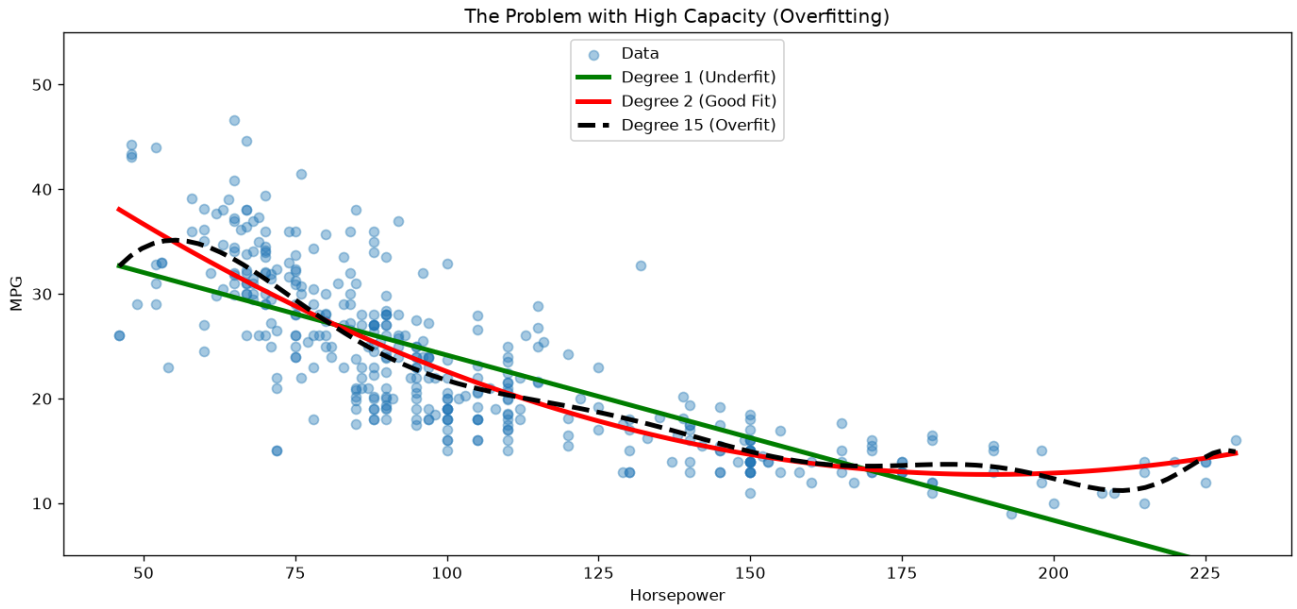
But this creates a new, dangerous question:

> **If $degree = 2$ is good, is $degree = 5$ better? What about $degree = 20$?**

A high-degree polynomial model has *enormous* capacity. It can "wiggle" as much as it needs to fit every single data point. This leads to the central problem of machine learning: **overfitting**.

Overfitting (or **high variance**) is when a model learns the *noise* in the training data, not the underlying *signal*. It will have a fantastic R^2 on the data it was trained on, but will fail (generalize) miserably on any new data.

Look at the example below:



We can see that, while the linear model is clearly **underfitting**, the high-degree polynomial is **overfitting**! The quadratic model seems to be in the sweet spot, but how shall we find the correct degree in a systematic way?

The Solution: Regularization

The plot above shows our dilemma: the $degree = 15$ model is "too powerful." How can we use a high-capacity model *without* it overfitting?

Unfortunately, as the model becomes complex, it's not straightforward to use plain statistics to answer this question.

Instead, following a pure machine learning perspective, we will use a technique called **regularization**.

Regularization adds a "penalty" to the cost function to *discourage* the model from becoming too complex.

Instead of just minimizing RSS, we minimize

$$RSS + \text{Penalty Term}$$

This penalty term is designed to **force the β coefficients to stay small**. A "wiggly" (overfit) curve *requires* massive β coefficients. By penalizing large coefficients, we force the model to find a *simpler, smoother* curve, even if it has many features.

Revisiting Collinearity

Regularization also happens to be a fantastic solution to our *other* major problem: **multicollinearity**.

As we saw in chapter 8, when predictors like `horsepower` and `weight` are highly correlated, the X^TX matrix is ill-conditioned, and the OLS coefficient estimates become **unstable** and their standard errors explode.

Regularization "stabilizes" this matrix, allowing the model to find sensible, stable coefficients even in the presence of multicollinearity.

Ridge Regression (L2 Penalty)

Ridge Regression is the most common type of regularization. It adds a penalty proportional to the **sum of the squares of the coefficients**. This is also called the L_2 **norm**.

The cost function becomes:

$$RSS_{L2} = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{n} \beta_j^2$$

- The RSS term pushes the model to **fit the data** (reducing bias).

- The $\lambda \sum \beta_j^2$ term pushes the coefficients to be **small** (reducing variance).

The **hyperparameter** λ (called `alpha` in code) controls this tradeoff.

- $\lambda = 0$: This is just standard OLS.
- $\lambda \to \infty$: The penalty is everything. All coefficients are forced to zero (a flat line).

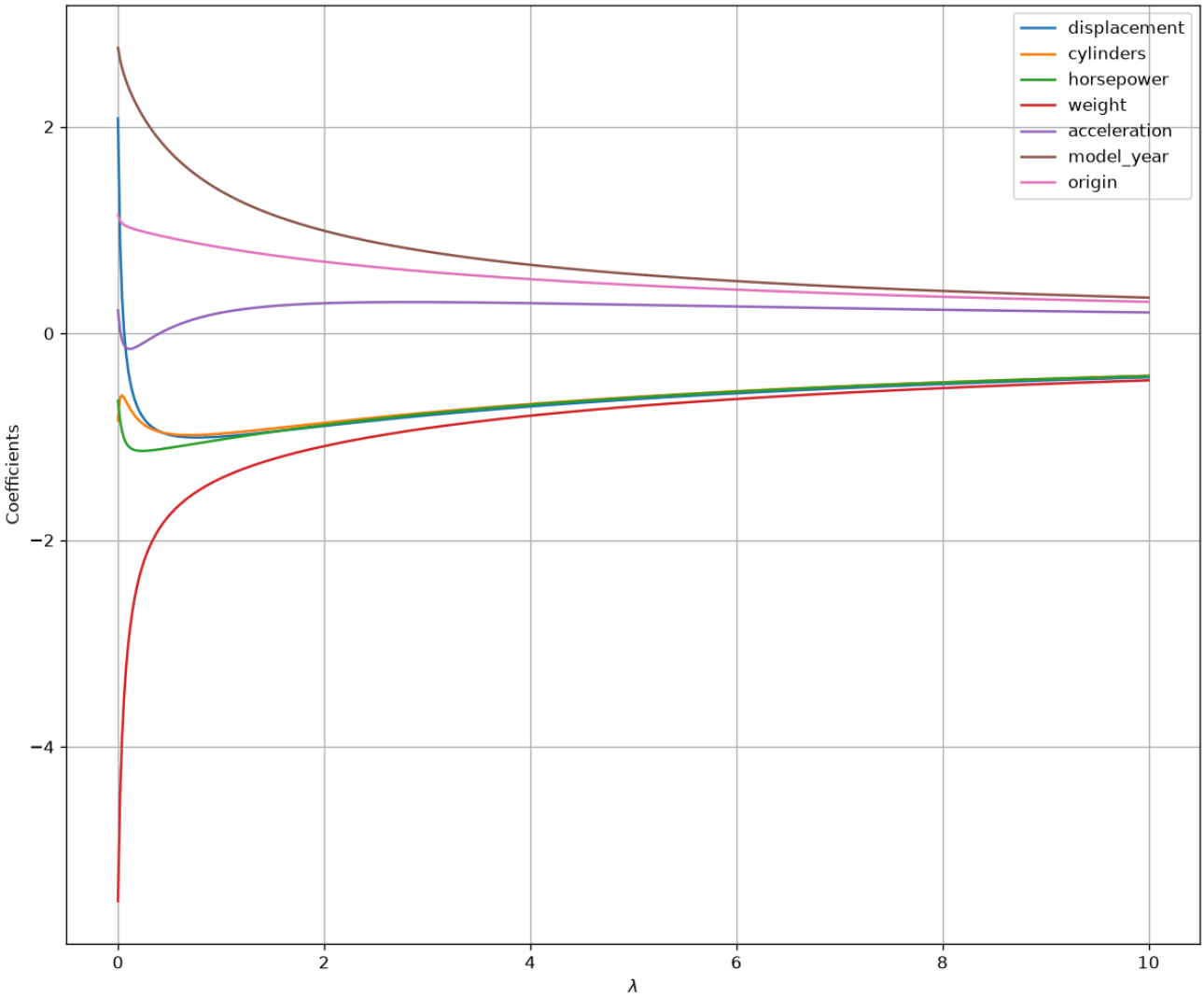
Key Property: Ridge Regression *shrinks* coefficients close to zero, but **it will not set them *exactly* to zero**.

Crucial Step: Because the penalty $\sum \beta_j^2$ is based on the *size* of the coefficients, it is critical that we **standardize (z-score)** all our features first. Otherwise, a feature measured in kilometers will be penalized differently than one measured in millimeters.

Note that the problem above is solved with OLS as follows:

$$\hat{\beta}_{Ridge} = (X^T X + \lambda I)^{-1} X^T y$$

where the $I $ term improves conditioning (invertibility) of the matrix even in the presence of multicollinearity.



As you can see, as λ increases, all coefficients are smoothly "shrunk" toward zero. Interestingly, the model **"decides"** which coefficients to shrink for a given value of λ. For instance, for low values of λ, `weight` is shrunk, while `acceleration` is first shrunk, then allowed to be larger than zero. This is due to the fact that, the regularization term acts as a sort of "soft constraint" encouraging the model to find smaller weights, while still finding a good solution.

It can be shown (but we will not see it formally), that ridge regression **reduces the variance of coefficient estimates**. At the same time, **the bias is increased**. So, finding a good value of λ allows to **control the bias-variance trade-off**.

Interpretation of the ridge regression coefficients

Let us compare the parameters obtained through a ridge regressor with those obtained with a linear regressor (OLS):

```
params = pd.DataFrame({'variables':['displacement','cylinders' , 'horsepower' ,
        'weight' , 'acceleration' , 'model_year' , 'origin'],
        'ridge_params':ols("mpg ~ displacement + cylinders + horsepower + weight +
        acceleration + model_year + origin", data2).fit_regularized(L1_wt=0,
        alpha=1).params[1:],'ols_params':ols("mpg ~ displacement + cylinders +
        horsepower + weight + acceleration + model_year + origin",
        data2).fit().params[1:]}).set_index('variables')
params
```

	ridge_params	ols_params
variables		
displacement	-0.998445	2.079303
cylinders	-0.969852	-0.840519
horsepower	-1.027231	-0.651636
weight	-1.401530	-5.492050
acceleration	0.199621	0.222014
model_year	1.375600	2.762119
origin	0.830411	1.147316

As we can see, the ridge parameters has a smaller scale. This is due to the regularization term. As a result, the parameters **cannot be interpreted statistically as the ones of a linear regressor**. Instead, we can interpret them as denoting the relative importance of the variable to the prediction.

Lasso Regression (L_1 Penalty)

Lasso is an alternative that has a different, very useful property. It adds a penalty proportional to the **sum of the absolute values of the coefficients**. This is also called the L_1 **norm**.

The cost function becomes:

$$RSS_{L1} = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{n} |\beta_j|$$

This subtle change has a profound effect. The “sharp corners” of the absolute value function allow the optimization to find solutions where “useless” coefficients are set to **exactly zero**.

Its Superpower: Lasso performs **automatic feature selection**, which is great when you have hundreds of features.

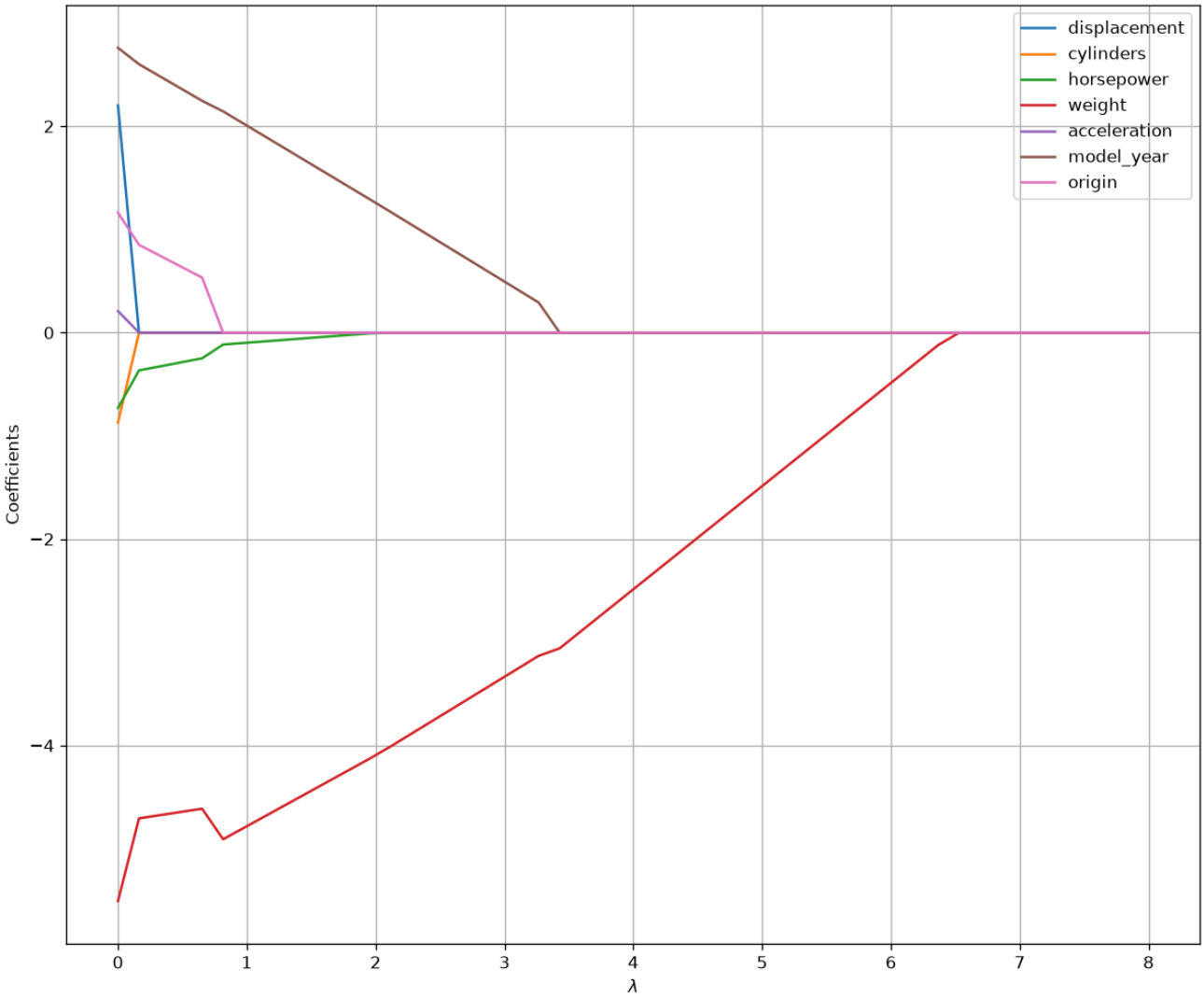
The figure below shows how the coefficient estimates change for different values of λ:

```
from scipy.stats import zscore
from statsmodels.formula.api import ols
import pandas as pd
column_names = ["mpg", "cylinders", "displacement", "horsepower", "weight",
        "acceleration", "model_year", "origin", "car_name"]
data = pd.read_csv(DATA + "auto_mpg.data", sep=r"\s+", names=column_names,
        na_values="?")
data = data[["displacement", "cylinders", "horsepower", "weight", "acceleration",
        "model_year", "origin", "mpg"]]

data2=data.dropna().drop('mpg',axis=1).apply(zscore).join(data.dropna()['mpg'])
dd=[]
for alpha in np.linspace(0,8,50):
    d=ols("mpg ~ displacement + cylinders + horsepower + weight + acceleration +
        model_year + origin", data2).fit_regularized(L1_wt=1, alpha=alpha).params



    d= pd.Series(
        {
            c:v for c,v in zip(["Intercept","displacement", "cylinders", "horsepower"
        ,"weight" ,"acceleration", "model_year" ,"origin"],d)
        }
        )
    d['alpha']=alpha
    dd.append(d)
dd=pd.concat(dd,axis=1).T
dd.drop('Intercept',axis=1).plot(x='alpha', figsize=(12,10))
plt.xlabel("$\lambda$")
plt.ylabel("Coefficients")
plt.grid()
```



As can be seen, the model now makes "hard choices" on whether a value should be set to zero or not, thus performing variable selection. Once Lasso regression identified the variables to remove, we could create a reduced dataset keeping only columns with nonzero coefficients and fit a regular, interpretable, linear regressor with the original, non standardized, data.

Regularization and the Bias-Variance Tradeoff

Regularization is our first, and most powerful, tool for **explicitly managing the Bias-Variance Tradeoff**.

Let's think about the two extremes:

1. **A Standard OLS Model (or high-degree polynomial)**
   - This model's only goal is to **minimize** RSS.
   - It will "succeed" by fitting the training data perfectly, including all its noise.
   - This results in a model with Low Bias (it's a perfect fit for the data it saw) but $\text{Extremely High Variance}$ (it's "nervous" and will fail on new data). This is **overfitting**.
2. **A Regularized Model (Ridge or Lasso)**
   - This model has a *different* goal: minimize $RSS + \text{Penalty Term}$.
   - To get the lowest total score, it is *forced* to make a compromise. It will *intentionally* stop fitting the training data perfectly, in order to make its coefficients smaller and reduce the

penalty.
    - This is the trade-off in action:
        - We **increase the bias** slightly (because our model is now "imperfect" for the training data).
        - In exchange, we **dramatically decrease the variance** (because the model is smoother and no longer chasing noise).

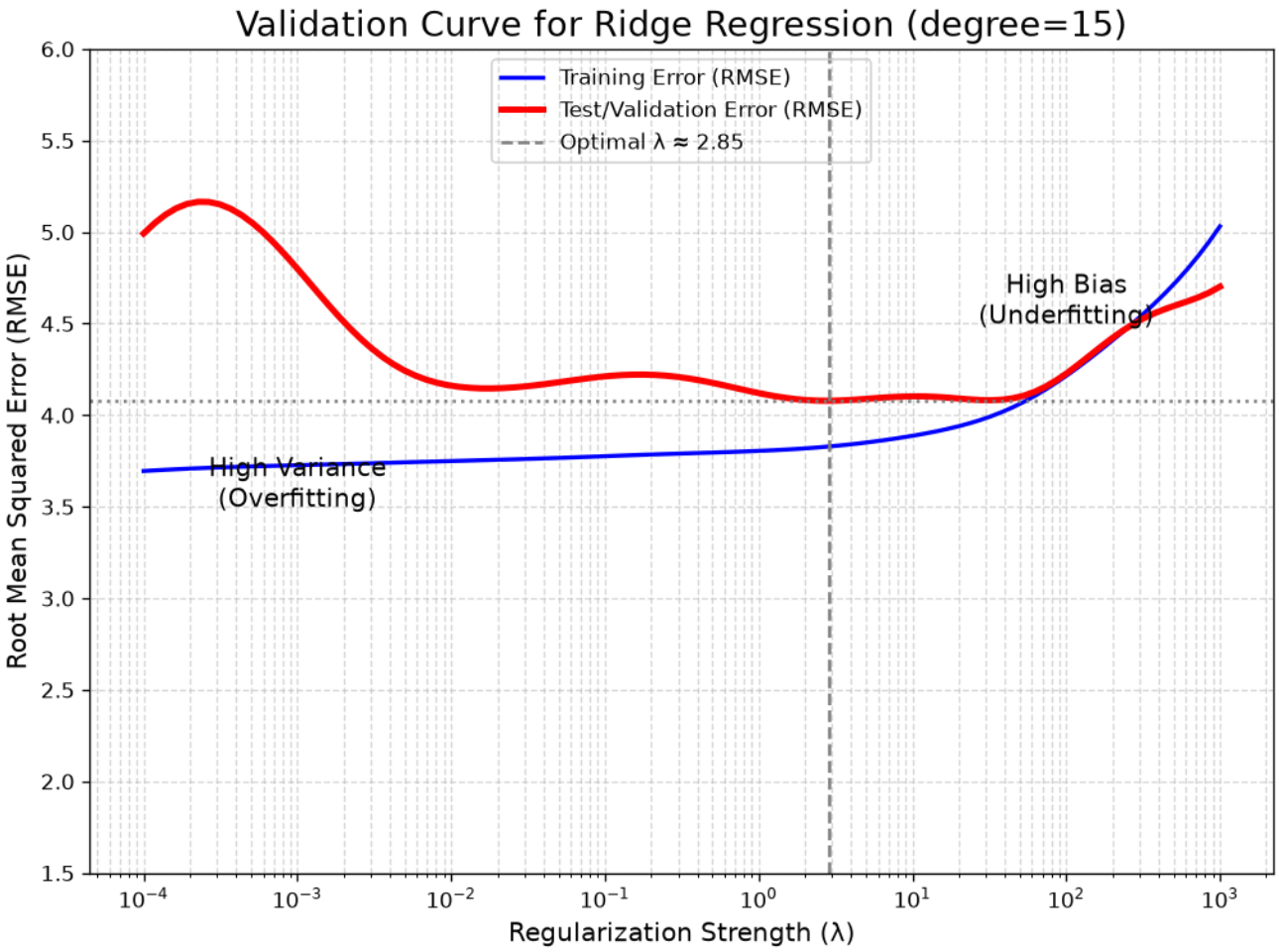
λ is the "Tradeoff Knob"

The **hyperparameter** λ (called `alpha` in `scikit-learn`) is our **explicit control knob** for moving along the bias-variance curve.

- **When $\lambda = 0$:** We are on the far right of the graph. We have a standard OLS model with Low Bias and High Variance (overfitting).
- **As we increase λ:** We "turn the knob," increasing the penalty. This moves us left on the graph. Bias starts to go up, but Variance drops *faster*. The **Total Error** (the black line) goes down.
- **At the "sweet spot":** We find the optimal λ that gives us the lowest possible **Total Error**.
- **As $\lambda \to \infty$:** We move to the far left of the graph. We are penalizing the coefficients so much that they all become zero (a flat line). The model has High Bias and Low Variance (underfitting).

The *entire goal* of the machine learning workflow, which we are about to see, is to find the **optimal value of** λ. We can't use our `statsmodels` p-values for this. We must use a new technique, **cross-validation**, to find the λ that gives the best predictive score on unseen data.

The plot below shows how training and validation error change when fitting a polynomial of degree 15 (a high-capacity model) as we increase the regularization hyperparameter λ, which illustrates how **regularization controls the bias-variance trade-off**.



Let's see how to interpret the plot:

- **Left Side (λ is tiny, e.g., 0.001):**
  - The **Training Error** (blue) is very low. The model is *overfitting* the training data.
  - The **Test Error** (red) is high. The model is *not generalizing* well.
  - This is the **High Variance** region.
- **Right Side (λ is large, e.g., 1000):**
  - The **Training Error** is high and almost equal to the test error. The model is *too simple* (too penalized).
  - The **Test Error** is also high.
  - This is the **High Bias** region (underfitting).
- **The "Sweet Spot" (Dashed Line):**
  - This is the **optimal** λ (our optimal λ).
  - It's the "Goldilocks" point where the Test Error is at its minimum.
  - This is the perfect balance between bias and variance, giving us the best possible predictive model.

From Theory to Practice: The `scikit-learn` Workflow

We have now established all the theory.

1. **Our Problem:** Simple linear models can **underfit** (high bias), like the "U-shape" in our `mpg ~ horsepower` plot.
2. **A Solution:** We can use **Polynomial Regression** to add non-linear features ($X^2, X^3, \ldots$) which gives our model higher **capacity** to fit the curve.
3. **The New Problem:** High-capacity models can easily **overfit** (high variance), "memorizing" the training data noise.
4. **A Better Solution:** We use **Regularization (Ridge L_2 or Lasso L_1)** to add a penalty (λ) that controls this high capacity, finding the "sweet spot" in the Bias-Variance tradeoff.

We've seen the *theory* of λ. Now we need a practical, robust workflow to actually *find* the best λ and build the best predictive model.

It's time to switch from `statsmodels` (built for **inference**) to `scikit-learn` (built for **prediction**).

Our New Toolkit

Our new workflow requires new rules and tools:

1. **Train/Test Split:** This is the **Golden Rule**. We *must* split our data to simulate a "future" test set. Our only goal is to get the best score (e.g., $RMSE$) on this test data.
2. **Standardization:** We *must* scale our features (e.g., `StandardScaler`) so the regularization penalty λ affects them all fairly.
3. **Pipelines:** We must bundle our steps (scaling, polynomial features, model) together. This is a best practice that prevents "data leakage" and makes our workflow reproducible.

Dataset

We'll use the California Housing dataset provided by the `scikit-learn` library. Let us load the data as a dataframe and have a look at the data description:

```
from sklearn.datasets import fetch_california_housing
data = fetch_california_housing(as_frame=True)
print(data['DESCR'])
data['data']
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

:Number of Instances: 20640

:Number of Attributes: 8 numeric, predictive attributes and the target

:Attribute Information:
    - MedInc        median income in block group
    - HouseAge      median house age in block group
    - AveRooms      average number of rooms per household
    - AveBedrms     average number of bedrooms per household
    - Population    block group population
    - AveOccup      average number of household members
    - Latitude      block group latitude
    - Longitude     block group longitude

:Missing Attribute Values: None

This dataset was obtained from the StatLib:
https://lib.stat.cmu.edu/datasets/houses.zip

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per census
block group. A block group is the smallest geographical unit for which the U.S.
Census Bureau publishes sample data (a block group typically has a population
of 600 to 3,000 people).

A household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, these
columns may take surprisingly large values for block groups with few households
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. rubric:: References

- Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
  Statistics and Probability Letters, 33:291-297, 1997.
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25
...	...	...	...	...	...	...	...	...
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.48	-121.09
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.49	-121.21
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.43	-121.22
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.43	-121.32
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.37	-121.24

20640 rows × 8 columns

The dataset contains 8 variables. The independent variable is `MedInc`, the average value of houses in a given suburb, while all other variables are independent. For our aims, we will treat the data as a matrix of numerical variables. We could easily convert the dataframe in this format, but scikit-learn allows to load the data directly in this format:

```
# let us load the data without passing as_frame=True
data = fetch_california_housing()
X = data.data # the features
y = data.target # the targets

print(X.shape, y.shape)
```

```
(20640, 8) (20640,)
```

Data Splitting

We will split the dataset into a training, a validation and a test set using the `train_test_split` function:

```
from sklearn.model_selection import train_test_split
# We'll do a 60:20:20 split
val_prop = 0.2
test_prop = 0.2

# We'll split the data in two steps - first let's create a test set and a combined
        trainval set
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=test_prop,
        random_state=42)

# We'll now split the combined trainval into train and val set with the chosen
        proportions
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval,
        test_size=test_prop/(1-test_prop), random_state=42)

# Let us check shapes and proportions
print(X_train.shape, X_val.shape, X_test.shape)
print(X_train.shape[0]/X.shape[0], X_val.shape[0]/X.shape[0],
        X_test.shape[0]/X.shape[0])
```

```
(12384, 8) (4128, 8) (4128, 8)
0.6 0.2 0.2
```

The `train_test_split` function will split the data randomly. We are passing a fixed `random_state` to be able to replicate the results, but, in general, we should avoid that if we want the split to be truly random (though it is common to use random seeds for splitting in research). Note that, while the split is random, the function makes sure that the i-th element of the y variable corresponds to the i-th element of the X variable after the split.

We will now reason mainly on the validation set, comparing different models and parameter configurations. Once we are done with our explorations, we'll check the final results on the test set.

Data Normalization

We'll start by normalizing the data with z-scoring. This will prove useful later when we use certain algorithms (e.g., regularization). Note that we have not normalized data before because we need to **make sure that even mean and standard deviation parameters are not computed on the validation or test set**. While this may seem a trivial detail, it is important to follow this rule as strictly as possible to avoid bias. We can normalize the data with the `StandardScaler` object:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train) # tunes the internal parameters of the standard scaler

X_train = scaler.transform(X_train) # does not tune the parameters anymore
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

Scikit-learn objects have a unified object-oriented interface. Each algorithm is an object (e.g., `StandardScaler`) with standard methods, such as:

- A `fit` method to tune the internal parameters of the algorithm. In this case, it is a vector of means and a vector of standard deviations, but in the case of a linear regression it will be a vector of weights;
- A `transform` method to transform the data. Note that in this stage no parameters are tuned, so we can safely apply this method to validation and test data. This method only applies to objects which transform the data, such as the standard scaler;
- A `predict` method to obtain predictions. This applies only to predictive models, such as a linear regressor;
- A `score` method to obtain a standard performance measure on the test or validation data. Also this only applies to predictive models.

We will see examples of the last two methods later.

Linear Regressor

We will start by training a linear regressor. We will use scikit-learn's implementation which does not provide statistical details (e.g., p-values) but is optimized for predictive modeling. The train/test interface is the same as above:

```
from sklearn.linear_model import LinearRegression

linear_regressor = LinearRegression()
linear_regressor.fit(X_train, y_train) # this tunes the internal parameters of the
        model

# Let us print the model's parameters
print(linear_regressor.coef_)
print(linear_regressor.intercept_)
```

```
[ 0.86025287  0.1200073  -0.28039183  0.31208687 -0.00957447 -0.02615781
 -0.88821331 -0.86190739]
2.068077419250432
```

We can obtain predictions on the validation set using the `predict` method:

```
y_val_pred = linear_regressor.predict(X_val)
print(y_val_pred.shape)
```

```
(4128,)
```

The function returns a vector of 4128 predictions, one for each example in the validation set. We can now evaluate the predictions using regression evaluation measures. We will use the standard implementation of the main evaluation measures as provided by scikit-learn:

```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

mae = mean_absolute_error(y_val, y_val_pred)
mse = mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print(mae, mse, rmse)
```

```
0.5333346447415043 0.5297481095803488 0.727837969317587
```

All evaluation measures in scikit-learn follow the `evaluation_measure(y_true, y_pred)` convention. Note that the target variable `MedInc` is measured in tens of thousands of dollars, so an MAE of about 0.5 corresponds to an average error of about 5000 dollars. This is not that bad if we consider the mean and standard deviation of targets:

```
y_train.mean(), y_train.std()
```

```
(np.float64(2.0680774192506464), np.float64(1.1509151433486544))
```

Each predictor in scikit-learn also provides a `score` method which takes as input the validation (or test) inputs and outputs and computes some standard evaluation measures. By default the linear regressor in scikit-learn returns the R^2 value:

```
linear_regressor.score(X_val, y_val)
```

```
0.6142000785497264
```

While we are mainly interested in the performance of the model on the validation set (and ultimately on those on the test set), it is still useful to assess the performance on the training set for model diagnostics. For instance, if we see a big discrepancy between training and validation errors, then we can imagine that some overfitting is going on:

```
y_train_pred = linear_regressor.predict(X_train)

mae_train = mean_absolute_error(y_train, y_train_pred)
mse_train = mean_squared_error(y_train, y_train_pred)
rmse_train = np.sqrt(mse_train)

print(mae_train, mse_train, rmse_train)
```

```
0.5266487515751345 0.5143795055231385 0.7172025554354491
```

We can see that, while there are some differences between training and test performance, those are minor, so we can deduce that there is no significant overfitting going on.

We should note that we should **always expect a certain degree of overfitting, depending on the task, the data and the model**. When the difference between train and test error is large, and hence there is significant overfitting, we can try to reduce this effect with regularization techniques.

To better compare models, we will now store the results of our analyses in a dataframe:

```
import pandas as pd
california_housing_val_results = pd.DataFrame({
    'Method': ['Linear Regressor'],
    'Parameters': [''],
    'MAE': [mae],
    'MSE': [mse],
    'RMSE': [rmse]
})

california_housing_val_results
```

	Method	Parameters	MAE	MSE	RMSE
0	Linear Regressor		0.533335	0.529748	0.727838

It is common to use the word “method” to refer to a predictive algorithm or pipeline.

Non-Linear Regression

Let us now try to fit a non-linear regressor. We will use polynomial regression with different polynomial degrees. To do so, we will perform an explicit polynomial expansion of the features using the `PolynomialFeatures` object. For convenience, we will define a function performing training and validation and returning both training and validation performance:

```
from sklearn.preprocessing import PolynomialFeatures
def trainval_polynomial(degree):
    pf = PolynomialFeatures(degree)
    # While the model does not have any learnable parameters, the "fit" method here
        is used to compute the output number of features
    pf.fit(X_train)
    X_train_poly = pf.transform(X_train)
    X_val_poly = pf.transform(X_val)

    polyreg = LinearRegression() # a Polynomial regressor is simply a linear
        regressor using polynomial features
    polyreg.fit(X_train_poly, y_train)

    y_poly_train_pred = polyreg.predict(X_train_poly)
    y_poly_val_pred = polyreg.predict(X_val_poly)

    mae_train = mean_absolute_error(y_train, y_poly_train_pred)
    mse_train = mean_squared_error(y_train, y_poly_train_pred)
    rmse_train = np.sqrt(mean_squared_error(y_train, y_poly_train_pred))

    mae_val = mean_absolute_error(y_val, y_poly_val_pred)
    mse_val = mean_squared_error(y_val, y_poly_val_pred)
    rmse_val = np.sqrt(mean_squared_error(y_val, y_poly_val_pred))

    return mae_train, mse_train, rmse_train, mae_val, mse_val, rmse_val
```

Let us now see what happens with different degrees:

```
for d in range(1,4):
    print("DEGREE: {} \n      {:>8s} {:>8s} {:>8s}\nTRAIN {:8.2f} {:8.2f} {:8.2f}
        \nVAL   {:8.2f} {:8.2f} {:8.2f}\n\n".format(d,"MAE", "MSE", "RMSE",
        *trainval_polynomial(d)))
```

```
DEGREE: 1
           MAE       MSE      RMSE
TRAIN     0.53      0.51      0.72
VAL       0.53      0.53      0.73


DEGREE: 2
           MAE       MSE      RMSE
TRAIN     0.46      0.42      0.65
VAL       0.48      0.91      0.95


DEGREE: 3
           MAE       MSE      RMSE
TRAIN     0.42      0.34      0.58
VAL      23.18 2102256.43   1449.92
```

Ridge Regularization

We can see that, as the polynomial gets larger, the effect of overfitting increases. We can try to reduce this effect with Ridge or Lasso regularization. We'll focus on degree 2 and try to apply ridge regression to it. Since Ridge regression relies on a parameter, we will try some values of the regularization parameter α (as it is called by sklearn). Let us define a function for convenience:

```
from sklearn.linear_model import Ridge
def trainval_polynomial_ridge(degree, alpha):
    pf = PolynomialFeatures(degree)
    # While the model does not have any learnable parameters, the "fit" method here
        is used to compute the output number of features
    pf.fit(X_train)
    X_train_poly = pf.transform(X_train)
    X_val_poly = pf.transform(X_val)

    polyreg = Ridge(alpha=alpha) # a Polynomial regressor is simply a linear
        regressor using polynomial features
    polyreg.fit(X_train_poly, y_train)

    y_poly_train_pred = polyreg.predict(X_train_poly)
    y_poly_val_pred = polyreg.predict(X_val_poly)

    mae_train = mean_absolute_error(y_train, y_poly_train_pred)
    mse_train = mean_squared_error(y_train, y_poly_train_pred)
    rmse_train = np.sqrt(mean_squared_error(y_train, y_poly_train_pred))

    mae_val = mean_absolute_error(y_val, y_poly_val_pred)
    mse_val = mean_squared_error(y_val, y_poly_val_pred)
    rmse_val = np.sqrt(mean_squared_error(y_val, y_poly_val_pred))

    return mae_train, mse_train, rmse_train, mae_val, mse_val, rmse_val
```

Let us now see the results for different values of α. $\alpha = 0$ means no regularization:

```
print("RIDGE, DEGREE: 2")
for alpha in [0,100,200,300,400]:
    print("Alpha: {:0.2f} \n      {:>8s} {:>8s} {:>8s}\nTRAIN {:8.2f} {:8.2f} {:8.2f}
        \nVAL   {:8.2f} {:8.2f} {:8.2f}\n\n".format(alpha,"MAE", "MSE", "RMSE",
        *trainval_polynomial_ridge(2,alpha)))
```

```
RIDGE, DEGREE: 2
Alpha: 0.00
           MAE       MSE      RMSE
TRAIN     0.46      0.42      0.65
VAL       0.48      0.91      0.96


Alpha: 100.00
           MAE       MSE      RMSE
TRAIN     0.47      0.43      0.66
VAL       0.48      0.54      0.74


Alpha: 200.00
           MAE       MSE      RMSE
TRAIN     0.48      0.44      0.67
VAL       0.49      0.51      0.72


Alpha: 300.00
           MAE       MSE      RMSE
TRAIN     0.49      0.46      0.68
VAL       0.50      0.50      0.71


Alpha: 400.00
           MAE       MSE      RMSE
TRAIN     0.50      0.47      0.68
VAL       0.51      0.51      0.71
```

We can see how, as alpha increases, the error on the training set increases, while the error on the test set decreases. For $\alpha = 300$ we obtained a slightly better result than our linear regressor:

```
california_housing_val_results
```

	Method	Parameters	MAE	MSE	RMSE
0	Linear Regressor		0.533335	0.529748	0.727838

Let us see if we can improve the results with a polynomial of degree 3:

```
print("RIDGE, DEGREE: 3")
for alpha in [0,1,10,20]:
    print("Alpha: {:0.2f} \n      {:>8s} {:>8s} {:>8s}\nTRAIN {:8.2f} {:8.2f} {:8.2f}
        \nVAL   {:8.2f} {:8.2f} {:8.2f}\n\n".format(alpha,"MAE", "MSE", "RMSE",
        *trainval_polynomial_ridge(3,alpha)))
```

```
RIDGE, DEGREE: 3
Alpha: 0.00
            MAE       MSE     RMSE
TRAIN      0.42      0.34     0.58
VAL       23.50 2162209.37  1470.45


Alpha: 1.00
            MAE       MSE     RMSE
TRAIN      0.42      0.34     0.58
VAL       15.59 934580.17   966.74


Alpha: 10.00
            MAE       MSE     RMSE
TRAIN      0.42      0.34     0.59
VAL        1.57   4867.65    69.77


Alpha: 20.00
            MAE       MSE     RMSE
TRAIN      0.42      0.35     0.59
VAL        1.78   6690.78    81.80
```

Let us add the results of Polynomial regression of degree 2 with and without regularization:

```
poly2 = trainval_polynomial_ridge(2,0)
poly2_ridge300 = trainval_polynomial_ridge(2,300)
california_housing_val_results = pd.concat([
    california_housing_val_results,
    pd.DataFrame({'Method':'Polynomial Regressor', 'Parameters': 'degree=2',
        'MAE':poly2[-3], 'MSE':poly2[-2], 'RMSE':poly2[-1]}, index=[1]),
    pd.DataFrame({'Method':'Polynomial Ridge Regressor', 'Parameters': 'degree=2,
        alpha=300', 'MAE':poly2_ridge300[-3], 'MSE':poly2_ridge300[-2],
        'RMSE':poly2_ridge300[-1]}, index=[2])
])
california_housing_val_results
```

	Method	Parameters	MAE	MSE	RMSE
0	Linear Regressor		0.533335	0.529748	0.727838
1	Polynomial Regressor	degree=2	0.480437	0.912971	0.955495
2	Polynomial Ridge Regressor	degree=2, alpha=300	0.499228	0.504155	0.710039

Lasso Regression

Let us now try the same with Lasso regression:

```
from sklearn.linear_model import Lasso
def trainval_polynomial_lasso(degree, alpha):
    pf = PolynomialFeatures(degree)
    # While the model does not have any learnable parameters, the "fit" method here
        is used to compute the output number of features
    pf.fit(X_train)
    X_train_poly = pf.transform(X_train)
    X_val_poly = pf.transform(X_val)

    polyreg = Lasso(alpha=alpha) # a Polynomial regressor is simply a linear
        regressor using polynomial features
    polyreg.fit(X_train_poly, y_train)

    y_poly_train_pred = polyreg.predict(X_train_poly)
    y_poly_val_pred = polyreg.predict(X_val_poly)

    mae_train = mean_absolute_error(y_train, y_poly_train_pred)
    mse_train = mean_squared_error(y_train, y_poly_train_pred)
    rmse_train = np.sqrt(mean_squared_error(y_train, y_poly_train_pred))

    mae_val = mean_absolute_error(y_val, y_poly_val_pred)
    mse_val = mean_squared_error(y_val, y_poly_val_pred)
    rmse_val = np.sqrt(mean_squared_error(y_val, y_poly_val_pred))

    return mae_train, mse_train, rmse_train, mae_val, mse_val, rmse_val
```

```
print("LSSO, DEGREE: 2")
for alpha in [0.02,0.03,0.04,0.05, 0.06]:
    print("Alpha: {:0.2f} \n      {:>8s} {:>8s} {:>8s}\nTRAIN {:8.2f} {:8.2f} {:8.2f}
        \nVAL   {:8.2f} {:8.2f} {:8.2f}\n\n".format(alpha,"MAE", "MSE", "RMSE",
        *trainval_polynomial_lasso(2,alpha)))
```

```
LSSO, DEGREE: 2
Alpha: 0.02
           MAE       MSE      RMSE
TRAIN     0.52      0.51      0.71
VAL       0.54      1.19      1.09


Alpha: 0.03
           MAE       MSE      RMSE
TRAIN     0.55      0.55      0.74
VAL       0.55      0.59      0.77


Alpha: 0.04
           MAE       MSE      RMSE
TRAIN     0.56      0.57      0.76
VAL       0.57      0.59      0.77


Alpha: 0.05
           MAE       MSE      RMSE
TRAIN     0.58      0.60      0.78
VAL       0.58      0.61      0.78


Alpha: 0.06
           MAE       MSE      RMSE
TRAIN     0.60      0.63      0.80
VAL       0.60      0.64      0.80
```

Lasso regression does not seem to improve results. Let us put the results obtained for $\alpha = 0.04$ to the dataframe:

```
poly2_lasso004 = trainval_polynomial_lasso(2,0.04)
california_housing_val_results = pd.concat([
    california_housing_val_results,
    pd.DataFrame({'Method':'Polynomial Lasso Regressor', 'Parameters': 'degree=2,
        alpha=0.04', 'MAE':poly2_lasso004[-3], 'MSE':poly2_lasso004[-2],
        'RMSE':poly2_lasso004[-1]}, index=[3])
])
california_housing_val_results
```

	Method	Parameters	MAE	MSE	RMSE
0	Linear Regressor		0.533335	0.529748	0.727838
1	Polynomial Regressor	degree=2	0.480437	0.912971	0.955495
2	Polynomial Ridge Regressor	degree=2, alpha=300	0.499228	0.504155	0.710039
3	Polynomial Lasso Regressor	degree=2, alpha=0.04	0.567318	0.590100	0.768180

Grid Search

Polynomial regression and ridge regression have parameters to optimize. We have so far optimized them manually. However, in practice, it is common to perform a grid search. This consists in defining a grid of possible values to try and train/validate many models, to finally choose the one with best performance.

This can be done manually as shown in the following example:

```
def grid_search(alphas=range(200,400,25), degrees=range(5)):
    best_mse = np.inf
    for a in alphas:
        for d in degrees:
            print(f"Evaluating a={a} d={d} MSE=", end='')
            results = trainval_polynomial_ridge(d,a)
            mse = results[-2]
            print(f"{mse:0.2f}")
            if mse<best_mse:
                best_mse = mse
                best_alpha = a
                best_degree = d

    return best_mse, best_alpha, best_degree
grid_search()
```

```
Evaluating a=200 d=0 MSE=1.37
Evaluating a=200 d=1 MSE=0.53
Evaluating a=200 d=2 MSE=0.51
Evaluating a=200 d=3 MSE=8161.95
Evaluating a=200 d=4 MSE=10884349.27
Evaluating a=225 d=0 MSE=1.37
Evaluating a=225 d=1 MSE=0.53
Evaluating a=225 d=2 MSE=0.51
Evaluating a=225 d=3 MSE=6710.18
Evaluating a=225 d=4 MSE=7244284.75
Evaluating a=250 d=0 MSE=1.37
Evaluating a=250 d=1 MSE=0.53
Evaluating a=250 d=2 MSE=0.51
Evaluating a=250 d=3 MSE=5542.92
Evaluating a=250 d=4 MSE=4880844.16
Evaluating a=275 d=0 MSE=1.37
Evaluating a=275 d=1 MSE=0.53
Evaluating a=275 d=2 MSE=0.50
Evaluating a=275 d=3 MSE=4598.99
Evaluating a=275 d=4 MSE=3313009.83
Evaluating a=300 d=0 MSE=1.37
Evaluating a=300 d=1 MSE=0.54
Evaluating a=300 d=2 MSE=0.50
Evaluating a=300 d=3 MSE=3831.00
Evaluating a=300 d=4 MSE=2255949.80
Evaluating a=325 d=0 MSE=1.37
Evaluating a=325 d=1 MSE=0.54
Evaluating a=325 d=2 MSE=0.50
Evaluating a=325 d=3 MSE=3202.40
Evaluating a=325 d=4 MSE=1534630.58
Evaluating a=350 d=0 MSE=1.37
Evaluating a=350 d=1 MSE=0.54
Evaluating a=350 d=2 MSE=0.51
Evaluating a=350 d=3 MSE=2684.99
Evaluating a=350 d=4 MSE=1038260.39
Evaluating a=375 d=0 MSE=1.37
Evaluating a=375 d=1 MSE=0.54
Evaluating a=375 d=2 MSE=0.51
Evaluating a=375 d=3 MSE=2256.89
Evaluating a=375 d=4 MSE=695246.02
```

```
(0.5041553563051261, 300, 2)
```

Testing a range of values, we found that best results are obtained with degree equal to 2 and $\alpha = 300$.

Scikit-Learn Pipelines

Often, a predictive model is obtained by stacking different components. For instance, in our example, a Polynomial regressor is obtained by following this pipeline:

- Data standardization;
- Polynomial feature expansion:
- Ridge regression.

While these steps can be carried out independently as seen before, scikit-learn offers the powerful interface of `Pipelines` to automate this process. A pipeline stacks different components together and makes it convenient to change some of elements of the pipeline or to optimize its parameters. Let us define a pipeline as the one discussed above:

```
from sklearn.pipeline import Pipeline
polynomial_regressor = Pipeline([
    ('scaler', StandardScaler()),
    ('polynomial_expansion', PolynomialFeatures()),
    ('ridge_regression', Ridge())
])
```

To take full advantage of the pipeline, we will re-load the dataset and avoid applying the standard scaler manually:

```
# let us load the data without passing as_frame=True
data = fetch_california_housing()
X = data.data # the features
y = data.target # the targets

# We'll do a 60:20:20 split
val_prop = 0.2
test_prop = 0.2

# We'll split the data in two steps - first let's create a test set and a combined
        trainval set
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=test_prop,
        random_state=42)

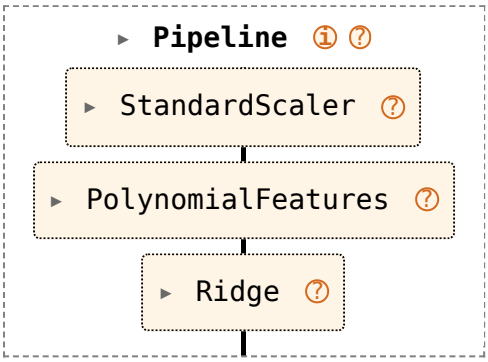
# We'll now split the combined trainval into train and val set with the chosen
        proportions
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval,
        test_size=test_prop/(1-test_prop), random_state=42)

# Let us check shapes and proportions
print(X_train.shape, X_val.shape, X_test.shape)
print(X_train.shape[0]/X.shape[0], X_val.shape[0]/X.shape[0],
        X_test.shape[0]/X.shape[0])
```

```
(12384, 8) (4128, 8) (4128, 8)
0.6 0.2 0.2
```

Our pipeline has two parameters that we need to set: the Ridge regressor's α and the degree of the polynomial. We can set these parameters as follows:

```
# We use the notation "object__parameter" to identify parameter names
polynomial_regressor.set_params(polynomial_expansion__degree=2,
        ridge_regression__alpha=300)
```



We can now fit and test the model as follows:

```
polynomial_regressor.fit(X_train, y_train)
y_val_pred = polynomial_regressor.predict(X_val)

mean_absolute_error(y_val, y_val_pred), mean_squared_error(y_val, y_val_pred),
        np.sqrt(mean_squared_error(y_val, y_val_pred))
```

```
(0.4992282794518713, 0.5041553563051261, np.float64(0.7100389822433174))
```

Grid Search with Cross Validation

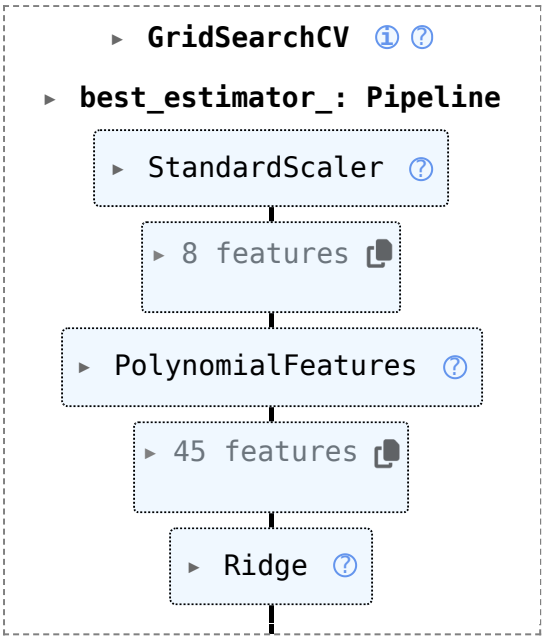
Scikit-learn offers a powerful interface to perform grid search with cross validation. In this case, rather than using a fixed training set, a K-Fold validation is performed for each parameter choice in order to find the best performing parameter combination. This is convenient when a validation set is not available. We will combine this approach with the pipelines to easily automate the search of optimal parameters:

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

gs = GridSearchCV(polynomial_regressor, param_grid=
        {'polynomial_expansion__degree':range(0,5),
        'ridge_regression__alpha':range(200,400,25)},
        scoring=make_scorer(mean_squared_error,greater_is_better=False))
```

We will now fit the model on the union of training and validation set:

```
gs.fit(X_trainval, y_trainval)
```



Let us check the best parameters:

```
gs.best_params_
```

```
{'polynomial_expansion__degree': 2, 'ridge_regression__alpha': 250}
```

These are similar to the ones found with our previous grid search. We can now fit the final model on the training set and evaluate on the validation set:

```
polynomial_regressor.set_params(**gs.best_params_)
polynomial_regressor.fit(X_train, y_train)
y_val_pred = polynomial_regressor.predict(X_val)
mae, mse, rmse = mean_absolute_error(y_val, y_val_pred), mean_squared_error(y_val,
        y_val_pred), np.sqrt(mean_squared_error(y_val, y_val_pred))

print(mae, mse, rmse)
```

```
0.4955697784505301 0.5062866264214417 0.7115382114977675
```

Let us add this result to our dataframe:

```
california_housing_val_results = pd.concat([
    california_housing_val_results,
    pd.DataFrame({'Method':'Cross-Validated Polynomial Ridge Regressor',
        'Parameters': 'degree=2, alpha=250', 'MAE':mae, 'MSE':mse, 'RMSE':rmse},
        index=[4])
])
california_housing_val_results
```

	Method	Parameters	MAE	MSE	RMSE
0	Linear Regressor		0.533335	0.529748	0.727838
1	Polynomial Regressor	degree=2	0.480437	0.912971	0.955495
2	Polynomial Ridge Regressor	degree=2, alpha=300	0.499228	0.504155	0.710039
3	Polynomial Lasso Regressor	degree=2, alpha=0.04	0.567318	0.590100	0.768180
4	Cross-Validated Polynomial Ridge Regressor	degree=2, alpha=250	0.495570	0.506287	0.711538

Other Regression Algorithms

Thanks to the unified interface of scikit-learn objects, we can easily train other algorithms even if we do not know how they work inside. Of course, to be able to optimize them in the most complex situations we will need to know how they work internally. The following code shows how to train a neural network (we will not see the algorithm formally):

```
from sklearn.neural_network import MLPRegressor

mlp_regressor = Pipeline([
    ('scaler', StandardScaler()),
    ('mlp_regression', MLPRegressor())
])

mlp_regressor.fit(X_train,y_train)
y_val_pred = mlp_regressor.predict(X_val)
mae, mse, rmse = mean_absolute_error(y_val, y_val_pred), mean_squared_error(y_val,
        y_val_pred), np.sqrt(mean_squared_error(y_val, y_val_pred))
mae, mse, rmse
```

```
(0.38390238923086073, 0.3077584657179924, np.float64(0.554759827058514))
```

```
california_housing_val_results = pd.concat([
    california_housing_val_results,
    pd.DataFrame({'Method':'Neural Network', 'Parameters': '', 'MAE':mae, 'MSE':mse,
        'RMSE':rmse}, index=[4])
])
california_housing_val_results
```

	Method	Parameters	MAE	MSE	RMSE
0	Linear Regressor		0.533335	0.529748	0.727838
1	Polynomial Regressor	degree=2	0.480437	0.912971	0.955495
2	Polynomial Ridge Regressor	degree=2, alpha=300	0.499228	0.504155	0.710039
3	Polynomial Lasso Regressor	degree=2, alpha=0.04	0.567318	0.590100	0.768180
4	Cross-Validated Polynomial Ridge Regressor	degree=2, alpha=250	0.495570	0.506287	0.711538
4	Neural Network		0.383902	0.307758	0.554760

Comparison and Model Selection

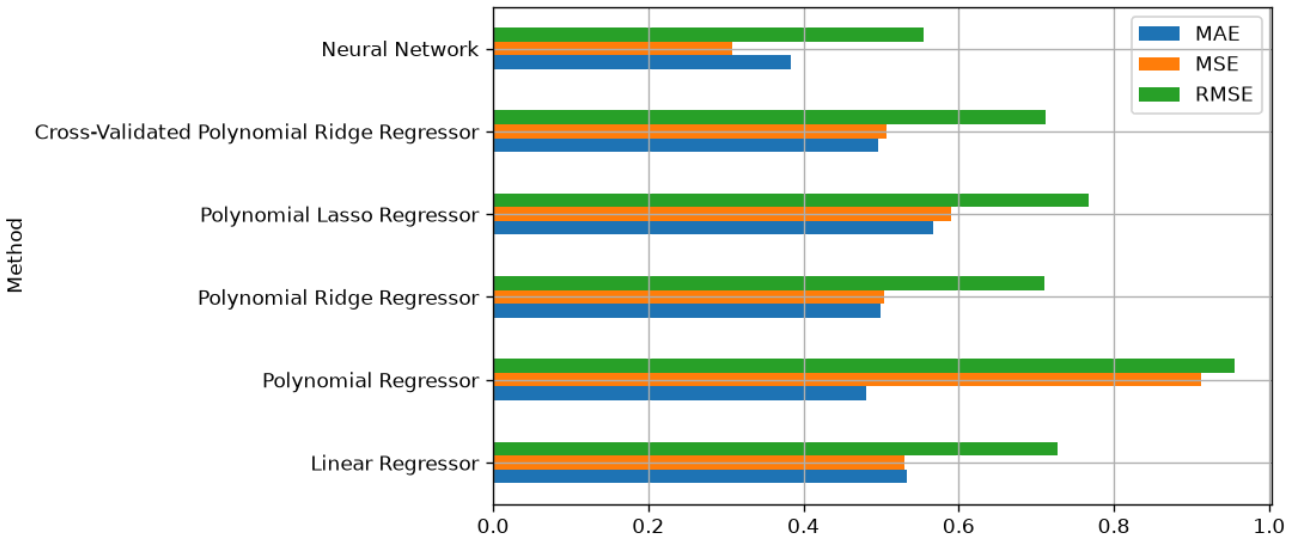
We can now compare the performance of the different models using the table:

```
california_housing_val_results
```


	Method	Parameters	MAE	MSE	RMSE
0	Linear Regressor		0.533335	0.529748	0.727838
1	Polynomial Regressor	degree=2	0.480437	0.912971	0.955495
2	Polynomial Ridge Regressor	degree=2, alpha=300	0.499228	0.504155	0.710039
3	Polynomial Lasso Regressor	degree=2, alpha=0.04	0.567318	0.590100	0.768180
4	Cross-Validated Polynomial Ridge Regressor	degree=2, alpha=250	0.495570	0.506287	0.711538
4	Neural Network		0.383902	0.307758	0.554760

Alternatively, we can visualize the results graphically:

```
california_housing_val_results.plot.barh(x='Method')
plt.grid()
plt.show()
```



From the analysis of the validation performance, it is clear that the neural network performs better. We can now compute the final performance on the test set:

```
y_test_pred = mlp_regressor.predict(X_test)
mae, mse, rmse = mean_absolute_error(y_test, y_test_pred), mean_squared_error(y_test,
        y_test_pred), np.sqrt(mean_squared_error(y_test, y_test_pred))
mae, mse, rmse
```

```
(0.3793900788199346, 0.3018830755077743, np.float64(0.5494388733132871))
```

Exercises

1. **Overfit on purpose.** Fit a polynomial of degree 15 to twenty points. Report the training and the test error, and plot the fitted curve. Describe what the model has learned.
2. **Ridge and lasso.** Fit both to the same dataset over a grid of λ. Plot the coefficient paths. At which λ does lasso set its first coefficient to exactly zero, and why can ridge never do that?
3. **Choosing λ.** Select λ by cross-validation, then report the test error. Explain why choosing λ on the test set would invalidate that number.
4. **Standardisation and penalties.** Fit ridge with and without standardising the features. Why does the penalty make the result depend on the units of each variable?
5. **Pipelines.** Rewrite your best model as a single `Pipeline` and run it through `GridSearchCV`. Explain which step of the pipeline is refitted inside each fold, and why that matters.

References

- Chapter 3 of [1]
- Parts of chapter 11 of [2]

[1] Heumann, Christian, and Michael Schomaker Shalabh. Introduction to statistics and data analysis. Springer International Publishing Switzerland, 2016.

[2] James, Gareth Gareth Michael. An introduction to statistical learning: with applications in Python, 2023.https://www.statlearning.com

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Classification, evaluation measures and KNN

Chapter resources

Slides (PDF) Open in Colab Practice quiz

Classification models are another broad class of predictive models. Following the Machine Learning terminology, it is common to talk about classification or regression "tasks". These are the two main classes of tasks belonging to the broader class of supervised learning.

Task Definition

Recall that a predictive model is a function defined as follows:

$$h : \mathcal{X} \rightarrow \mathcal{Y}$$

If in the case of regression $\mathcal{X} = \Re^n$ and $\mathcal{Y} = \Re^m$, in the case of classification, the output targets are discrete values **which are generally referred to as "classes"**. Without loss of generality, if there are M classes, then we define $\mathcal{Y} = \{1, \dots, M\}$. If the inputs are numerical vectors (it does not have to always be this way, but we can usually employ a representation function to map the input to a numerical vector), a classification model can be defined as:

$$h : \Re^n \rightarrow \{0, \dots, M-1\}$$

Also in this case, we will assume to have a set of data to train and evaluate our model

$$\mathrm{D} = \{(x_i, y_i)\}_{i=1}^{N}$$

Note that in this case $x_i \in \Re^n$ and $y_i \in \{0, \dots, M-1\}$. Here, the values y_i are generally called "**labels**", while the values $\hat{y} = h(x)$ predicted using the classifier h are called **predicted labels**.

We can find the optimal model h by minimizing the empirical risk. A possible loss function is the following one:

$$L(\hat{y}, y) = \begin{cases} 1 & \text{if } \hat{y} \neq y \\ 0 & \text{if } \hat{y} = y \end{cases}$$

Hence, the empirical risk will be the fraction of incorrect predictions:

$$R_{emp}(h) = \frac{1}{N} \sum_{i=1}^{N} L(\hat{y}_i, y_i) = \frac{\text{number of incorrect predictions}}{N}$$

The empirical risk computed as defined above is also known as **error rate**. It is a number comprised between 0 and 1 which can also be interpreted as a percentage.

The classifier can be learned by minimizing the empirical risk:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} R_{emp}(h)$$

Examples

Classifiers support decision making any time that observations need to be categorized in one of a predefined set of classes. Examples of such problems are:

- Detecting spam emails (**spam vs legitimate email classification**).
- Classifying social media posts as being about politics or something else (**politics vs non-politics classification**).
- Recognizing the object depicted in an image out of 1000 different objects (**object recognition**).

For example, we can define spam detection as follows:

- *Task*: given an e-mail, classify it as spam or non-spam.
- *Input example* $\mathbf{e}$: the text of the e-mail. This can be a sequence of characters of arbitrary length. We can use some **representation function** to map an email $\mathbf{e}$ to a vector of real numbers $\mathbf{x} \in \Re^n$. We also assume that a training set pairing the vectors $\mathbf{x}$ with labels $y \in \{0, 1\}$ is available.
- *Classifier*: a function $h : \Re^n \rightarrow \{0, 1\}$.
- *Output*: a predicted label $\widehat{y} \in \{0, 1\}$ indicating if the e-mail is legitimate or spam. Here we have a **binary classification task**, hence $M = 2$.

Evaluation Measures

As with the case of regression, we need to define evaluation measures. These will be useful to **guide training** (e.g., modify the parameters in order to improve the performance of the algorithm on the training set), **tune hyper-parameters** and to **finally assess that the algorithm works on unseen data** (the test set).

As in the case of regression, we will consider the set of **ground truth test labels**:

$$Y_{TE} = \left\{ y^{(i)} | \left(\mathbf{x}^{(i)}, y^{(i)} \right) \in TE \right\}_i$$

and the set of **predicted test labels** :

$$\widehat{Y}_{TE} = \left\{ h\left(\mathbf{x}^{(i)} \right) | \left(\mathbf{x}^{(i)}, y^{(i)} \right) \in TE \right\}_i$$

which will be used as inputs of our performance measures.

Accuracy

Accuracy is a very common performance measure. We define accuracy as **the percentage of test examples for which our algorithm has predicted the correct label**:

$$Accuracy\left(Y_{TE}, \widehat{Y}_{TE}\right) = \frac{\left|\left\{y^{(i)} : y^{(i)} = \widehat{y}^{(i)}\right\}\right|}{|Y_{TE}|}$$

For instance, if the test set contains 100 examples and for 70 of them we have predicted the correct label, then we will have:

$$Accuracy\left(Y_{TE}, \widehat{Y}_{TE}\right) = \frac{70}{100} = 0.7$$

Note that accuracy is always a number comprised between 0 and 1. We can see the accuracy as a percentage. For instance, in the example above we could say that we have an accuracy of 70%.

Error Rate

The error rate is closely associated to accuracy and defined as:

$$ErrorRate\left(Y_{TE}, \widehat{Y}_{TE}\right) = \frac{\left|\left\{y^{(i)} : y^{(i)} \neq \widehat{y}^{(i)}\right\}\right|}{|Y_{TE}|} = 1 - Accuracy\left(Y_{TE}, \widehat{Y}_{TE}\right)$$

Example - Imbalanced Dataset

To see the limits of accuracy, let us consider a dataset containing 10000 elements of two classes distributed as follows:

- 500 data points from class 0;
- 9500 data points from class 1.

Let us now consider a naïve classifier which always predicts class 1:

$$f\left(\mathbf{x}\right) = 1$$

Intuitively, we see that this classifier is not a good one, as it discards its input and just predicts the most frequent class. However, it can be easily seen that its accuracy is 0.95.

Error Types

The main limitation of the accuracy is that it counts the number of mistakes made by the algorithm, but it does not consider which types of mistakes it makes. In general, on each example, we can make two kinds of mistakes:

- **Type 1**: we classify an example as belonging to the considered class, but it does not. These kinds of misclassifications are often called **False Positives (FP)**.
- **Type 2**: we classify the example as not belonging to the considered class, but it does belong to it. These kinds of misclassifications are often called **False Negatives (FN)**.

We can also have two kinds of correct predictions:

- **True Positives (TP)**: these are elements from the considered class which have been classified as actually belonging to that class.
- **True Negatives (TN)**: these are elements which are not from the considered class and have been classified as actually not belonging to that class.

Confusion Matrix

To have a more complete view of how our classifier is performing, we can put these numbers in a table which we will call a **confusion matrix**:

CONFUSION MATRIX		PREDICTED LABELS	
		POSITIVE	NEGATIVE
TRUE LABELS	POSITIVE	TP	FN
	NEGATIVE	FP	TN

In the matrix, the rows indicate the true labels, whereas the columns indicate the predicted labels. A good confusion matrix has large numbers in the main diagonal (TP and TN) and low numbers in the rest of the matrix (where the errors are).

The accuracy can be recovered from the confusion matrix as follows:

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}$$

which consists in **summing the numbers on the diagonal and dividing by the sum of all numbers**. We can see the computation of the accuracy from the confusion matrix graphically as follows:

CONFUSION MATRIX		PREDICTED LABELS	
		POSITIVE	NEGATIVE
TRUE LABELS	POSITIVE	TP	FN
	NEGATIVE	FP	TN

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}$$

Example 1 – Spam Detector

Let us consider a spam detector which correctly detects 40 out of 50 spam emails, while it only recognizes 30 out of 50 legitimate emails. The confusion matrix associated to this classifier will be as follows:

CONFUSION MATRIX		PREDICTED LABELS	
		SPAM	LEGITIMATE
TRUE LABELS	SPAM	40	10
	LEGITIMATE	20	30

Its accuracy will be:

$$Accuracy = \frac{40 + 30}{40 + 10 + 20 + 30} = \frac{70}{100} = 0.7$$

Example 2 – Imbalanced Dataset

Let us now consider the example of our imbalanced dataset with 9500 data points of class 1 and 500 data points of class 0. The confusion matrix of the naïve classifier $f(\mathbf{x}) = 1$ will be:

CONFUSION MATRIX		PREDICTED LABELS	
		POSITIVE	NEGATIVE
TRUE LABELS	POSITIVE	9500	0
	NEGATIVE	500	0

If we compute the accuracy of this classifier, we will obtain a good performance:

$$Accuracy = \frac{9500}{9500 + 500} = 0.95$$

However, looking at the confusion matrix, **it is clear that something is wrong, and our classifier is not working well**.

Precision and Recall

The confusion matrix allows to understand if there is an issue with the classifier in the case of imbalanced data. However, it is still convenient to have scalar measures which can tell us something about how the classifier is doing. In practice, it is common to define two complementary measures: precision and recall.

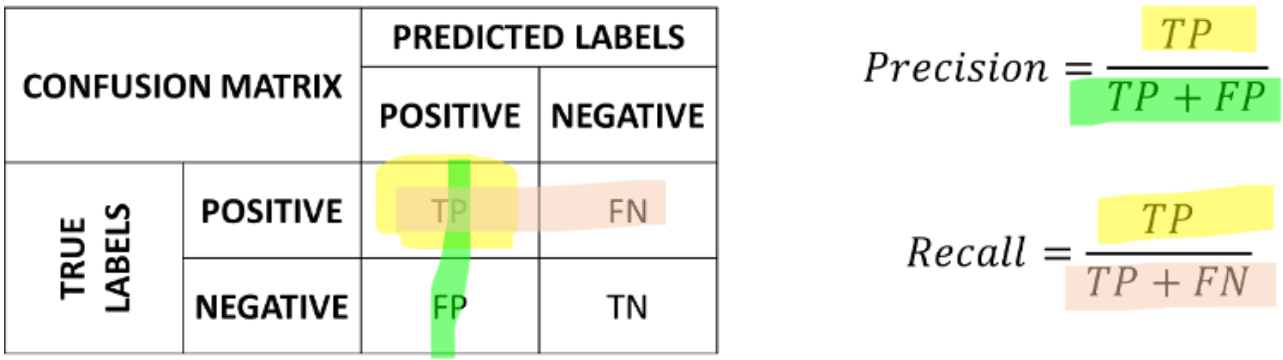
Precision measures **how many of the examples which have been classified as positives were actually positives** and is defined as follows:

$$Precision = \frac{TP}{TP + FP}$$

Recall measures **how many of the examples which are positives, have been correctly classified as positives** and is defined as follows:

$$Recall = \frac{TP}{TP + FN}$$

We can see graphically the computation of precision and recall as follows:



High Precision vs High Recall

These values capture different properties of the classifier. Depending on the application, we may want to have a higher precision or a higher recall. For example:

- Consider a **spam detector**: we may want to have a very **high precision**, even at the cost of a **low recall**. Indeed, we want to make sure that if we classify an e-mail as spam (and hence we filter it out), it is actually spam (hence a high precision). This is acceptable even if sometimes we let a spam email get through the filter (hence a low recall).

- Consider a **medical pre-screening** test which is used to assess if a patient is likely to have a given pathology. The test is cheap (e.g., a blood test) and can be made on a large sample of patients. If the test is positive, we then perform a more expensive but accurate test. In this case, we want to have a **high recall**. Indeed, if a patient has the pathology, we want to detect it and send the patient for the second, more accurate test (hence a high precision). This is acceptable even if sometimes we have false positives (hence a low precision). Indeed, if we wrongly detect a pathology, the second test will give the correct result.

Precision and recall can often have contrasting values (e.g., we can obtain a high precision but a low recall and vice versa), hence it is generally **necessary to look at both numbers together**.

Example – Spam Detector

Let us consider again the spam example, with the classifier obtaining this confusion matrix:

CONFUSION MATRIX		PREDICTED LABELS	
		SPAM	LEGITIMATE
TRUE LABELS	SPAM	40	10
	LEGITIMATE	20	30

From the confusion matrix, we see that:

- TP=40.
- FN=10.
- FP=20.
- TN=30.

We can compute the following precision and recall values:

- $Precision = \frac{TP}{TP+FP} = \frac{40}{40+20} = 0.67$;
- $Recall = \frac{TP}{TP+FN} = \frac{40}{40+10} = 0.8$.

Like the accuracy, precision and recall are telling us that the classifier is not perfect, and they disagree about *how*. Recall is the higher of the two: **most spam e-mails are caught**. Precision is the lower one: **one in three of the e-mails we filter out is legitimate.** For a spam filter this is the wrong way round. Filtering a legitimate e-mail is the expensive error here, so we would want a very high precision (if we flag an e-mail as spam, it had better be spam) **even at the cost of a lower recall** — which is exactly what this classifier fails to deliver. Reading the two numbers in the wrong order would have hidden that.

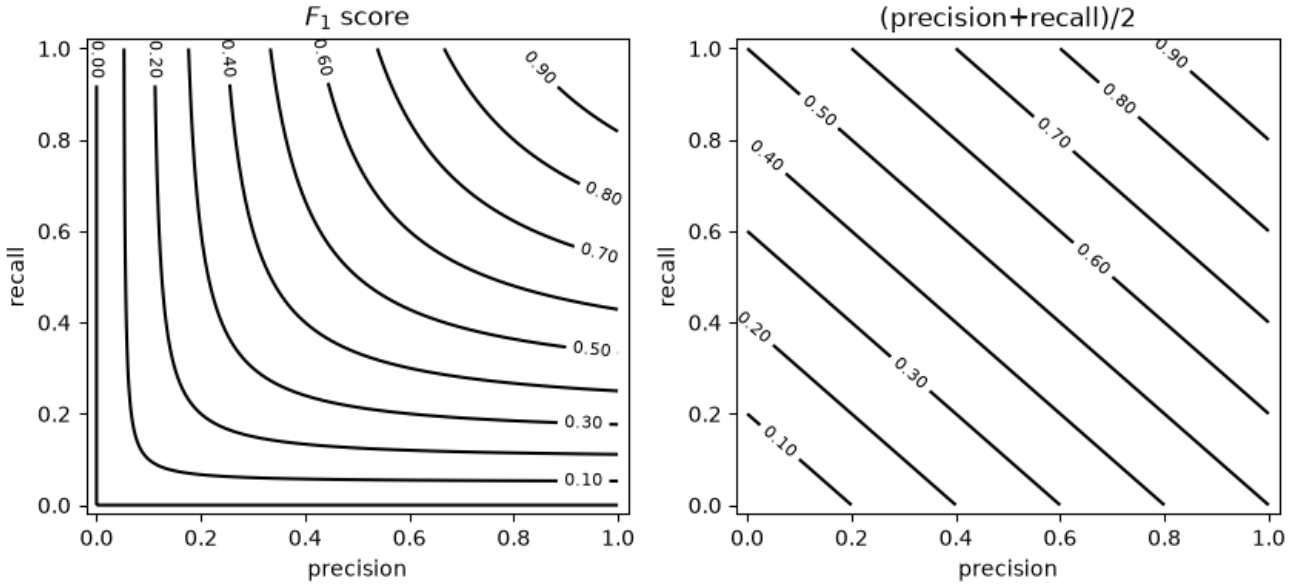
F_1 Score

We have seen that precision and recall describe different aspects of the classifier and hence it is often a good idea to look at them jointly. However, it is often convenient to have a single number which classifies both numbers.

The F_1 score allows to do exactly this, by computing the **harmonic mean** of precision and recall:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

We can note that, in order to obtain a large F_1 score, we need to obtain **both a large precision and a large recall.** This is a property of the harmonic mean, as it is illustrated in the following example which compares the arithmetic mean (precision/2+recall/2) to the harmonic mean (the F_1 score):



The example above shows the isocurves obtained by considering given precision and recall values. As can be noted, to obtain a large F_1 score, we need to have both a large precision and a large recall.

Example - Spam Detector

Let us consider again the spam example, with the classifier obtaining this confusion function:

CONFUSION MATRIX		PREDICTED LABELS	
		SPAM	LEGITIMATE
TRUE LABELS	SPAM	40	10
	LEGITIMATE	20	30

Starting from the precision and recall values previously computed:

- $Precision = \frac{TP}{TP+FP} = \frac{40}{40+20} = 0.67$;
- $Recall = \frac{TP}{TP+FN} = \frac{40}{40+10} = 0.8$.

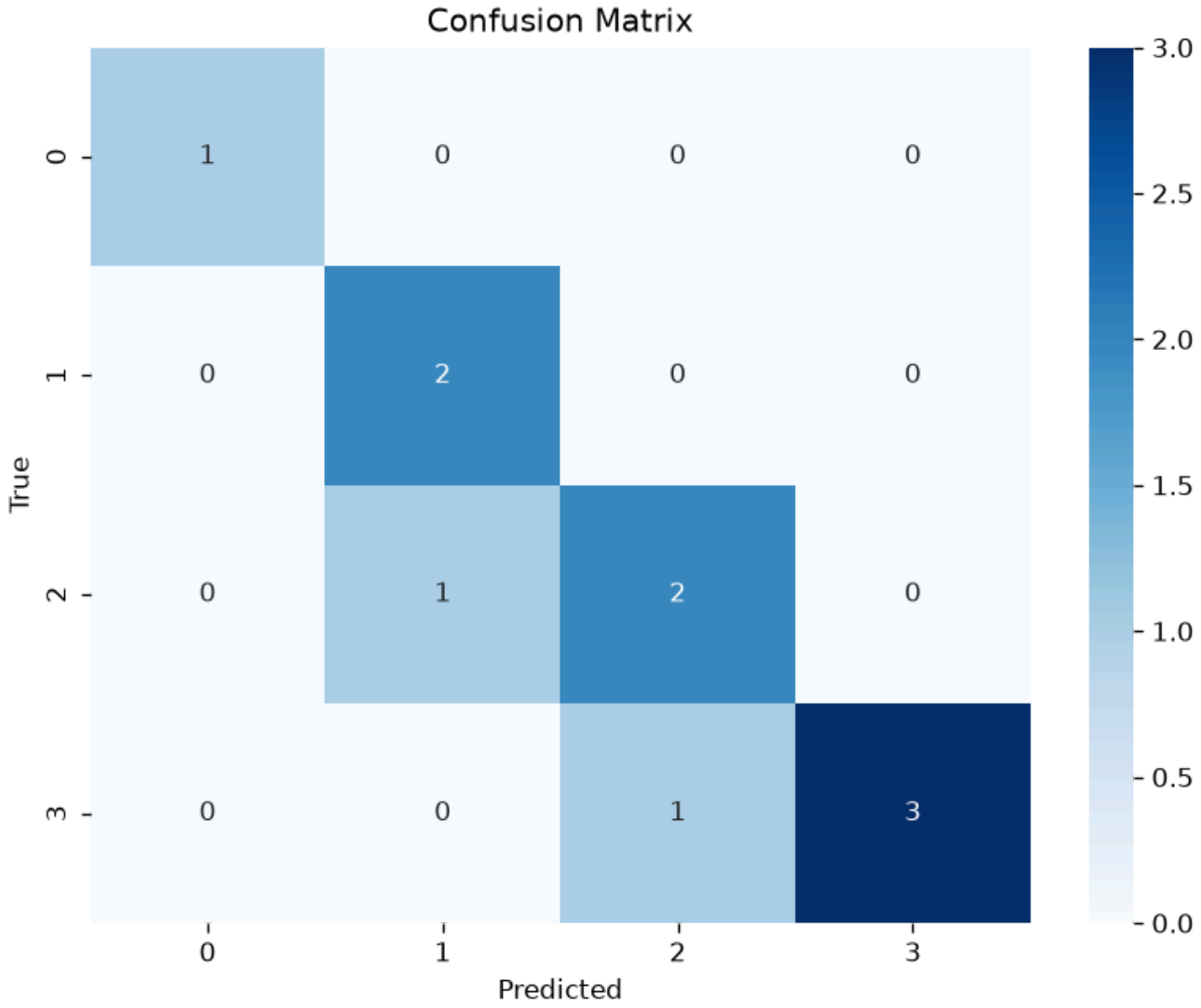
we can compute the F_1 score as:

$$F_1 = 2\frac{precision \cdot recall}{precision + recall} = 2 \cdot \frac{0.67 \cdot 0.8}{0.67 + 0.8} = 0.72$$

Note that, since the dataset is balanced, this value is not very different from the accuracy of 0.7.

Confusion Matrix for Multi-Class Classification

We have seen the confusion matrix in the case of binary classification. However, it should be noted that the confusion matrix generalizes to the case in which there are M classes. In that case, the confusion matrix is $M \times M$ and its general element C_{ij} indicates the number of elements **belonging to class i, which have been classified as belonging to class j**. An example of a confusion matrix in the case of three classes is the following:



Similar to the binary case, we expect to have large numbers on the diagonal and small number in all other cells. The concepts of precision, recall and F_1 score generalize considering a binary classification task for each of the classes (i.e., distinguishing each class from all the others). Hence, in the example shown above, we would have three F_1 scores:

```
[1.         0.8        0.66666667 0.85714286]
```

ROC (Receiver Operating Characteristic) Curve and Area Under the Curve (AUC) Measures

As previously mentioned, some binary classifiers output a probability or a confidence score and allow to obtain class predictions by thresholding on such probabilities or scores. When the confidence value is not a probability, it might not be easy to interpret it and find a good threshold. Even when the classifier outputs a probability, the optimal threshold might not be 0.5. For instance, we may want to build an intrusion detection system which is more or less sensitive to potential intrusions.

The ROC curve allows to evaluate the performance of a classifier independently from the threshold. Specifically, let

$$c(\mathbf{x})$$

be a function predicting a confidence value from an input vector $\mathbf{x}$. For instance, in the case of the logistic regressor:

$$c(\mathbf{x}) = \sigma(\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n)$$

We will define our classification function as:

$$h_\theta(\mathbf{x}) = [\sigma(\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n) \geq \theta]$$

where $[\cdot]$ denotes the Iverson brackets and $\theta \in \Re$ is a real-valued threshold.

Depending on the chosen value of the threshold, we will have a given number of true positives, true negatives, false positives and false negatives:

$$TP_\theta$$

$$TN_\theta$$

$$FP_\theta$$

$$FN_\theta$$

We will define the true positive rate (TPR) and false positive rate (FPR) as follows:

$$TPR_\theta = \frac{TP_\theta}{TP_\theta + FN_\theta}$$

$$FPR_\theta = \frac{FP_\theta}{FP_\theta + TN_\theta}$$

In practice:

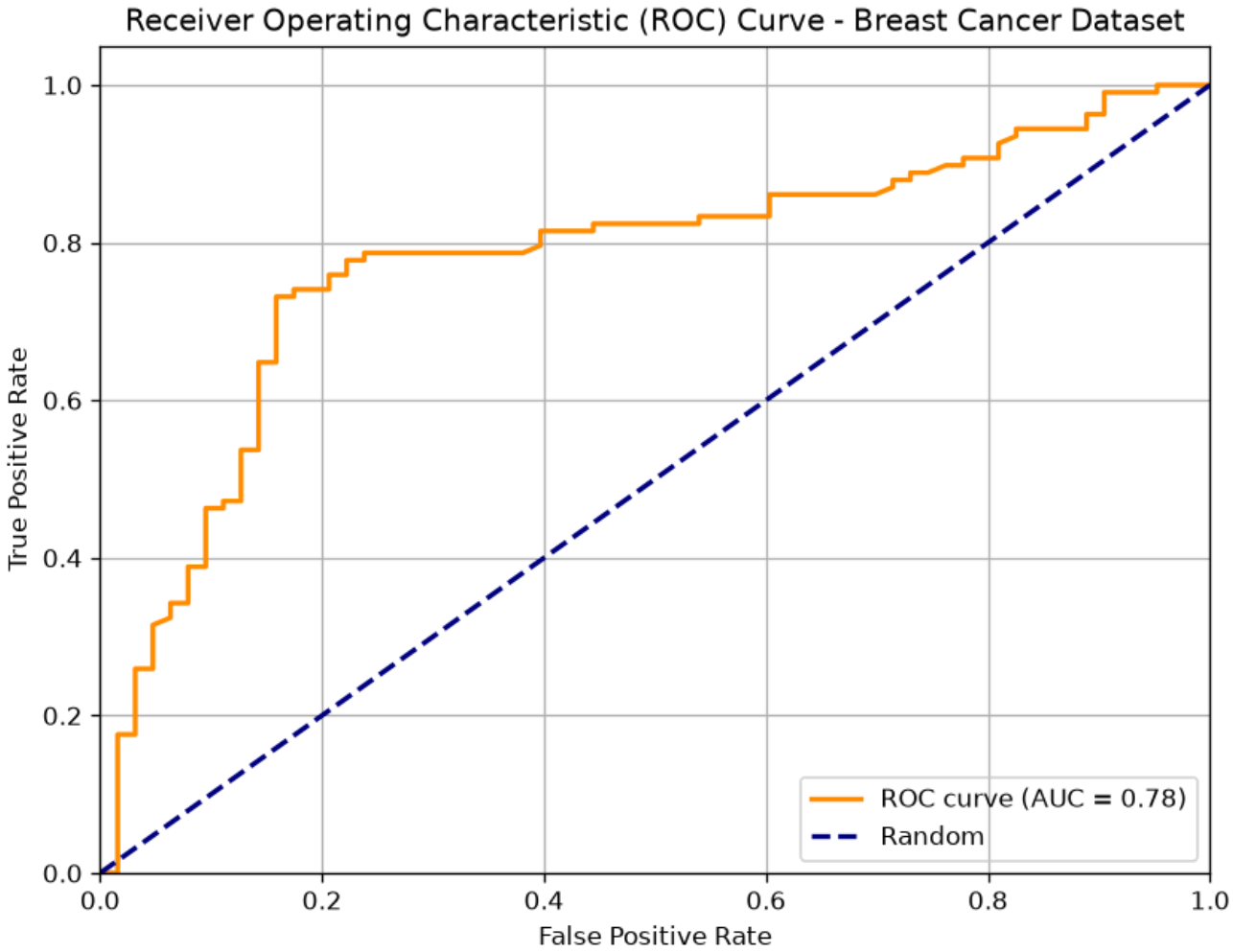
- The TPR is the fraction of true positives over all positive elements - **this is the same as the recall**;
- The FPR on the contrary is the fraction of false positive predictions over all negative elements.

We note that:

- If we pick **low threshold values, both the TPR and the FPR will be equal to 1**. Indeed, with a small enough threshold, all elements will be classified as positives and there will be no predicted negatives, so the TPR will be equal to 1. At the same time, the FPR will be 1 because we will have no true negatives;
- If we pick **high threshold values, both the TPR and the TNR will be zero**. Indeed, with a large enough threshold, all elements will be classified as negatives and there will be no positive predictions, so the TPR will be zero. At the same time, since all elements will be classified as negatives and there will be no false positive predictions, the FPR will be equal to 0;

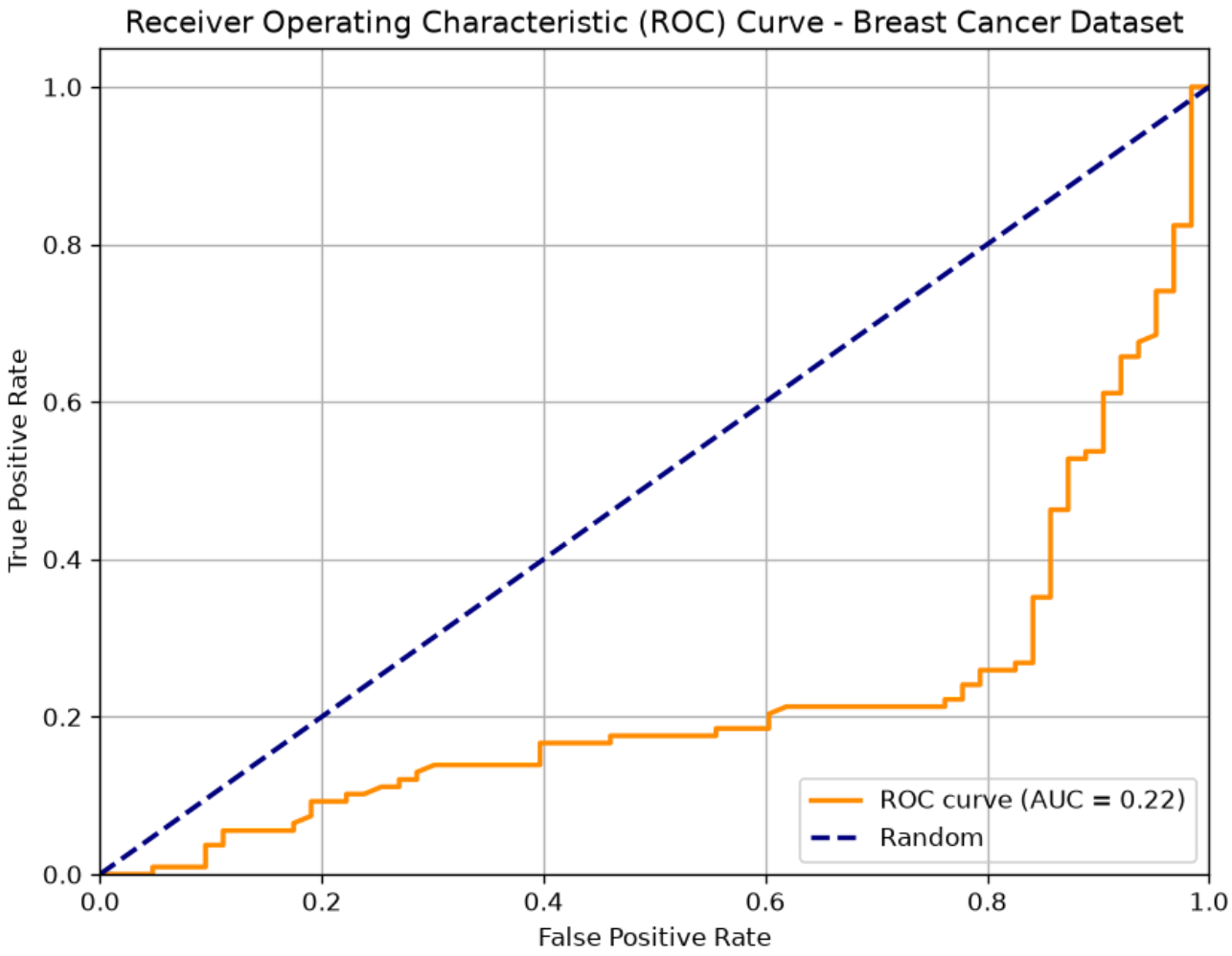
An ROC curve is obtained by picking a threshold value θ and plotting a 2D point (TPR_θ, TNR_θ). By varying the threshold θ, we obtain a curve which tell us what is the trade-off between TPR and TNR regardless of the threshold.

The following plot show an example of an ROC curve for a binary classifier on the breast cancer dataset:

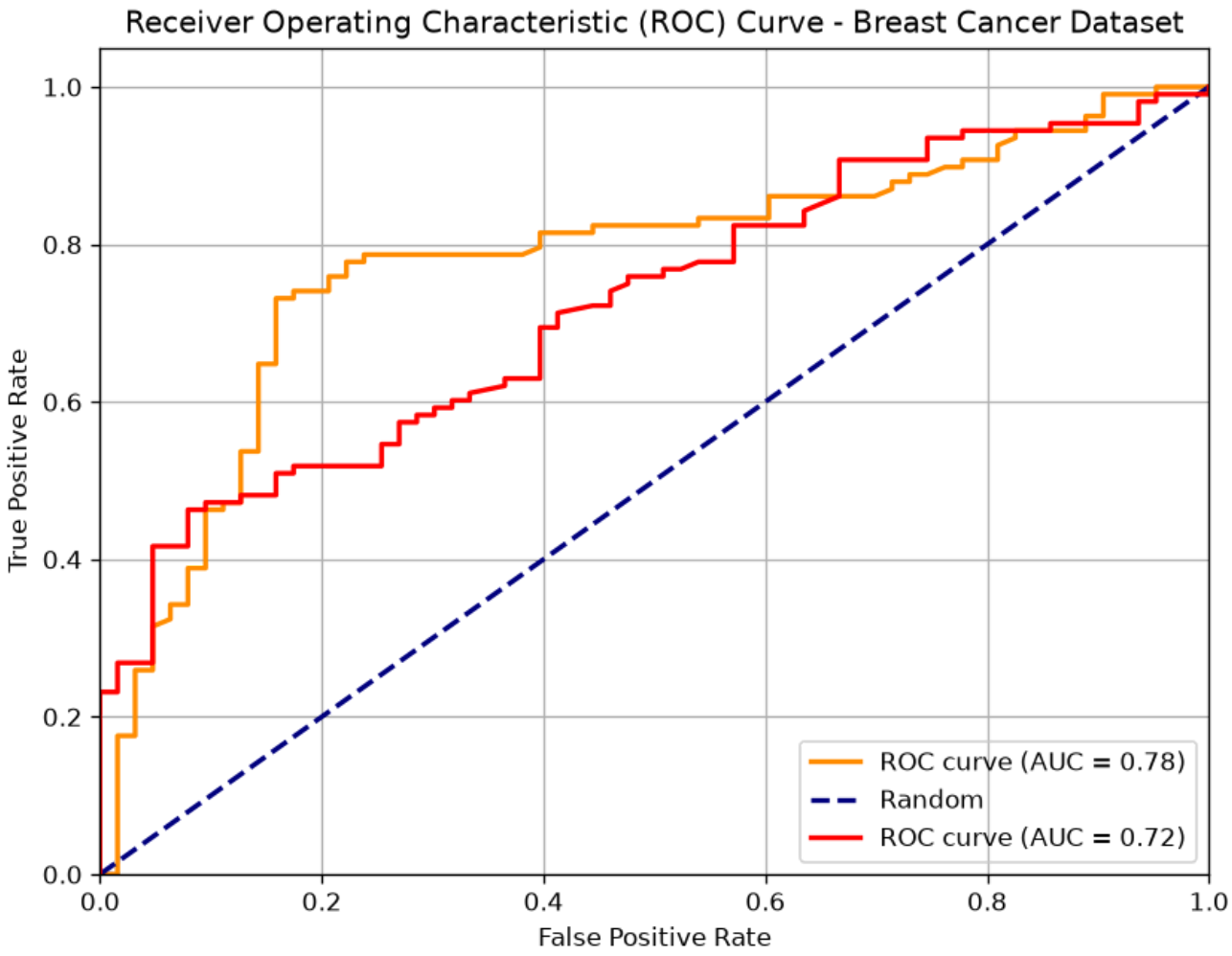


We know that the two points $(0, 0)$ and $(1, 1)$ belong to the curve. Ideally, starting from a low threshold (with TPR=FPR=1), as we move the threshold up enough, **we would expect the FPR to decrease (we are discarding false positives), while the TPR is still high (we are not discarding true positives)**. Hence the ideal curve should be a rectangular curve touching point $(0, 1)$. In practice, we can measure the area under the ROC curve to see how well the classifier is doing. This value is generally referred to as "AUC".

The dashed line indicates the performance of a random predictor. Note the the area under the curve identified by this dashed line will be equal to 0.5. Any curve which is systematically below this line indicates a classifier which is doing thresholding in the wrong way (i.e. we should invert the sign of the thresholding). For instance, this is the curve of the same classifier when we invert the sign of thresholding:



We can use ROC curves to compare two different classifiers as shown in the following:



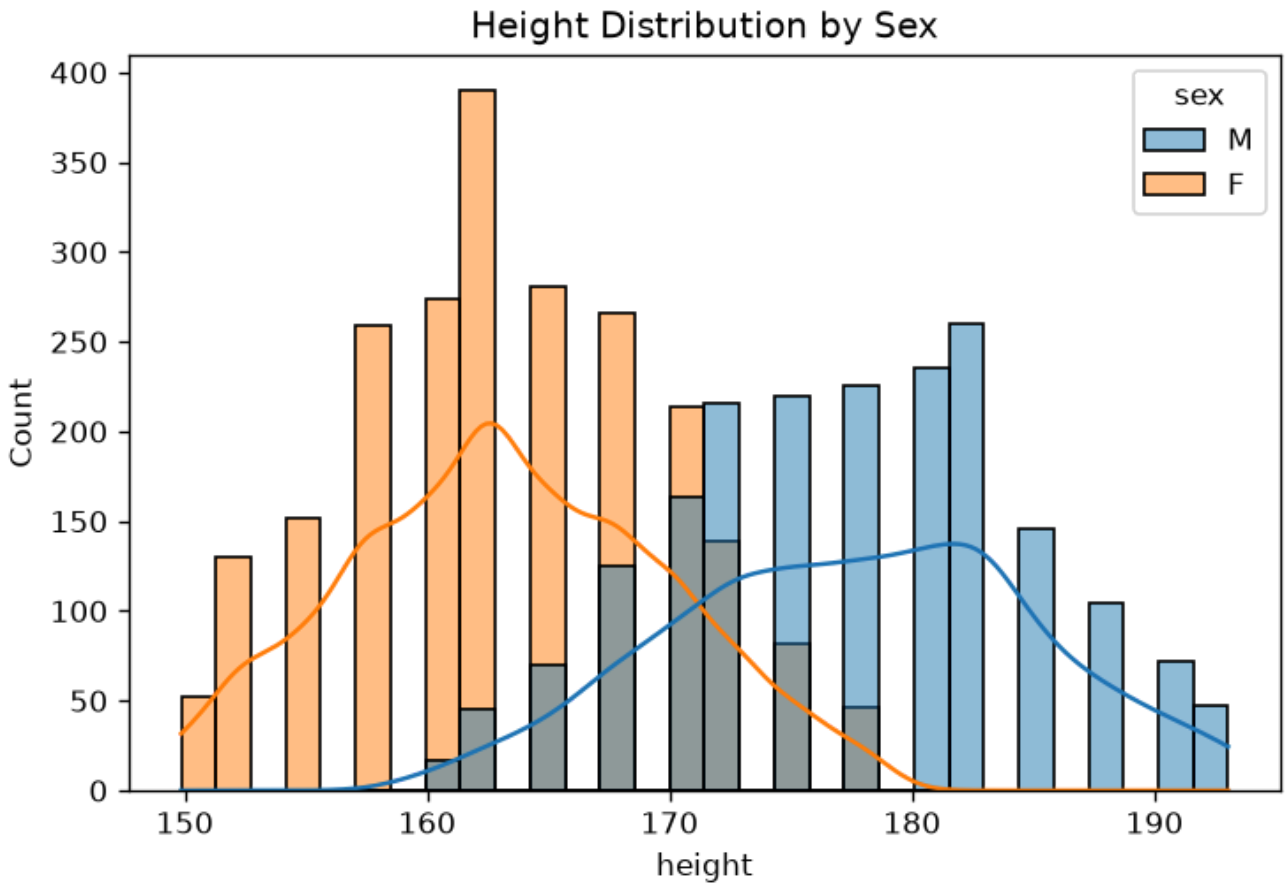
A Simple Classifier: The Threshold Model

Let's start with the simplest possible classifier. We'll use a dataset of heights and weights to predict a person's `sex` (Male or Female).

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, roc_curve, auc, classification_report

# Load the dataset
data = pd.read_csv(DATA + "height_weight.csv")

# Let's look at the height distributions
sns.histplot(data=data, x='height', hue='sex', kde=True, bins=30)
plt.title('Height Distribution by Sex')
plt.show()
```



The plot clearly shows that, on average, males are taller than females. This suggests we could build a simple classifier:

"If a person's `height` is above a certain *threshold*, we predict 'Male'."

Let's arbitrarily pick a threshold of **170 cm**.

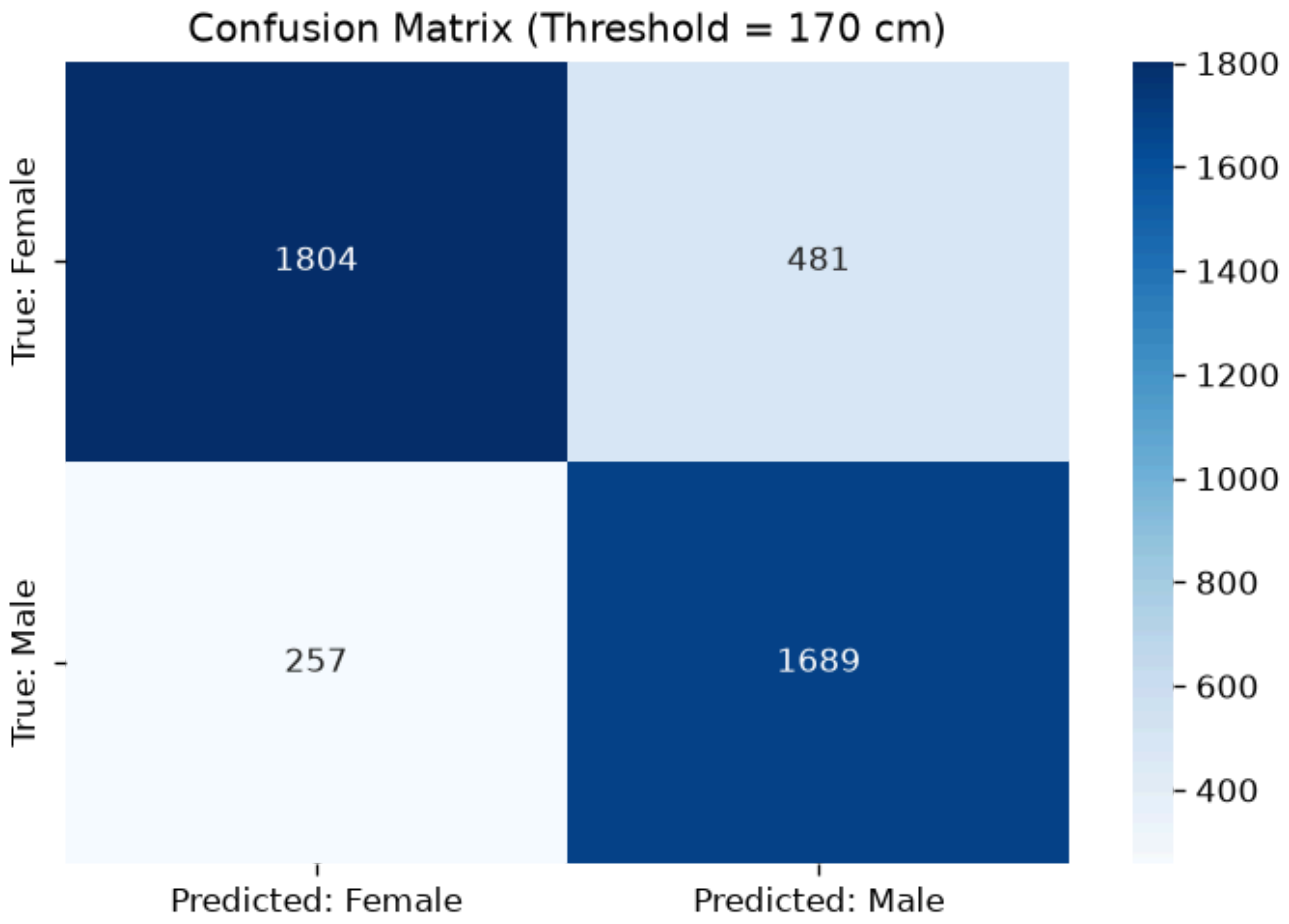
```python
# Our "model" is just a simple threshold
threshold = 170

# 1. Get the predicted labels (True for 'Male', False for 'Female')
# This is our model's prediction, ŷ
male_pred = (data['height'] ⩾ threshold)

# 2. Get the "Ground Truth" labels
# This is the true y
male_gt = (data['sex'] == 'M')

# Now, let's see how well our simple "model" did.
# We will build a Confusion Matrix.
cm = confusion_matrix(male_gt, male_pred)

# Let's visualize it
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted: Female', 'Predicted: Male'],
            yticklabels=['True: Female', 'True: Male'])
plt.title(f'Confusion Matrix (Threshold = {threshold} cm)')
plt.show()
print(classification_report(male_gt, male_pred))
```



```
              precision    recall  f1-score   support

       False       0.88      0.79      0.83      2285
        True       0.78      0.87      0.82      1946

    accuracy                           0.83      4231
   macro avg       0.83      0.83      0.83      4231
weighted avg       0.83      0.83      0.83      4231
```

Our model has only a single threshold parameter. As we can see, changing it affects performance. For instance, setting the threshold ot $180cm$ gives this confusion matrix:

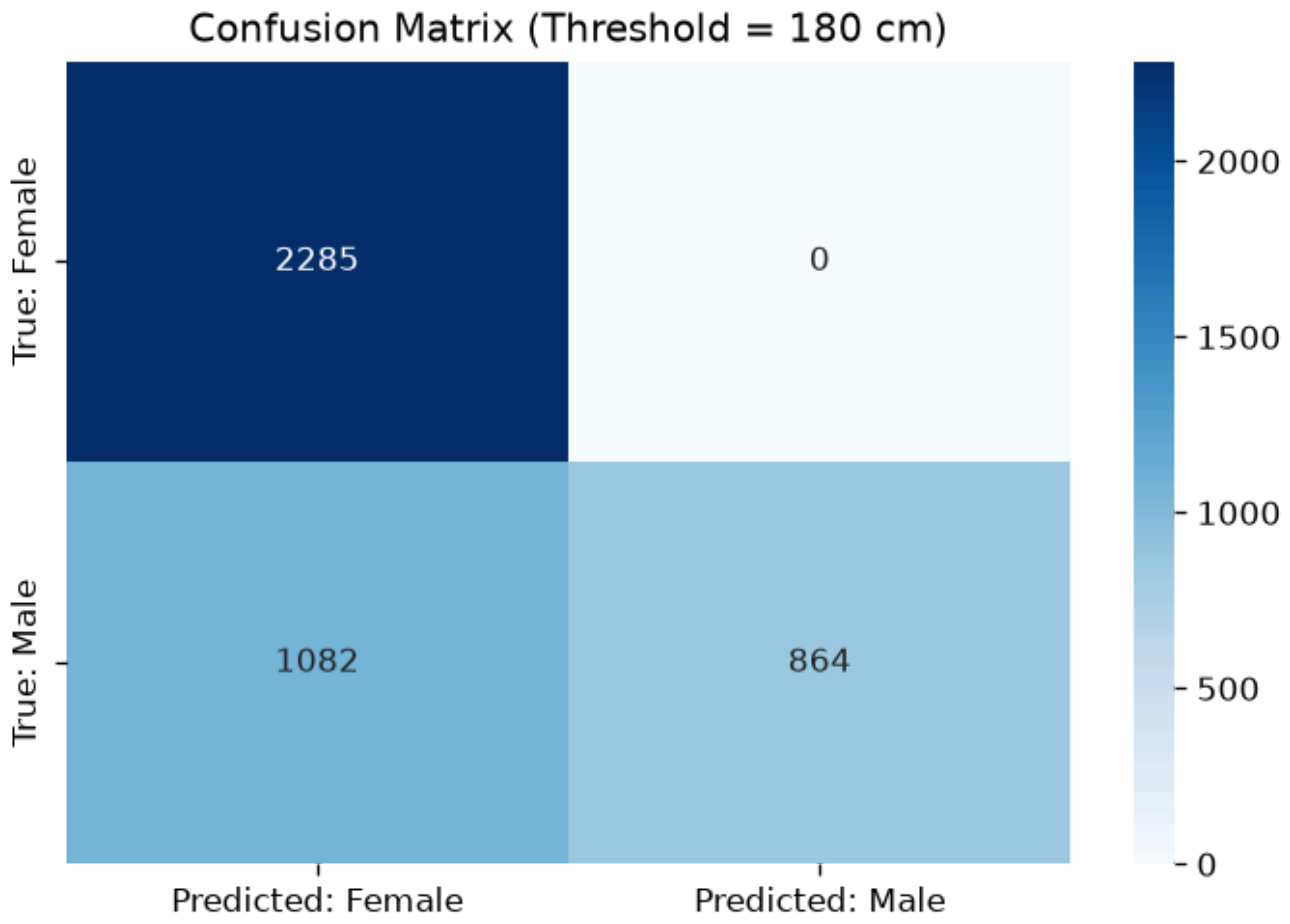
```python
# Our "model" is just a simple threshold
threshold = 180

# 1. Get the predicted labels (True for 'Male', False for 'Female')
# This is our model's prediction, ŷ
male_pred = (data['height'] ⩾ threshold)

# 2. Get the "Ground Truth" labels
# This is the true y
male_gt = (data['sex'] == 'M')

# Now, let's see how well our simple "model" did.
# We will build a Confusion Matrix.
cm = confusion_matrix(male_gt, male_pred)

# Let's visualize it
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted: Female', 'Predicted: Male'],
            yticklabels=['True: Female', 'True: Male'])
plt.title(f'Confusion Matrix (Threshold = {threshold} cm)')
plt.show()
print(classification_report(male_gt, male_pred))
```



	precision	recall	f1-score	support
False	0.68	1.00	0.81	2285
True	1.00	0.44	0.61	1946
accuracy			0.74	4231
macro avg	0.84	0.72	0.71	4231
weighted avg	0.83	0.74	0.72	4231

The model is “worse” (more elements off diagonal), but it also found many more females. In practice, moving the threshold changes the balance between true positives and false positives. In practice, we can compute the ROC curve to visualize this trade-off:

```
# The "ground truth" labels (1 for Male, 0 for Female)
male_gt = (data['sex'] == 'M')

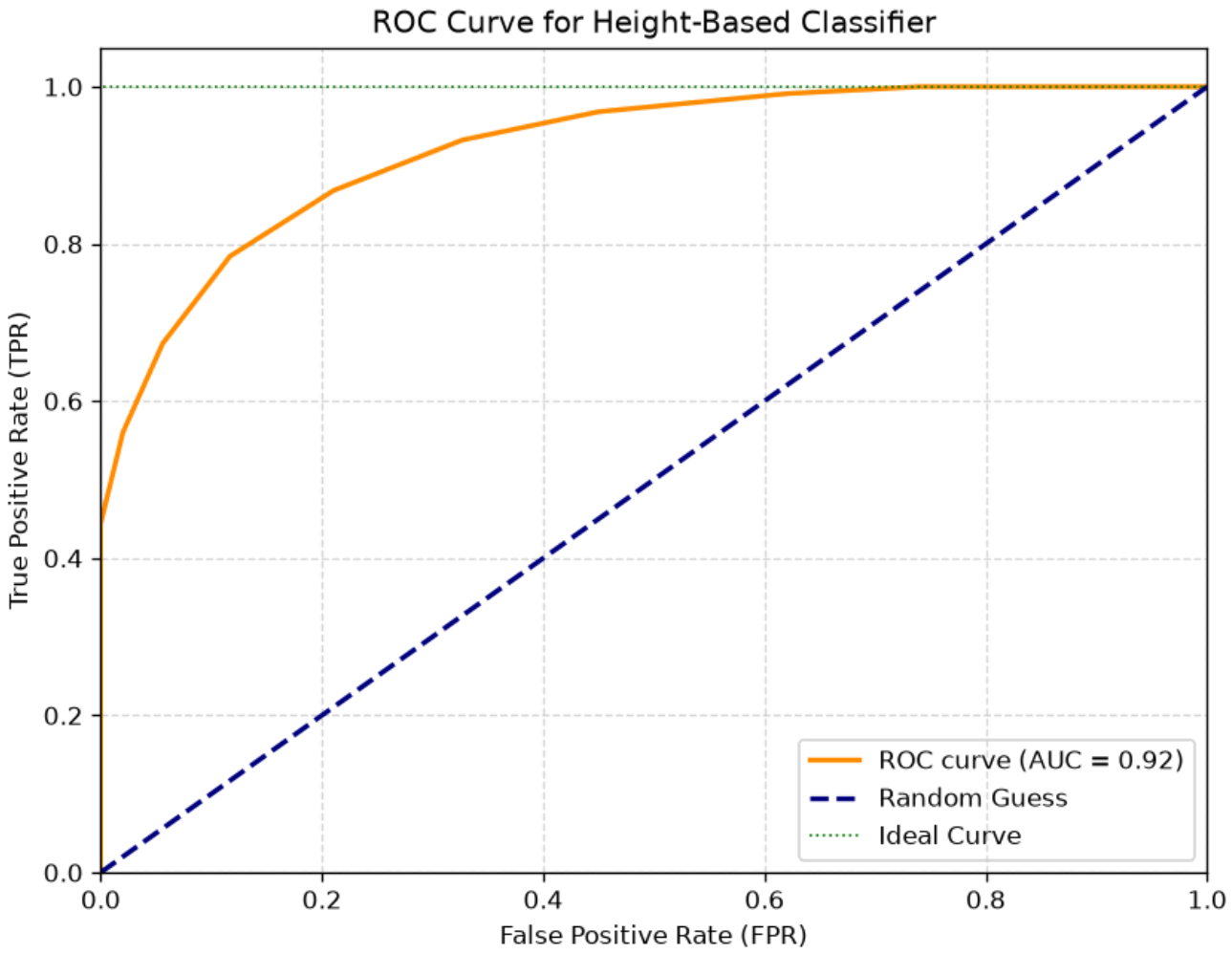
# The "score" our classifier uses. In this case, it's just the height itself.
# A higher score = more likely to be 'Male'.
scores = data['height']

# 1. Calculate the ROC curve
fpr, tpr, thresholds = roc_curve(male_gt, scores)

# 2. Calculate the Area Under the Curve (AUC)
roc_auc = auc(fpr, tpr)

# 3. Plot
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC =
        {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess') #
        The "worst" curve
plt.plot([0, 0, 1], [0, 1, 1], color='green', lw=1, linestyle=':', label='Ideal
        Curve') # The "best" curve

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve for Height-Based Classifier')
plt.legend(loc='lower right')
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```



Interpreting the ROC Curve & AUC

- **The Curve (Orange):** This shows our classifier. Each point on this line is a different threshold.
- **The "Random Guess" Line (Blue Dashed):** This is the performance of a coin flip (AUC = 0.5). Any model below this line is useless.
- **The "Ideal" Curve (Green Dotted):** A perfect classifier would go straight to the top-left corner (100% TPR, 0% FPR).

The **Area Under the Curve (AUC)** summarizes this entire plot into a single number from 0 to 1.

- **AUC = 1.0:** A perfect classifier.
- **AUC = 0.5:** A useless (random) classifier.
- **AUC < 0.5:** A classifier that is actively wrong (it's better if you reverse its predictions).

Our model, based *only on height*, has an **AUC of 0.92**, which is extremely good!

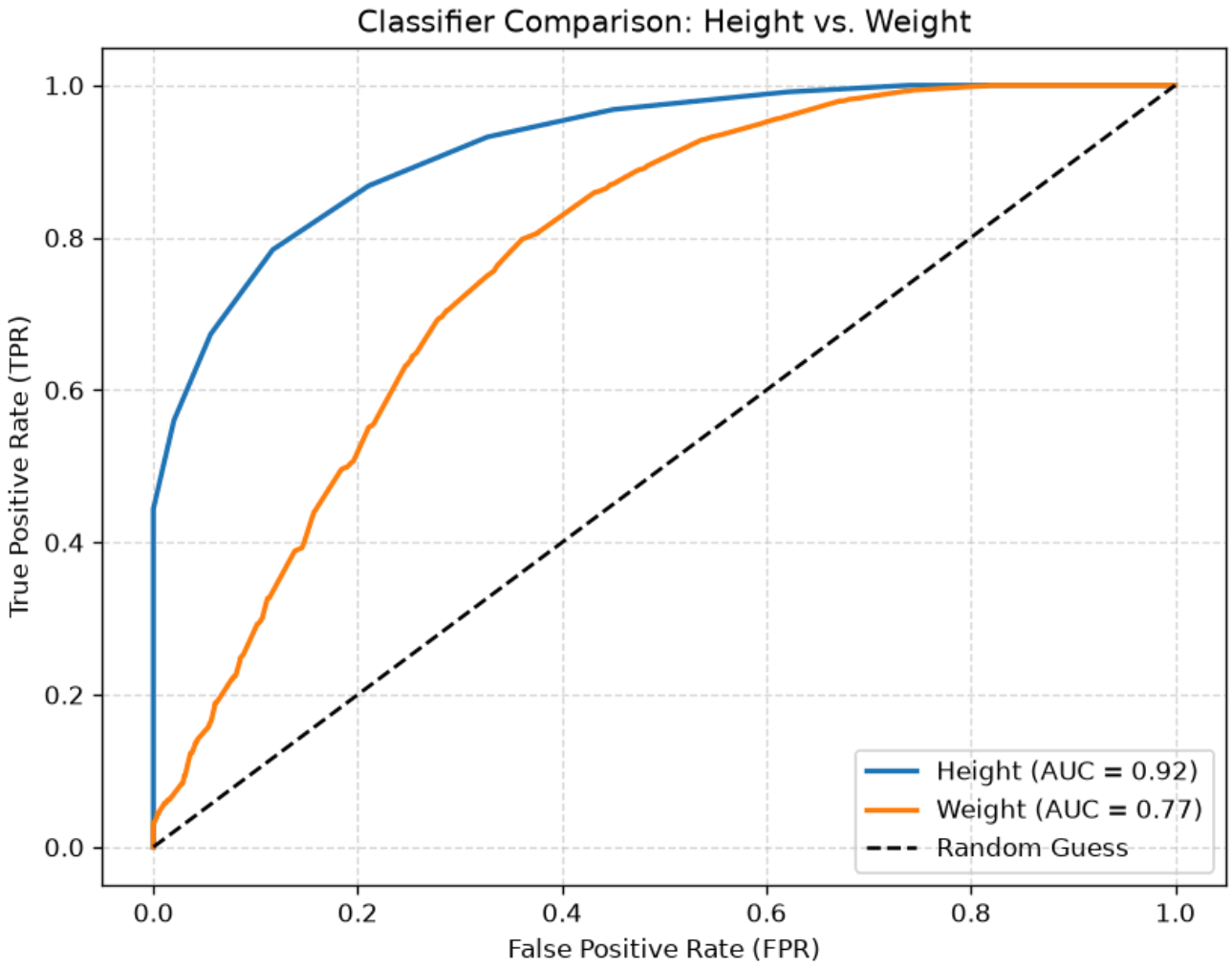
We can even use this method to compare classifiers. For example, is `height` or `weight` a better predictor of `sex`?

```python
# Calculate ROC for both features
fpr_h, tpr_h, _ = roc_curve(male_gt, data['height'])
auc_h = auc(fpr_h, tpr_h)

fpr_w, tpr_w, _ = roc_curve(male_gt, data['weight'])
auc_w = auc(fpr_w, tpr_w)

# Plot both
plt.figure(figsize=(8, 6))
plt.plot(fpr_h, tpr_h, lw=2, label=f'Height (AUC = {auc_h:.2f})')
plt.plot(fpr_w, tpr_w, lw=2, label=f'Weight (AUC = {auc_w:.2f})')
plt.plot([0, 1], [0, 1], 'k--', label='Random Guess')

plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Classifier Comparison: Height vs. Weight')
plt.legend(loc='lower right')
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```



The plot clearly shows that `height` (AUC = 0.92) is a significantly better predictor than `weight` (AUC = 0.88).

This entire analysis (Confusion Matrix, TPR, FPR, ROC, AUC) is our toolkit for evaluating *any* classifier.

How to Choose an Optimal Threshold

The ROC curve shows us *all* possible tradeoffs, but in the end, we must choose **one single threshold** for our model.

The "best" threshold depends on your goal:

- **High-Sensitivity (e.g., medical screening):** You might choose a threshold that gives a **high TPR (e.g., 0.95)**, even if it means a higher FPR.
- **High-Specificity (e.g., spam filter):** You might choose a threshold that gives a **very low FPR (e.g., 0.01)**, even if you miss some positives.

If you don't have a specific preference and want a "balanced" model, a common method is to find the threshold that is "closest" to the top-left corner (0 FPR, 1 TPR).

A simple way to do this is to find the threshold that maximizes the sum of **TPR** and **Specificity (1 - FPR)**. This is also known as **Youden's J statistic**.

$$J = TPR + (1 - FPR)$$

Let's find the threshold for our `height` classifier that maximizes this balanced score.

```python
# We already have fpr_h, tpr_h, and thresholds_h from the 'height' model

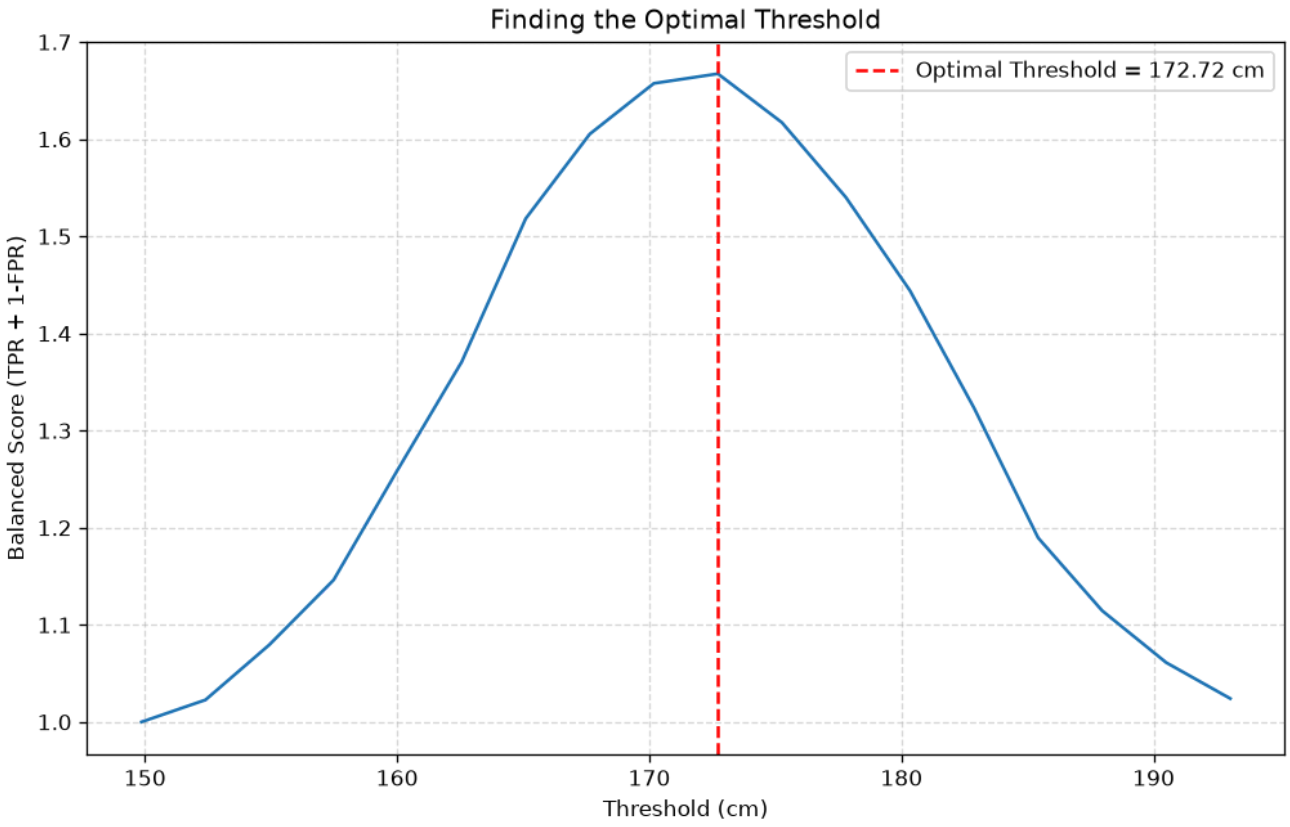
# 1. Calculate the balanced score (Youden's J) for each threshold
balanced_score = tpr_h + (1 - fpr_h)

# 2. Find the index of the threshold that gives the maximum score
best_index = np.argmax(balanced_score)
best_threshold = thresholds[best_index]
best_tpr = tpr_h[best_index]
best_fpr = fpr_h[best_index]

# 3. Plot the score vs. the thresholds
plt.figure(figsize=(10, 6))
# We slice [1:] to avoid a plotting artifact from the 'inf' threshold
plt.plot(thresholds[1:], balanced_score[1:])
plt.axvline(best_threshold, color='red', linestyle='--',
            label=f'Optimal Threshold = {best_threshold:.2f} cm')

plt.title('Finding the Optimal Threshold')
plt.xlabel('Threshold (cm)')
plt.ylabel('Balanced Score (TPR + 1-FPR)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()

print(f"Optimal threshold was found at {best_threshold:.2f} cm")
```



```
Optimal threshold was found at 172.72 cm
```

Final Evaluation at the Optimal Threshold

The plot shows us exactly how the performance changes. The peak of the curve (our optimal threshold) is at **169.57 cm**.

Let's see the final TPR and FPR if we use this single, balanced threshold for our classifier.

```
# We already found the best TPR/FPR from the argmax
print(f"--- Final Model at Optimal Threshold ({best_threshold:.2f} cm) ---")
print(f"True Positive Rate (TPR):  {best_tpr:.2f} (We find {best_tpr*100:.0f}% of all
        Males)")
print(f"False Positive Rate (FPR): {best_fpr:.2f} (We mislabel only
        {best_fpr*100:.0f}% of Females)")
print("-" * 25)

# You can also re-calculate it manually to double-check
print("\nVerifying with confusion_matrix:")
male_pred_optimal = (data['height'] ⩾ best_threshold)
male_gt = (data['sex'] == 'M')

tn, fp, fn, tp = confusion_matrix(male_gt, male_pred_optimal).ravel()
tpr_check = tp / (tp + fn)
fpr_check = fp / (fp + tn)

print(f"TPR Check: {tpr_check:.2f}")
print(f"FPR Check: {fpr_check:.2f}")

print("\nClassification Report at Optimal Threshold:")
print(classification_report(male_gt, male_pred_optimal))
```

```
--- Final Model at Optimal Threshold (172.72 cm) ---
True Positive Rate (TPR):  0.78 (We find 78% of all Males)
False Positive Rate (FPR): 0.12 (We mislabel only 12% of Females)
-------------------------

Verifying with confusion_matrix:
TPR Check: 0.78
FPR Check: 0.12

Classification Report at Optimal Threshold:
              precision    recall  f1-score   support

       False       0.83      0.88      0.85      2285
        True       0.85      0.78      0.82      1946

    accuracy                           0.84      4231
   macro avg       0.84      0.83      0.84      4231
weighted avg       0.84      0.84      0.84      4231
```

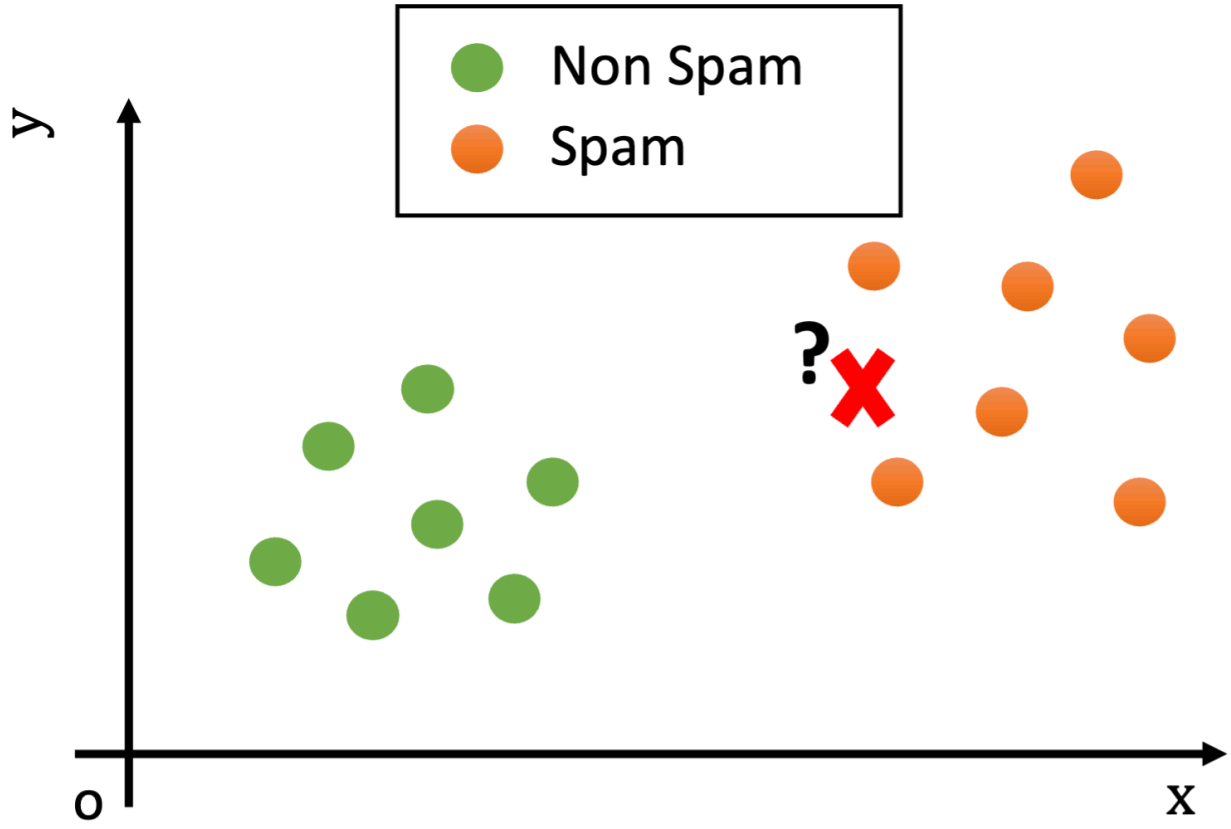
This method gives us a principled, balanced way to select a single threshold from our ROC curve. We find that a threshold of ~172.2 cm gives us a 78% TPR while only mislabeling 12% of females, which is a strong, balanced result.

K-Nearest Neighbor Classification

Threshold-based classification is very limited as it works when we only have one feature. We will now see the K-Nearest Neighbor classifier, an intuitive, non-parametric algorithm based on the concept of **similarity** between data points. We will first start by introducing the simple 1-NN algorithm (or nearest neighbor).

The Nearest Neighbor (or 1-NN) Classification Algorithm

Given an observation $\mathbf{x}$, the basic principle of Nearest Neighbor classification is to look at the nearest example in the training set and assign the same class. The idea is that nearby elements in the representation space will likely be similar. Consider the following example in which each data point is an email:



We could think of the x and y features as characteristics useful to determine if an e-mail is spam. For instance x may count the number of orthographic errors, while y may count the number of occurrences of given keywords such as “buy” or “viagra”.

Looking at the plot above, what would be the class of the new example (the red cross)?

We notice that, while the test example is not exactly equal to any training example, it is still reasonably similar to **some training examples** (i.e., there are points nearby in the Euclidean space).

Since the training and test sets have been collected in similar ways (e.g., both contain spam and legitimate e-mails), we can hypothesize that the *test example will be of the same class of similar examples belonging to the training set*.

For instance, if two documents have similar word frequencies, they are probably of the same class (e.g., if both contain the words “viagra” and “sales” many times, they probably belong to “spam” class).

We can measure how “similar” two examples $\mathbf{x}$ and $\mathbf{x}$’ are using a suitable distance function d:

$$d(\mathbf{x}, \mathbf{x}')$$

We expect similar examples to have a small distance. *A very common choice for d is the Euclidean distance*:

$$d\left(\mathbf{x},\mathbf{x}^{'}\right) = \left\|\mathbf{x}-\mathbf{x}^{'}\right\|_2 = \sqrt{\sum_{i=1}^{N}\left(x_i - x_i^{'}\right)^2}$$

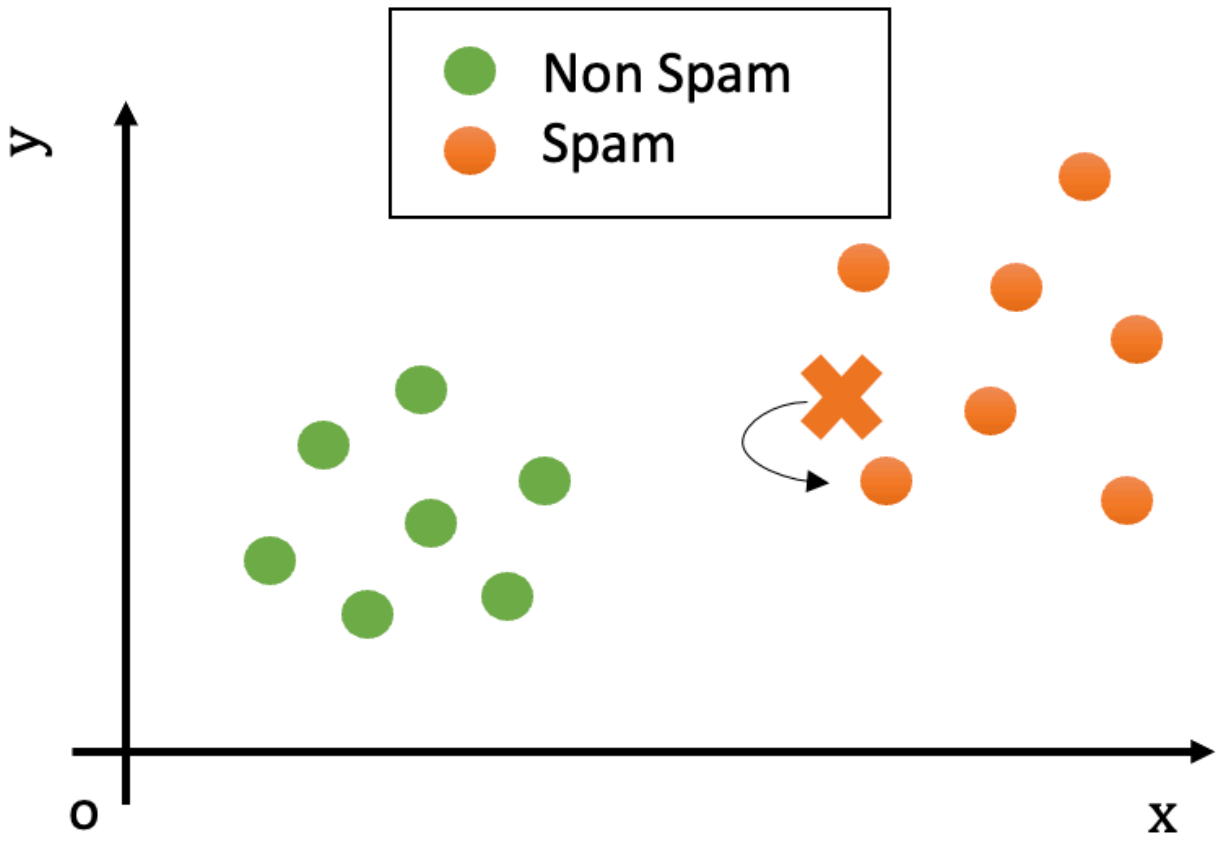
We can hence define a classifier which leverages our intuition **assigning to a test example $\mathbf{x}^{'}$ the class of the closest example in the training set**:

$$h\left(\mathbf{x}^{'}\right) = \arg_y \min \left\{d\left(\mathbf{x}^{'},\ \mathbf{x}\right) \mid (\mathbf{x},y) \in TR\right\}$$

This algorithm is referred to as the **nearest neighbor algorithm (or 1-NN as we will see in its generalization in a moment)**. In practice, given a test example $\mathbf{x}^{'}$, the nearest neighbor algorithm works in two steps:

- Find the element $\overline{\mathbf{x}}$ of the training set with the smallest distance with $\mathbf{x}'$ (i.e., such that $d(\overline{\mathbf{x}}, \mathbf{x}^{'})$ is minimum). The element $\overline{\mathbf{x}}$ is called the nearest neighbor of $\mathbf{x}^{'}$.
- Return the ground truth label associated to $\overline{\mathbf{x}}$, i.e., $y \mid (\overline{\mathbf{x}}, y) \in TR$.

In the example above, the test observation will be assigned the “spam” class, as shown in the following figure:



The K-Nearest Neighbour Classification Algorithm

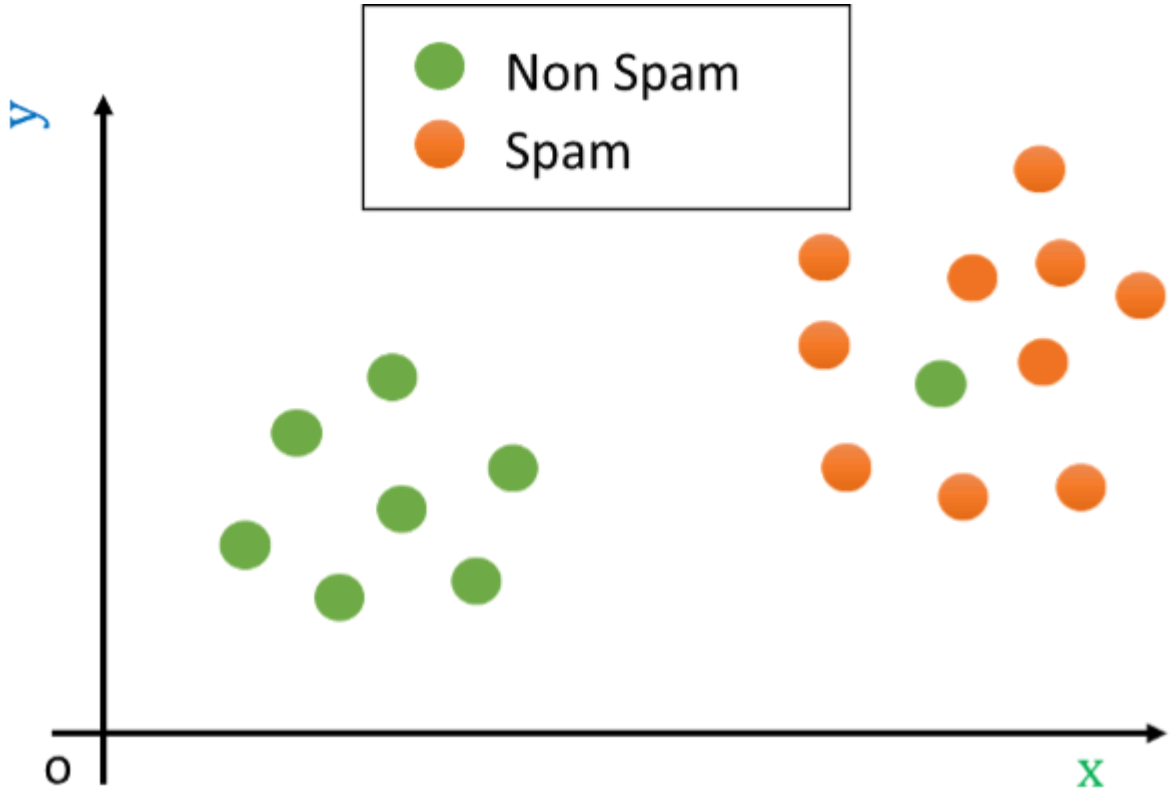
The Nearest Neighbor (or 1-NN) algorithm assumes that data points of the same class are close to each other in the representation space. This can be reasonably true when the representation space is

ideal for the classification task and the data is *clean and simple enough*. For instance, we expect similar documents to have similar word frequencies.

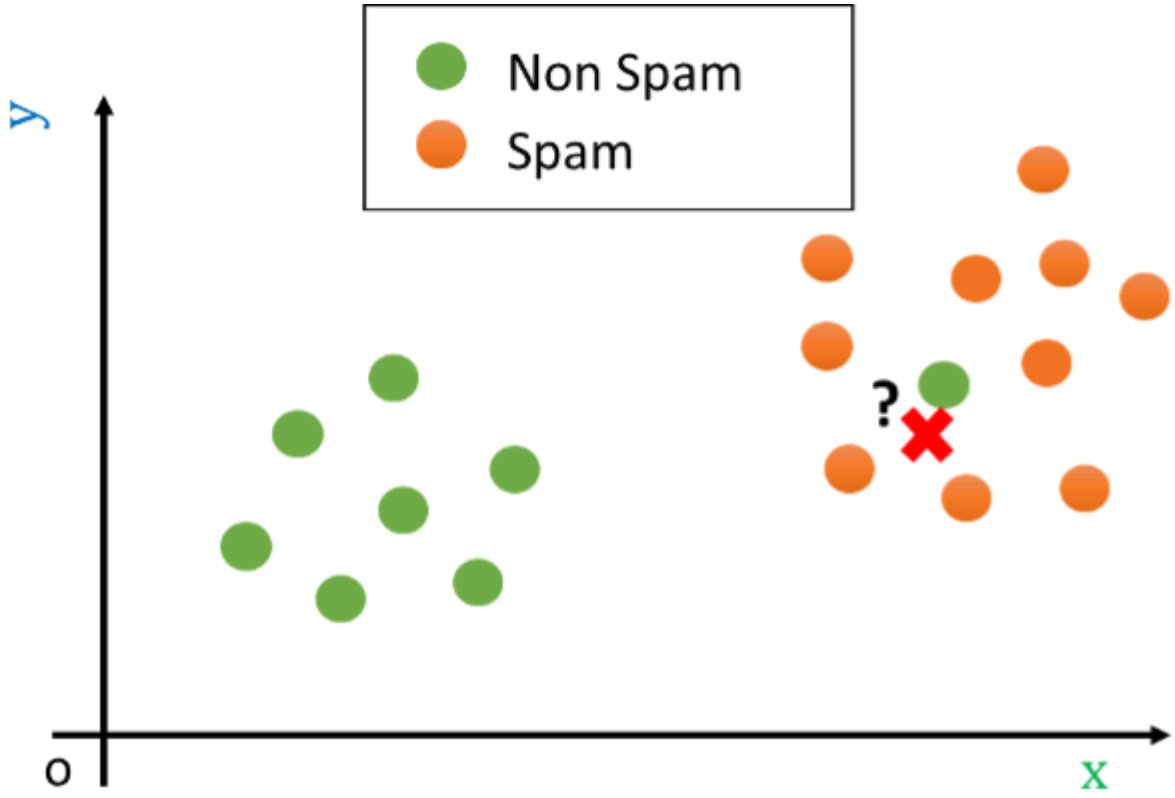
However, it is often common to have **'outliers'** in the training data, i.e., data points which do not closely follow the distribution of the other data points. This can be due to different factors:

- *The data may not be clean*: maybe an email has been wrongly classified as "spam" when it's actually not spam;
- *The data representation may not be ideal*: there could be legitimate email in which the word "viagra" is used and there are many orthographical errors. *Think of a legitimate email forwarding a spam email*. Our simple representation does not account for that, which leads to outliers.

Let us consider as an outlier a **legitimate e-mail containing the word 'viagra'**. This example can be seen graphically as follows:

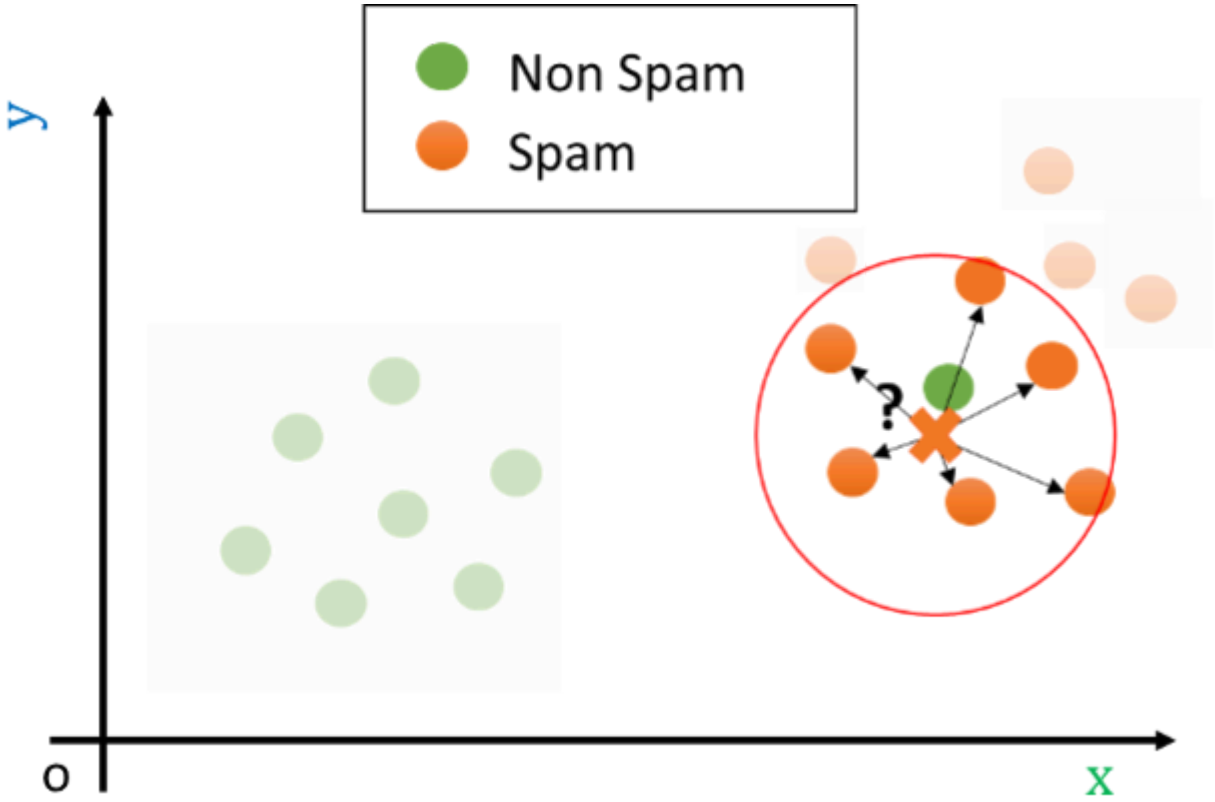


Let us now assume that we are presented with a test example which is shown as a red cross in the following figure:



In the example above, the nearest neighbor algorithm would classify the test example (the red cross) as “non spam” since the closest point is the green outlier, while it is clear that the example is most probably a “spam” e-mail. Indeed, while the closest example is “non spam”, all other examples nearby belong to the “spam” class.

Reasonably, in cases like this, **we should not look just at the closest point in space**, but instead, we should look at a **neighborhood** of the data point. Consider the following example:



If we look at a sufficiently large neighborhood, we find that most of the points in the neighborhood are actually spam! Hence, it is wiser to classify the data point as belonging to the spam class, rather than to the non-spam one.

In practice, setting an appropriate radius for the neighborhood is not easy. For instance, if the space is not uniformly dense (and usually it is not – as in the example above!), a given radius could lead to neighborhoods containing different numbers of elements. Indeed, in some cases, they may even include just zero elements. Hence, rather than considering a neighborhood of a given radius, we consider neighborhoods of the point containing **at most $\mathbf{K}$ elements**, where $\mathbf{K}$ **is a *hyper-parameter* of the algorithm**.

Similarly to what we have defined in the case of density estimation, given a point $\mathbf{x'}$, we will define the neighborhood of training points of size K centered at $\mathbf{x'}$ as follows:

$$N_K(\mathbf{x'}) = N(\mathbf{x'}, R_K(\mathbf{x'}))$$

where $N(x, r)$ denotes a neighborhood centered at x and with radius r, and:

$$R_K(\mathbf{x'}) = \sup\{r : |N(\mathbf{x'}, r) \setminus \{\mathbf{x'}\}| \leq K\}$$

Finally, we define the K-Nearest Neighbor Classification Algorithm (also called K-NN) as follows:

$$h\left(\mathbf{x'}\right) = mode\{y|\left(\mathbf{x}, y\right) \in N\left(\mathbf{x'}; TR,\ K\right)\}$$

Where $mode$ is the "statistical mode" function, which returns the **most frequent element of a set**.

$\mathbf{K}$ **is in practice an hyperparameter of the algorithm**. It can be set to some arbitrary value or optimized using a **validation set or cross-validation**.

We should note that this definition **generalizes the nearest neighbor algorithm defined before**. Indeed, a 1-NN is exactly the nearest neighbor classifier seen above.

The Curse of Dimensionality

The K-Nearest Neighbors algorithm is simple, intuitive, and powerful. However, it suffers from a famous and critical problem known as the **"Curse of Dimensionality."**

This "curse" refers to a set of problems that arise when your data has a very large number of features (or "dimensions").

The core idea is: **As dimensions increase, your data becomes incredibly sparse, and the concept of "distance" becomes meaningless.**

The "Empty Space" Problem (2D vs. 3D)

Let's visualize how the same number of data points become increasingly sparse as we add dimensions. We'll generate 50 random data points and plot them in 1, 2, and 3 dimensions. Imagine these points are uniformly distributed within a unit hypercube (from 0 to 1 on each axis).

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # Required for 3D plots

n_points = 50 # Let's use 50 points for better visibility

plt.figure(figsize=(18, 6))

# --- 1 Dimension ---
ax1 = plt.subplot(1, 3, 1)
data_1d = np.random.rand(n_points, 1) # 50 points, 1 feature
y_1d = np.zeros(n_points) # For plotting on a line

ax1.scatter(data_1d, y_1d, s=50, alpha=0.7, color='blue')
ax1.set_title(f'1D Space with {n_points} Points', fontsize=14)
ax1.set_xlabel('Feature 1')
ax1.set_yticks([]) # Hide y-axis ticks for a line plot
ax1.set_xlim(-0.1, 1.1)
ax1.set_ylim(-0.1, 0.1)
ax1.grid(True, linestyle='--', alpha=0.6)

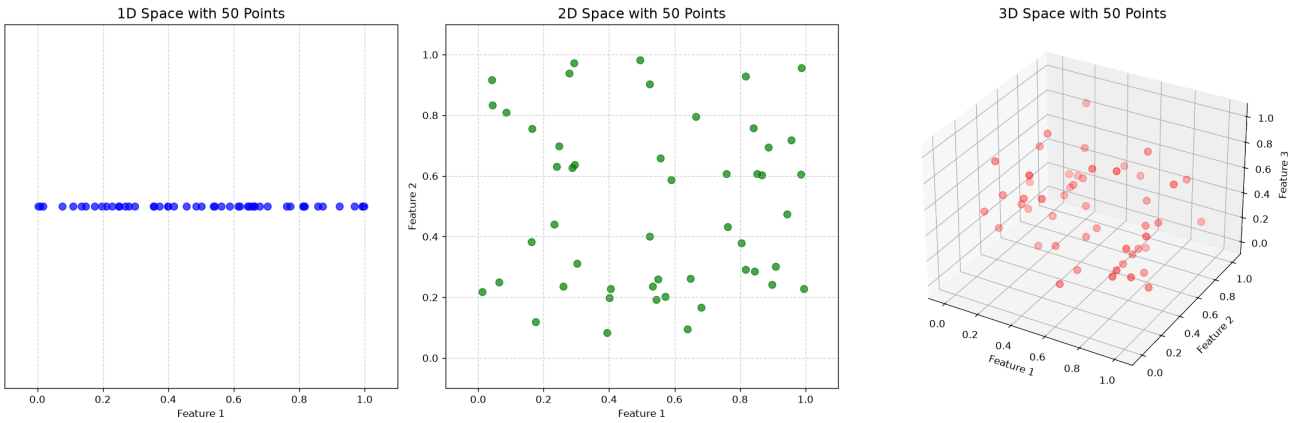

# --- 2 Dimensions ---
ax2 = plt.subplot(1, 3, 2)
data_2d = np.random.rand(n_points, 2) # 50 points, 2 features

ax2.scatter(data_2d[:, 0], data_2d[:, 1], s=50, alpha=0.7, color='green')
ax2.set_title(f'2D Space with {n_points} Points', fontsize=14)
ax2.set_xlabel('Feature 1')
ax2.set_ylabel('Feature 2')
ax2.set_xlim(-0.1, 1.1)
ax2.set_ylim(-0.1, 1.1)
ax2.grid(True, linestyle='--', alpha=0.6)


# --- 3 Dimensions ---
ax3 = plt.subplot(1, 3, 3, projection='3d')
data_3d = np.random.rand(n_points, 3) # 50 points, 3 features

ax3.scatter(data_3d[:, 0], data_3d[:, 1], data_3d[:, 2], s=50, alpha=0.7,
        color='red')
ax3.set_title(f'3D Space with {n_points} Points', fontsize=14)
ax3.set_xlabel('Feature 1')
ax3.set_ylabel('Feature 2')
ax3.set_zlabel('Feature 3')
ax3.set_xlim(-0.1, 1.1)
ax3.set_ylim(-0.1, 1.1)
ax3.set_zlim(-0.1, 1.1)
ax3.grid(True, linestyle='--', alpha=0.6)
```

```
plt.tight_layout()
plt.show()
```



Interpretation of the Plots:

- 1D: The 50 points are quite crowded on the line. You can easily find very close neighbors.
- 2D: The same 50 points are now spread out over a square. The space feels much emptier, and points are farther apart.
- 3D: In a cube, the 50 points are truly sparse. It's hard to even see individual points, and the space between them is vast.

Now, imagine doing this for dozens, hundreds, or even thousands of dimensions. The space becomes astronomically large, and your few data points are like individual dust particles lost in an empty galaxy.

The Impact on KNN

This creates two massive problems for KNN, which relies *entirely* on the idea of "nearness":

1. **Sparsity:** In high dimensions, your "nearest neighbor" might still be incredibly far away. Is it *really* "similar" to you if it's so far? Its "vote" is probably not reliable.
2. **Distance Becomes Meaningless:** This is the more critical point. In high-dimensional space, a strange mathematical property emerges: the distance to your *nearest* point and the distance to your *farthest* point become almost the same.

When all points are "equally far away" from each other, the concept of a "nearest neighbor" becomes random and meaningless. The algorithm's foundation breaks down, and its predictions become unstable.

The plot below illustrates this:

```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.spatial.distance import pdist

# 1. Define our parameters
n_points = 100
# We will test a range of dimensions, from 2 up to 1000
dimensions_to_test = range(2, 5001, 100)

# Store the results
min_distances = []
max_distances = []
relative_spread = []

# 2. Run the simulation
for d in dimensions_to_test:
    # Generate random points in a d-dimensional unit hypercube
    points = np.random.rand(n_points, d)

    # Calculate all pairwise distances
    all_distances = pdist(points)

    # Find the min and max
    min_d = np.min(all_distances)
    max_d = np.max(all_distances)

    # Store the results
    min_distances.append(min_d)
    max_distances.append(max_d)
    # This is the key metric: how "different" is max from min, relative to max?
    relative_spread.append((max_d - min_d) / max_d)

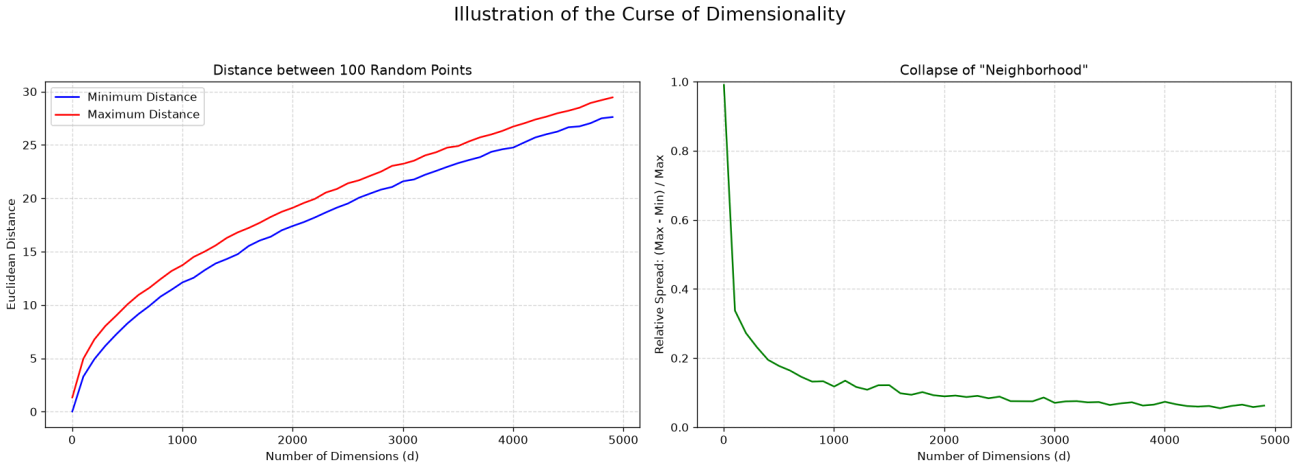
# --- 3. Plot the Results ---
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))
fig.suptitle('Illustration of the Curse of Dimensionality', fontsize=16)

# Plot 1: The actual min and max distances
ax1.plot(dimensions_to_test, min_distances, 'b-', label='Minimum Distance')
ax1.plot(dimensions_to_test, max_distances, 'r-', label='Maximum Distance')
ax1.set_xlabel('Number of Dimensions (d)')
ax1.set_ylabel('Euclidean Distance')
ax1.set_title('Distance between 100 Random Points')
ax1.legend()
ax1.grid(True, linestyle='--', alpha=0.5)

# Plot 2: The relative spread
ax2.plot(dimensions_to_test, relative_spread, 'g-')
ax2.set_xlabel('Number of Dimensions (d)')
```

```
ax2.set_ylabel('Relative Spread: (Max - Min) / Max')
ax2.set_title('Collapse of "Neighborhood"')
ax2.grid(True, linestyle='--', alpha=0.5)
ax2.set_ylim(0, 1) # Force y-axis to be 0 to 1 (or 100%)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



The plot on the left shows how both the minimum and maximum distance increase with the number of dimensions. The plot on the right computes the average relative spread between the maximum and minimum distance for each point. The realtive spread is computed as:

$$\text{Relative Spread} = \frac{\max - \min}{\max}$$

The plot shows that relative spread collapses to almost zero in high dimensions.

Takeaways

- KNN works extremely well for **low-dimensional data** (e.g., our 2-feature or 4-feature Iris example).
- It is almost always a **poor choice** for **high-dimensional data** (like images, which can have 10,000+ features, or text data with 5,000+ features) unless you use dimensionality reduction first (like PCA, as we will see later).

A Deeper Look: KNN as a Discriminative Classifier

This is a good time to introduce a fundamental concept in classification. Classifiers are generally split into two main types: **Generative** and **Discriminative**.

Generative Classifiers

- **What they do:** A generative model learns the underlying probability distribution of *each class*. It builds a "story" for what each class looks like (e.g., "What's the average and standard deviation of `sepal length` for a `setosa`?").

- **The Question it Answers:** "What does *Class A* look like?"
- **How it Classifies:** It uses this knowledge and Bayes' theorem to calculate $P(y|X)$—the probability of a class y given a new X.
- **Example:** Naive Bayes is the classic generative model.

Discriminative Classifiers

- **What they do:** A discriminative model *ignores* the underlying distribution of each class. It doesn't care what "Class A" looks like. It *only* cares about finding the **decision boundary** that separates Class A from Class B.
- **The Question it Answers:** "What is the *difference* between Class A and Class B?"
- **How it Classifies:** It directly learns the function $h(x)$ or the boundary $P(y|X)$.
- **Example: K-Nearest Neighbors is a classic discriminative model.**

The decision boundary plots we saw earlier are the *perfect* illustration of this. KNN is literally *all about* the boundary. It carves up the feature space into regions based on the training data, and it doesn't spend *any* time modeling the probability distribution of the `setosa` class itself.

Logistic Regression, which we will see later, is another famous discriminative classifier.

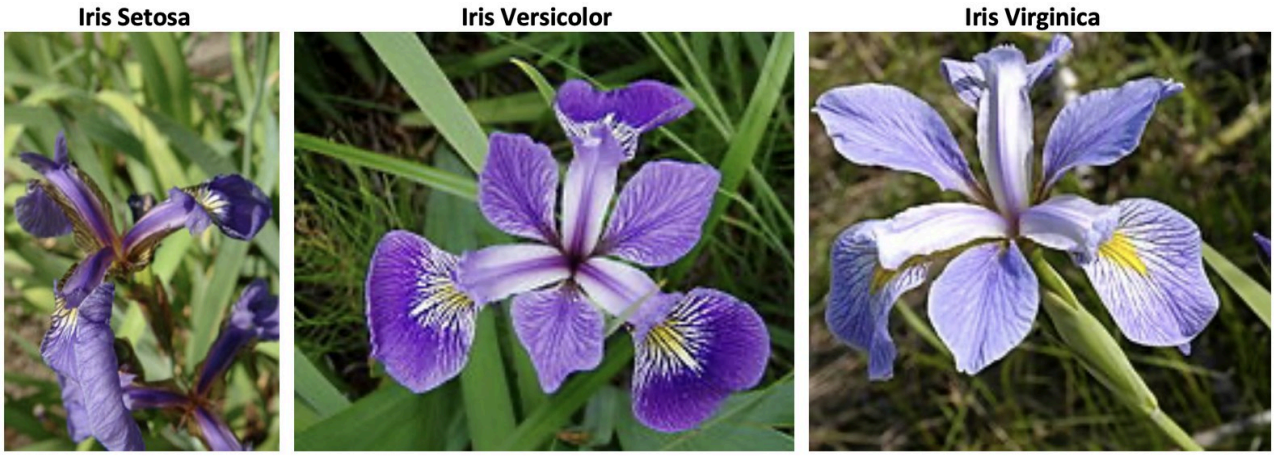
KNN Example: The Fisher Iris Dataset

To understand K-Nearest Neighbors, we will use a classic, famous dataset: the **Fisher Iris Dataset**. This is the "hello, world" of machine learning classification.

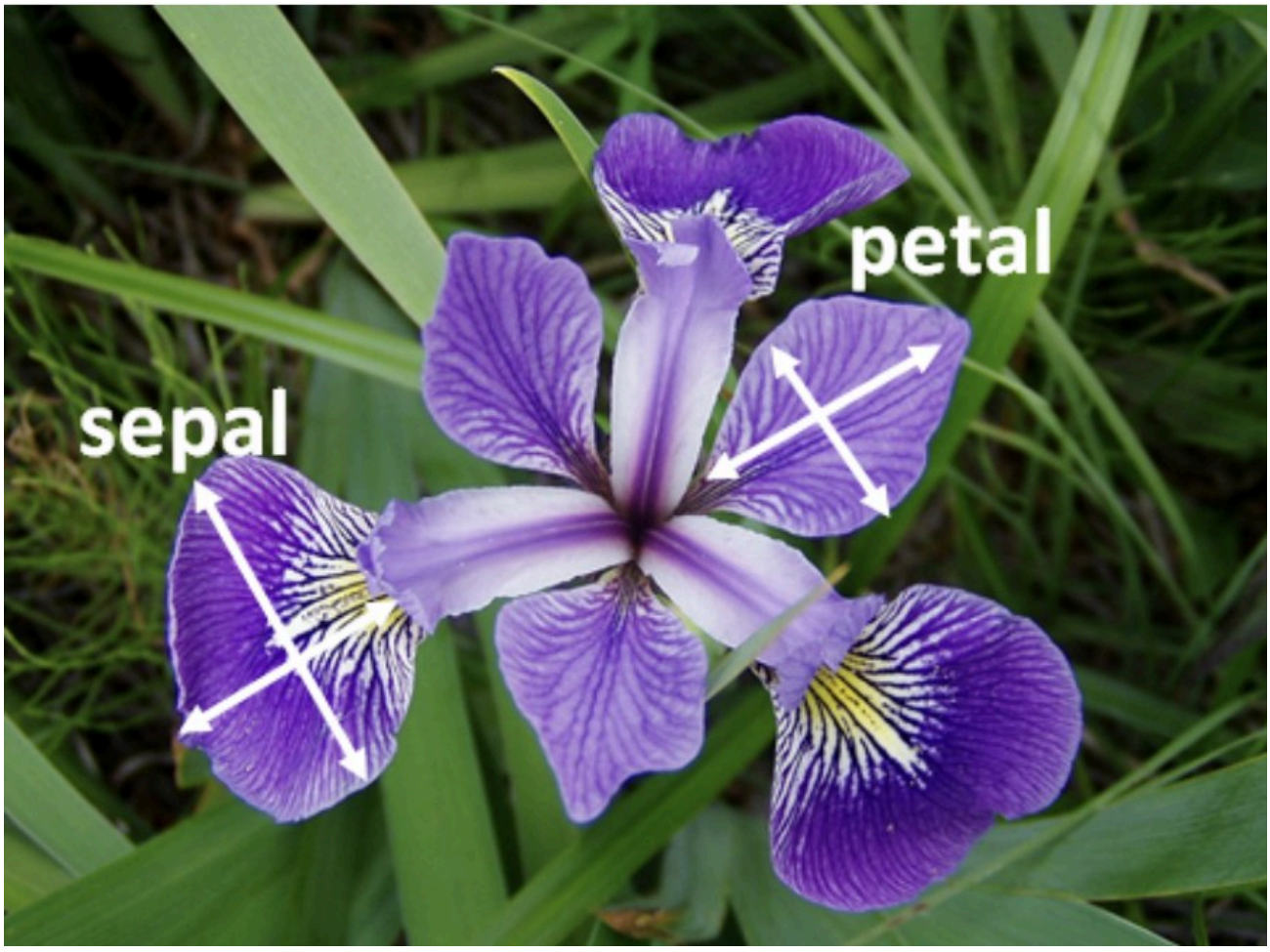
Fisher's Iris Dataset

- **Origin:** Introduced by the statistician and biologist Ronald Fisher in 1936.
- **Goal:** To classify Iris flowers into one of three species (the target variable) based on four physical measurements (the features).
- **Data Structure:**
  - **Target (Y):** 3 classes (species)
    1. *Iris setosa*
    2. *Iris versicolor*
    3. *Iris virginica*
  - **Features (X):** 4 continuous measurements
    1. Sepal Length (in cm)
    2. Sepal Width (in cm)
    3. Petal Length (in cm)
    4. Petal Width (in cm)
  - **Data Size:** 150 total samples, perfectly balanced with 50 samples for each of the three species.

The figure below depicts the three categories of flowers:



The following one illustrates the features:



The Iris dataset is famous because it perfectly illustrates the main challenge of classification.

- One species, *Iris setosa*, is **linearly separable** from the other two (it's very easy to classify).
- The other two species, *Iris versicolor* and *Iris virginica*, **are *not* linearly separable**. They overlap, meaning there is no simple, straight line that can perfectly separate them.

This overlap is what makes the problem interesting and is a perfect scenario to test a flexible, non-linear classifier like K-Nearest Neighbors (KNN). Our goal will be to train a KNN model that can learn the "boundaries" between these species based on their petal and sepal measurements.

Loading and Exploring the Data

First, we'll load the dataset. It's included directly in `scikit-learn`. We will load it and convert it into a `pandas` DataFrame to make it easier to work with.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

# Load the dataset
iris = load_iris()

# Create a DataFrame
# We use iris.feature_names as the column headers
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# Add the target (species) to the DataFrame
# iris.target is numeric (0, 1, 2)
df['target'] = iris.target

# Add a column for the actual species names for easier plotting
df['species'] = df['target'].map({
    0: 'setosa',
    1: 'versicolor',
    2: 'virginica'
})

# Display the first 5 rows
print("--- Data Head ---")
print(df.head())

# Check the data types
print("\n--- Data Info ---")
df.info()

# Check the class balance
print("\n--- Class Balance ---")
print(df['species'].value_counts())
```

```
--- Data Head ---
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target species
0       0  setosa
1       0  setosa
2       0  setosa
3       0  setosa
4       0  setosa

--- Data Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int64
 5   species            150 non-null    object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB

--- Class Balance ---
species
setosa        50
versicolor    50
virginica     50
Name: count, dtype: int64
```

Preparing for Modeling

Before we can train any model, we must perform two critical steps:

1. **Train/Test Split:** This is the **Golden Rule of Machine Learning**. We will split our 150 samples into two groups: a `training set` (which the model learns from) and a `test set` (which we hold back to get an honest, unbiased score).
2. **Feature Scaling:** KNN is a **distance-based algorithm**. It finds the “nearest” neighbors by calculating Euclidean distance. A feature like `sepal length` (range 4.3-7.9) would have a much larger influence on this distance than `petal width` (range 0.1-2.5). To prevent this, we must

standardize our features (e.g., scale them to have a mean of 0 and a standard deviation of 1) so they are all on a level playing field.

For this example, we will use only **two features**—`petal length (cm)` and `petal width (cm)`—because they are highly predictive and allow us to easily visualize our model's decisions in 2D later.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# 1. Define our features (X) and target (y)
# We select only the two petal features for this example
X = df[['petal length (cm)', 'petal width (cm)']]
y = df['target']

# 2. Create the Train/Test Split
# test_size=0.3 means 30% of the data is held back for testing.
# random_state=42 ensures our split is reproducible.
# stratify=y ensures that the proportion of each class (setosa, etc.)
# is the same in both the train and test sets. This is a best practice.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
        random_state=42, stratify=y)

print(f"Training samples: {X_train.shape[0]}")
print(f"Test samples:     {X_test.shape[0]}")

# 3. Scale the Features
scaler = StandardScaler()

# We .fit() the scaler ONLY on the training data
X_train_scaled = scaler.fit_transform(X_train)

# We .transform() the test data using the *same* scaler
# (We don't fit on the test data, as that would be "cheating")
X_test_scaled = scaler.transform(X_test)
```

```
Training samples: 105
Test samples:     45
```

Let's visualize the data, both training and test:

```python
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Load Iris dataset
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target
df['target_name'] = df['target'].apply(lambda x: iris.target_names[x])

# Select features and target
X = df[['petal length (cm)', 'petal width (cm)']]
y = df['target_name']

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y)

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

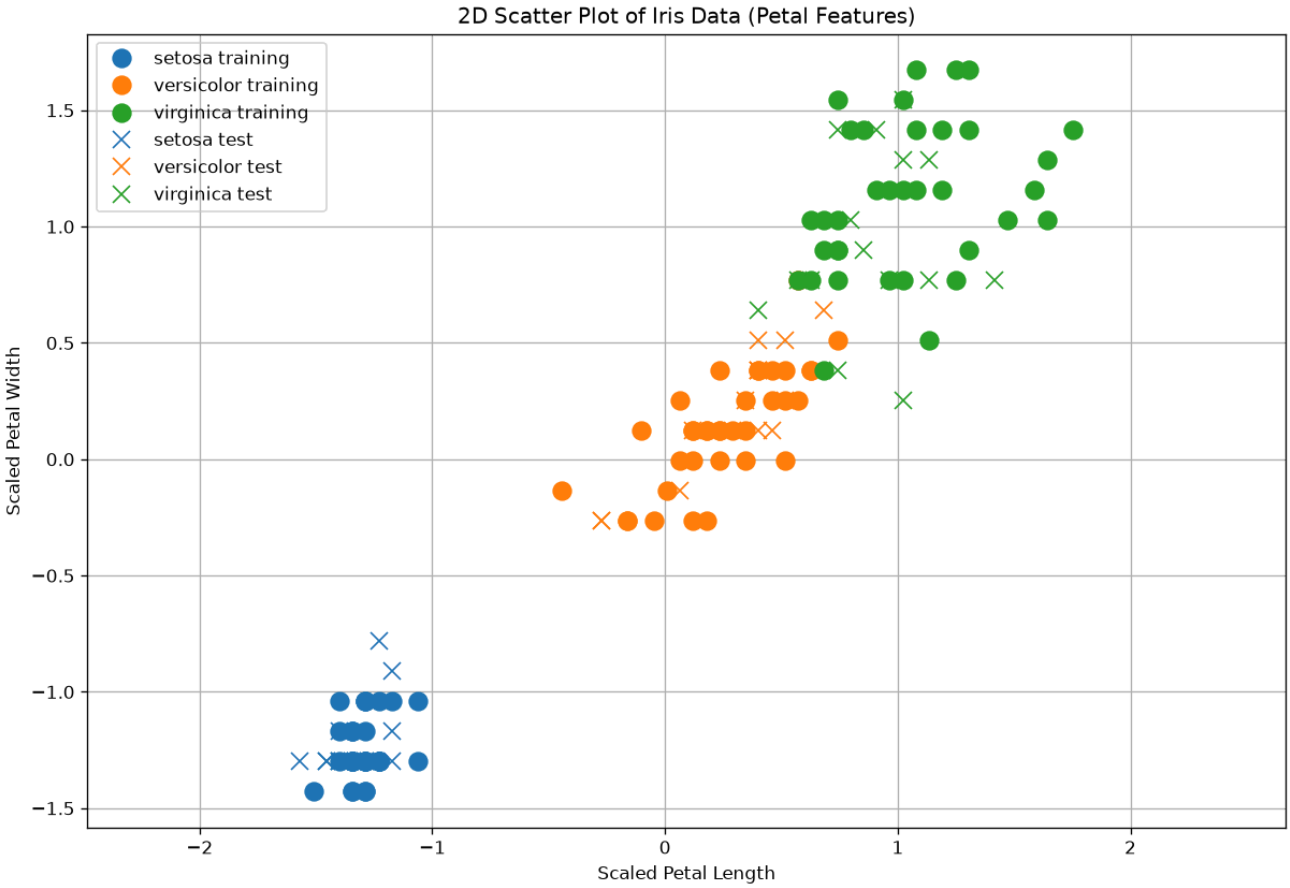
# Prepare dataframes for plotting
data_training = pd.DataFrame(X_train_scaled, columns=['X', 'Y'])
data_training['C'] = y_train.values

data_test = pd.DataFrame(X_test_scaled, columns=['X', 'Y'])
data_test['C'] = y_test.values

# Plotting function
def plot2d(data, label_suffix='', marker='o'):
    classes = sorted(data['C'].unique())
    for c in classes:
        subset = data[data['C'] == c]
        plt.plot(subset['X'].values, subset['Y'].values, marker,
                 label=c + label_suffix, markersize=10)

# Create plot
plt.figure(figsize=(12, 8))
plot2d(data_training, ' training', marker='o')
plt.gca().set_prop_cycle(None)  # Reset color cycle
plot2d(data_test, ' test', marker='x')
plt.axis('equal')
plt.grid()
```

```
plt.legend()
plt.title('2D Scatter Plot of Iris Data (Petal Features)')
plt.xlabel('Scaled Petal Length')
plt.ylabel('Scaled Petal Width')
plt.show()
```



Training the KNN Model

Now we can train our model. The KNN algorithm is very simple:

- **Training:** The "training" step is just storing the (scaled) `X_train` and `y_train` data. That's it.
- **Predicting:** When we get a new, unseen flower, the model calculates the distance to *all* points in its "memory" (the training data) and finds the k nearest ones. It then takes a "majority vote" of those k neighbors to decide the species.

The most important choice we have to make is the **hyperparameter** k (the number of neighbors).

- A **small** k (e.g., $k = 1$) creates a "nervous" model that is very sensitive to noise. It has **low bias** but **high variance** (it overfits).
- A **large** k (e.g., $k = 50$) creates a "smooth" model that ignores local patterns. It has **high bias** but **low variance** (it underfits).

We will start with a common, balanced choice: $k = 5$.

```
# 1. Choose k
k = 5

# 2. Create the classifier
knn = KNeighborsClassifier(n_neighbors=k)

# 3. "Train" the model (i.e., let it memorize the scaled training data)
knn.fit(X_train_scaled, y_train)

print(f"KNN model trained with k={k}")
```

```
KNN model trained with k=5
```

Evaluating the Model

Now for the moment of truth. We will use our trained model to make predictions on the `X_test_scaled` data, which it has *never seen before*. We will then compare its predictions (`y_pred`) to the *true* labels (`y_test`).

We will use three key metrics:

1. **Accuracy:** The simplest metric. What percentage of predictions did the model get right?
2. **Confusion Matrix:** The most important diagnostic. It shows us *where* the model got confused.
3. **Classification Report:** A full summary of Precision, Recall, and F1-Score for each class.

```
# 1. Make predictions on the test set
y_pred = knn.predict(X_test_scaled)

# 2. Calculate Accuracy
acc = accuracy_score(y_test, y_pred)
print(f"Model Accuracy (k=5): {acc * 100:.2f}%")

# 3. Calculate and print the Confusion Matrix
# Rows are the "True" class, Columns are the "Predicted" class
print("\n--- Confusion Matrix ---")
cm = confusion_matrix(y_test, y_pred)
print(cm)

# 4. Print the Classification Report
print("\n--- Classification Report ---")
report = classification_report(y_test, y_pred, target_names=iris.target_names)
print(report)
```

```
Model Accuracy (k=5): 91.11%

--- Confusion Matrix ---
[[15  0  0]
 [ 0 14  1]
 [ 0  3 12]]

--- Classification Report ---
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.82      0.93      0.88        15
   virginica       0.92      0.80      0.86        15

    accuracy                           0.91        45
   macro avg       0.92      0.91      0.91        45
weighted avg       0.92      0.91      0.91        45
```

Interpreting the Results

Let's analyze the output:

- **Accuracy:** We achieved a very high accuracy! This is common on the Iris dataset, especially when using the petal features.
- **Confusion Matrix:** The matrix is the best way to see this: `[[15  0  0]  ← (True: setosa)` `[ 0 14  1]  ← (True: versicolor)  [ 0  3 12]] ← (True: virginica)`
  - The numbers on the diagonal (15, 14, 12) are our **correct** predictions.
  - The numbers *off* the diagonal are our **errors**. 1 versicolor has been classified as virginica, while 2 virginica have been classified as versicolor.
  - It correctly identified all 15 `setosa`, 14 `versicolor`, and 12 `virginica` samples in the test set.
- **Classification Report:**
  - This confirms the perfect score. The **precision** (of the predictions) and **recall** (of the true classes) are very high for the three species.

This shows that for this dataset, a simple KNN (k=5) model using only two features is a good classifier.

Macro vs. Micro Averaging

The report gives us scores for `setosa`, `versicolor`, and `virginica` individually. But how do we get a single, overall score for "Precision" or "Recall"? We have to average the per-class scores. There are three main ways to do this:

1. **Macro Averaging**

- **How:** Calculates the metric (e.g., Precision) for each class *independently*, then computes the simple, unweighted average.
- **Formula:** `Macro Precision = (Precision_setosa + Precision_versicolor + Precision_virginica) / 3`
- **What it means:** Every **class** gets an equal vote, regardless of its size. Use this if you care about performance on rare classes.

2. **Weighted Averaging**
   - **How:** Calculates the metric for each class, then computes a weighted average based on the number of true samples for each class (its "support").
   - **Formula:** `Weighted Precision = (Prec_setosa * 15 + Prec_versicolor * 15 + Prec_virginica * 15) / (15 + 15 + 15)`
   - **What it means:** Every class's score is weighted by its *size*. This is a good, "fair" average for imbalanced datasets.
3. **Micro Averaging**
   - **How:** Sums up *all individual* True Positives, False Positives, and False Negatives from *all classes* first, then computes one global score.
   - **Formula:** `Micro Precision = (TP_set + TP_ver + TP_vir) / (TP_set + TP_ver + TP_vir + FP_set + FP_ver + FP_vir)`
   - **What it means:** Every **sample** gets an equal vote. In multi-class classification, this number will always be **exactly the same as the overall Accuracy**.

Classification Map/Decision Boundary Of K-NN and Importance of Parameter K

A classifier f assigns a class to each input $\mathbf{x}$. Since $\mathbf{x}$ can be seen as a geometrical point in the n-dimensional space, it is generally interesting to see how the classification function f works on a portion of the representation space. This is done by generating a **classification map** or **decision boundary**, which is obtained computing the label that the classifier would assign to a dense grid of data points.

This is the decision boundary with $K = 1$ in our case:

```python
import numpy as np
from matplotlib import pyplot as plt
from sklearn import neighbors, datasets
from matplotlib.colors import ListedColormap

# Load Iris dataset
iris = datasets.load_iris()

# Use petal length and petal width (features 2 and 3)
X = iris.data[:, 2:4]
y = iris.target

# Train KNN classifier
knn = neighbors.KNeighborsClassifier(n_neighbors=1)
knn.fit(X, y)

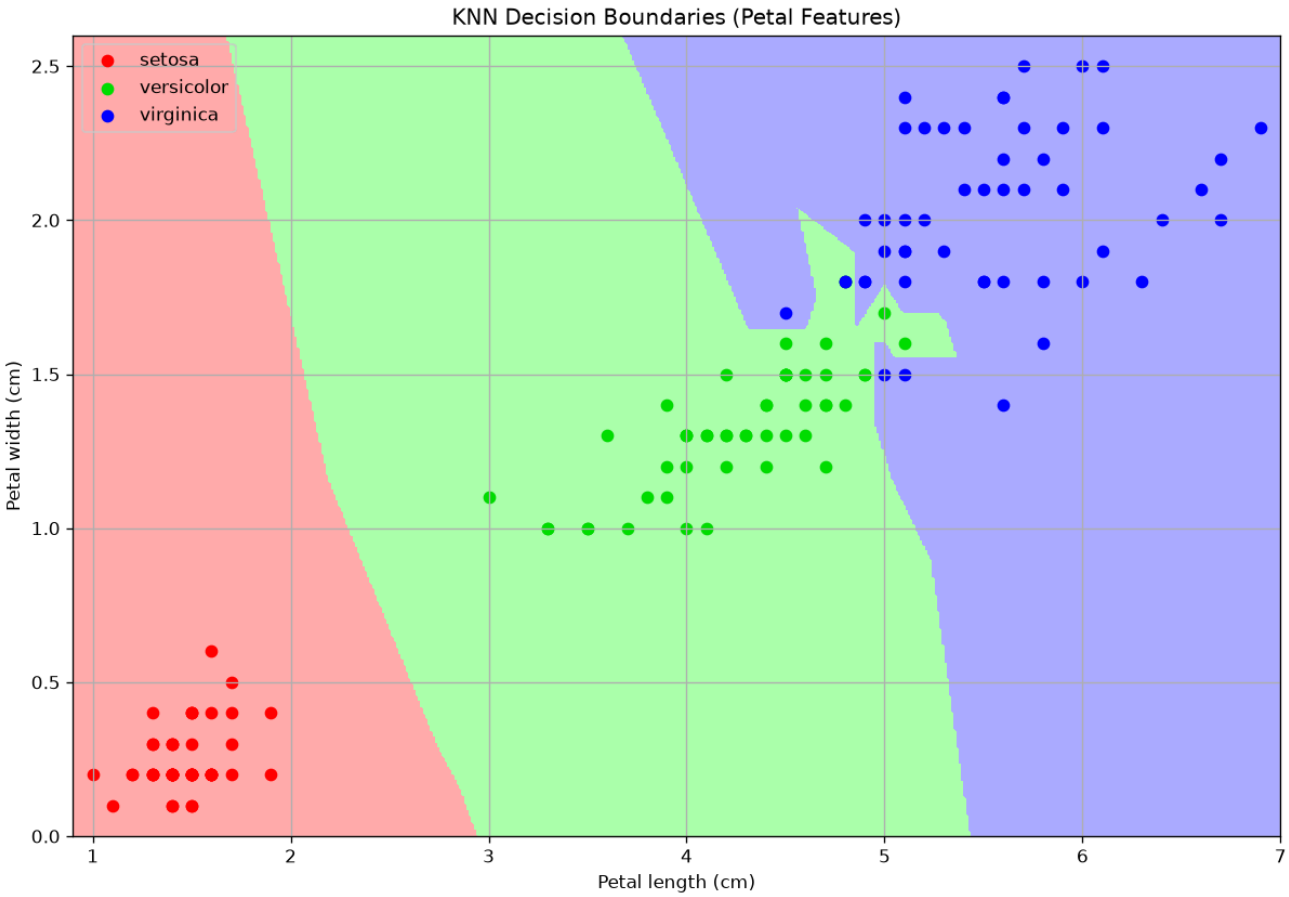
# Create meshgrid for decision boundary
x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500),
                     np.linspace(y_min, y_max, 500))
Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Define color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00DD00', '#0000FF'])

# Plot decision boundaries
plt.figure(figsize=(12, 8))
plt.contourf(xx, yy, Z, cmap=cmap_light)

# Plot training points
plt.scatter(X[y == 0, 0], X[y == 0, 1], color='#FF0000', label=iris.target_names[0])
plt.scatter(X[y == 1, 0], X[y == 1, 1], color='#00DD00', label=iris.target_names[1])
plt.scatter(X[y == 2, 0], X[y == 2, 1], color='#0000FF', label=iris.target_names[2])

# Label axes and finalize plot
plt.xlabel('Petal length (cm)')
plt.ylabel('Petal width (cm)')
plt.legend()
plt.axis('tight')
plt.title('KNN Decision Boundaries (Petal Features)')
plt.grid(True)
plt.show()
```



In the plot above, points of different colors represent the data points belonging to the three classes of the dataset, while regions of different colors represent how that point in space would be classified by the model.

K controls how the space is partitioned (bias-variance tradeoff).

Examples of classification maps for a 1-NN, a 5-NN, a 10-NN, and a 20-NN classifiers are shown below:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors, datasets
from matplotlib.colors import ListedColormap

# Load Iris dataset
iris = datasets.load_iris()
X = iris.data[:, 2:4]  # Petal length and petal width
y = iris.target

# Define color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00DD00', '#0000FF'])

# Create meshgrid
x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500),
                     np.linspace(y_min, y_max, 500))

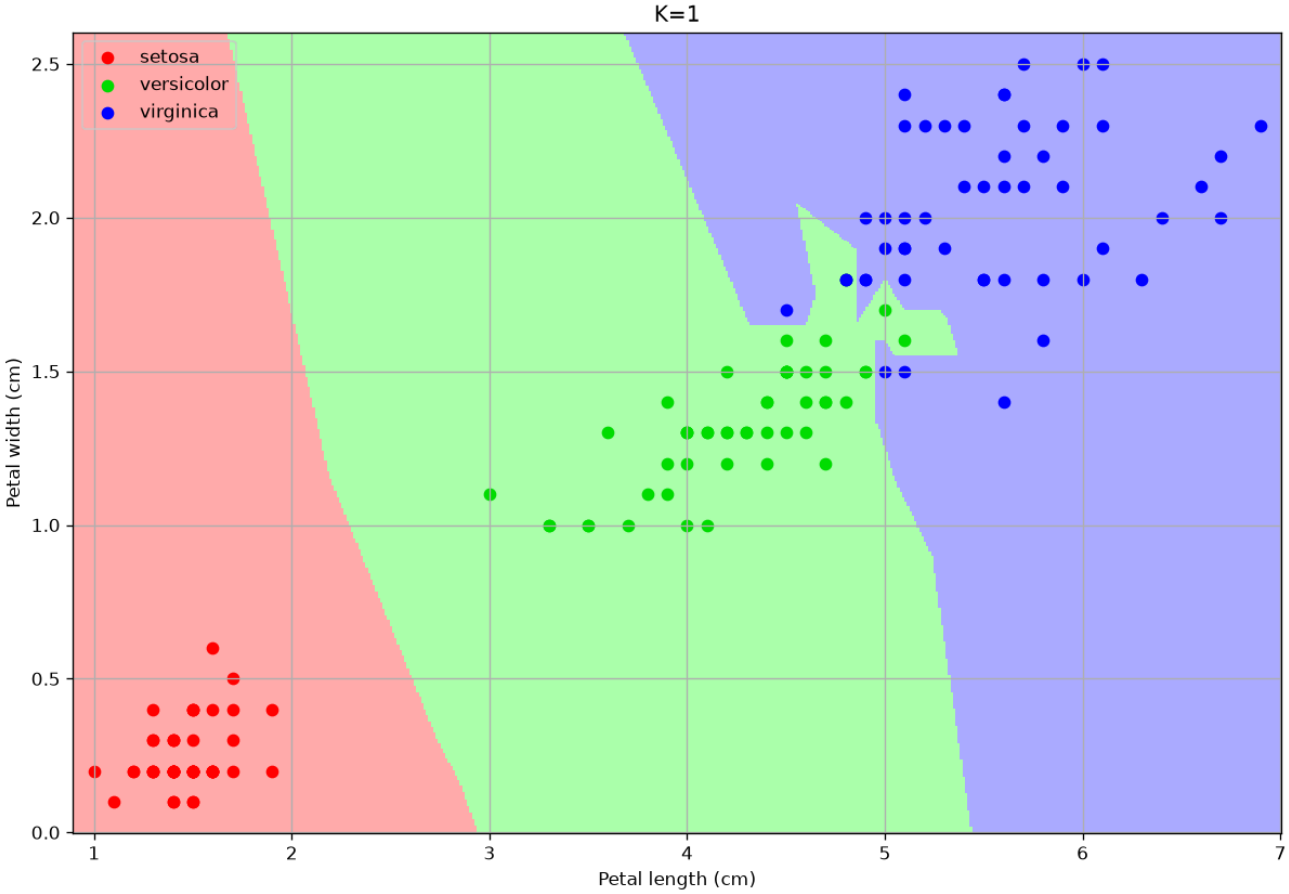
# Loop through different k values
for k in [1, 5, 10, 20]:
    knn = neighbors.KNeighborsClassifier(n_neighbors=k)
    knn.fit(X, y)
    Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

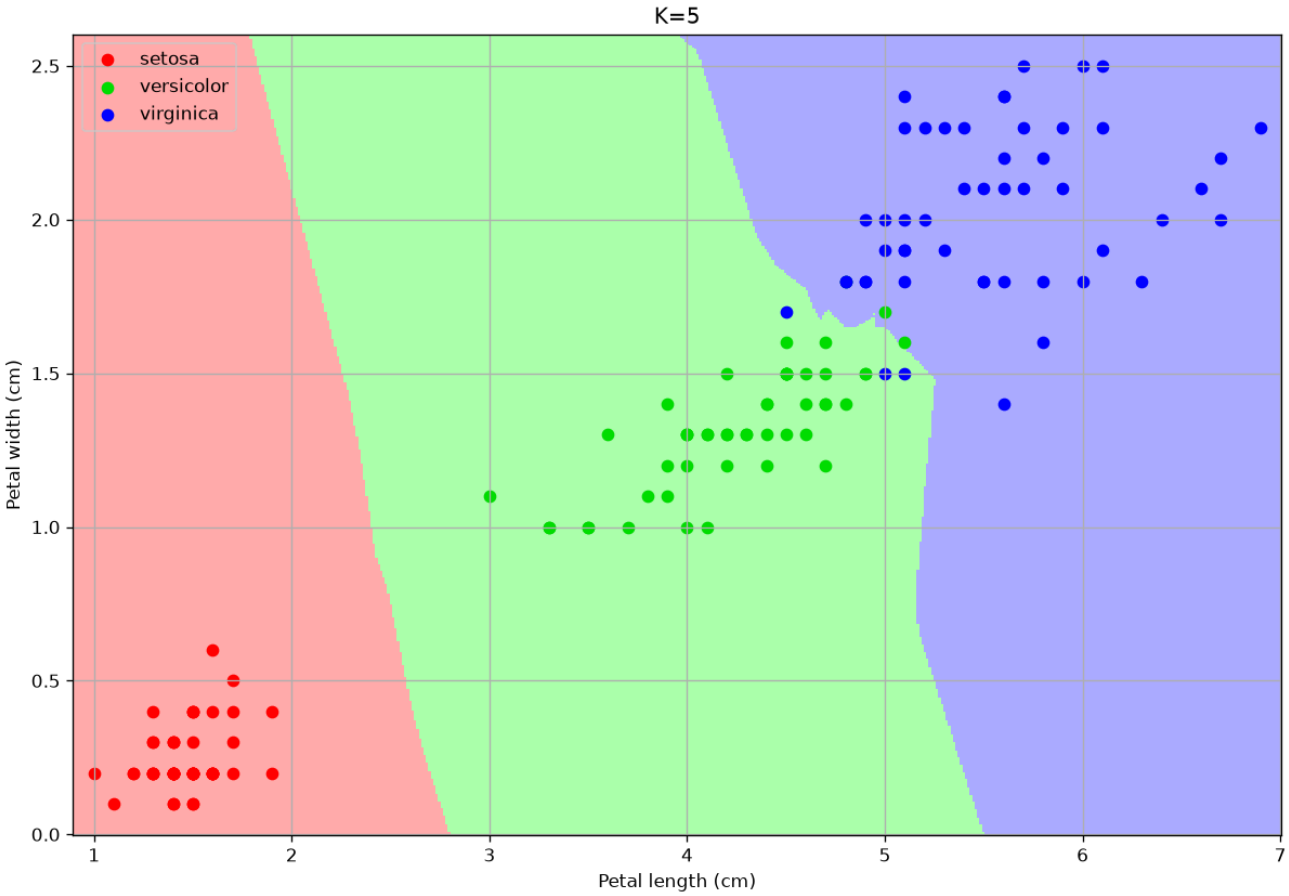
    plt.figure(figsize=(12, 8))
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light, shading='auto')

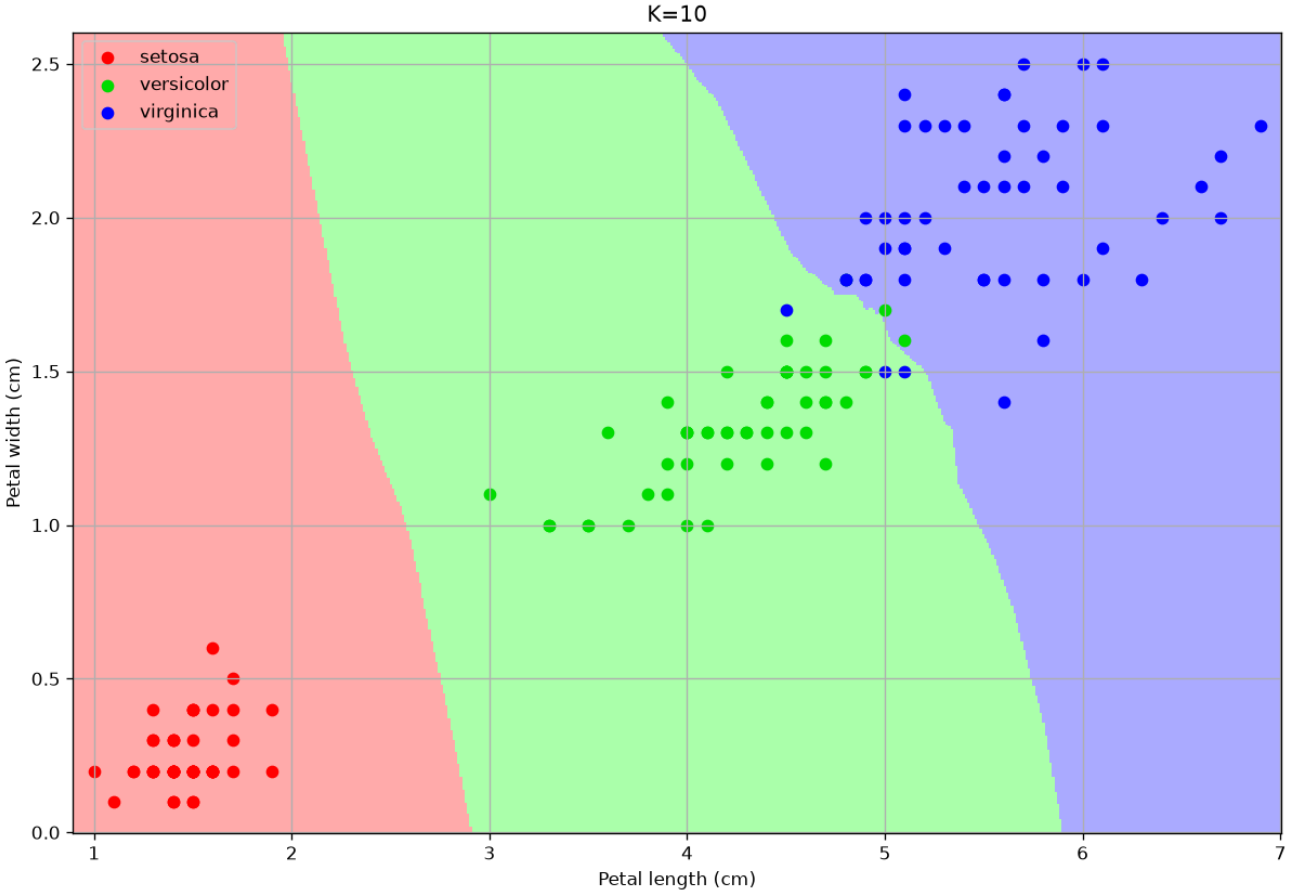
    # Plot training points
    plt.scatter(X[y == 0, 0], X[y == 0, 1], color='#FF0000',
        label=iris.target_names[0])
    plt.scatter(X[y == 1, 0], X[y == 1, 1], color='#00DD00',
        label=iris.target_names[1])
    plt.scatter(X[y == 2, 0], X[y == 2, 1], color='#0000FF',
        label=iris.target_names[2])

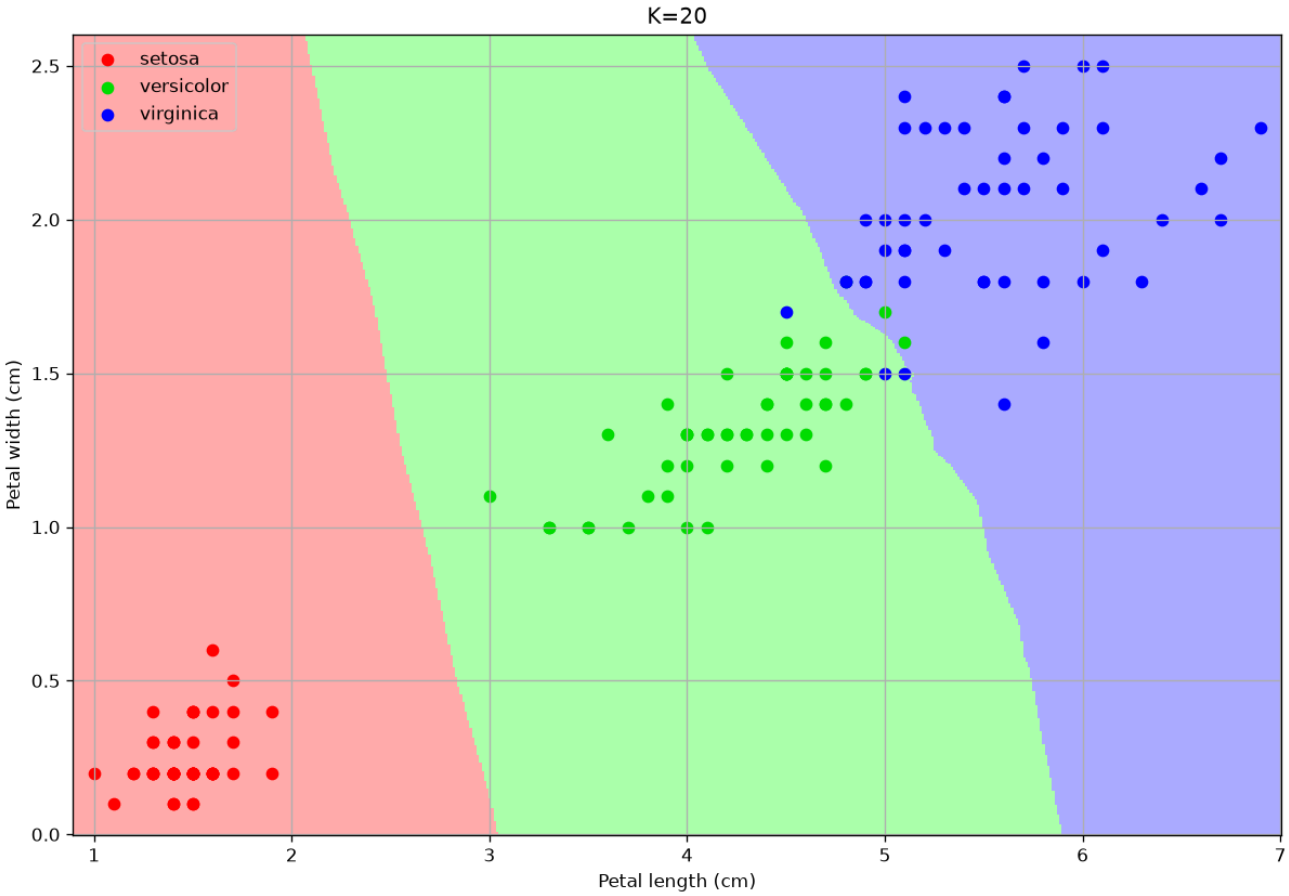
    plt.xlabel('Petal length (cm)')
    plt.ylabel('Petal width (cm)')
    plt.title(f'K={k}')
    plt.legend()
    plt.axis('tight')
    plt.grid(True)

plt.show()
```


K=1
setosa
versicolor
virginica
Petal width (cm)
Petal length (cm)
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


K=5
setosa
versicolor
virginica
Petal width (cm)
Petal length (cm)
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7





The number of neighbors changes the decision map:

- For small values, the algorithm tends to over-segment the space and creates very small regions for isolated training data-points.
- For larger values, the regions tend to be smoother and isolated data points are ignored.
- Choosing **a larger K can** reduce overfitting (indeed the isolated data-points can be seen as outliers).
- However, choosing a **too large K** can encourage underfitting, by completing ignoring some of the decision regions.
- In particular, setting **K to the size of the training set**, any data point is classified with the most numerous class.

Finding the best K

We saw that K affects the quality of our classifier, but how to choose the appropriate one? We will see how to do it with cross-validation.

Step 1: Prepare Data

For this, we'll use all 4 features to make the problem more robust. We'll do a single 70/30 split and then lock the test set away.

```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

# 1. Load data
iris = load_iris()
X = iris.data
y = iris.target

# 2. Create the one-and-only Train/Test Split
# We'll call it X_train_full, y_train_full to be clear
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.3,
        random_state=42, stratify=y)

# 3. Scale the data
# We fit the scaler ONLY on the full training set
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_full)

# We transform the test set using the *same* scaler
X_test_scaled = scaler.transform(X_test)
```

Step 2: Run Cross-Validation to Find Best K

Now we loop through our K values (e.g., 1 to 20). For each K, we will use `cross_val_score` to automatically perform 5-fold cross-validation *only on the training data*.

```
# 1. Define a range of K values to test
k_range = range(1, 21)
cv_scores = [] # This will store the *mean* accuracy for each k

# 2. Loop through each K
for k in k_range:
    # Create a new KNN model for this k
    knn = KNeighborsClassifier(n_neighbors=k)

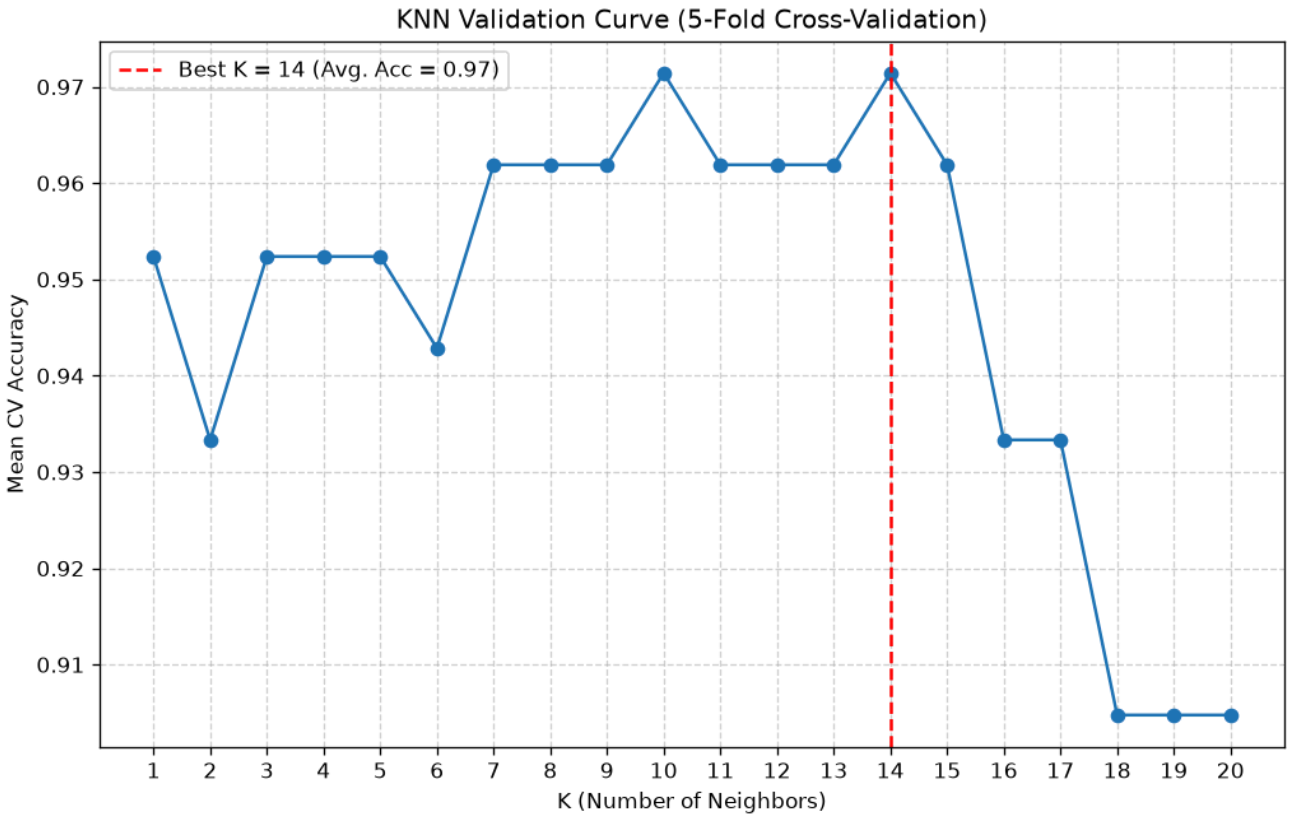
    # Perform 5-fold cross-validation
    # We use X_train_scaled and y_train_full
    # 'cv=5' means 5 folds
    # 'scoring='accuracy'' is the metric we want
    scores = cross_val_score(knn, X_train_scaled, y_train_full, cv=5,
        scoring='accuracy')

    # Store the average accuracy from the 5 folds
    cv_scores.append(scores.mean())

# 3. Plot the Validation Curve
plt.figure(figsize=(10, 6))
plt.plot(k_range, cv_scores, 'o-')
plt.title('KNN Validation Curve (5-Fold Cross-Validation)')
plt.xlabel('K (Number of Neighbors)')
plt.ylabel('Mean CV Accuracy')
plt.xticks(k_range)

# Find and mark the best K
best_k_index = np.argmax(cv_scores)
best_k = k_range[best_k_index]
best_acc = cv_scores[best_k_index]
plt.axvline(best_k, linestyle='--', color='red', label=f'Best K = {best_k} (Avg. Acc
        = {best_acc:.2f})')

plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```



Step 3: Final Training and Evaluation

The plot shows us the bias-variance tradeoff.

- **Low K (left):** The accuracy is jumpy and lower. The model is overfitting to the *folds* (High Variance).
- **High K (right):** The accuracy starts to drop as the model becomes too "simple" (High Bias).
- **The Peak:** Our plot shows the "sweet spot" is around $K = 10$ or $K = 14$. This is our chosen hyperparameter.

Now, we perform our final two steps.

```python
# 1. Get the best K we found from our CV
best_k = k_range[np.argmax(cv_scores)]
print(f"Cross-validation complete. The best K is {best_k}.")

# 2. Train a *new* model on the *ENTIRE* training set
final_knn_model = KNeighborsClassifier(n_neighbors=best_k)
final_knn_model.fit(X_train_scaled, y_train_full)

# 3. Finally, use the "locked away" Test Set
# This is our one and only time.
final_predictions = final_knn_model.predict(X_test_scaled)

# 4. Report the final, unbiased score
final_accuracy = accuracy_score(y_test, final_predictions)
print(f"\n--- Final Model Evaluation ---")
print(f"Final Test Set Accuracy (with K={best_k}): {final_accuracy * 100:.2f}%")

print("\n--- Final Classification Report ---")
print(classification_report(y_test, final_predictions,
        target_names=iris.target_names))
```

```
Cross-validation complete. The best K is 14.

--- Final Model Evaluation ---
Final Test Set Accuracy (with K=14): 95.56%

--- Final Classification Report ---
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.88      1.00      0.94        15
   virginica       1.00      0.87      0.93        15

    accuracy                           0.96        45
   macro avg       0.96      0.96      0.96        45
weighted avg       0.96      0.96      0.96        45
```

Conclusion

This is the principled, robust machine learning workflow.

1. We split our data once.
2. We used the **training data** and **cross-validation** to find the best hyperparameter (K).
3. We trained a single, final model on *all* the training data.
4. We reported a final, honest accuracy by testing on the **test data**.

The final classification report gives us the unbiased performance of our *tuned* model.

K-Nearest Neighbors (KNN) for Regression

We've seen how KNN works for **Classification** (predicting a label) by taking a "majority vote" of the nearest neighbors.

The algorithm is extremely flexible and can also be used for **Regression** (predicting a continuous value). The logic is identical, with one small change in the final step:

- **KNN Classifier (Iris):** Find the k nearest flowers $\rightarrow$ Predict the **most common species** (the *mode*) of the neighbors.
- **KNN Regressor (MPG):** Find the k nearest cars $\rightarrow$ Predict the **average** `mpg` (the *mean*) of the neighbors.

Let's use our `mpg` dataset to build a `KNeighborsRegressor` that predicts `mpg` using `horsepower`.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Import the Regressor and other tools
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error

# Load and clean the data
mpg = sns.load_dataset('mpg').dropna()

# 1. Define our features (X) and target (y)
# We use .values.reshape(-1, 1) to turn the pandas Series into a 2D numpy array
X = mpg[['horsepower']].values
y = mpg['mpg'].values

# 2. Create the Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
        random_state=42)

# 3. Scale the Features (CRITICAL for KNN)
# KNN is distance-based, so scaling is not optional.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Visualizing the Bias-Variance Tradeoff (Overfit vs. Underfit)

Just like in classification, the choice of k controls the bias-variance tradeoff. Let's build two models to see this:

1. A $k = 3$ **model (Low Bias, High Variance):** This model will be "nervous" and "wiggly," as it only looks at its 3 closest neighbors. It will overfit to the training data.
2. A $k = 50$ **model (High Bias, Low Variance):** This model will be very "smooth" and "simple," as it averages the opinions of 50 neighbors. It will be stable but will underfit, missing the local patterns.

```
# 1. Create the two models (using 'uniform' weights)
knn_overfit = KNeighborsRegressor(n_neighbors=3, weights='uniform')
knn_underfit = KNeighborsRegressor(n_neighbors=50, weights='uniform')

# 2. Fit both models on the scaled training data
knn_overfit.fit(X_train_scaled, y_train)
knn_underfit.fit(X_train_scaled, y_train)

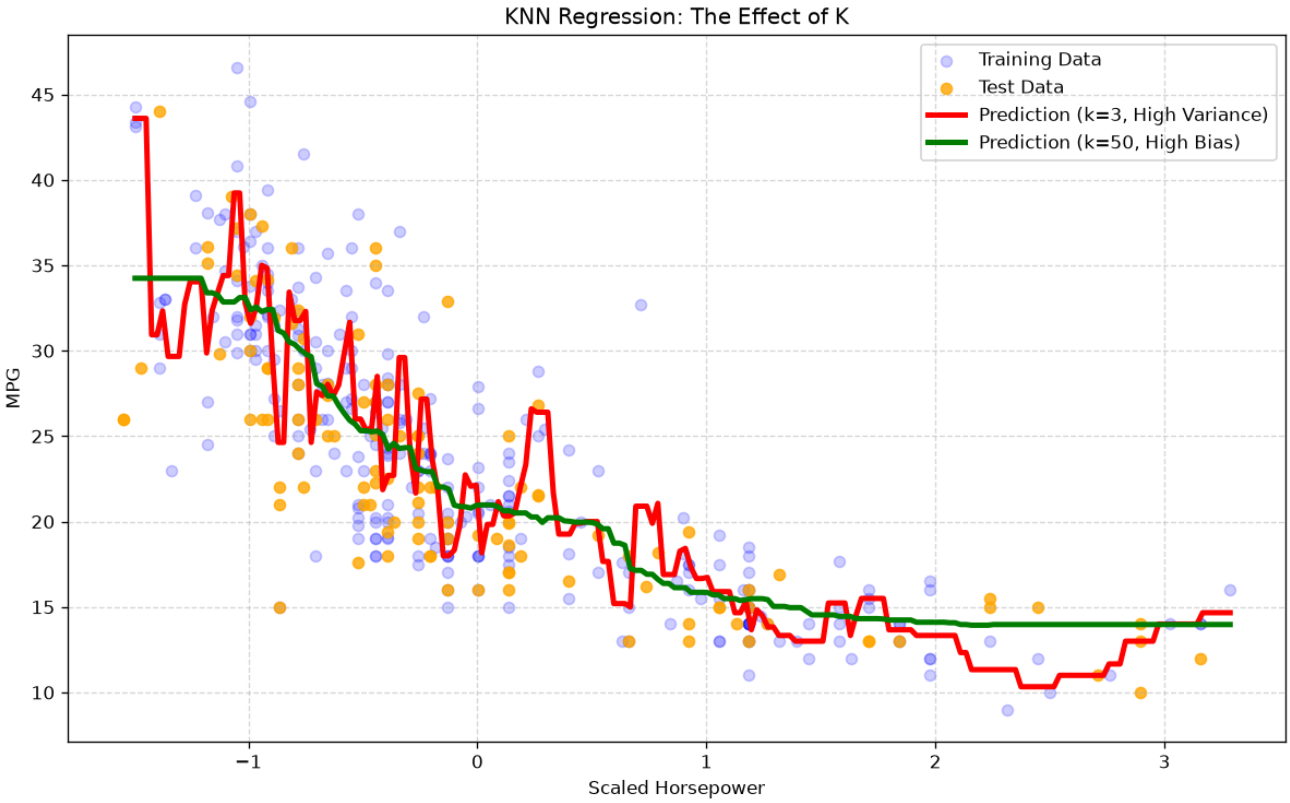
# 3. Create a smooth line of x-values for plotting
X_plot = np.linspace(X_train_scaled.min(), X_train_scaled.max(), 200).reshape(-1, 1)

# 4. Get the predictions from both models
y_pred_overfit = knn_overfit.predict(X_plot)
y_pred_underfit = knn_underfit.predict(X_plot)

# 5. Plot the results
plt.figure(figsize=(12, 7))
plt.scatter(X_train_scaled, y_train, c='blue', alpha=0.2, label='Training Data')
plt.scatter(X_test_scaled, y_test, c='orange', alpha=0.8, label='Test Data')

plt.plot(X_plot, y_pred_overfit, 'r-', lw=3, label='Prediction (k=3, High Variance)')
plt.plot(X_plot, y_pred_underfit, 'g-', lw=3, label='Prediction (k=50, High Bias)')

plt.title('KNN Regression: The Effect of K')
plt.xlabel('Scaled Horsepower')
plt.ylabel('MPG')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```



Interpretation of the Plot

You can clearly see the bias-variance tradeoff in action:

- The **red line ($k = 3$)** is "spiky" and "nervous." It tries to follow the training data (blue dots) too closely. This is **overfitting (High Variance)**.
- The **green line ($k = 50$)** is very smooth, acting almost like a simple average. It misses the clear non-linear curve in the data. This is **underfitting (High Bias)**.

Our goal is to find the "sweet spot" k in the middle.

Finding the Optimal K with Cross-Validation

Just like in our classification example, we can use K-Fold Cross-Validation on the **training set** to find the best k.

The only difference is our `scoring` metric. Instead of `accuracy`, we use `neg_mean_squared_error` (scikit-learn uses *negative* MSE because its internal optimizer always tries to *maximize* a score).

```
# 1. Define a range of K values to test
k_range = range(1, 41)
cv_scores = [] # This will store the *mean* RMSE for each k

# 2. Loop through each K
for k in k_range:
    knn = KNeighborsRegressor(n_neighbors=k, weights='uniform')

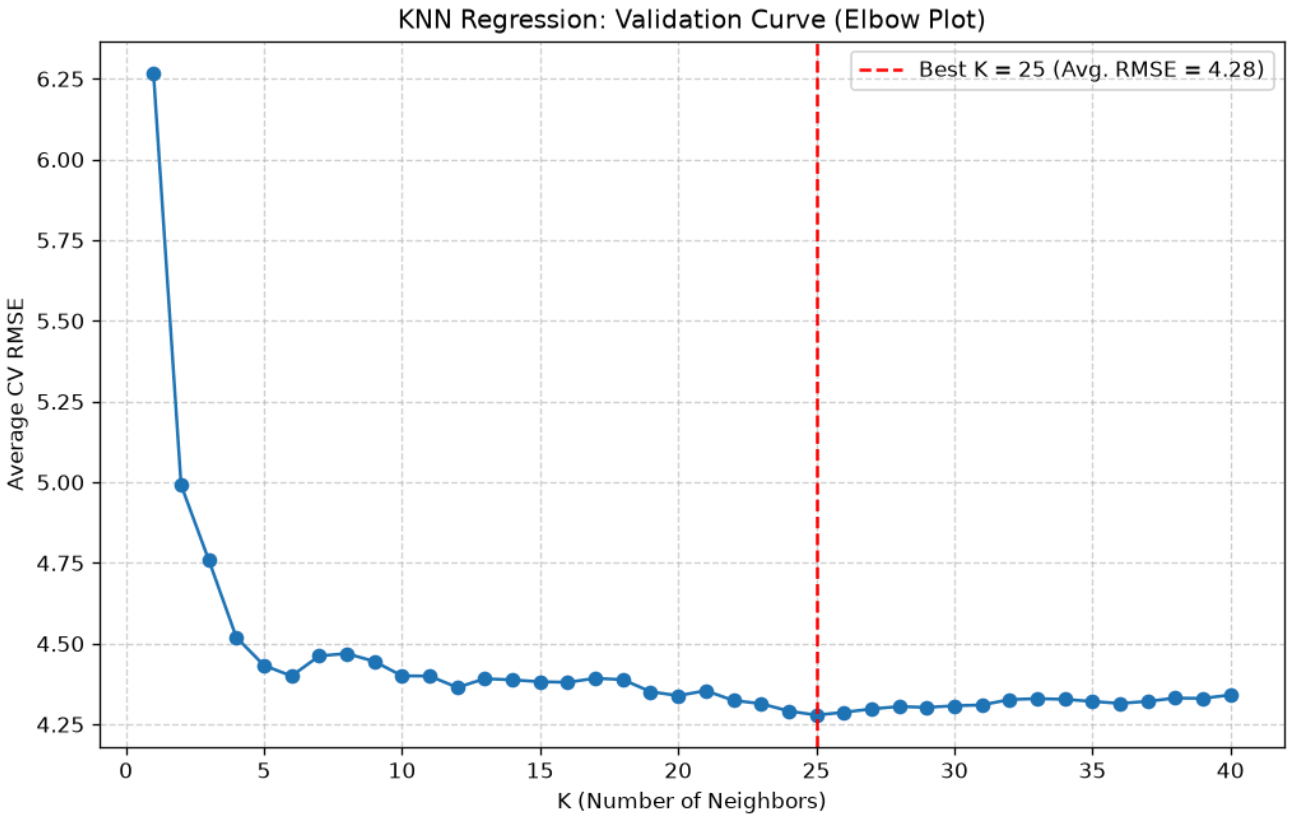
    # Perform 5-fold cross-validation on the training data
    # We ask for 'neg_mean_squared_error'
    scores = cross_val_score(knn, X_train_scaled, y_train, cv=5,
        scoring='neg_mean_squared_error')

    # We store the *positive* *root* of the MSE
    cv_scores.append(np.sqrt(-scores.mean()))

# 3. Plot the Validation "Elbow" Curve
plt.figure(figsize=(10, 6))
plt.plot(k_range, cv_scores, 'o-')
plt.title('KNN Regression: Validation Curve (Elbow Plot)')
plt.xlabel('K (Number of Neighbors)')
plt.ylabel('Average CV RMSE') # Lower is better!
plt.xticks(np.arange(0, 41, 5))

# Find and mark the best K (lowest RMSE)
best_k_index = np.argmin(cv_scores)
best_k = k_range[best_k_index]
best_rmse = cv_scores[best_k_index]
plt.axvline(best_k, linestyle='--', color='red', label=f'Best K = {best_k} (Avg. RMSE
        = {best_rmse:.2f})')

plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```



Final Evaluation

The validation plot shows us that the "sweet spot" (lowest error) is at $k = 25$.

Now we can train our final, optimized model on the *entire* training set and get our final, unbiased score from the **test set**.

```
# 1. Train the *final* model with the best K
final_knn_regressor = KNeighborsRegressor(n_neighbors=best_k, weights='uniform')
final_knn_regressor.fit(X_train_scaled, y_train)

# 2. Get final predictions on the (held-out) test set
final_predictions = final_knn_regressor.predict(X_test_scaled)

# 3. Calculate the final, honest RMSE
final_rmse = np.sqrt(mean_squared_error(y_test, final_predictions))

print(f"--- Final Model Evaluation ---")
print(f"The best K found by CV: {best_k}")
print(f"Final Test Set RMSE: {final_rmse:.4f} (mpg)")
```

```
--- Final Model Evaluation ---
The best K found by CV: 25
Final Test Set RMSE: 4.2943 (mpg)
```

From Binary to Multi-Class: One-vs-Rest (OvR)

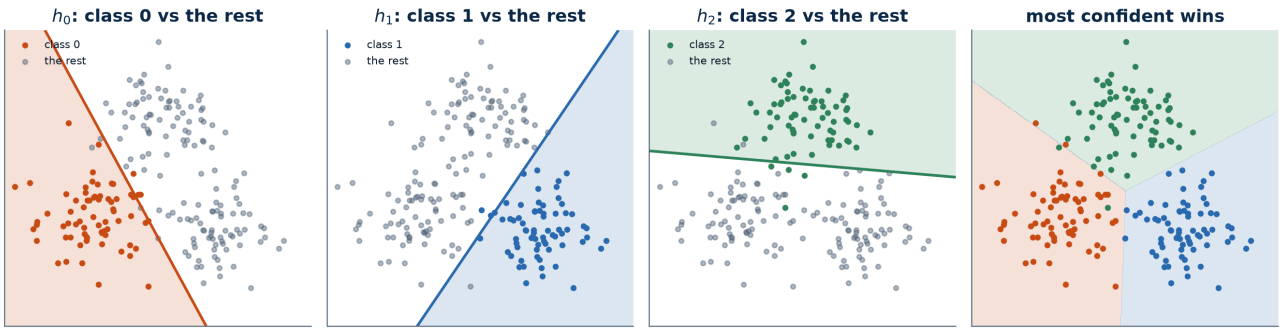
One-vs-Rest provides a technique to turn any binary classification model into a multi-class model. This is also known as “one vs all”.

The OvR approach turns one K-class problem into K separate binary classification problems. To do this, we need a binary classifier (like Logistic Regression) that outputs a **confidence score** (a probability), which we can use to find the most likely class.

Given a classification task with K classes, the OvR approach works as follows:

1. **Deconstruct the Problem:** We train K independent binary classifiers, h_k. Each classifier is trained to distinguish *one* class from *all other* classes (the “rest”).
   - **Classifier 1 (h_1):** Trains on `Class 1` vs. `(Class 2 + Class 3 + ... + Class K)`
   - **Classifier 2 (h_2):** Trains on `Class 2` vs. `(Class 1 + Class 3 + ... + Class K)`
   - ...
   - **Classifier K (h_K):** Trains on `Class K` vs. `(Class 1 + Class 2 + ... + Class K-1)`
2. **Classify a New Point:** To classify a new example $\mathbf{x}$, we feed it to **all K classifiers**. Each one will output a probability or confidence score:
   - $h_1(\mathbf{x}) \rightarrow P(y = 1|\mathbf{x})$
   - $h_2(\mathbf{x}) \rightarrow P(y = 2|\mathbf{x})$
   - $h_3(\mathbf{x}) \rightarrow P(y = 3|\mathbf{x})$
3. **Choose the Winner:** The final predicted class is simply the one whose classifier gave the **highest confidence score**.

This is exemplified in the figure below, on three classes: the first three panels are the binary problems each classifier solves, the fourth is the resulting decision.



One-vs-rest on three classes: three binary classifiers, and the multi-class decision they produce together.

This OvR strategy is a convenient “wrapper” that lets us use a simple binary model for a multi-class task. The alternative is to fit **multinomial logistic regression** directly, as in chapter 10: one model that handles all K classes at once, and normalises across them from the start rather than arbitrating between separate classifiers afterwards. The lab below compares the two.

Multiclass Logistic Regression in Python (Machine Learning Perspective)

In chapters 9 and 10 we used `statsmodels` to build logistic regression models for **inference** and **understanding**. We focused on interpreting p-values and coefficients (log-odds).

Now, we will switch to the **Machine Learning** perspective. Our goal is no longer *understanding* but **predictive accuracy**.

We will use `scikit-learn` to build robust, optimized models. Our new workflow will be:

1. **Start with a Train/Test Split** to get an honest, unbiased evaluation.
2. Build a `Pipeline` that includes `StandardScaler`, as logistic regression is sensitive to feature scales (especially when regularized).
3. Use `GridSearchCV` to find the best **hyperparameters** (specifically the regularization strength `C`).
4. Evaluate our final, tuned model on the **test set** using metrics like `accuracy`, `classification_report`, and the `confusion_matrix`.

Part 1: Binary Logistic Regression

We will use the **Breast Cancer Wisconsin** dataset, which is a classic binary classification problem: predict whether a tumor is `malignant` (1) or `benign` (0) based on 30 continuous features.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
from sklearn.datasets import load_breast_cancer

# All our models and tools
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# 1. Load Data
data = load_breast_cancer()
X = data.data
y = data.target
target_names = data.target_names

print(f"Features shape: {X.shape}")
print(f"Target shape: {y.shape}")
print(f"Classes: 0 = {target_names[0]}, 1 = {target_names[1]}")

# 2. Create the Train/Test Split
# We hold back 30% for the final test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
        random_state=42, stratify=y)
```

```
Features shape: (569, 30)
Target shape: (569,)
Classes: 0 = malignant, 1 = benign
```

Building the Pipeline and Hyperparameter Grid

We will create a `Pipeline` that first scales the data, then fits the logistic regression model.

Note that the `LogisticRegression` is regularized with L2 norm by default with a parameter `C`.

- `C` is the *inverse* of the regularization strength λ we saw in theory.
- **Small** `C` = Strong regularization (simpler model, high bias).
- **Large** `C` = Weak regularization (complex model, high variance).

We will use `GridSearchCV` to find the best value for `C`.

```
# 1. Create the pipeline
pipe_lr = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(solver='liblinear')) # 'liblinear' is a good solver
        for binary
])

# 2. Define a search grid for 'C'
# We'll test a wide range of C values on a log scale
param_grid_lr = {
    'model__C': np.logspace(-3, 3, 7) # [0.001, 0.01, 0.1, 1, 10, 100, 1000]
}

# 3. Create and fit the Grid Search
# cv=5 means 5-fold cross-validation on the training set
print("Starting Grid Search for Binary Logistic Regression...")
search_lr = GridSearchCV(pipe_lr, param_grid_lr, cv=5, scoring='accuracy')
search_lr.fit(X_train, y_train)

print("Grid Search complete.")
print(f"Best 'C' found by CV: {search_lr.best_params_['model__C']}")
print(f"Best CV accuracy:     {search_lr.best_score_:.4f}")
```

```
Starting Grid Search for Binary Logistic Regression...
Grid Search complete.
Best 'C' found by CV: 1.0
Best CV accuracy:     0.9799
```

Final Evaluation (Binary)

The `GridSearchCV` has automatically found the best `C` and re-trained a final model on the entire training set (available as `search_lr.best_estimator_`).

Now we “unlock” our test set for the first and only time to get our final, unbiased score.

```python
# 1. Get predictions from the best-tuned model
y_pred = search_lr.predict(X_test)

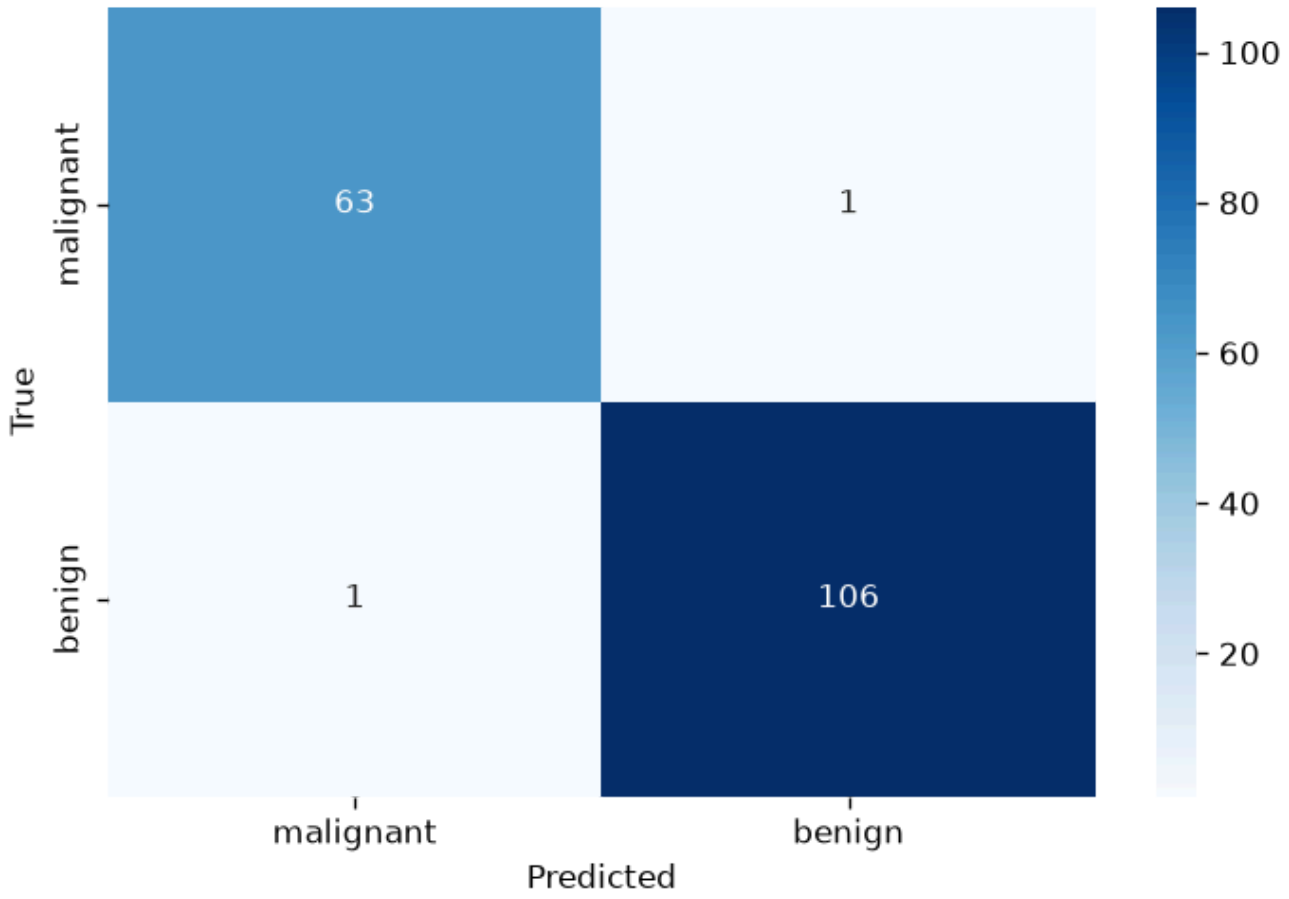
# 2. Calculate final metrics
accuracy = accuracy_score(y_test, y_pred)
print(f"--- Final Model Evaluation (Test Set) ---")
print(f"Overall Accuracy: {accuracy * 100:.2f}%")

print("\n--- Confusion Matrix ---")
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names,
            yticklabels=target_names)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()

print("\n--- Classification Report ---")
print(classification_report(y_test, y_pred, target_names=target_names))
```

```
--- Final Model Evaluation (Test Set) ---
Overall Accuracy: 98.83%

--- Confusion Matrix ---
```



```
--- Classification Report ---
              precision    recall  f1-score   support

   malignant       0.98      0.98      0.98        64
      benign       0.99      0.99      0.99       107

    accuracy                           0.99       171
   macro avg       0.99      0.99      0.99       171
weighted avg       0.99      0.99      0.99       171
```

Part 2: Multi-Class Logistic Regression

What happens when we have more than two classes, like in the **Iris dataset** (3 classes)? A binary model that predicts $P(y = 1)$ can't work.

We have two main strategies:

1. **One-vs-Rest (OvR):**
   - **How it works:** This strategy trains **K** separate binary classifiers.
     - Classifier 1: `setosa` vs. (`versicolor` + `virginica`)
     - Classifier 2: `versicolor` vs. (`setosa` + `virginica`)
     - Classifier 3: `virginica` vs. (`setosa` + `versicolor`)
   - When a new flower comes in, all 3 classifiers give a probability. The one with the highest confidence wins.
   - This is the default strategy in `sklearn`.
2. **Multinomial logistic regression:**
   - **How it works:** the model of chapter 10, fitted as a single unit over all K classes rather than as K separate problems.
   - It outputs a *vector* of probabilities, one per class, guaranteed to sum to 1.
     - e.g., `[P(setosa)=0.9, P(versicolor)=0.05, P(virginica)=0.05]`
   - `scikit-learn` fits it in the softmax parametrisation; `statsmodels` in the baseline one. Same model, and nothing here depends on which.

Let's use the Iris dataset and compare both strategies in a "bake-off".

```
from sklearn.datasets import load_iris
from sklearn.multiclass import OneVsRestClassifier # ⟵ Import the OvR wrapper

# 1. Load Iris data (using all 4 features)
iris = load_iris()
X_i = iris.data
y_i = iris.target
target_names_i = iris.target_names

# 2. Create the Train/Test Split
X_train_i, X_test_i, y_train_i, y_test_i = train_test_split(X_i, y_i, test_size=0.3,
        random_state=42, stratify=y_i)

# 3. Define the parameter grid (we'll use this for both)
param_grid = {
    'C': np.logspace(-3, 3, 7) # [0.001, 0.01, 0.1, 1, 10, 100, 1000]
}
```

Model 1: Multinomial logistic regression

This is the standard for multi-class logistic regression in `sklearn` — the softmax parametrisation of the model of chapter 10. We build a pipeline and tune the `C` parameter of the `LogisticRegression` model.

```
# 1. Create the pipeline for the multinomial model
pipe_multi = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(solver='lbfgs', max_iter=200))
])

# 2. Define the grid search parameters
# We must prefix parameters with the pipeline step name: 'model__'
param_grid_multi = {
    'model__C': np.logspace(-3, 3, 7)
}

# 3. Run Grid Search
print("Running Grid Search for the multinomial model...")
search_multi = GridSearchCV(pipe_multi, param_grid_multi, cv=5, scoring='accuracy')
search_multi.fit(X_train_i, y_train_i)

print(f"Best 'C' for Multinomial: {search_multi.best_params_['model__C']}")
print(f"Best CV Accuracy: {search_multi.best_score_:.4f}")
```

```
Running Grid Search for the multinomial model...
Best 'C' for Multinomial: 1.0
Best CV Accuracy: 0.9810
```

Model 2: One-vs-Rest (OvR)

Now we'll use the `OneVsRestClassifier` wrapper. This wrapper takes a *base* estimator (our binary `LogisticRegression` model) and will train 3 of them "under the hood."

Important: Because our model is now a wrapper, the hyperparameter `C` belongs to the *base estimator*, not the wrapper. When we build our `GridSearchCV`, the parameter name changes to `model__estimator__C`.

```
# 1. Create the pipeline for OvR
# We pass a *new* LogisticRegression() model *inside* the wrapper
pipe_ovr = Pipeline([
    ('scaler', StandardScaler()),
    ('model', OneVsRestClassifier(LogisticRegression(solver='liblinear')))
])

# 2. Define the grid search parameters
# Note the new parameter name: 'model__estimator__C'
param_grid_ovr = {
    'model__estimator__C': np.logspace(-3, 3, 7)
}

# 3. Run Grid Search
print("\nRunning Grid Search for One-vs-Rest (OvR)...")
search_ovr = GridSearchCV(pipe_ovr, param_grid_ovr, cv=5, scoring='accuracy')
search_ovr.fit(X_train_i, y_train_i)

print(f"Best 'C' for OvR: {search_ovr.best_params_['model__estimator__C']}")
print(f"Best CV Accuracy: {search_ovr.best_score_:.4f}")
```

```
Running Grid Search for One-vs-Rest (OvR)...
Best 'C' for OvR: 10.0
Best CV Accuracy: 0.9714
```

Final Comparison: multinomial against one-vs-rest

Now we do our final, unbiased evaluation on the **test set**.

```
# 1. Get predictions from the best multinomial model
y_pred_multi = search_multi.predict(X_test_i)
acc_multi = accuracy_score(y_test_i, y_pred_multi)

# 2. Get predictions from the best OvR model
y_pred_ovr = search_ovr.predict(X_test_i)
acc_ovr = accuracy_score(y_test_i, y_pred_ovr)

# 3. Create a results table
results = {
    "Multinomial": acc_multi,
    "One-vs-Rest (OvR)": acc_ovr
}
results_df = pd.DataFrame.from_dict(results, orient='index', columns=['Test
        Accuracy'])

print("\n--- Multi-Class Model Comparison ---")
print(results_df.sort_values(by='Test Accuracy', ascending=False))

print("\n--- Report for the Multinomial Model ---")
print(classification_report(y_test_i, y_pred_multi, target_names=target_names_i))

print("\n--- Report for One-vs-Rest (OvR) Model ---")
print(classification_report(y_test_i, y_pred_ovr, target_names=target_names_i))
```

```
--- Multi-Class Model Comparison ---
                  Test Accuracy
Multinomial            0.911111
One-vs-Rest (OvR)      0.866667

--- Report for the Multinomial Model ---
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.82      0.93      0.88        15
   virginica       0.92      0.80      0.86        15

    accuracy                           0.91        45
   macro avg       0.92      0.91      0.91        45
weighted avg       0.92      0.91      0.91        45


--- Report for One-vs-Rest (OvR) Model ---
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.80      0.80      0.80        15
   virginica       0.80      0.80      0.80        15

    accuracy                           0.87        45
   macro avg       0.87      0.87      0.87        45
weighted avg       0.87      0.87      0.87        45
```

Exercises

1. **Beyond accuracy.** On an imbalanced problem, build a classifier that achieves high accuracy and is useless. Report its precision, recall and confusion matrix, and say which metric you would put in a report and why.
2. **Threshold.** Take a logistic regression and vary the decision threshold from 0 to 1. Plot precision and recall against the threshold. Choose a threshold for (a) a spam filter and (b) a screening test, and justify each choice in terms of the cost of each error.
3. **Choosing K.** Cross-validate KNN for K from 1 to 50. Plot the curve. Explain the behaviour at both ends in terms of the bias-variance tradeoff.
4. **Scaling and distance.** Run KNN with and without standardising the features on a dataset whose variables have very different units. Explain the difference in terms of how the distance is computed.
5. **Multinomial against one-vs-rest.** Fit both to a three-class problem. Compare the cross-validated scores and the number of parameters. Are the predictions identical? Where do they disagree?

References

- Evaluation Measures for Classifcation: https://en.wikipedia.org/wiki/Precision_and_recall
- Nearest Neighbor: Section 2.5.2 of [1]

[1] Bishop, Christopher M. *Pattern recognition and machine learning*. springer, 2006. https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Generative classifiers: QDA, LDA and naive Bayes

Chapter resources

Slides (PDF) Open in Colab Practice quiz

In previous lectures, we have seen **discriminative models** (like Logistic Regression). These models have one goal: learn the *decision boundary* between classes. They directly model the conditional probability $P(Y = k|X = \mathbf{x})$.

Recall that in this stage Y is discrete and finite (our classes), while X can be either continuous or discrete, scalar or multidimensional.

In this lecture, we will learn about a completely different family of classifiers: **generative models**, and in particular, we will see the **Naive Bayes** model.

Generative models do not explicitly model the decision boundary. Instead, their goal is to **learn the "story" or "profile" of each class independently**. They aim to build a full probabilistic model of what each class *looks like*.

Technically, this means they model the **joint probability distribution** $P(X, Y)$. Recalling that $P(X, Y) = P(X|Y)P(Y)$, they do this by modeling two separate pieces:

1. **The Likelihood** $P(X = \mathbf{x}|Y = k)$**:** "What does a typical X look like *for this class*?"
2. **The Prior** $P(Y = k)$**:** "How common is *this class* in general?"

Once the model has learned these two things, it can be used for classification.

The Core Principle: Maximum A Posteriori (MAP)

Generative classifiers are built on the **Maximum A Posteriori (MAP)** principle. We want to find the class k that is *most probable* after we observe the data $\mathbf{x}$.

$$h(\mathbf{x}) = \arg\max_k P(Y = k|X = \mathbf{x})$$

Using Bayes' theorem, we can "flip" this equation to use the *generative* pieces we've learned:

$$h(\mathbf{x}) = \arg\max_k \frac{P(X = \mathbf{x}|Y = k)P(Y = k)}{P(X = \mathbf{x})}$$

We note that, if we are not interested in computing the actual probabilities $P(Y = k|\mathbf{x})$, but we only want to assign $\mathbf{x}$ to the most likely class, we can drop the evidence $P(X = \mathbf{x})$, which is independent of class k. Indeed we note that:

$$P(Y = k|\mathbf{x}) \propto P(X|Y = k)P(Y = k)$$

Which leads to the final MAP classification rule:

$$h(\mathbf{x}) = \arg\max_k \underbrace{P(X|Y = k)}_{\text{Likelihood}}\underbrace{P(Y = k)}_{\text{Prior}}$$

This approach is known as Maximum A Posteriori (MAP) classification as we aim to maximize the posterior probability.

In order to implement this principle, we need to compute the following two quantities:

- The likelihood $P(X|Y)$;
- The prior $P(Y)$;

We will now see how to compute each of these quantities.

The Prior $P(Y)$

$P(Y)$: this is the **prior probability of a given class**. If observing a class $Y = k$ is not very common, then $P(Y = k)$ will be small. We can use different approaches to estimate $P(Y = k)$:

- We can estimate $P(Y = k)$ by **considering the number of examples in the dataset**. For instance, if our dataset contains 800 non-spam e-mails and 200 spam e-mails and we set `Class 0 = Spam` and `Class 1 = Non Spam`, we can assume that $P(Y = 0) = 0.2$ and $P(Y = 1) = 0.8$.
- Alternatively, we could ***study* what is the proportion of examples in each class in the real world**. In the case of spam detection, we could ask a large sample of people how many e-mails they receive in average and how many spam e-mails they receive. These numbers can be used to define the prior probability.
- Another common choice, when we don't have enough information on the phenomenon is to **assume that all classes are equally probable**, in which case $P(Y = k) = \frac{1}{m}$, where m is the number of classes.

There are many ways to define the prior probability. However, it should be considered that this quantity should be interpreted in Bayesian terms. This means that, by specifying a prior probability, we are introducing our **degree of belief** on what classes are more or less likely in the system.

The Likelihood $P(X|Y)$

While estimating the prior is easy and estimating the evidence is not necessary for classification purposes (it would be indeed necessary if we were to compute probabilities), computing the likelihood term is less straightforward.

If we have M different classes, a general approach to estimate the likelihood consists in group all observations belonging to a given class $Y = k$ (let's call X_k the random variable of the examples belonging to this group) and estimate the probability $P(X_k)$.

If we repeat this process for every possible value of C, we have concretely estimated $P(X|Y = k)$ as:

$$P(X = x|Y = k) = P(X_k)$$

To estimate $P(X_k)$ we will generally need to make a few assumptions. Depending on such assumptions, we obtain different generative models.

The "Likelihood" Problem: From Ideal to Practical

The MAP rule is simple: $h(\mathbf{x}) = \arg\max_k P(X = \mathbf{x}|Y = k)P(Y = k)$.

The prior $P(Y = k)$ is easy to estimate (we just count). The *entire* challenge of generative modeling lies in estimating the **Likelihood,** $P(X = \mathbf{x}|Y = k)$.

How we do this depends on our features, and this is where we run (again) into the **Curse of Dimensionality**.

Guiding Example: Spam Detector

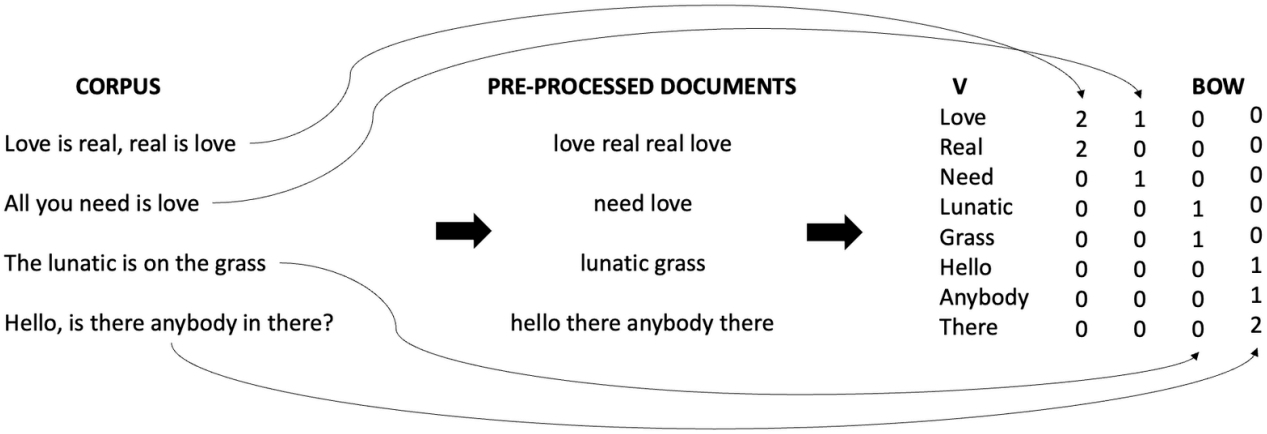
Spam detection is framed as a generative classification task, where the goal is to determine whether an incoming email is "spam" or "ham" based on its features.

Each email is represented as a high-dimensional vector, with features derived from word frequencies across a potentially massive vocabulary—often tens of thousands of words.

Since most emails contain only a small subset of these words, the resulting feature vectors are typically sparse.

This sparsity and dimensionality make estimating the likelihood function $P(X \mid Y = k)$ challenging, requiring strategic simplifications to model the data effectively.

The figure below shows how a text is converted into a vector of word counts. Note that in this process, we usually remove some less meaningful works such as articles and prepositions, which are usually called "stop words":



The “Ideal” Model for Discrete Data (and its Limitations)

If our features are discrete (e.g., `Offer=Yes`, `Free=No`), we could try to model $P(X|Y)$ by building a giant contingency table for every single combination of features.

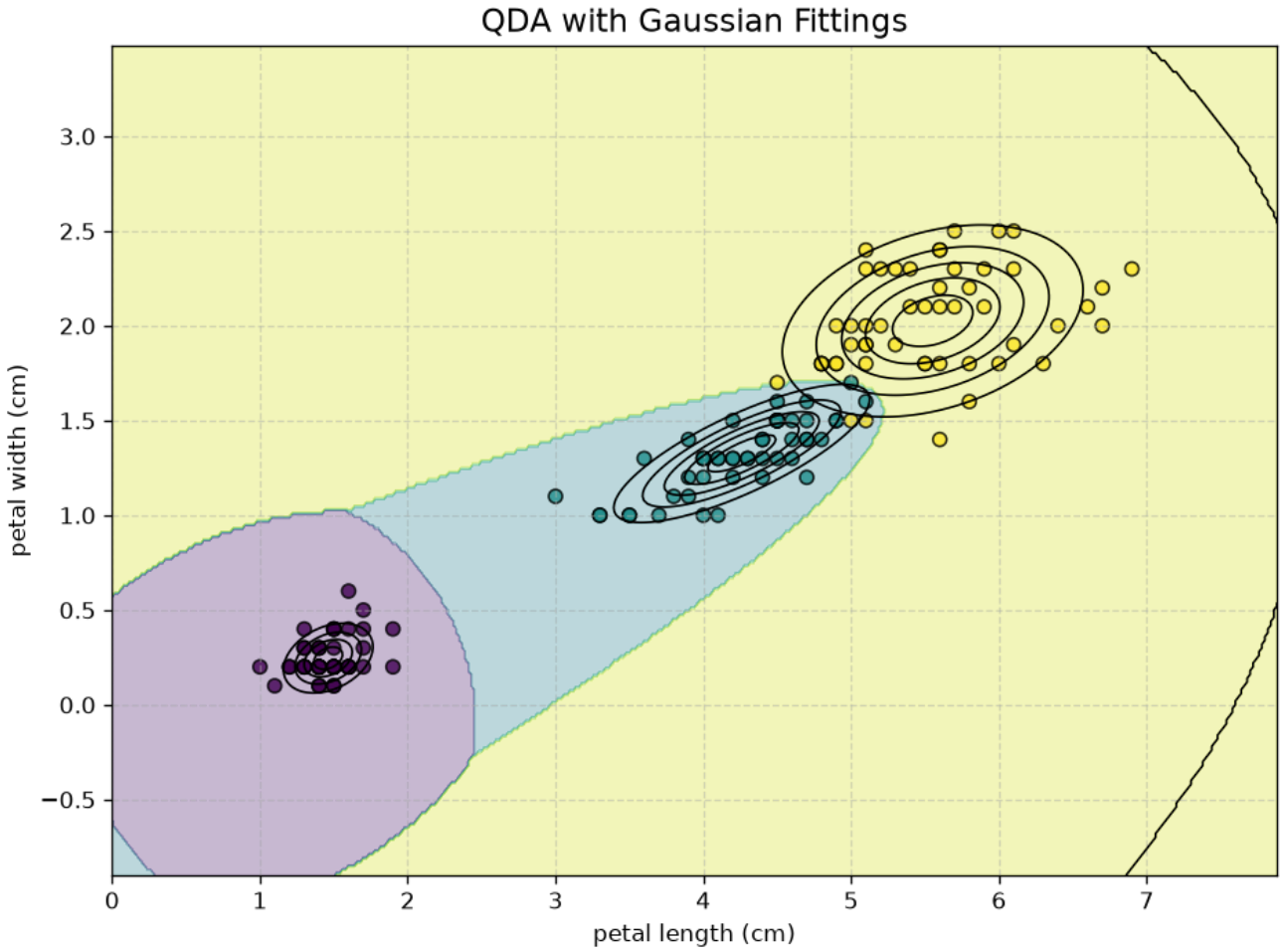
- **The Problem:** If we have 10,000 words (features) in our vocabulary, we would need to estimate the probability for $2^{10,000}$ possible combinations. This is computationally and statistically impossible.

The “Ideal” Model for Continuous Data: QDA

If our features are continuous (like the Iris dataset), we can’t build a table. The “ideal” approach is to model $P(X = \mathbf{x}|Y = k)$ as a **Multivariate Gaussian (Normal) Distribution**, $N(\mu_k, \mathbf{\Sigma}_k)$.

This is **Quadratic Discriminant Analysis (QDA)**. It’s the most flexible Gaussian model.

- **Assumption:** Each class k has its *own* mean vector μ_k and its *own, full* covariance matrix $\mathbf{\Sigma}_k$.
- **The “Nightmare”:** This is computationally a nightmare. To estimate a full covariance matrix $\mathbf{\Sigma}_k$ for d features, we must estimate $\sim \frac{d^2}{2}$ parameters. For 100 features, that’s $\approx 5,000$ parameters, *per class*. This requires massive amounts of data and overfits easily.
- **The Result:** A powerful, *curved (quadratic)* decision boundary.



QDA produces a powerful, curved (quadratic) decision boundary between classes, reflecting its flexibility in modelling distinct class distributions.

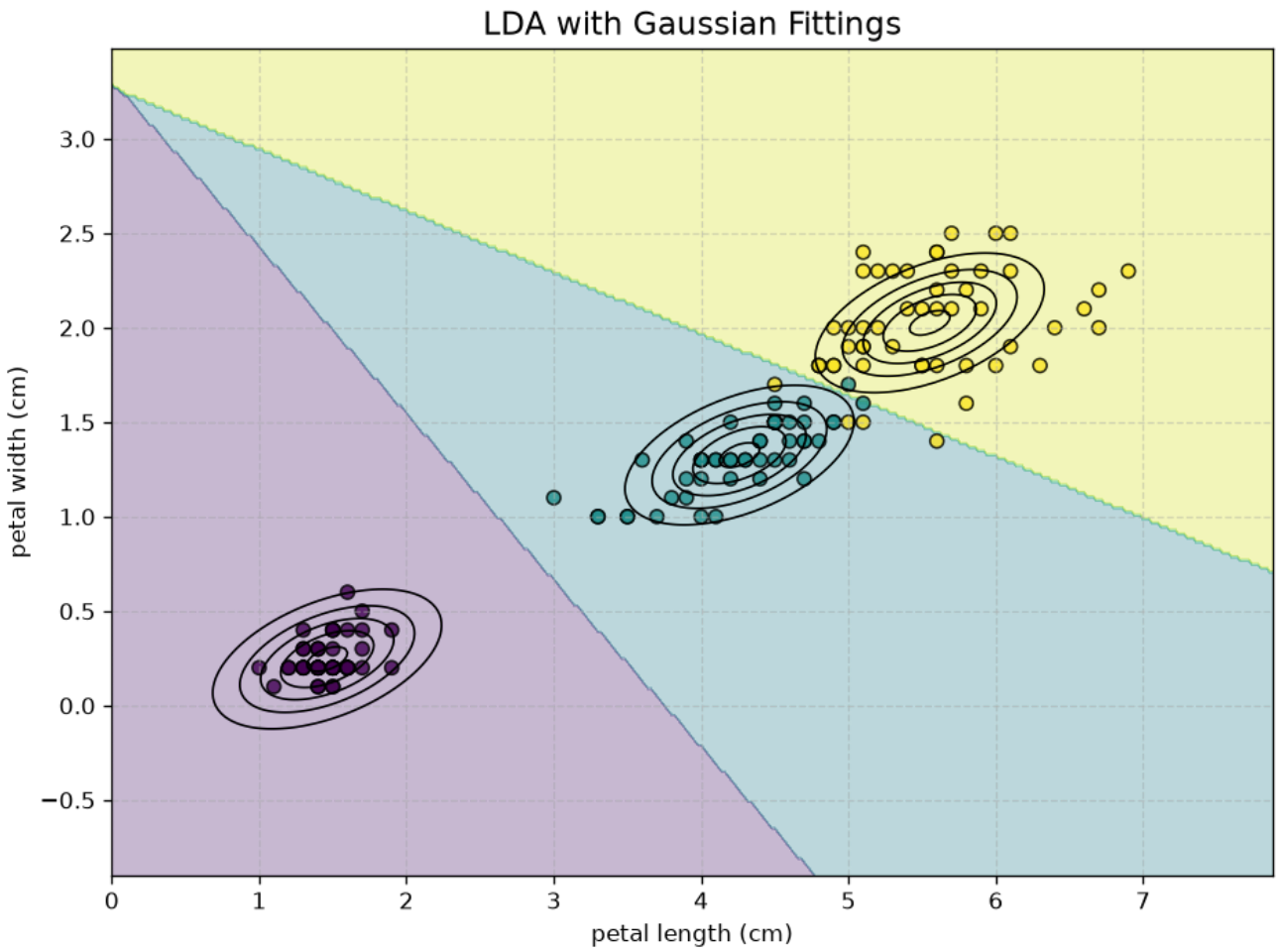
When we have enough data, QDA can lead to good results, but it can easily overfit when we don't have enough data.

Note that each class has its own Gaussian distribution and that distributions can be very different from each other.

The First Simplification: Linear Discriminant Analysis (LDA)

The "nightmare" of QDA leads to our first compromise: **Linear Discriminant Analysis (LDA)**.

- **Assumption:** We simplify the problem by assuming all classes *share* the **same covariance matrix** ($\boldsymbol{\Sigma}_k = \boldsymbol{\Sigma}$ for all k).
- **The Benefit:** This is much more stable. We only have to estimate *one* full $\boldsymbol{\Sigma}$ matrix, not k of them.
- **The "Nightmare" (Still):** We *still* have to estimate one full $\frac{d^2}{2}$ matrix. For text data with $d = 10,000$, this is still impossible.
- **The Result:** This assumption forces the math to simplify, and the decision boundary is always *linear*.



This critical assumption forces the mathematical derivations to simplify, resulting in a decision boundary that is always linear.

While less flexible than QDA's quadratic boundaries, LDA's linear approach is more robust and less prone to overfitting when data is scarce or dimensionality is high.

Note how now each class has the same "shape" (covariance). This makes the model less flexible and leads to a linear decision boundary.

Naive Bayes: The Pragmatic Solution

While LDA makes it easier to fit the model to the data, it can still be very computationally demanding.

Indeed, for **very high-dimensional data** ($d = 10,000$ features, like in text classification), estimating even *one* full covariance matrix $\hat{\boldsymbol{\Sigma}}$ can be computationally and statistically challenging. This is an aspect of the **Curse of Dimensionality**.

This leads us to our final and most extreme simplification: **Naive Bayes**.

The "Naive" Assumption

Recall that our problem is estimating

$$P(X|Y)$$

Considering a multidimensional $X = (X_1, X_2, \ldots, X_n)$, we may factorize the expression above as follows:

$$P(X|Y) = P(X_1, \ldots, X_n|Y) = P(X_1|Y)\, P(X_2|X_1, Y)\, P(X_3|X_2, X_1, Y) \ldots P(X_n|X_1, ..$$

However, this is not a very helpful factorization as the terms $P(X_i|X_1, \ldots, X_{i-1}, C)$ are conditioned also on other features, which makes them not easy to model.

The Naive Bayes classifier makes one, very strong ("naive") assumption:

It assumes that all features X_i are **conditionally independent** given the class $Y = k$.

This can be written as follows:

$$X_i \perp X_j \mid Y,\ \forall i \neq j$$

Spam Classification Example Let's assume a spam classification example and imagine that features X_i and X_j are related to the number of appearance of specific words in the current email represented by X. If X_i and X_j are **word counts**, then we are saying that if I take all non-spam e-mails, then the number of occurrences of a given word **does not influence** the number of occurrences of **another word**. This is **obviously not true in general!** For instance, there can be legitimate e-mails of different topics. If the e-mail is about a vacation, the words 'trip', 'flight', 'luggage' will appear often. Instead, if the e-mail is about work, the words 'meeting', 'report', 'time', will appear more often. This means that, within the same category (non-spam e-mails), the number of occurrences of a word (e.g., 'trip') may be related to the number of occurrences of another words (e.g., 'flight'), which breaks the assumption of **conditional independence**. This is why this assumption is called **naïve assumption**. With this in mind, it should be considered that, despite such naïve assumption, the Naïve Bayes Classifier works surprisingly well in many contexts.

We know that:

$$X \perp Y|Z \Leftrightarrow P(X, Y|Z) = P(X|Z)\, P(Y|Z)$$

Hence, we discover that, under the assumption of conditional independence:

$$P\left(X_1, \ldots, X_n|Y\right) = P\left(X_1|Y\right) P\left(X_2|Y\right) \ldots P(X_n|Y)$$

So, we can re-write the MAP classification rule as:

$$f(\mathbf{x}) = \arg_k max\ P\left(\mathbf{x}_1|Y=k\right)\left(\mathbf{x}_2|Y=k\right) \ldots P\left(\mathbf{x}_n|Y=k\right) P(Y=k)$$

Flavors of Naive Bayes

The single $P(X_1|Y=k)$ terms are now easy to model, since X_1 is mono-dimensional. In practice, depending on the considered problem, we can model these terms in different ways. Two common approaches, depending on the kind of data we have, are to use a **Gaussian distribution** or a **Multinomial distribution**.

When we use Gaussian distributions to model the $P(X_i|Y)$ terms, the classification method is called "**Gaussian Naïve Bayes**". Similarly, if we consider a multinomial distribution, the classification method is called "**Multinomial Naïve Bayes**".

Gaussian Naïve Bayes

Let us consider again our sex classification example based on height and weight. We will consider $X = [H, W]$, which are random variables representing heights and weights of subjects. If we assume that the data is approximately Gaussian, the probabilities $P(H|C)$ and $P(W|C)$ can be modeled with univariate (1D) Gaussian distributions. This is done by first obtaining four samples:

- H_1: the heights of subjects when $C = 1$;
- W_1: the weights of subjects when $C = 1$;
- H_0: the heights of subjects when $C = 0$;
- W_0: the weights of subjects when $C = 0$.

We hence model each sample as a 1D Gaussian distribution by computing a mean and a variance value from each of these samples to obtain four Gaussian distributions:

- $P\left(H = h|C = 0\right) = N(x; \mu_1, \sigma_1)$;
- $P\left(W = w|C = 0\right) = N(x; \mu_2, \sigma_2)$;
- $P\left(H = h|C = 1\right) = N(x; \mu_3, \sigma_3)$;
- $P\left(W = w|C = 1\right) = N(x; \mu_4, \sigma_4)$;

After this, we can apply the classification rule:

- The example (h, w) is classified as class 1 if

$$P(h|C=1)\,P(w|C=1)\,P(C=1) > P(h|C=0)\,P(w|C=0)\,P(C=0);$$

- The example (h, w) is classified as class 0 otherwise.

We can exemplify this process as follows:

$P(W|C)$



$P(H|C)$

Implications of the Naive Assumption in Gaussian Naive Bayes

Gaussian Naive Bayes assumes that features are conditionally independent given the class label. This leads to a **diagonal covariance matrix** for each class Σ_k, meaning that the off-diagonal entries—representing feature correlations—are zero. Mathematically, this is expressed as:

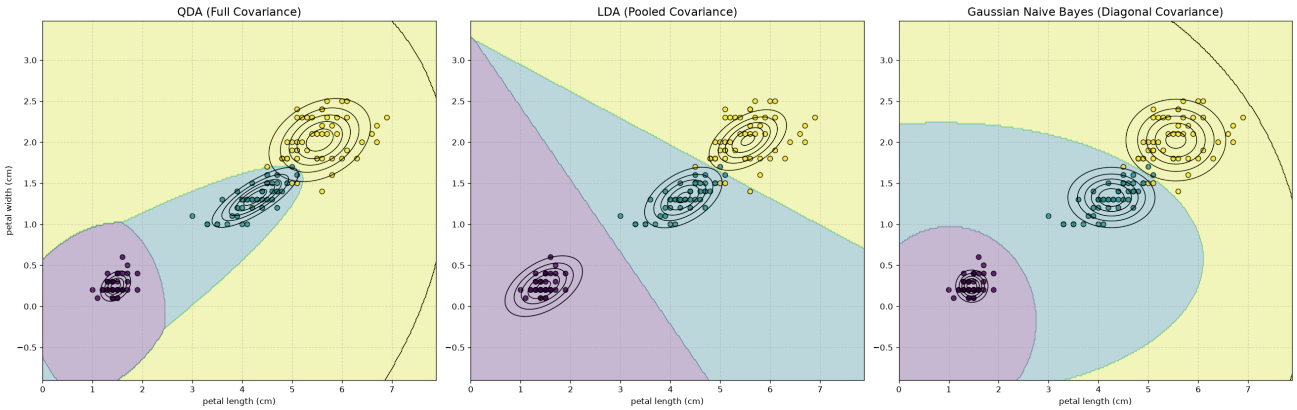
$$\Sigma_k = \begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \sigma_2^2 & 0 \\ 0 & 0 & \sigma_3^2 \end{bmatrix}$$

This structure implies that the model treats each feature as contributing independently to the likelihood $P(X \mid Y = k)$. As a result, the fitted Gaussian distributions are **axis-aligned**, meaning their contours are parallel to the coordinate axes. This simplifies computation and estimation, especially in high-dimensional spaces, but limits the model's ability to capture **intra-class correlations**.

Despite this simplification, Gaussian Naive Bayes allows each class to have its own diagonal covariance matrix. That is, $\Sigma_k \neq \Sigma_j$ for $k \neq j$, which enables the model to produce **non-linear decision boundaries**. These boundaries arise from differences in class-specific variances, even though the distributions themselves remain axis-aligned. However, the inability to model feature interactions within a class remains a core limitation of the naive assumption.

Decision Boundaries and Comparison with QDA and LDA

Let's compare the decision boundaries of the three classifiers:



This figure perfectly summarizes the three models:

- **QDA (Left):** Most flexible. It learns a separate, *rotated* ellipse (full $\boldsymbol{\Sigma}_k$) for each class. Its boundaries are quadratic.
- **LDA (Middle):** The compromise. It learns *one* pooled, *rotated* ellipse (full $\boldsymbol{\Sigma}$) and uses it for all classes. Its boundaries are linear.
- **GNB (Right):** Most simple. It learns a separate, *axis-aligned* ellipse (diagonal $\boldsymbol{\Sigma}_k$) for each class. Its boundaries are quadratic but can't capture any feature correlation (tilt).

Multinomial Naive Bayes (MNB)

In the previous section, we saw **Gaussian Naive Bayes (GNB)**, which is ideal for **continuous features** (like `petal_length`) that we can model with a Gaussian (bell curve).

When the input features are **discrete**, we must use a different "flavor" of Naive Bayes. The most common case is **text classification** (e.g., spam filtering).

If we represent text using **word counts** (a "Bag of Words" or BoW representation), our input features are a vector of discrete, natural numbers:

$$\mathbf{x} = (x_1, x_2, x_3, \dots, x_d)$$

where d is the size of our vocabulary (i.e., $d = |V|$) and $x_i \in \mathbb{N}$ is the number of times the i-th word in the vocabulary appears in the document $\mathbf{x}$.

Our goal is to model the likelihood, $P(\mathbf{x}|Y = k)$, which is:

$$P(x_1, x_2, \dots, x_d|Y = k)$$

If we imagine the process of "generating" a document as pulling n words out of a "bag" (our vocabulary V), we can model this probability $P(\mathbf{x}|Y = k)$ with a **Multinomial Distribution**. We will need to estimate a different Multinomial distribution for each class k.

The Multinomial Distribution: Remember that the multinomial distribution models the probability of obtaining exactly $(\mathbf{x}_1, \dots, \mathbf{x}_d)$ occurrences for each of the d possible outcomes (words), in a sequence of n independent experiments (total words), where each experiment follows a categorical distribution with probabilities $(p_{k1}, \dots, p_{kd})$.

In our case:

- $n = \sum_{i=1}^{d} x_i$: The total number of words in the document $\mathbf{x}$.
- d **possible outcomes**: The d words in the vocabulary.
- p_{ki}: The probability of randomly picking word i, given that we are in class k.

Fitting the Multinomial Distribution

Using the analytical form of the multinomial distribution of chapter 5, we can write the full likelihood:

$$P(\mathbf{x}|Y = k) = \frac{(\sum_{i=1}^{d} x_i)!}{x_1! \dots x_d!} \cdot p_{k1}^{x_1} \cdot p_{k2}^{x_2} \cdot \dots \cdot p_{kd}^{x_d}$$

Now, we plug this into our MAP classification rule:

$$h(\mathbf{x}) = \arg\max_k P(Y = k) \cdot \frac{(\sum_{i=1}^{d} x_i)!}{x_1! \dots x_d!} \prod_{i=1}^{d} (p_{ki})^{x_i}$$

We can immediately simplify this. The big factorial term

$$\frac{(\sum_{i=1}^{d} x_i)!}{x_1! \dots x_d!}$$

...is a constant *with respect to* k. It only depends on the input $\mathbf{x}$, not the class. Since we are just trying to find the `arg max`, we can drop this term entirely, as it doesn't change which k gives the highest score.

This leaves us with a much simpler (and faster) classification rule:

$$h(\mathbf{x}) = \arg\max_k P(Y = k) \prod_{i=1}^{d} (p_{ki})^{x_i}$$

Estimating the Parameters

To use this model, we must estimate two sets of parameters from our training data: the priors $P(Y = k)$ and the word likelihoods p_{ki}.

Estimating the Word Likelihoods p_{ki}

We estimate the probability of word i for class k by counting:

$$p_{ki} = \frac{\text{Total count of word } i \text{ in all documents of class } k}{\text{Total count of *all words* in all documents of class } k}$$

Using the Iverson bracket notation of chapter 15, where $[y^{(j)} = k]$ is 1 if true and 0 otherwise:

$$p_{ki} = \frac{\sum_j [y^{(j)} = k] x_i^{(j)}}{\sum_{l=1}^{d} \left(\sum_j [y^{(j)} = k] x_l^{(j)} \right)}$$

- **Numerator:** Sums the counts of word i *only* in documents j from class k.
- **Denominator:** Sums the counts of *all* words ($l = 1$ to d) in *all* documents j from class k.

Estimating the Prior $P(Y = k)$

This is the same as before. We estimate the prior as the fraction of documents in our training set that belong to class k:

$$P(Y = k) = \frac{\sum_j [y^{(j)} = k]}{N}$$

Where N is the total number of documents.

Practical "Gotchas": Two Critical Fixes

If we use the formulas above directly, our model will fail. We need to fix two major problems.

Problem 1: The Zero-Probability Problem

What happens if we get a new email that contains the word "crypto"? If our training data for `Class='Spam'` *never* contained the word "crypto," then its estimated probability is: $p_{k,\text{'crypto'}} = 0$

When we classify this email, our MAP rule will multiply by this zero:

$$h(\mathbf{x}) \propto P(Y = \text{Spam}) \cdot \ldots \cdot p_{k,\text{'crypto'}}^{x_{\text{crypto}}} \cdot \ldots = P(Y = \text{Spam}) \cdot \ldots \cdot 0 \cdot \ldots = 0$$

Because of this one **zero**, the entire score for the "Spam" class becomes zero. The model breaks.

The Solution: Laplace (Add-One) Smoothing The fix is simple: we pretend we've seen every word in our vocabulary *at least one time* before we start. This is called **Laplace Smoothing**.

We add 1 (the "pseudo-count") to the numerator, and we add $d = |V|$ (the vocabulary size) to the denominator to keep the probabilities summing to 1. This ensures no probability is ever zero.

$$p_{ki} = \frac{\left(\sum_j [y^{(j)} = k] x_i^{(j)}\right) + 1}{\left(\sum_{l=1}^{d} \sum_j [y^{(j)} = k] x_l^{(j)}\right) + |V|}$$

Problem 2: Arithmetic Underflow

Our final rule multiplies thousands of tiny probabilities together (e.g., $0.001 \times 0.0004 \times \ldots$). This number gets so small, so fast, that the computer runs out of precision and rounds it to 0. This is called **arithmetic underflow**.

The Solution: Log-Probabilities Since the $\log$ function is monotonic (it doesn't change the `arg max`), we can maximize the *log-probability* instead.

Using the log-rule $\log(a \cdot b) = \log(a) + \log(b)$ and $\log(p^x) = x \cdot \log(p)$, we get:

$$h(\mathbf{x}) = \arg\max_k \left(\log P(Y = k) + \sum_{i=1}^{d} x_i \cdot \log p_{ki} \right)$$

This is the **final classification rule** that is actually implemented in `scikit-learn`. It's computationally stable, extremely fast (it's just a sum!), and is the foundation of modern text classification.

Lab Bake-Off: The Right vs. The Wrong Tool for the Job

We've now seen a "family" of generative models (QDA, LDA, GNB) and know about discriminative models like KNN and Logistic Regression.

Let's run a "bake-off" to see what happens when we apply these models to a **high-dimensional text classification** problem (spam filtering). This will clearly demonstrate *why* `Multinomial Naive Bayes (MNB)` is a standard tool for this task, and why the other models we've learned are the **wrong tool for the job**.

The Competitors:

1. **Text-Native Models:**
   - `MultinomialNB` (Generative, for counts)
   - `LogisticRegression` (Discriminative, works well with high-d text)
2. **The "Wrong" Models:**
   - `KNeighborsClassifier` (Distance-based, suffers from Curse of Dimensionality)
   - `GaussianNB` (Generative, assumes features are Gaussian, not counts)
   - `LinearDiscriminantAnalysis` (Generative, needs a full covariance matrix)
   - `QuadraticDiscriminantAnalysis` (Generative, needs *multiple* full covariance matrices)

The Workflow:

1. Load and clean the SMS Spam dataset.
2. Create a single **Train/Test Split**.

3. Build a `Pipeline` for each model. This is crucial:
   - `MNB` will use `CountVectorizer` (raw counts).
   - All other models will use `TfidfVectorizer` (normalized features).
   - The Gaussian models (GNB, LDA, QDA) *cannot* work with the default "sparse" matrix from a vectorizer, so we must build a custom "hack" to make them dense, which will highlight their inefficiency.
4. Fit all models, capture their accuracy and training time, and catch the errors for the models that will (and should) fail.
5. Compare all results in a final table and plot.

```
# --- 1. All Imports ---
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

# For data loading
import io
import zipfile
import urllib.request

# Sklearn Workflow Tools
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Sklearn Feature Engineering
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.base import TransformerMixin, BaseEstimator # For our custom transformer

# Sklearn Models (The Competitors)
from sklearn.naive_bayes import MultinomialNB, GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,
        QuadraticDiscriminantAnalysis
```

Step 1: Load and Split the Data

First, we load the data from the UCI zip file, map the labels, and create our single, "locked-away" test set.

```
# 1. Load the dataset
try:
    zip_data = data_bytes("sms_spam_collection.zip")

    # 2. Unzip the file in memory
    with zipfile.ZipFile(io.BytesIO(zip_data)) as z:
        # 3. Read the specific file from the zip archive into pandas
        with z.open('SMSSpamCollection') as f:
            df = pd.read_csv(f,
                             sep='\t',
                             header=None,
                             names=['label', 'message'],
                             encoding='latin-1')

    # 4. Map labels to 0 (ham) and 1 (spam)
    df['label'] = df['label'].map({'ham': 0, 'spam': 1})

    print("--- Data Head ---")
    print(df.head())

    # 5. Define X and y
    X = df['message'] # The raw text
    y = df['label']   # The 0 or 1

    # 6. Create the Train/Test Split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
        random_state=42, stratify=y)
    print(f"\nTraining samples: {len(X_train)}, Test samples: {len(X_test)}")

except Exception as e:
    print(f"Error loading or processing data: {e}")
```

```
--- Data Head ---
   label                                            message
0      0  Go until jurong point, crazy.. Available only ...
1      0                      Ok lar... Joking wif u oni...
2      1  Free entry in 2 a wkly comp to win FA Cup fina...
3      0  U dun say so early hor... U c already then say...
4      0  Nah I don't think he goes to usf, he lives aro...

Training samples: 3900, Test samples: 1672
```

Step 2: Define Pipelines and Custom Transformer

We need a custom transformer to convert the sparse output of `TfidfVectorizer` into a dense `numpy` array, which GNB, LDA, and QDA require. This is intentionally inefficient and will highlight the "nightmare" of using these models on high-dimensional data.

```
# Custom Transformer to convert sparse matrix to dense
class DenseTransformer(TransformerMixin, BaseEstimator):
    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.toarray()

# --- 1. Define the models we want to test ---
models_to_test = {
    "Multinomial NB (Text-Native)": Pipeline([
        ('vectorizer', CountVectorizer(stop_words='english')),
        ('classifier', MultinomialNB())
    ]),

    "Logistic Regression (Discriminative)": Pipeline([
        ('vectorizer', TfidfVectorizer(stop_words='english')),
        ('classifier', LogisticRegression(random_state=42))
    ]),

    "K-Neighbors (Distance-Based)": Pipeline([
        ('vectorizer', TfidfVectorizer(stop_words='english')),
        ('classifier', KNeighborsClassifier(n_neighbors=5))
    ]),

    "Gaussian NB (Wrong Assumption)": Pipeline([
        ('vectorizer', TfidfVectorizer(stop_words='english')),
        ('to_dense', DenseTransformer()), # Must be dense for GNB
        ('classifier', GaussianNB())
    ]),

    "Linear Discriminant (LDA)": Pipeline([
        ('vectorizer', TfidfVectorizer(stop_words='english')),
        ('to_dense', DenseTransformer()), # Must be dense for LDA
        ('classifier', LinearDiscriminantAnalysis())
    ]),

    "Quadratic Discriminant (QDA)": Pipeline([
        ('vectorizer', TfidfVectorizer(stop_words='english')),
        ('to_dense', DenseTransformer()), # Must be dense for QDA
        ('classifier', QuadraticDiscriminantAnalysis())
    ])
}

# --- 2. Store the results ---
results = {}
```

Step 3: Run the Bake-Off

Now we loop, fit, and time each model. Watch what happens when we try to run the models that aren't designed for this kind of data.

```
for name, pipeline in models_to_test.items():
    print(f"--- Testing: {name} ---")
    start_time = time.time()

    try:
        # 1. Fit the pipeline
        pipeline.fit(X_train, y_train)

        # 2. Make predictions on the TEST set
        y_pred = pipeline.predict(X_test)

        # 3. Get the accuracy
        accuracy = accuracy_score(y_test, y_pred)

        # 4. Store results
        elapsed = time.time() - start_time
        results[name] = {'Test Accuracy': accuracy, 'Time (sec)': elapsed, 'Error':
        None}
        print(f"  Success! Accuracy: {accuracy*100:.2f}%, Time: {elapsed:.2f}s")

    except Exception as e:
        # Catch models that fail to train
        elapsed = time.time() - start_time
        results[name] = {'Test Accuracy': 0.0, 'Time (sec)': elapsed, 'Error':
        str(e)}
        print(f"  FAILED! Error: {e}")
```

```
--- Testing: Multinomial NB (Text-Native) ---
  Success! Accuracy: 98.74%, Time: 0.05s
--- Testing: Logistic Regression (Discriminative) ---
  Success! Accuracy: 96.23%, Time: 0.06s
--- Testing: K-Neighbors (Distance-Based) ---
  Success! Accuracy: 90.79%, Time: 0.14s
--- Testing: Gaussian NB (Wrong Assumption) ---
  Success! Accuracy: 87.50%, Time: 0.44s
--- Testing: Linear Discriminant (LDA) ---
  Success! Accuracy: 97.49%, Time: 61.56s
--- Testing: Quadratic Discriminant (QDA) ---
  FAILED! Error: The covariance matrix of class 0 is not full rank. When using
`solver='svd'` the number of samples in each class should be more than the number
of features, but class 0 has 3377 samples and 6889 features. Try using
`solver='eigen'` and setting the parameter `shrinkage` for regularization.
```

Step 4: Show the Final Results

```
# Convert results to a pandas DataFrame for a "nice chart"
results_df = pd.DataFrame.from_dict(results, orient='index')
results_df = results_df.sort_values(by='Test Accuracy', ascending=False)

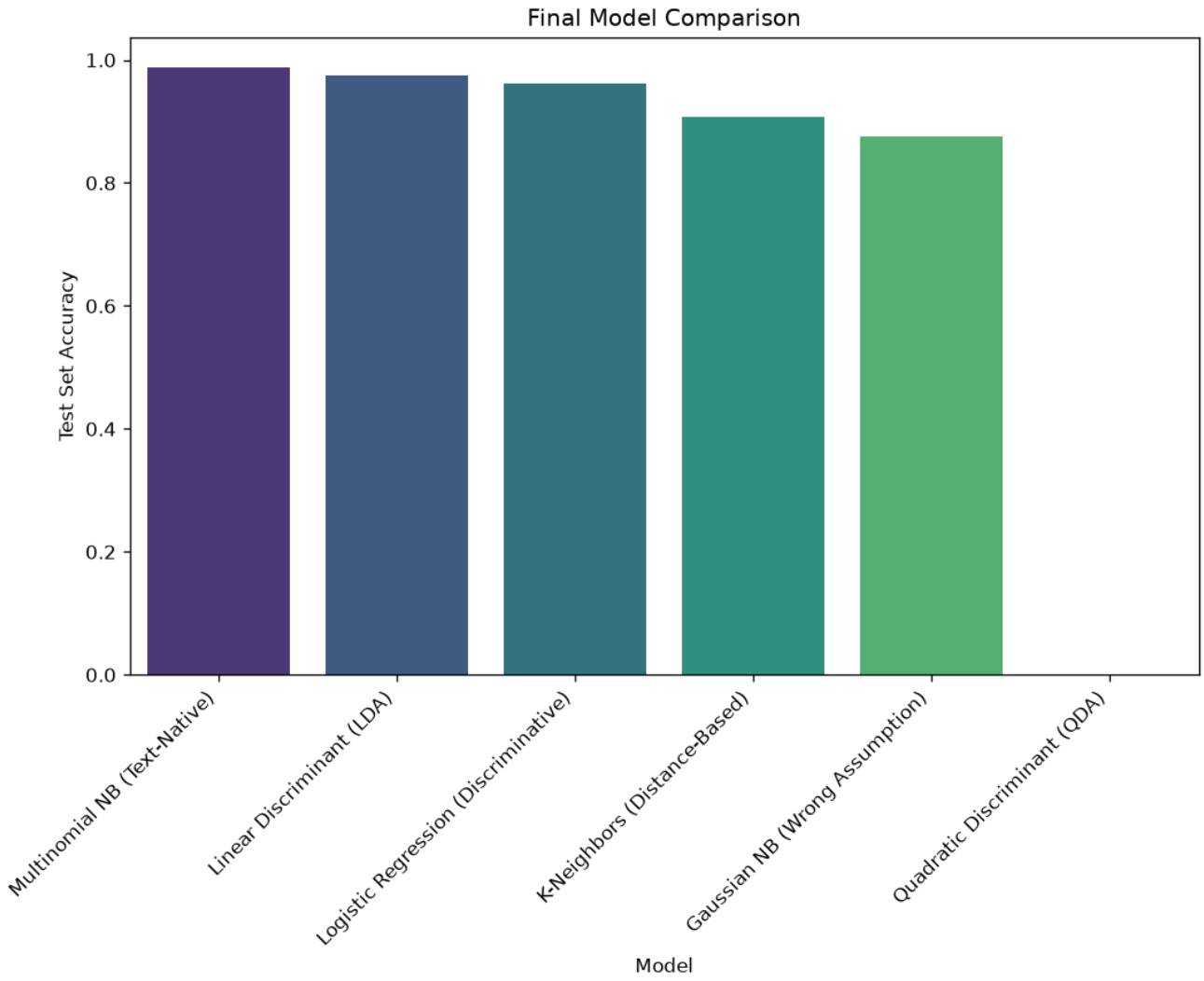
print("\n--- FINAL BAKE-OFF RANKING (Spam Filter) ---")
# We'll also show Time and Error to tell the full story
pd.set_option('display.max_colwidth', 100) # Show more of the error message
print(results_df)

# Plot the accuracy
plt.figure(figsize=(10, 6))
# We create a new column for plotting, setting failed models' accuracy to 0
plot_df = results_df.fillna(0)
sns.barplot(x=plot_df.index, y=plot_df['Test Accuracy'], hue=plot_df.index,
        palette='viridis', legend=False)
plt.xlabel('Model')
plt.ylabel('Test Set Accuracy')
plt.title('Final Model Comparison')
plt.xticks(rotation=45, ha='right')
plt.show()
```

```
--- FINAL BAKE-OFF RANKING (Spam Filter) ---
                                      Test Accuracy  Time (sec)  \
Multinomial NB (Text-Native)               0.987440    0.052082
Linear Discriminant (LDA)                  0.974880   61.555204
Logistic Regression (Discriminative)       0.962321    0.064013
K-Neighbors (Distance-Based)               0.907895    0.137356
Gaussian NB (Wrong Assumption)             0.875000    0.438121
Quadratic Discriminant (QDA)               0.000000   30.011519



Error
Multinomial NB (Text-Native)
None
Linear Discriminant (LDA)
None
Logistic Regression (Discriminative)
None
K-Neighbors (Distance-Based)
None
Gaussian NB (Wrong Assumption)
None
Quadratic Discriminant (QDA)          The covariance matrix of class 0 is not full
rank. When using `solver='svd'` the number of sampl...
```



Conclusion: The Right Tool for the Job

This bake-off tells a very clear story:

1. `Logistic Regression` **and** `MultinomialNB` were the **winners**. They are designed to handle high-dimensional, sparse text data and performed excellently.
2. `GaussianNB` performed poorly, but at least it ran. This is because our data is *counts* (mostly zeros), which does not follow a *Gaussian* (bell curve) distribution at all. This is a classic "Wrong Assumption" failure.
3. `KNeighborsClassifier` was also very inaccurate. This is a perfect demonstration of the **Curse of Dimensionality**. With over 7,000 features, the "distance" between any two messages becomes meaningless, so the "nearest neighbor" is no longer a reliable predictor.
4. `LDA` **and** `QDA` **took a long time to run and gave warnings**.

The final lesson is clear: You must choose a model that matches the assumptions and dimensionality of your data. For high-dimensional, sparse count data like text, **Multinomial Naive Bayes** and **Logistic Regression** are the correct, industry-standard tools.

Exercises

1. **Count the parameters.** For d features and K classes, write the number of parameters estimated by QDA, by LDA and by Gaussian naive Bayes. Evaluate each for $d = 10, K = 3$ and for $d = 1000, K = 3$.
2. **QDA against LDA.** Generate two classes with clearly different covariance matrices. Fit both models and plot the decision boundaries. Then shrink the difference between the covariances and watch the two boundaries converge.
3. **When naive is enough.** Compare Gaussian naive Bayes with LDA on a dataset with strongly correlated features. Naive Bayes is wrong about the correlations — does it classify worse? Explain the discrepancy.
4. **Priors matter.** Fit a generative classifier on an imbalanced dataset, then refit with `priors` set to uniform. How do the predictions change, and which choice is right if your training sample is not representative of the population?
5. **Zero counts.** Fit a multinomial naive Bayes on text with `alpha=0`. Find an input that produces a probability of exactly zero, and explain how Laplace smoothing fixes it.
6. **Generative or discriminative?** Compare naive Bayes and logistic regression on the same data at several training-set sizes. Which is better with very few examples, and which wins as the data grows?

References

- Naïve Bayes Classifier: https://en.wikipedia.org/wiki/Naive_Bayes_classifier;
- https://scikit-learn.org/stable/modules/lda_qda.html#lda-qda
- Section 4.4 of [1]

[1] James, Gareth Gareth Michael. An introduction to statistical learning: with applications in Python, 2023.https://www.statlearning.com

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Data representation

Chapter resources

Slides (PDF) Open in Colab Practice quiz

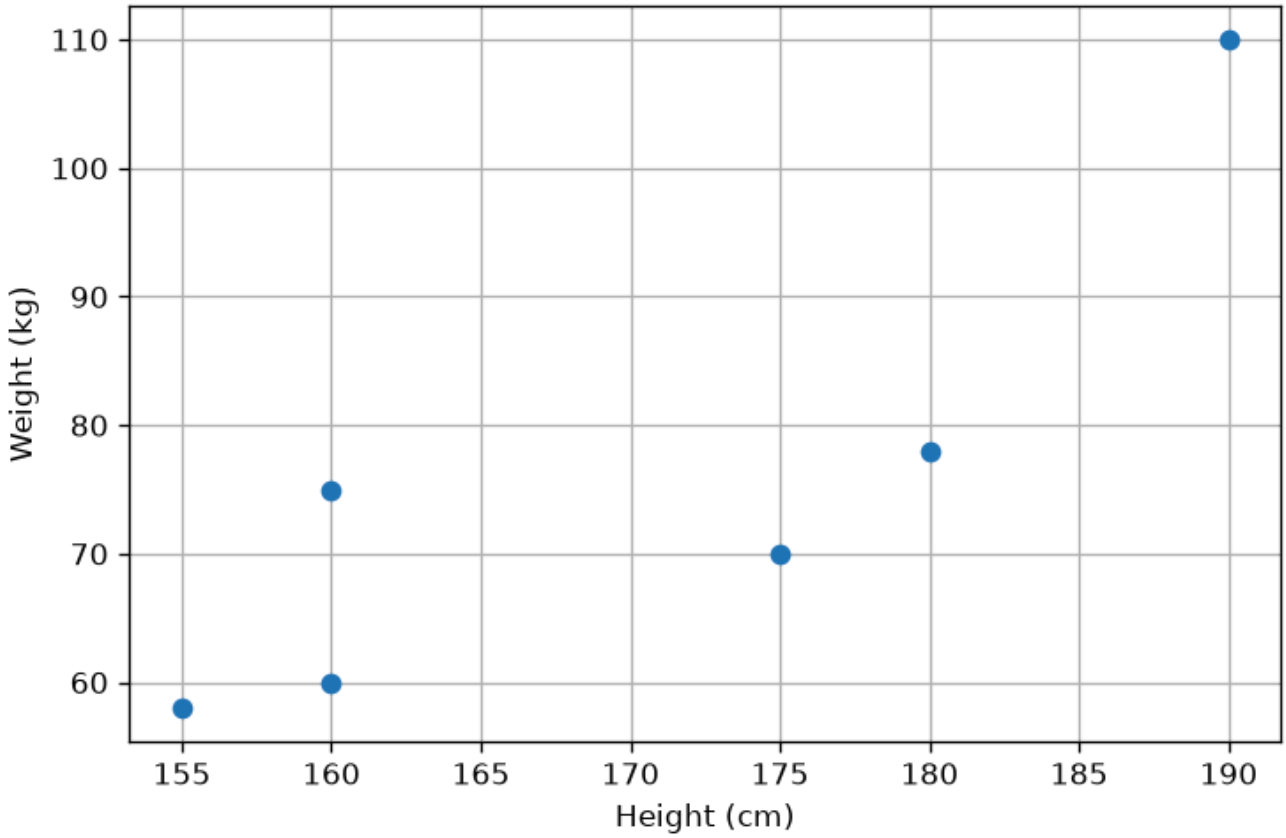
So far, we have seen data as a collection of observations of multiple variables. We often represented a dataset as a table or a data matrix. In all previous analyses, we have focused on the interpretation of the values of each variable.

However, in many cases (and especially when we have many variables), it is useful to given a geometric interpretation of data and see observations more generically as **d-dimensional data points**, not paying too much attention to variable names.

Consider the following data:

Subject	Height (cm)	Weight (Kg)
1	175	70
2	160	60
3	180	78
4	160	75
5	155	58
6	190	110

We can see each of these observations (the rows of the matrix) as a data point. Further, we can geometrically represent such points as follows:



In this context in which we are not always **explicitly giving names or interpretation to the variables**, each of the variables will also be referred to as **a feature of the data**. For instance, **a dataset of images of 100×100 pixels can be seen as a collection of 10000-dimensional vectors**, where each vector's dimension is a "feature", i.e., the value of a specific pixel of the image.

Given this interpretation of variables as features, it is often common to talk about **feature extraction**, in the case of any process transforming the data $\mathbf{x} \in \mathfrak{R}^d$ to another form $\mathbf{y} \in \mathfrak{R}^m$. In this context, a function:

$$f : \mathfrak{R}^d \rightarrow \mathfrak{R}^m$$

will be often referred to as a "**feature extraction function**" (of feature extractor) or a "**representation function**". We will recall these terms going further in the course. We will also refer to the process of mapping data using a representation function as **data representation**.

We will also refer to $\mathfrak{R}^d$ and $\mathfrak{R}^m$ as "**feature spaces**" or "**representation spaces**" as they will be **vector spaces** to which data points (also called **feature vectors**) belong.

Important Properties of Feature Spaces

Since observations $\mathbf{x}$ are vectors living in some vector spaces, all **basic properties of vectors and vector spaces seen in linear algebra will apply here as well**. We'll recall the most important concepts as we need them in the course, but, for the moment, it is useful to summarize the main important properties.

Norms

Given a vector space S, a norm p is a function from S to a non negative real number:

$$p : S \rightarrow \Re_0^+$$

A commonly used family of norms is the one of the L-p norms:

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{d} |x_i|^p \right)^{1/p}$$

Where $\mathbf{x}$ is d-dimensional, x_i is the i-th component of $\mathbf{x}$, and p is a parameter defining the behavior of the norm.

Note that a norm computes some kind of measure of distance of the vector from the origin of the space. In practice, the following norms are commonly used:

L-2 Norm

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{d} x_i^2}$$

This is what we know as "computing the magnitude (or modulus) of the vector". It is the Euclidean distance between the origin and the vector.

When we use the L2 norm, we can omit the $p = 2$ subscript:

$$||\mathbf{x}||_2 = ||\mathbf{x}||$$

Another commonly used notation is the one for the squared L2 norm:

$$||\mathbf{x}||_2^2 = ||\mathbf{x}||^2 = \sum_{i=1}^{d} x_i^2$$

L-1 Norm

$$||\mathbf{x}||_1 = \sum_{i=1}^{d} |x_i|$$

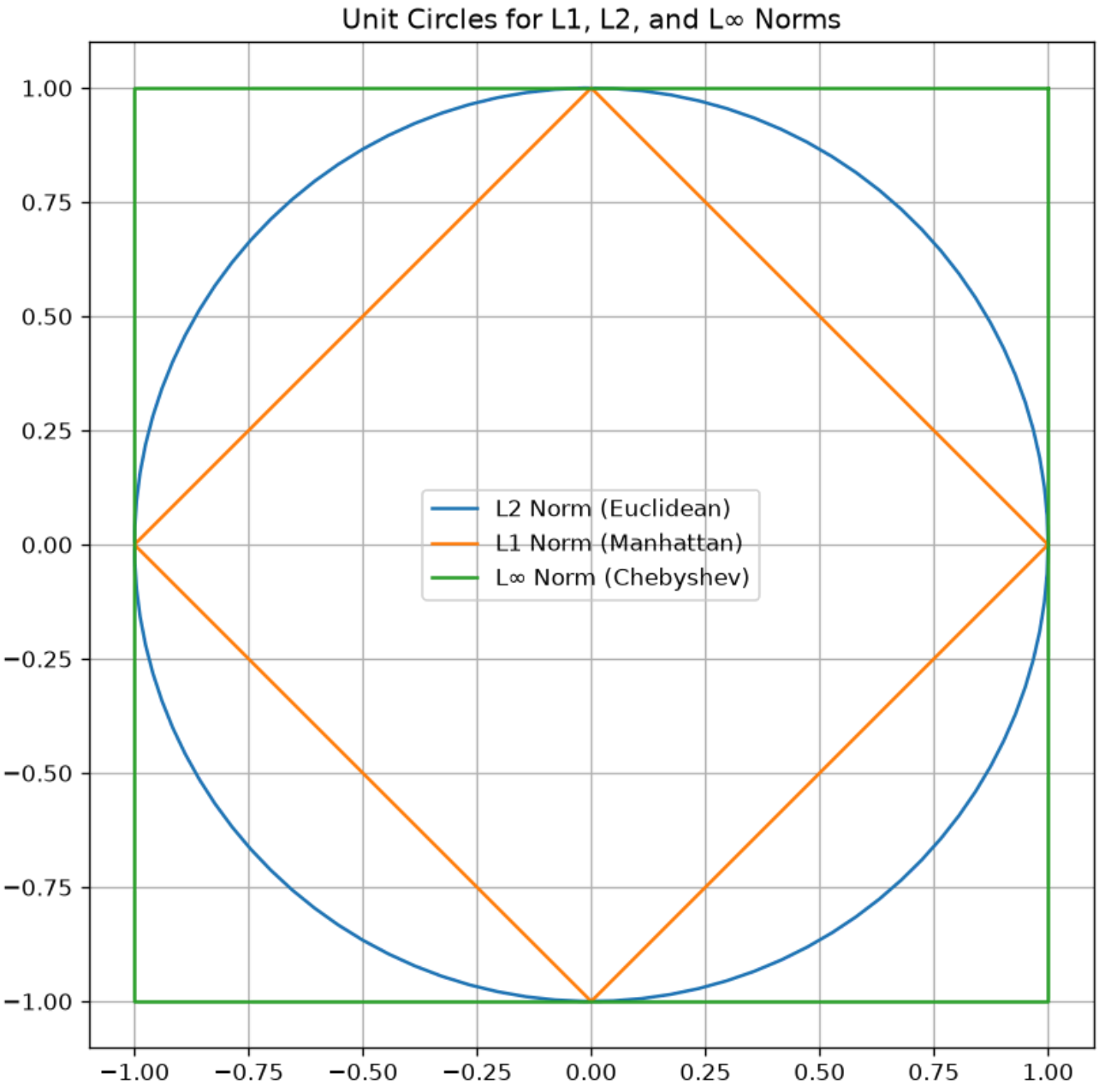
This is the sum of the absolute values of the components of the vector.

L-∞ Norm

$$||\mathbf{x}||_\infty = \max\{|x_i|\}_{i=1}^{d}$$

This is the maximum absolute value of the components of the vector.

To visualize the difference between the different norms, it often common to display the **unit circles according to the different norms**. Each of the shapes is made of **vectors with unit norm** according to specific norm:



Metrics

Given a space S, a function

$$m : S \times S \rightarrow \Re$$

is **a metric** if the following properties are satisfied:

1. The distance of a point from itself is zero: $m(\mathbf{x}, \mathbf{x}) = 0$;
2. The distance between two distinct points is always positive (positivity) if $\mathbf{x} \neq \mathbf{y}$: $m(\mathbf{x}, \mathbf{y}) > 0, \forall \mathbf{x}, \mathbf{y} \in S, \mathbf{x} \neq \mathbf{y}$;

3. The distance between x and y is the same as the distance between y and x (symmetry): $m(\mathbf{x}, \mathbf{y}) = m(\mathbf{y}, \mathbf{x})$.
4. Triangle inequality: $m(\mathbf{x}, \mathbf{y}) \leq m(\mathbf{x}, \mathbf{z}) + m(\mathbf{z}, \mathbf{y}), \forall \mathbf{x}, \mathbf{y}, \mathbf{z} \in S$

L-p Metrics

From the L-p norms, we can derive **a family of metrics** as follows:

$$L_p(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_p = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p}$$

L2 Distance

Note that with $p = 2$ we have the Euclidean (or L-2) distance:

$$L_2(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

L1 Distance

With $p = 1$, we obtain the L1 distance, which is also known as the **Manhattan distance**:

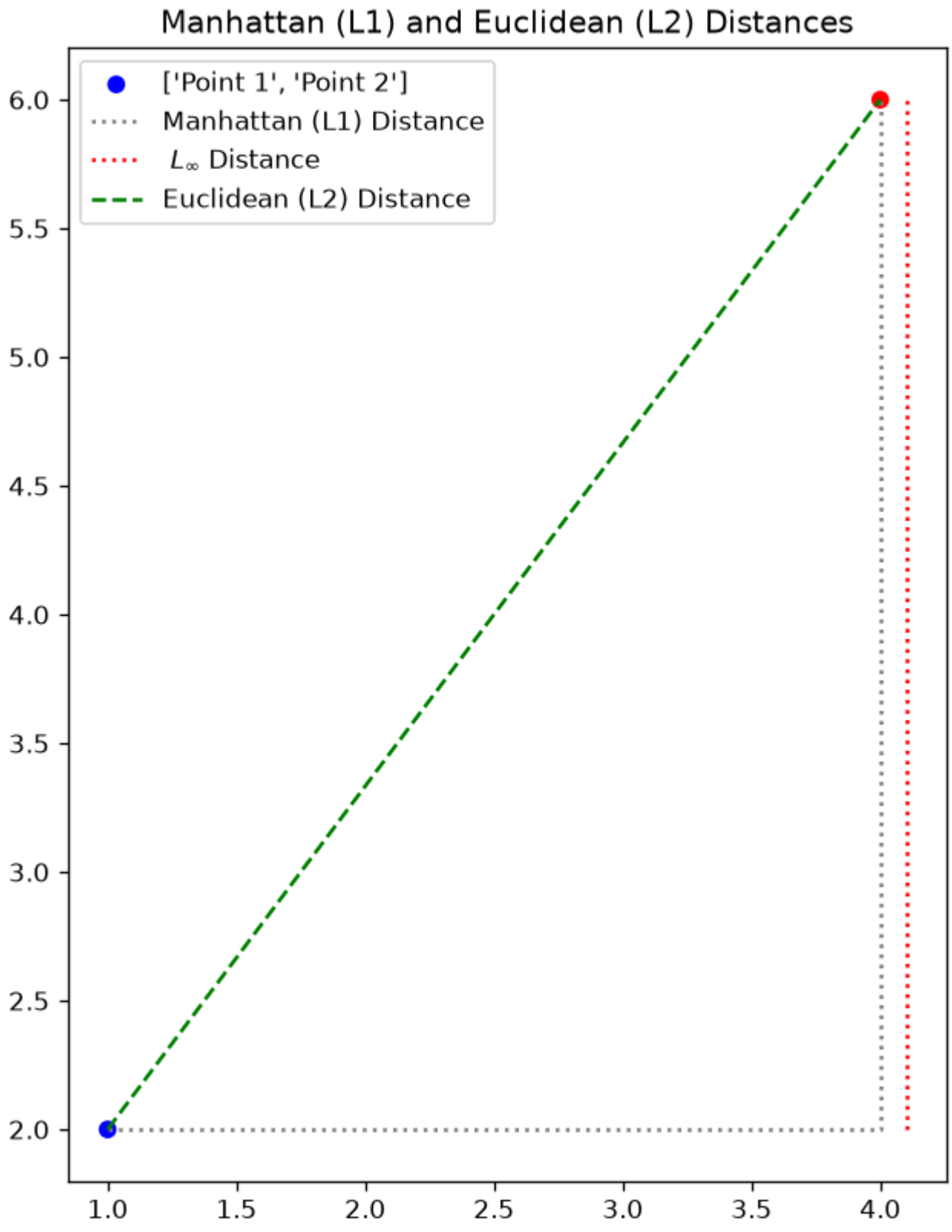
$$L_1(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_1 = \sum_{i=1}^{d} |x_i - y_i|$$

L-∞ Distance

With $p = \inf$, we obtain the L-∞ distance:

$$L_\infty(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_\infty = \max_{i=1}^{d} \{|x_i - y_i|\}$$

The difference between the L1 and L2 distances is notable, as shown in the figure blow:




```
Manhattan (L1) Distance: 7.00
Euclidean (L2) Distance: 5.00
L-Inf Distance: 4.00
```

While the Euclidean distance measures the length of the straight segment connecting the two points, the Manhattan distance is the one in grey (dashed), which measures the distance that a taxi driver should drive in Manhattan (or any other square-block based city) to reach the destination.

Cosine Distance

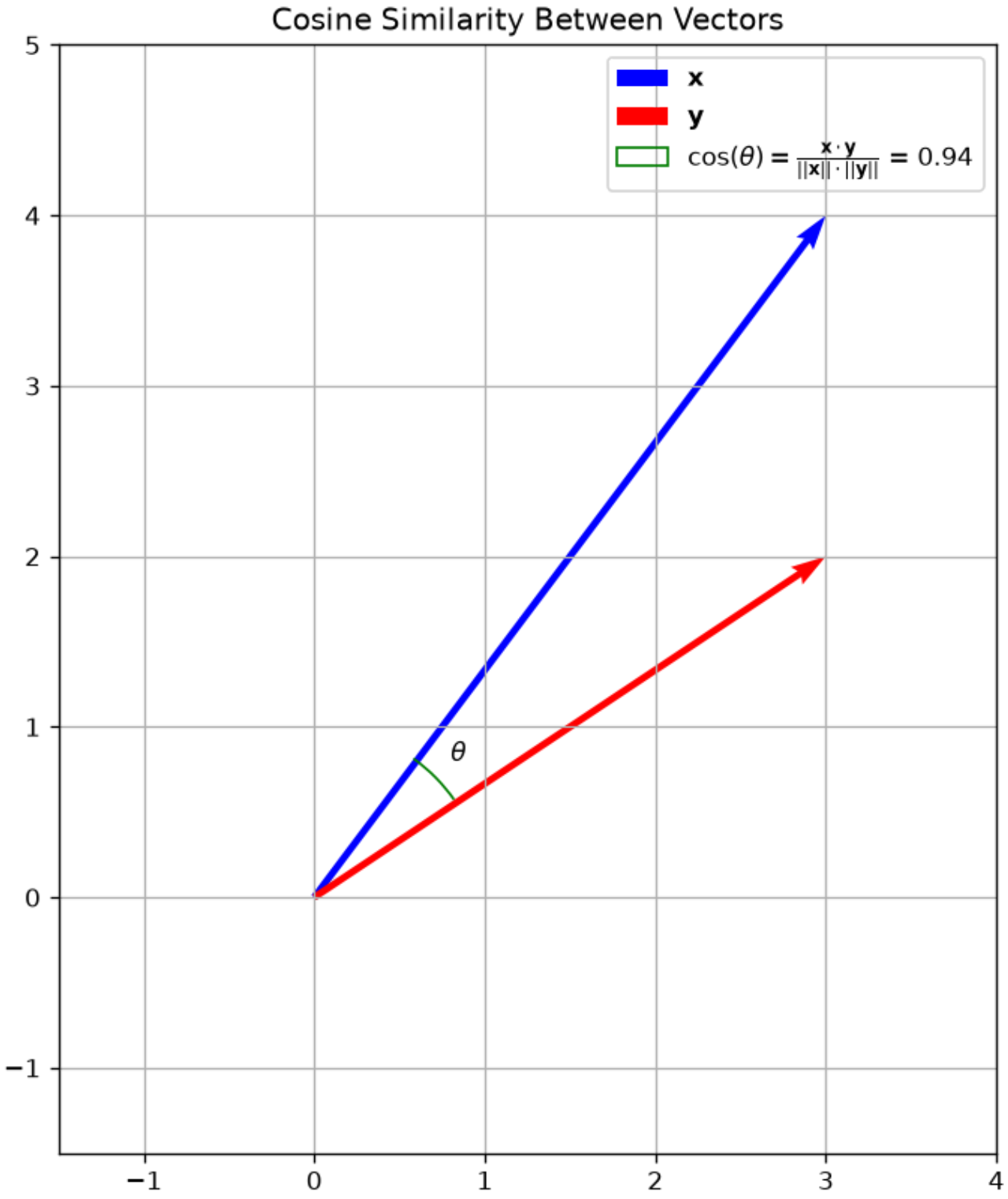
The cosine distance is useful when we need to compare two vectors but we do not care about differences arising from scaling factors. Note that the Euclidean distance of two proportional vectors is in general non-zero:

$$||\mathbf{x} - \alpha\mathbf{x}||^2 > 0, \alpha \neq 1$$

Nevertheless, $\mathbf{x}$ and $\alpha\mathbf{x}$ are very similar. If we want to compare two vectors considering only **the relationships between their coordinates, rather than their values**, we can use the cosine similarity, which is defined as follows:

$$S_C(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||||\mathbf{y}||}$$

The cosine similarity computes the cosine of the angle θ comprised between two vectors, as shown in the plot below:



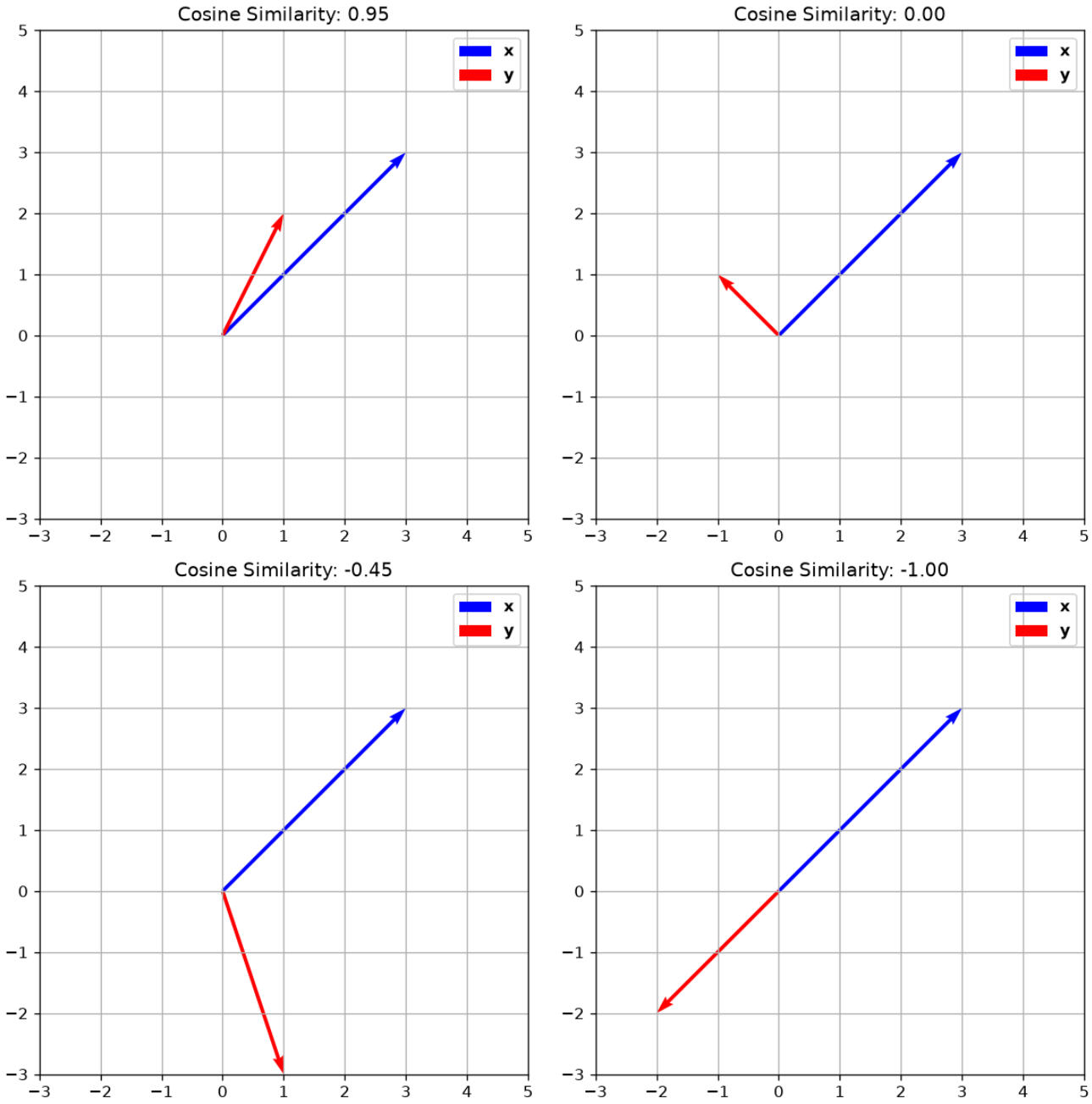
Note that the cosine similarity would be the same if the two vectors had different scales (i.e., if they were longer but they would have the same orientation).

This kind of similarity measure is useful when the scale of the vector is not important. For instance, if the vectors $\mathbf{x}$ and $\mathbf{y}$ are vectors of word counts of two documents of different lengths, we care about the **proportions of words in each document, while longer document will have vectors with larger L2 norms**.

The cosine similarity is a normalized number between -1 and 1, where:

- $+1$ means maximum alignment (similarity) between the two vectors;
- -1 means that the two vectors are dissimilar - they are indeed opposite.
- 0 means that the two vectors are orthogonal.

The plot below shows different examples of cosine similarity of vector pairs:



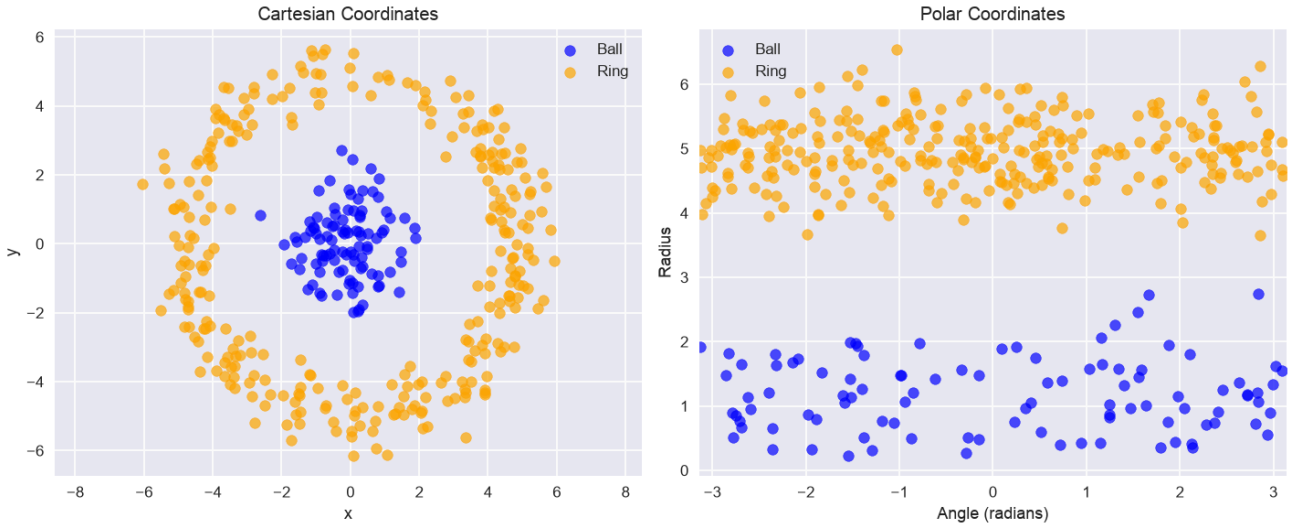
The cosine distance is defined from the cosine similarity as:

$$D_C = 1 - S_C = 1 - \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}|| \cdot ||\mathbf{x}||}$$

It should be noted that the cosine similarity is not a metric as it does not satisfy the triangular inequality.

Features and Representation Functions

Data representation can help making data analysis processes simpler. Consider for example the example below:



The data on the left in not linearly separable. However, taking polar coordinates in this form (in practice the case-defined `atan2` function is used instead of `arctan` to handle quadrants correctly):

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \arctan\left(\frac{y}{x}\right)$$

We obtain a linearly separable set of data which could be classified with a linear classifier.

From Hand-Crafted Features to Learned Representations

The polar coordinate example is powerful. It shows that a problem that is impossible for a linear model in one space (Cartesian) can become trivial in another space (Polar).

In this case, we used our human intuition to "hand-craft" a new feature representation (r, θ) that solved the problem. This is the traditional approach to machine learning.

The "Black Box" Feature Extractor

In practice, for complex data like images, audio, or text, we can't manually discover the perfect feature function.

- What is the "radius" of a picture of a cat?
- What is the "angle" of a legal document?

This is where modern machine learning, particularly **Deep Learning**, comes in. The dominant approach today is to *learn* the representation function f from the data itself.

We can treat these massive, pre-trained models as powerful "black-box" feature extractors.

- **For Text:** We can use models like **BERT**, **GPT**, or **Sentence Transformers**. You can feed a raw sentence into one of these models and get a 768-dimensional vector as output. This vector is a rich, numerical "feature vector" that understands the *semantic meaning* of the sentence. Libraries like **Hugging Face** `transformers` are the standard toolkit for this.
- **For Images:** We can use pre-trained **Convolutional Neural Networks (CNNs)** like **ResNet** or **Vision Transformers (ViT)**. You feed in a raw image and get a 2048-dimensional vector that represents its visual content.
- **For Audio:** We can use traditional features like **MFCCs** (Mel-Frequency Cepstral Coefficients) or use deep models like **1D CNNs** or **Audio Transformers** like **Wav2Vec2**. You feed in a raw waveform and get a feature vector that represents the acoustic content, speaker identity, or spoken words.

In all cases, the principle is the same. Whether hand-crafted (like r and θ) or learned (like a Transformer), the goal of the representation function $f : \mathfrak{R}^d \rightarrow \mathfrak{R}^m$ is to map the "messy" raw data into a new **feature space** where the relationships are simpler and a classifier can easily find the decision boundary.

Data Processing Inequality

The Data Processing Inequality (DPI) states that applying any data transformation, whether deterministic or stochastic, can never increase the information content about an original random variable. In simple terms: processing data can only lose or preserve information, never create it.

Mathematically, if we have a Markov chain $X \rightarrow Y \rightarrow Z$, meaning Z is conditionally independent of X given Y, then the mutual information satisfies:

$$I(X;Y) \geq I(X;Z)$$

Here, $I(X;Y)$ represents the shared information between random variables X and Y. This inequality implies that as data X is successively transformed into Y and then Z, the amount of information about the original variable X can only decrease or stay the same; it can never increase.

This principle has profound implications for feature engineering:

- **Irreversible Loss** If hand-crafted features inadvertently discard crucial information about the underlying signal, that information is permanently lost and cannot be recovered by subsequent modeling steps.
- **Information Bottleneck** Poor feature design creates an information bottleneck, limiting the maximum performance achievable by any machine learning model built upon those features.
- **The Advantage of Learned Features** Deep learning models aim to learn representations that preserve maximal task-relevant information directly from raw data, minimizing the risk of premature information loss.

Lab: The Critical Importance of Feature Representation

So far, we have learned about algorithms (like KNN) and the "Curse of Dimensionality." We've also discussed how we can *represent* data, from raw vectors (pixels) to hand-crafted features (polar coordinates) to "learned" features (like from your `DLFeat` library).

In this lab, we will prove that **feature representation is often more important than the choice of algorithm.**

We will conduct two "bake-offs" to show how a simple `KNeighborsClassifier` (which we know is sensitive to dimensionality and distance) performs in different feature spaces.

1. **Image Task (CIFAR):** We'll compare KNN on **raw pixels** vs. KNN on **"smart" CNN features**.
2. **Text Task (Spam):** We'll compare the "classic" `MultinomialNB` vs. KNN on word counts vs. KNN on "smart" deep features.

We will use the `sklearn` library for KNN and other utilities, and `DLFeat` for deep learning-based text features.

Let's start by installing libraries:

```
!pip install torch torchvision torchaudio scikit-learn Pillow numpy scipy av
!pip install transformers sentence-transformers timm requests
!pip install --upgrade torch torchvision torchaudio transformers sentence-
        transformers timm requests
!pip install git+https://github.com/antoninofurnari/dlfeat.git
```

Let's import dependencies:

```
# --- 1. All Imports ---
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time
import os
import joblib # Using joblib for efficient saving of numpy arrays

# --- Sklearn Tools ---
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, accuracy_score

# --- Models ---
from sklearn.neighbors import KNeighborsClassifier

# --- Data and Feature Extractors ---
from torchvision.datasets import CIFAR10
from PIL import Image
from dlfeat import DLFeatExtractor
```

Part 1: The Image Challenge (Handwritten Digits)

We will build two KNN models to classify images from the **CIFAR-10** dataset.

- **Model 1 (The "Dumb" Baseline):** We will train KNN on **raw pixel data**. This is a "dumb," high-dimensional space (32x32x3 = 3,072 features). We expect this to perform poorly.
- **Model 2 (The "Smart" Model):** We will first use a powerful, pre-trained Convolutional Neural Network (CNN) from your `DLFeat` library to extract "smart" **semantic features**. We will then train KNN on this new, "clean" feature space.

We hypothesize that the simple KNN model will perform *dramatically* better when using the "smart" features.

Step 1: Load and Visualize the CIFAR-10 Data

First, let's load the data. CIFAR-10 consists of 60,000 32x32 color images in 10 classes. For speed, we'll use a subset of 5,000 for training and 1,000 for testing.

We will prepare our data in two formats:

1. `X_train_pil`: A list of PIL Image objects, required by `DLFeatExtractor`.
2. `X_train_raw`: A NumPy array of flattened pixel vectors (shape `(N, 3072)`), for our baseline KNN.

```
# --- 1. Load CIFAR-10 Dataset ---
print("1. Loading CIFAR-10 dataset...")
try:
    cifar10_train_raw = CIFAR10(root='./cifar_data', train=True, download=True)
    cifar10_test_raw = CIFAR10(root='./cifar_data', train=False, download=True)

    class_names = cifar10_train_raw.classes
    print(f"   Classes: {class_names}")

    # Extract a subset for faster demonstration
    num_train_samples = 5000
    num_test_samples = 1000

    # Get PIL Images (for DLFeat)
    X_train_pil = [cifar10_train_raw[i][0] for i in range(num_train_samples)]
    y_train = [cifar10_train_raw[i][1] for i in range(num_train_samples)]

    X_test_pil = [cifar10_test_raw[i][0] for i in range(num_test_samples)]
    y_test = [cifar10_test_raw[i][1] for i in range(num_test_samples)]

    # Get Raw Pixel vectors (for Baseline KNN)
    # Shape = (N, 32*32*3) = (N, 3072)
    X_train_raw = np.array([np.array(img).flatten() for img in X_train_pil])
    X_test_raw = np.array([np.array(img).flatten() for img in X_test_pil])

    print(f"   Subset loaded: {len(X_train_pil)} training samples, {len(X_test_pil)}
        testing samples.")
    print(f"   Raw pixel data shape: {X_train_raw.shape}")

except Exception as e:
    print(f"Error loading or processing CIFAR-10: {e}")
```

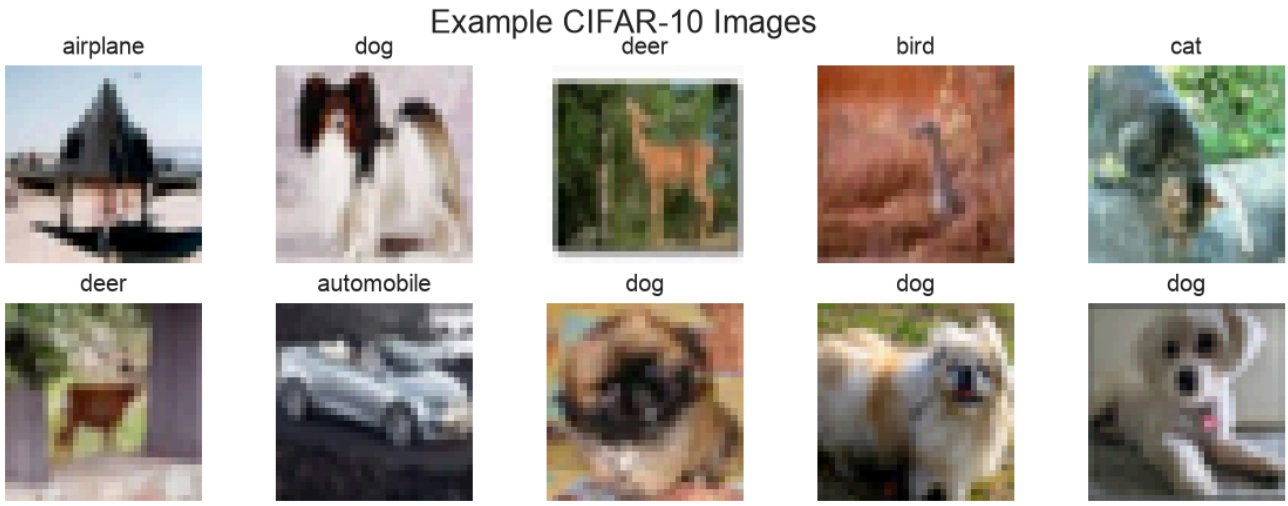
```
1. Loading CIFAR-10 dataset...
```

```
   Classes: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']
   Subset loaded: 5000 training samples, 1000 testing samples.
   Raw pixel data shape: (5000, 3072)
```

Step 2: Look at the Task

Let's see what these images actually look like. They are small (32x32) and can be hard to recognize.

```
# Show 10 random sample images
plt.figure(figsize=(12, 4))
for i in range(10):
    ax = plt.subplot(2, 5, i + 1)
    idx = np.random.randint(0, len(X_train_pil))
    plt.imshow(X_train_pil[idx])
    plt.title(class_names[y_train[idx]])
    plt.axis('off')
plt.suptitle('Example CIFAR-10 Images', fontsize=16)
plt.show()
```



Step 3: Baseline (KNN on Raw Pixels)

First, we'll train our baseline model. We create a `Pipeline` that scales the 3,072 pixel values (a critical step) and then runs our KNN classifier. We expect this to perform poorly due to the **Curse of Dimensionality**—the 3,072-dimensional space is too large and "empty" for "nearest neighbor" to be a meaningful concept.

```
# 1. Create a simple KNN pipeline for raw pixels
# We must scale the pixel data!
pipe_knn_raw = Pipeline([
    ('scaler', StandardScaler()),
    ('model', KNeighborsClassifier(n_neighbors=5))
])

# 2. Fit and score
print("Fitting KNN on RAW 3072-pixel features...")
start_time = time.time()
pipe_knn_raw.fit(X_train_raw, y_train)
y_pred_raw = pipe_knn_raw.predict(X_test_raw)
acc_raw = accuracy_score(y_test, y_pred_raw)
time_raw = time.time() - start_time

print(f"   Accuracy (KNN on Raw Pixels): {acc_raw*100:.2f}%")
print(f"   Time taken: {time_raw:.2f} seconds")

# Store results for our final chart
results = {
    'KNN on Raw Pixels': acc_raw
}
```

```
Fitting KNN on RAW 3072-pixel features...
   Accuracy (KNN on Raw Pixels): 26.20%
   Time taken: 0.79 seconds
```

Step 4: Extract “Smart” Features with `DLFeat` (The Expensive Step)

Now, we’ll use a pre-trained **ResNet18** model via your `DLFeatExtractor` to convert our images into “smart” feature vectors. This model has already learned about edges, shapes, and objects from the ImageNet dataset.

This step is **computationally expensive** and only needs to be done *once*. We will run it and then **save the resulting feature vectors to disk** (`.pkl` files) to save time in the future.

```
# Define file paths for our cached features
FEATURE_FILE_TRAIN = 'cifar_train_resnet18.pkl'
FEATURE_FILE_TEST = 'cifar_test_resnet18.pkl'
MODEL_NAME = "resnet18"

# Check if features are already saved
if os.path.exists(FEATURE_FILE_TRAIN) and os.path.exists(FEATURE_FILE_TEST):
    print("Loading 'smart' features from disk... (This is fast!)")
    X_train_deep = joblib.load(FEATURE_FILE_TRAIN)
    X_test_deep = joblib.load(FEATURE_FILE_TEST)
    print("   Features loaded.")

else:
    # If not on disk, we must extract them
    print(f"No cached features found. Initializing DLFeatExtractor with model:
        {MODEL_NAME}...")
    try:
        feature_extractor = DLFeatExtractor(model_name=MODEL_NAME, task_type="image")
        print(f"   Feature dimension will be:
        {feature_extractor.get_feature_dimension()}")

        # 1. Extract Training Features
        print("   Extracting training features (this is the slow part)...")
        start_time_extract = time.time()
        # We pass the list of PIL Images: X_train_pil
        X_train_deep = feature_extractor.fit(X_train_pil)
        X_train_deep = feature_extractor.transform(X_train_pil)

        # 2. Extract Test Features
        print("   Extracting test features...")
        X_test_deep = feature_extractor.transform(X_test_pil)
        time_extract = time.time() - start_time_extract
        print(f"   Feature extraction complete. Time: {time_extract:.2f}s")

        # 3. Save features to disk for next time
        print("   Saving features to disk...")
        joblib.dump(X_train_deep, FEATURE_FILE_TRAIN)
        joblib.dump(X_test_deep, FEATURE_FILE_TEST)

    except Exception as e:
        print(f"   Error initializing or using DLFeatExtractor: {e}")
        # In case of error, create dummy variables so the lab can continue
        X_train_deep, X_test_deep = None, None

print(f"\n'Smart' feature shape: {X_train_deep.shape}")
```

```
Loading 'smart' features from disk... (This is fast!)
   Features loaded.

'Smart' feature shape: (5000, 512)
```

Step 5: KNN on “Smart” Features

Now we have our new “smart” feature set, `X_train_deep`. Instead of 3072 “noisy” pixel features, we have 512 “smart” semantic features.

Let’s train our *same* KNN pipeline on this new, better data.

```
# 1. Create a *new* pipeline for these *new* features
pipe_knn_deep = Pipeline([
    ('scaler', StandardScaler()), # Still good practice to scale the CNN features
    ('model', KNeighborsClassifier(n_neighbors=5))
])

# 2. Fit and score
print("Fitting KNN on 'smart' CNN features...")
start_time = time.time()
pipe_knn_deep.fit(X_train_deep, y_train)
y_pred_deep = pipe_knn_deep.predict(X_test_deep)
acc_deep = accuracy_score(y_test, y_pred_deep)
time_deep = time.time() - start_time

print(f"   Accuracy (KNN on CNN Features): {acc_deep*100:.2f}%")
print(f"   Time taken (training only): {time_deep:.2f} seconds")

# Store the result
results['KNN on Smart Features'] = acc_deep
```

```
Fitting KNN on 'smart' CNN features...
   Accuracy (KNN on CNN Features): 73.90%
   Time taken (training only): 0.06 seconds
```

Step 6: The Final Showdown (Bar Chart)

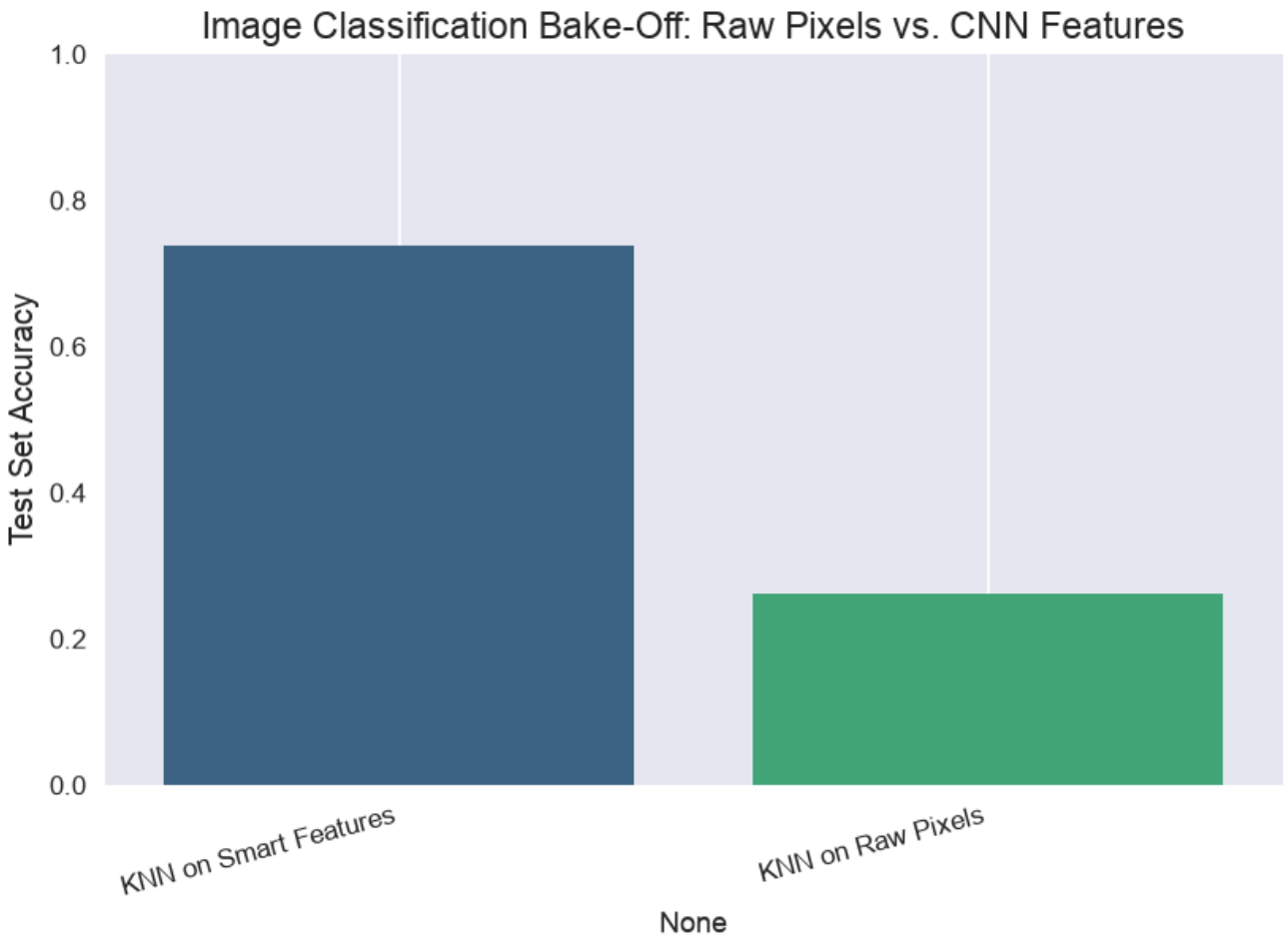
Let’s compare the final accuracy of our two models.

```
# Convert results to a pandas DataFrame
results_df = pd.DataFrame.from_dict(results, orient='index', columns=['Test
        Accuracy'])
results_df = results_df.sort_values(by='Test Accuracy', ascending=False)

print("\n--- FINAL BAKE-OFF RANKING ---")
print(results_df)

# Plot the comparison
plt.figure(figsize=(8, 5))
sns.barplot(
    x=results_df.index,
    y=results_df['Test Accuracy'],
    hue=results_df.index,
    palette='viridis',
    legend=False
)
plt.title('Image Classification Bake-Off: Raw Pixels vs. CNN Features', fontsize=14)
plt.ylabel('Test Set Accuracy', fontsize=12)
plt.xticks(rotation=15, ha='right')
plt.ylim(0, 1.0) # Show full 0-100% scale
plt.grid()
plt.show()
```

```
--- FINAL BAKE-OFF RANKING ---
                      Test Accuracy
KNN on Smart Features          0.739
KNN on Raw Pixels              0.262
```



Analysis

The results are dramatic and clear.

1. **KNN on Raw Pixels** performed poorly (e.g., ~26% accuracy). This is a perfect demonstration of the **Curse of Dimensionality**. The model was trying to calculate distances in a "noisy" 3072-dimensional space, and the "nearest neighbor" was meaningless.
2. **KNN on "Smart" CNN Features** performed *dramatically* better (e.g., ~74% accuracy).

Why? The pre-trained ResNet model acted as an intelligent "feature extractor." It transformed the "dumb" 3072-pixel space into a "smart" 512-dimensional **semantic space**, where similar-looking images (like "dog" and "cat") are placed closer together than "dog" and "airplane."

In this new, lower-dimensional, and more meaningful feature space, our simple KNN model was able to find "truly" similar images and make accurate predictions. This proves our main lesson: **The quality of your feature representation is the most important part of your model.**

Multinomial Naive Bayes vs. KNN vs. "Smart" KNN

We have seen `Multinomial Naive Bayes (MNB)` as the classic, fast, and effective model for text. We also saw `K-Nearest Neighbors (KNN)` and learned about its main weakness: the **Curse of Dimensionality**.

In this lab, we will run a "bake-off" to prove these concepts. We will build three different models for our spam classification task:

1. **Model 1 (The Baseline): `MultinomialNB` on `Count` features.**
   - This is the "classic" generative model that is built for this type of data.
2. **Model 2 (The "Wrong Tool"): `KNN` on `Count` features.**
   - This is a distance-based model. We expect it to fail *miserably* because it will be trying to calculate distances in a "dumb," unscaled, high-dimensional (e.g., 7000+ features) space.
3. **Model 3 (The "Smart" Model): `KNN` on `DLFeat` features.**
   - We will first use a `sentence-bert` model from your `DLFeat` library to create a "smart," low-dimensional (e.g., 384) feature space that understands *meaning*.
   - We hypothesize that KNN will perform *brilliantly* in this new space.

Our goal is to compare their final accuracy on the **test set**.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time
import io
import zipfile
import urllib.request

# --- Sklearn Tools ---
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, accuracy_score

# --- Models ---
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB

# --- Feature Extractors ---
from sklearn.feature_extraction.text import CountVectorizer
from dlfeat import DLFeatExtractor
```

Step 1: Load and Split the Data

First, we load the spam dataset, map the labels, and create our single, "locked-away" test set.

```
# 1. Load the dataset
try:
    zip_data = data_bytes("sms_spam_collection.zip")
    with zipfile.ZipFile(io.BytesIO(zip_data)) as z:
        with z.open('SMSSpamCollection') as f:
            df = pd.read_csv(f, sep='\t', header=None, names=['label', 'message'],
        encoding='latin-1')

    # 2. Map labels to 0 (ham) and 1 (spam)
    df['label'] = df['label'].map({'ham': 0, 'spam': 1})

    # 3. Define X and y
    X = df['message'] # The raw text
    y = df['label']   # The 0 or 1

    # 4. Create the Train/Test Split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
         random_state=42, stratify=y)

    print("Spam data loaded and split.")
    print(f"Training samples: {len(X_train)}, Test samples: {len(X_test)}")

except Exception as e:
    print(f"Error loading or processing data: {e}")
```

```
Spam data loaded and split.
Training samples: 3900, Test samples: 1672
```

Part 1: The “Bag of Words” Competitors

First, we will build pipelines using `CountVectorizer`. This converts our text into a large, sparse matrix of raw word counts (e.g., `(3900, 7000)` shape).

```
# --- 1. Model 1: MultinomialNB (The Baseline) ---
# This model is designed for raw counts
pipe_mnb = Pipeline([
    ('vectorizer', CountVectorizer(stop_words='english')),
    ('model', MultinomialNB())
])

print("Fitting MultinomialNB...")
start_time_mnb = time.time()
pipe_mnb.fit(X_train, y_train)
y_pred_mnb = pipe_mnb.predict(X_test)
acc_mnb = accuracy_score(y_test, y_pred_mnb)
time_mnb = time.time() - start_time_mnb

print(f"Accuracy (MNB on Counts): {acc_mnb*100:.2f}% (Time: {time_mnb:.2f}s)")


# --- 2. Model 2: KNN on Counts (The "Wrong Tool") ---
# KNN is not designed for this. It will suffer from:
# 1. Curse of Dimensionality (thousands of features)
# 2. Unscaled Data (word counts like 'the'=50 vs 'win'=1)
pipe_knn_counts = Pipeline([
    ('vectorizer', CountVectorizer(stop_words='english')),
    ('model', KNeighborsClassifier(n_neighbors=5))
])

print("\nFitting KNN on 'dumb' Counts...")
start_time_knn = time.time()
pipe_knn_counts.fit(X_train, y_train)
y_pred_counts = pipe_knn_counts.predict(X_test)
acc_knn_counts = accuracy_score(y_test, y_pred_counts)
time_knn = time.time() - start_time_knn

print(f"Accuracy (KNN on Counts): {acc_knn_counts*100:.2f}% (Time: {time_knn:.2f}s)")
```

```
Fitting MultinomialNB...
Accuracy (MNB on Counts): 98.74% (Time: 0.04s)

Fitting KNN on 'dumb' Counts...
Accuracy (KNN on Counts): 91.15% (Time: 0.13s)
```

Part 2: The “Smart” Features Competitor

The `KNN on Counts` model performed terribly, as expected.

Now, let’s create a *new* feature space. We will use `DLFeatExtractor` with `'sentence-bert'` to convert each text message into a **384-dimensional “smart” vector** that understands semantic meaning. This space is **low-dimensional**, **dense**, and **meaningful**.

```
# --- 3. Model 3: KNN on "Smart" DLFeat Features ---

# 3a. Extract Deep Features (The expensive, one-time step)
print("\nExtracting deep semantic features with DLFeat...")
# We use .to_list() to pass the raw text strings to the extractor
X_train_list = X_train.to_list()
X_test_list = X_test.to_list()

try:
    feature_extractor = DLFeatExtractor(model_name="sentence-bert", task_type="text")

    start_time_extract = time.time()
    X_train_deep = feature_extractor.transform(X_train_list)
    X_test_deep = feature_extractor.transform(X_test_list)
    extract_time = time.time() - start_time_extract

    print(f"   Feature extraction complete. Time: {extract_time:.2f}s")
    print(f"   New feature shape: {X_train_deep.shape} (vs. {len(X_train_list)}
        docs)")

    # 3b. Build a *new* pipeline for these features
    # This pipeline *only* needs to scale the features, not vectorize text
    pipe_knn_deep = Pipeline([
        ('scaler', StandardScaler()),
        ('model', KNeighborsClassifier(n_neighbors=5))
    ])

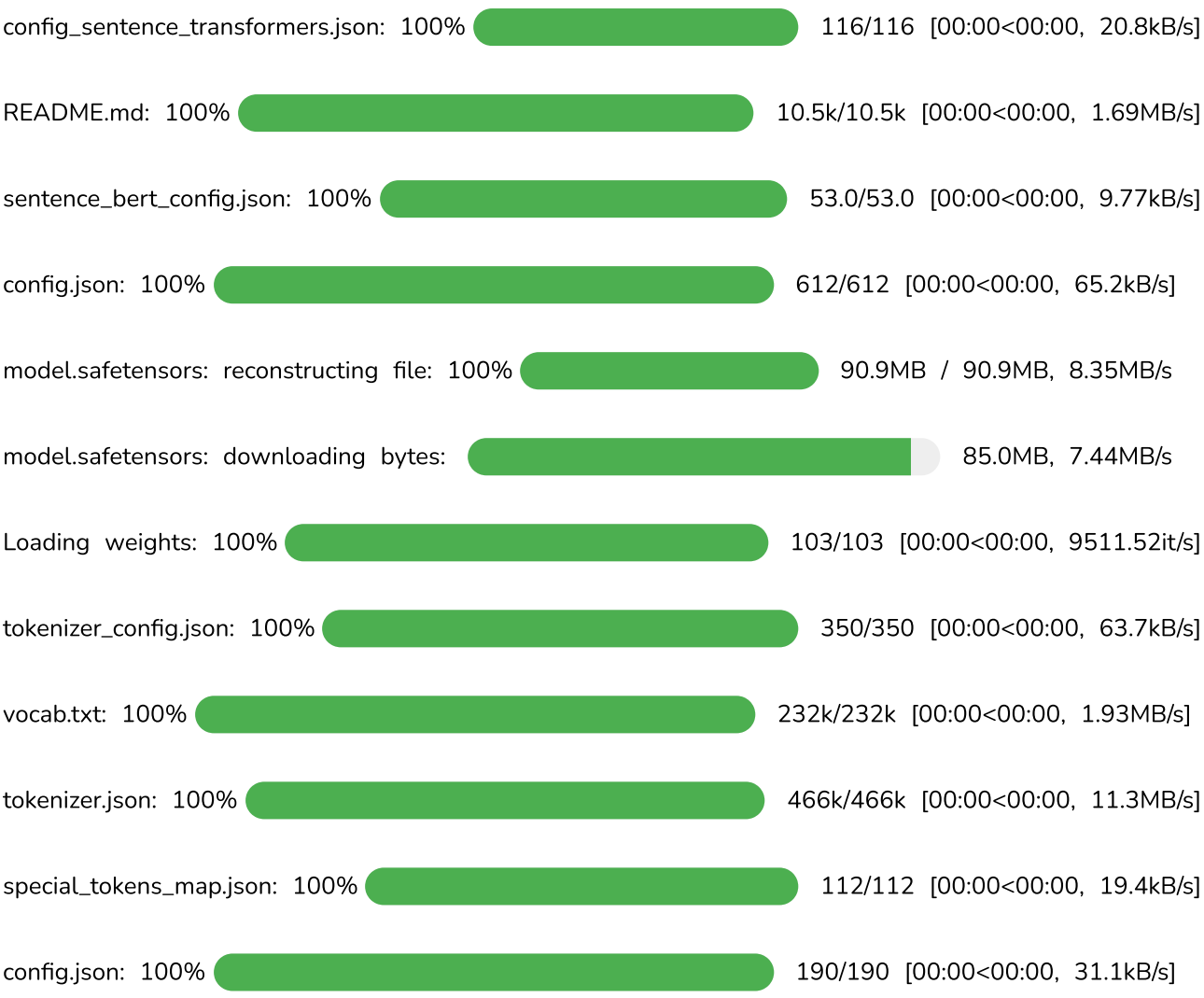
    # 3c. Fit and score
    print("Fitting KNN on 'smart' DLFeat features...")
    start_time_knn_deep = time.time()
    pipe_knn_deep.fit(X_train_deep, y_train)
    y_pred_deep = pipe_knn_deep.predict(X_test_deep)
    acc_knn_deep = accuracy_score(y_test, y_pred_deep)
    time_knn_deep = time.time() - start_time_knn_deep

    print(f"   Accuracy (KNN on DLFeat): {acc_knn_deep*100:.2f}%")
    print(f"   Time taken (training only): {time_knn_deep:.2f}s")

except Exception as e:
    print(f"Error in text feature extraction: {e}")
    print("Ensure dependencies for text models (like sentence-transformers) are
        installed.")
    acc_knn_deep = 0.0 # Set to 0 for the final chart
```

```
Extracting deep semantic features with DLFeat...
```

modules.json: 100% 349/349 [00:00<00:00, 45.7kB/s]




```
   Feature extraction complete. Time: 12.27s
   New feature shape: (3900, 384) (vs. 3900 docs)
Fitting KNN on 'smart' DLFeat features...
   Accuracy (KNN on DLFeat): 97.97%
   Time taken (training only): 0.05s
```

Part 3: The Final Results

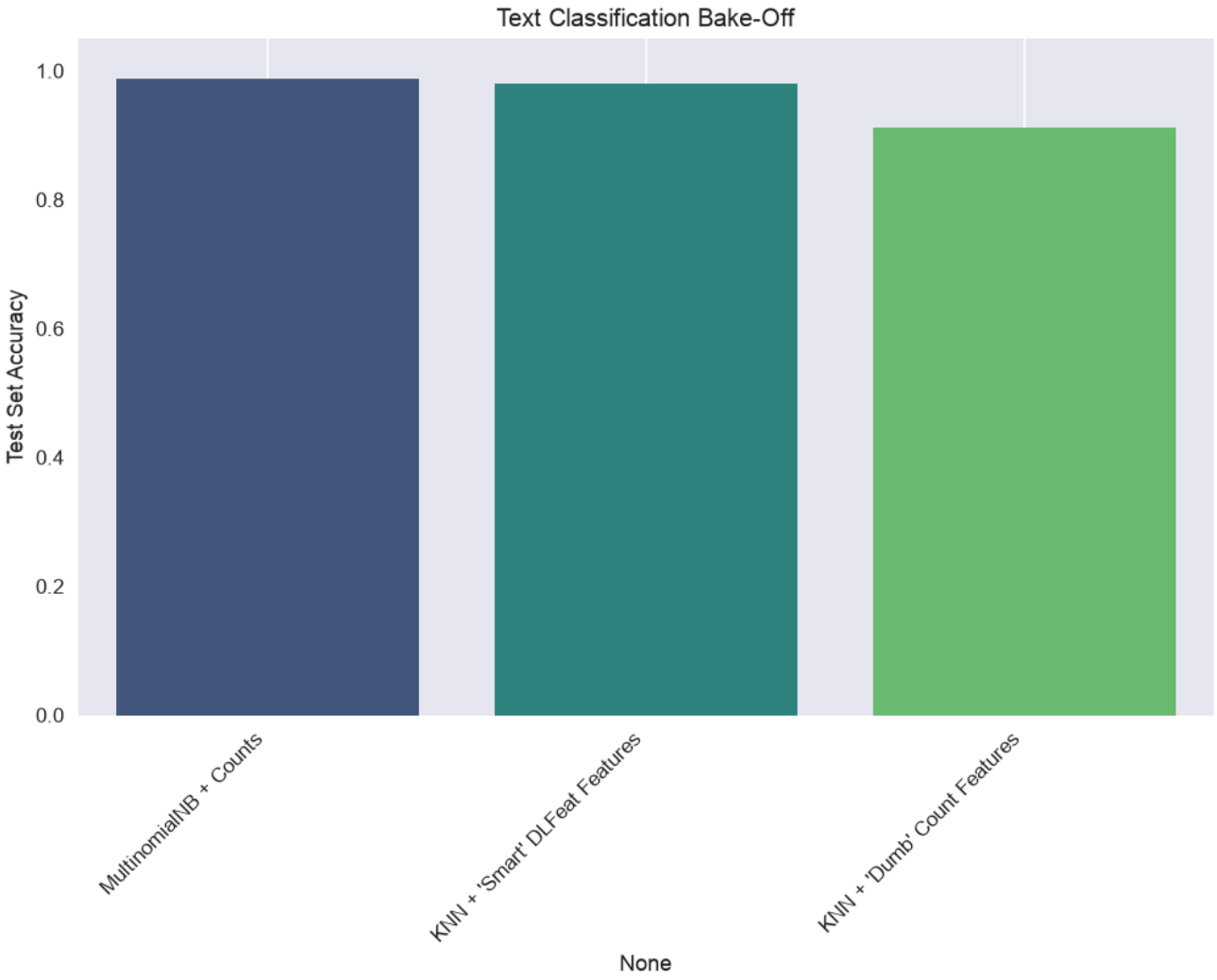
```
# Create a final comparison table
results = {
    "MultinomialNB + Counts": acc_mnb,
    "KNN + 'Smart' DLFeat Features": acc_knn_deep,
    "KNN + 'Dumb' Count Features": acc_knn_counts
}

results_df = pd.DataFrame.from_dict(results, orient='index', columns=['Test
        Accuracy'])
results_df = results_df.sort_values(by='Test Accuracy', ascending=False)

print("\n--- FINAL BAKE-OFF RANKING (Spam Filter) ---")
print(results_df)

# Plot the comparison
plt.figure(figsize=(10, 6))
sns.barplot(x=results_df.index, y=results_df['Test Accuracy'], hue=results_df.index,
        palette='viridis', legend=False)
plt.title('Text Classification Bake-Off')
plt.ylabel('Test Set Accuracy')
plt.xticks(rotation=45, ha='right')
plt.ylim(0, 1.05) # Show full 0-100% scale
plt.grid()
plt.show()
```

```
--- FINAL BAKE-OFF RANKING (Spam Filter) ---
                              Test Accuracy
MultinomialNB + Counts             0.987440
KNN + 'Smart' DLFeat Features      0.979665
KNN + 'Dumb' Count Features        0.911483
```



Analysis & Conclusion

The results are clear:

1. `MultinomialNB` won, and it is the model with the least machinery behind it. Word counts are exactly what a multinomial distribution is built for, and spam is largely a question of *which words occur* — no understanding of the sentence required.
2. `KNN` **on** `CountVectorizer` (raw counts) came last. This is the **curse of dimensionality** in action: the model is computing distances in a 7000-dimensional, unscaled space, where "nearest" means very little.
3. `KNN` **on** `DLFeat` came second, within a point of the winner. The same "dumb" KNN that failed on raw counts works well once the text is mapped into a *low-dimensional (384), semantically rich* space — the model did not change, the representation did.

That is the lesson of this lab: **the quality of the representation matters more than the choice of the algorithm.** It does not say that deep features always win; on this task a well-matched classical model is still ahead, and it fits in a fraction of the time. The next lab is the task where the semantics do make the difference.

Lab: The Power of Feature Representation (Sentiment Analysis)

We've seen `MultinomialNB` is great for text. We also know `KNN` is a simple, powerful classifier that suffers from the **Curse of Dimensionality**.

In this lab, we'll run a "bake-off" to prove that **feature representation is the most important part of a model.** We'll use a **sentiment analysis** task: classifying product reviews as "positive" (1) or "negative" (0).

The Competitors:

1. **Model 1 (Baseline): `MultinomialNB` on `Count` features.**
   - This is the "classic" generative model that is built for this type of data.
2. **Model 2 (The "Wrong Tool"): `KNN` on `Count` features.**
   - We expect this to fail due to the Curse of Dimensionality (thousands of unscaled features).
3. **Model 3 (The "Smart" Model): `KNN` on `DLFeat` features.**
   - We'll use a `sentence-bert` model to create a "smart," low-dimensional feature space that understands *meaning*. We expect KNN to perform very well here.

The Workflow:

1. **Dataset:** A set of 1000 Amazon product reviews, labeled 0 (negative) or 1 (positive).
2. **Split:** A standard Train/Test split.
3. **Pipelines:** We'll build the correct `Pipeline` for each model.
4. **Evaluation:** We'll compare all models on the **test set** accuracy.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time
import io
import zipfile
import urllib.request

# --- Sklearn Tools ---
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# --- Models ---
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB

# --- Feature Extractors ---
from sklearn.feature_extraction.text import CountVectorizer
from dlfeat import DLFeatExtractor
```

Step 1: Load and Split the Data

First, we'll load a labeled dataset of Amazon reviews from the UCI archive.

```
# 1. Load the dataset
# This file contains 1000 reviews, 500 positive (1) and 500 negative (0).

try:
    zip_data = data_bytes("sentiment_labelled_sentences.zip")

    # 2. Unzip the file in memory
    with zipfile.ZipFile(io.BytesIO(zip_data)) as z:
        # 3. Read the 'amazon_cells_labelled.txt' file
        with z.open('sentiment labelled sentences/amazon_cells_labelled.txt') as f:
            df = pd.read_csv(f,
                             sep='\t',
                             header=None,
                             names=['message', 'label'])

    print("--- Data Head ---")
    print(df.head())

    # 4. Define X and y
    X = df['message'] # The raw text
    y = df['label']   # The 0 (negative) or 1 (positive)

    # 5. Create the Train/Test Split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
        random_state=42, stratify=y)

    print(f"\nTraining samples: {len(X_train)}, Test samples: {len(X_test)}")

except Exception as e:
    print(f"Error loading or processing data: {e}")
```

```
--- Data Head ---
                                             message  label
0  So there is no way for me to plug it in here i...      0
1                        Good case, Excellent value.      1
2                             Great for the jawbone.      1
3  Tied to charger for conversations lasting more...      0
4                                   The mic is great.      1

Training samples: 700, Test samples: 300
```

Part 1: The “Bag of Words” Competitors

First, we will build pipelines using `CountVectorizer`. This converts our text into a large, sparse matrix of raw word counts.

```
# --- 1. Model 1: MultinomialNB (The Baseline) ---
# This model is designed for raw counts
pipe_mnb = Pipeline([
    ('vectorizer', CountVectorizer(stop_words='english')),
    ('model', MultinomialNB())
])

print("Fitting MultinomialNB...")
pipe_mnb.fit(X_train, y_train)
y_pred_mnb = pipe_mnb.predict(X_test)
acc_mnb = accuracy_score(y_test, y_pred_mnb)
print(f"Accuracy (MNB on Counts): {acc_mnb*100:.2f}%")


# --- 2. Model 2: KNN on Counts (The "Wrong Tool") ---
# This will suffer from the Curse of Dimensionality
pipe_knn_counts = Pipeline([
    ('vectorizer', CountVectorizer(stop_words='english')),
    ('model', KNeighborsClassifier(n_neighbors=5))
])

print("\nFitting KNN on 'dumb' Counts...")
pipe_knn_counts.fit(X_train, y_train)
y_pred_counts = pipe_knn_counts.predict(X_test)
acc_knn_counts = accuracy_score(y_test, y_pred_counts)
print(f"Accuracy (KNN on Counts): {acc_knn_counts*100:.2f}%")
```

```
Fitting MultinomialNB...
Accuracy (MNB on Counts): 76.00%

Fitting KNN on 'dumb' Counts...
Accuracy (KNN on Counts): 67.67%
```

Part 2: The “Smart” Features Competitor

The `KNN on Counts` model performed terribly (likely ~60-70% accuracy), as expected.

Now, let’s create a *new* feature space. We will use `DLFeatExtractor` with `'sentence-bert'` to convert each text review into a **384-dimensional “smart” vector** that understands semantic meaning. This space is **low-dimensional**, **dense**, and **meaningful**.

```
# --- 3. Model 3: KNN on "Smart" DLFeat Features ---

# 3a. Extract Deep Features (The expensive, one-time step)
print("\nExtracting deep semantic features with DLFeat...")
# We use .to_list() to pass the raw text strings to the extractor
X_train_list = X_train.to_list()
X_test_list = X_test.to_list()

try:
    feature_extractor = DLFeatExtractor(model_name="sentence-bert", task_type="text")

    start_time_extract = time.time()
    X_train_deep = feature_extractor.transform(X_train_list)
    X_test_deep = feature_extractor.transform(X_test_list)
    extract_time = time.time() - start_time_extract

    print(f"   Feature extraction complete. Time: {extract_time:.2f}s")
    print(f"   New feature shape: {X_train_deep.shape}")

    # 3b. Build a *new* pipeline for these features
    # This pipeline *only* needs to scale the features
    pipe_knn_deep = Pipeline([
        ('scaler', StandardScaler()),
        ('model', KNeighborsClassifier(n_neighbors=5))
    ])

    # 3c. Fit and score
    print("Fitting KNN on 'smart' DLFeat features...")
    pipe_knn_deep.fit(X_train_deep, y_train)
    y_pred_deep = pipe_knn_deep.predict(X_test_deep)
    acc_knn_deep = accuracy_score(y_test, y_pred_deep)

    print(f"   Accuracy (KNN on DLFeat): {acc_knn_deep*100:.2f}%")

except Exception as e:
    print(f"Error in text feature extraction: {e}")
    print("Ensure dependencies for text models (like sentence-transformers) are
        installed.")
    acc_knn_deep = 0.0 # Set to 0 for the final chart
```

```
Extracting deep semantic features with DLFeat...
```

Loading weights: 100%  103/103 [00:00<00:00, 5451.13it/s]

```
    Feature extraction complete. Time: 1.43s
    New feature shape: (700, 384)
 Fitting KNN on 'smart' DLFeat features...
    Accuracy (KNN on DLFeat): 85.00%
```

Part 3: The Final Results

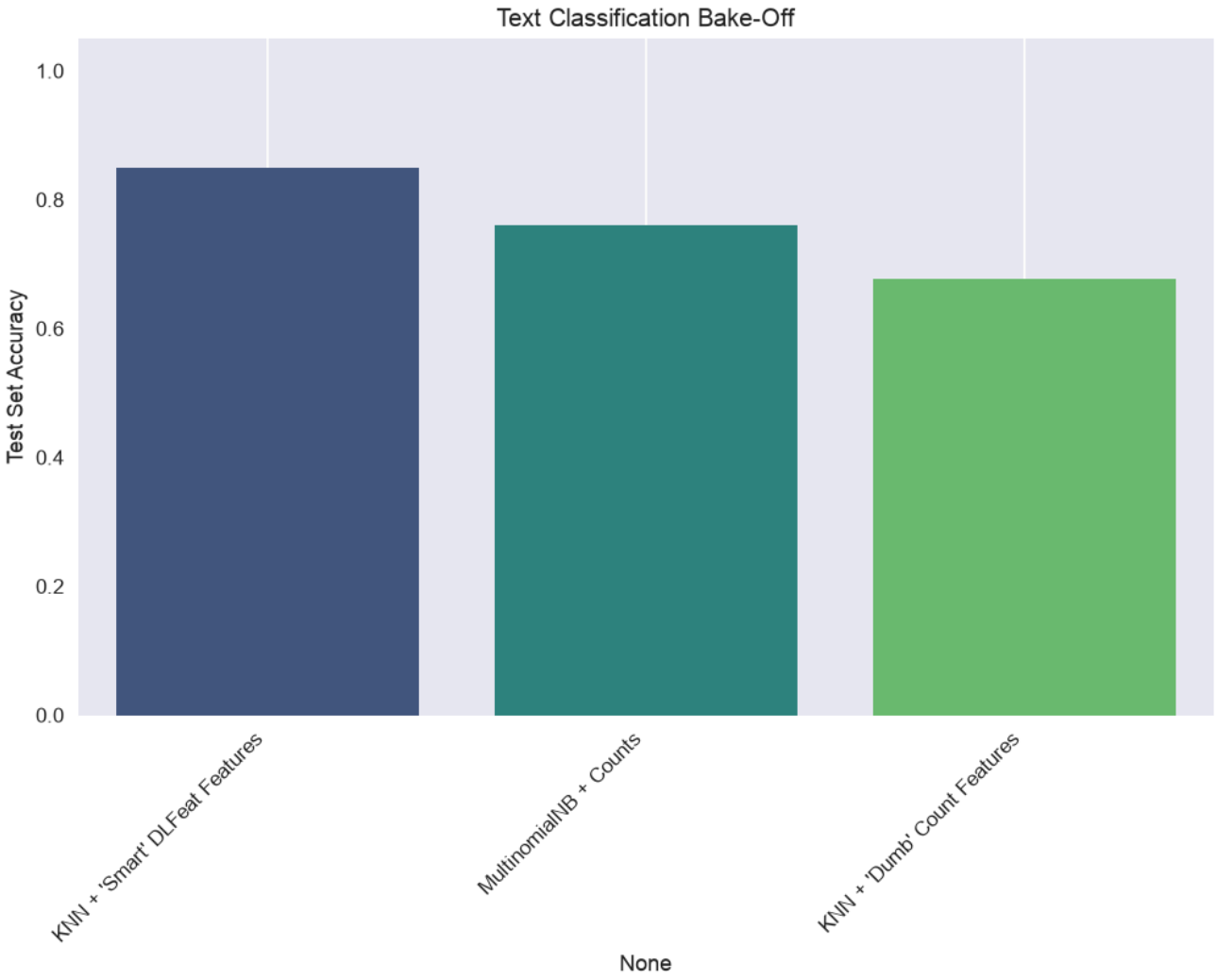
```
# Create a final comparison table
results = {
    "MultinomialNB + Counts": acc_mnb,
    "KNN + 'Smart' DLFeat Features": acc_knn_deep,
    "KNN + 'Dumb' Count Features": acc_knn_counts
}

results_df = pd.DataFrame.from_dict(results, orient='index', columns=['Test
        Accuracy'])
results_df = results_df.sort_values(by='Test Accuracy', ascending=False)

print("\n--- FINAL BAKE-OFF RANKING (Sentiment Analysis) ---")
print(results_df)

# Plot the comparison
plt.figure(figsize=(10, 6))
sns.barplot(x=results_df.index, y=results_df['Test Accuracy'], hue=results_df.index,
        palette='viridis', legend=False)
plt.title('Text Classification Bake-Off')
plt.ylabel('Test Set Accuracy')
plt.xticks(rotation=45, ha='right')
plt.ylim(0, 1.05) # Show full 0-100% scale
plt.grid()
plt.show()
```

```
 --- FINAL BAKE-OFF RANKING (Sentiment Analysis) ---
                               Test Accuracy
 KNN + 'Smart' DLFeat Features       0.850000
 MultinomialNB + Counts              0.760000
 KNN + 'Dumb' Count Features         0.676667
```



Analysis & Conclusion

The results are clear, and they are the opposite of the spam task's:

1. `KNN` **on** `DLFeat` was the **winner**, by nine points over the classical baseline. Mapping each review into a *low-dimensional (384), semantically rich* space first lets a simple KNN find reviews that *mean* the same thing.
2. `MultinomialNB` came second. It is fast and well matched to word counts, but word counts are not what sentiment is made of.
3. `KNN` **on** `CountVectorizer` (raw counts) came last, for the same reason as before: thousands of unscaled dimensions, and distances that stop discriminating.

This task highlights the importance of semantics. MNB and "dumb" KNN can't tell that "I *hated* this product" and "This product is *terrible*" are semantically similar, even though they share no words. The `DLFeat` model *does* understand this, creating a feature space where these two sentences are "near" each other.

This proves our main lesson: **The quality of your feature representation is the most important part of your model.** A "dumb" model on smart features can easily beat a "smart" model on dumb features.

For Further Exploration: The `DLFeat` Library

In this lab, we used `DLFeat` to quickly extract powerful features from images (`resnet18`) and text (`sentence-bert`). This is just the beginning.

Your `DLFeat` library is a powerful toolkit designed to make feature extraction from many different types of data (or "modalities") simple.

For more information, examples, and a full list of available models, **check the official documentation:**

https://antoninofurnari.github.io/DLFeat/

In the documentation, you will find:

- **The Model Zoo:** A complete list of all supported "black-box" feature extractors, including:
  - Advanced **Image** models (like `vit_base_patch16_224`, `dinov2_base`)
  - Powerful **Video** models (like `video_swin_t`, `r2plus1d_18`)
  - **Audio** models (like `wav2vec2_base`)
  - **Multimodal** (Image+Text) models (like `clip_vit_b32`)
- **Basic Usage Guides:** Code examples for how to extract features from all these different data types (Audio, Video, etc.), not just the ones we used today.
- **Installation & Self-Tests:** How to install the full library with all optional dependencies and how to run `dlfeat.run_self_tests()` to see which models are available in your environment.

Exercises

1. **Features are choices.** Take a dataset of your choice and build three different feature representations of the same raw data. Compare a single classifier across the three. How much of the final performance is the model, and how much is the representation?
2. **Distance depends on scale.** Compute the pairwise distances between a few points before and after standardisation. Which pairs change rank, and what does that imply for every distance-based method in this course?
3. **One-hot and dimensionality.** One-hot encode a categorical variable with 50 levels. How many columns does it add? What does that do to the distances, and what would you do instead?
4. **Text as vectors.** Represent a small set of documents as raw counts and as TF-IDF. Which pairs of documents are nearest under each representation, and why do they differ?
5. **Similarity is a modelling decision.** Compare Euclidean and cosine distance on the same text vectors. Which one ignores document length, and when is that what you want?

References

- https://en.wikipedia.org/wiki/Norm_(mathematics)

- https://en.wikipedia.org/wiki/Lp_space
- https://en.wikipedia.org/wiki/Metric_space
- https://en.wikipedia.org/wiki/Cosine_similarity
- Parts of Chapter 2 of [1]

[1] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Clustering

Chapter resources

Slides (PDF) Open in Colab Practice quiz

When we deal with complex datasets of multiple observations and variables, it can be useful to be able to understand the underlying structure of the data.

One approach to do so is to determine whether the data can be grouped into **clusters containing points with similar characteristics**. This can be useful in different application scenarios in data analysis:

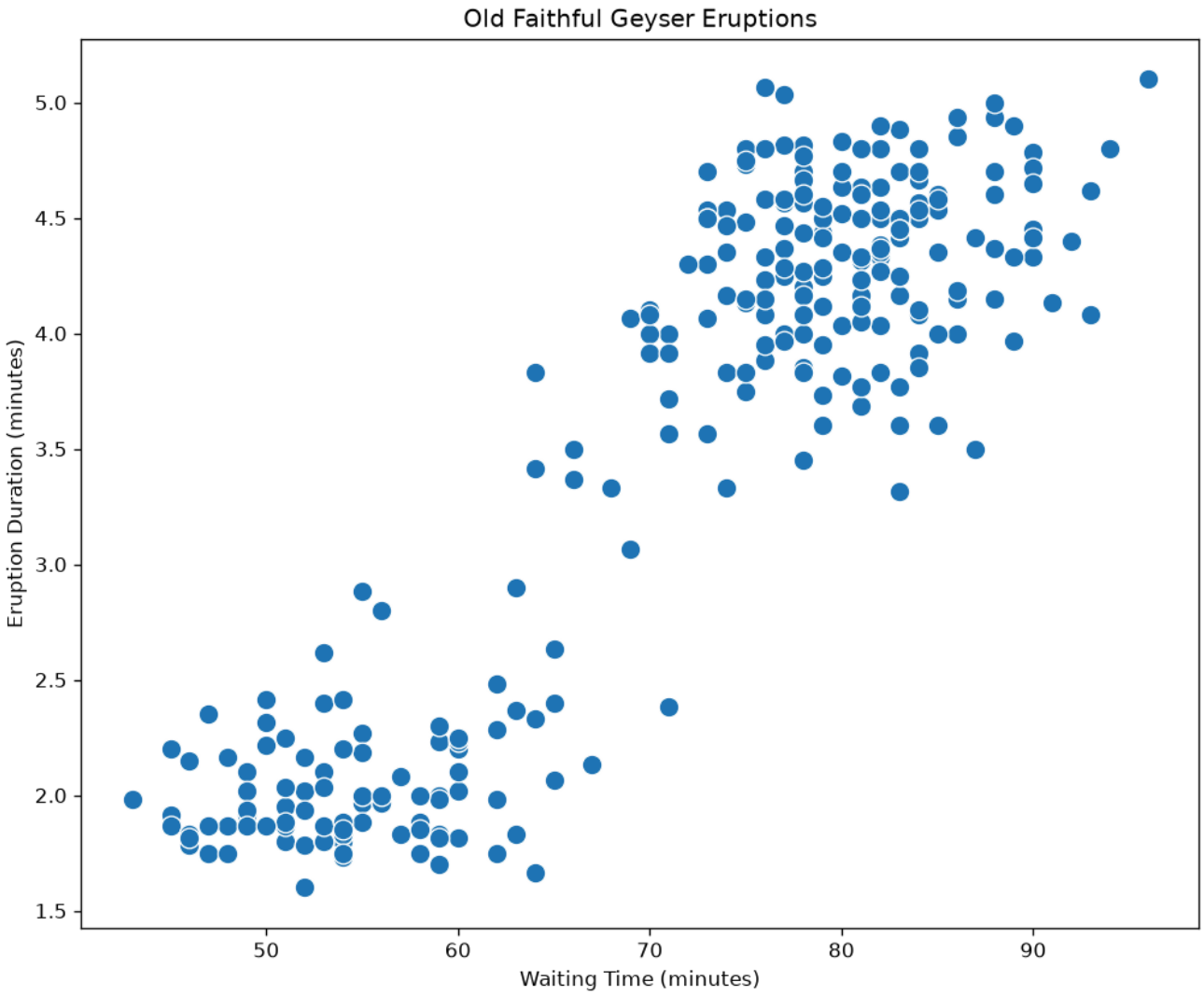
1. **Customer Segmentation:**
   - Grouping customers based on their purchasing behavior, preferences, or demographics for targeted marketing strategies.
2. **Image Segmentation:**
   - Dividing an image into meaningful segments or regions based on similarity, aiding in object recognition and computer vision tasks.
3. **Text Document Clustering:**
   - Grouping similar documents together based on their content for tasks such as topic modeling, document organization, and information retrieval.
4. **Speech and Audio Processing:**
   - Clustering **audio data for tasks such as speaker identification, music genre classification, and speech analysis**.
5. **E-commerce:**
   - Grouping products or users based on purchasing patterns to optimize inventory management and improve the user experience.

Problem Definition

Clustering aims to break the data into distinct group with similar properties. Let us consider the “Old Faithful” dataset, which comprises 272 observations of the eruption of the Old Faithful geyser at the Yellowstone National Park in USA. Each observation includes two variables:

- The **duration** of the eruption in minutes;
- The time in minutes to the **next eruption**.

We can see each observation as 2D vector which can be plotted in a Cartesian coordinate system:



We can see how we clearly have two kinds of eruptions:

- Short eruptions followed by other eruptions in a short time (bottom left);
- Long eruptions followed by other eruptions in a long time (upper right).

A clustering algorithm aims to automatically find the two groups and assign each data point to the most likely group. Note that the two groups will be referred generically as "cluster i", meaning that the algorithm will not assign any semantic meaning to the clusters, but it will aim to put "similar" data points in the same cluster.

Let

$$\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{N}, \ \mathbf{x}^{(i)} \in \mathfrak{R}^n$$

be a set of observations. The goal of clustering is to split $\mathbf{X}$ into K groups (called clusters):

$$S = \{S_1, S_2, \ldots, S_K\}$$

Such that:

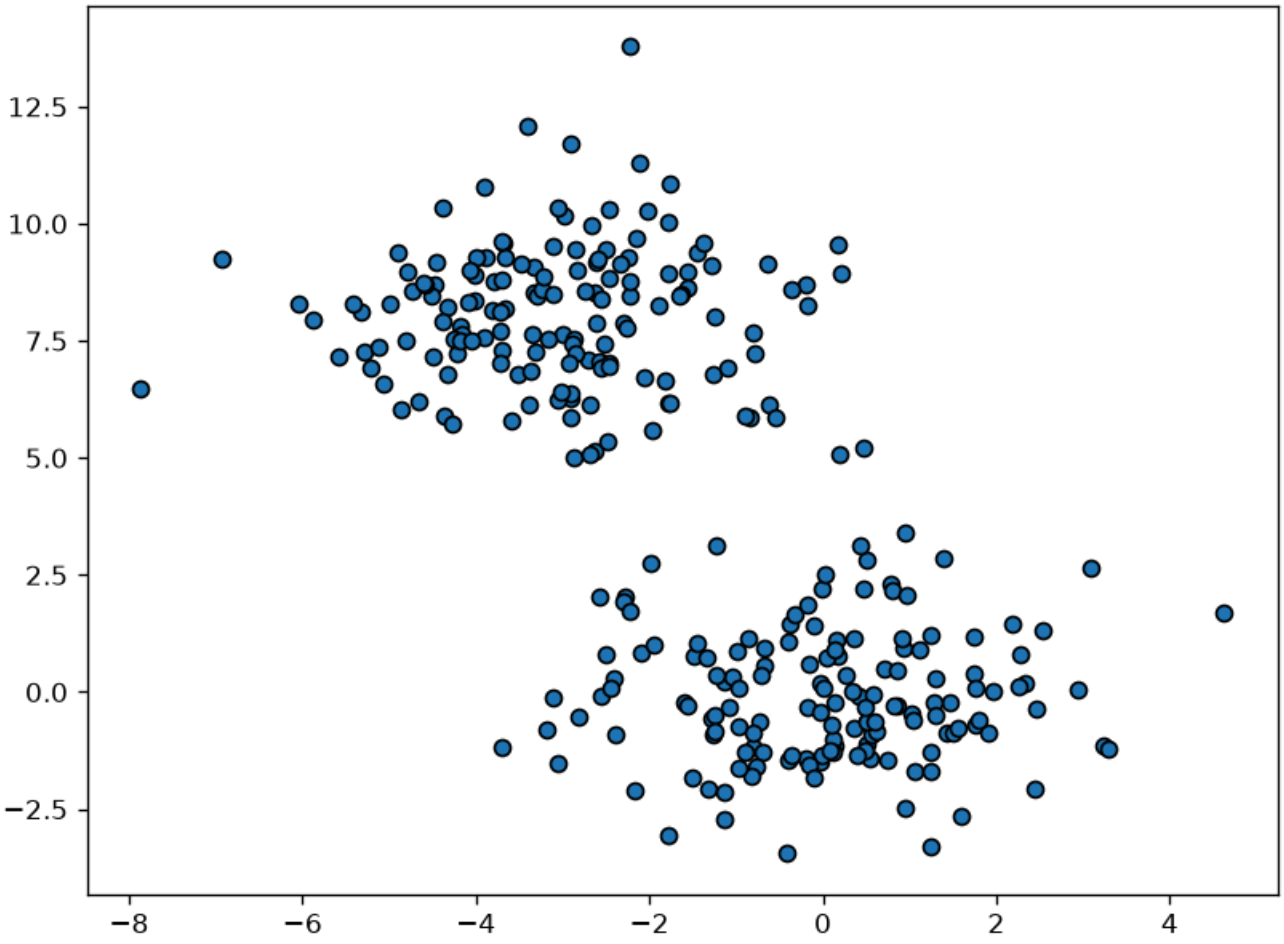
$$\forall \mathbf{x}^{(i)} \in \mathbf{X} \; \exists! S_j \in S \; : \mathbf{x}^{(i)} \in S_j$$

The clusters are expected to be such that elements within a group are **similar to each other**, whereas elements belonging to different groups are different from each other.

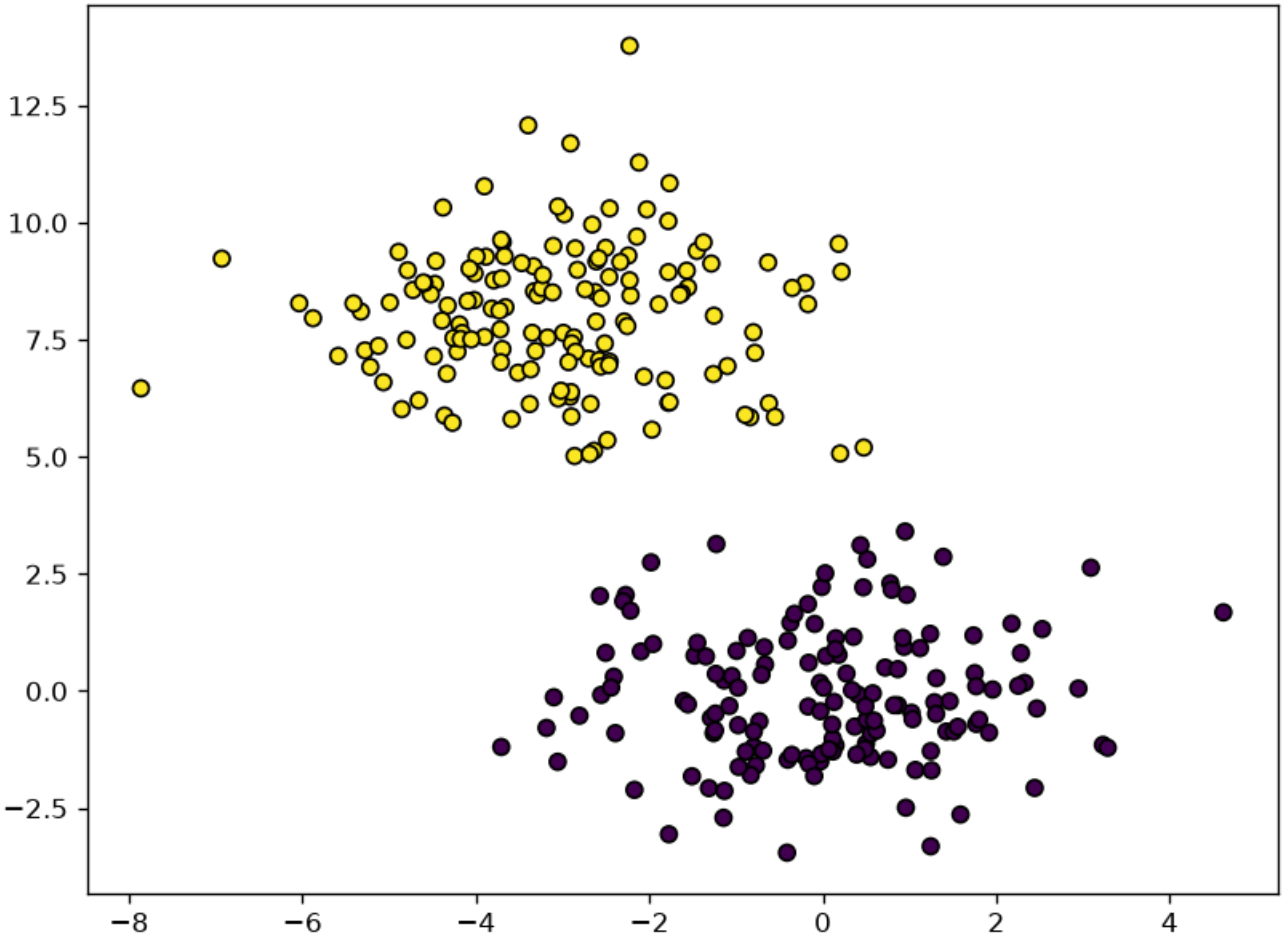
The number of clusters $\mathbf{K}$ is often a hyper-parameter of the algorithm.

Graphical Example

For instance, consider the following 2D points:



Even if we the points are not explicitly assigned to given groups (i.e., we are not observing any discrete variable y indicating the group to which each point belong), we can clearly see that there are two distinct groups (the one in the top-left part of the plot, and the one in bottom-right). The goal of clustering is to split the data into two groups, as it is shown below:



Also note that, while different possible partitions of the 2D points are possible, the one outlined in the plot guarantees that points within the same group are similar to one another (i.e., their Euclidean distance is small), whereas points from different groups have are dissimilar (i.e., their Euclidean distance is large).

K-Means Clustering

K-Means is the most popular clustering algorithm. As in the definition we just gave of clustering, the goal of K-Means is to break the data

$$\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{N}, \ \mathbf{x}^{(i)} \in \mathfrak{R}^n$$

into K clusters

$$S = \{S_1, S_2, \ldots, S_K\}$$

such that:

$$\forall \, \mathbf{x}^{(i)} \in \mathbf{X} \; \exists! S_j \in S \; : \mathbf{x}^{(i)} \in S_j$$

More specifically, K-Means attempts to create K clusters which are **as compact as possible**, where K is a parameter specified by the user (its application may depend on the application).

We will start by defining K different vectors $\mu_k \in \Re^n$ which will act as prototypes for our clusters. We will shortly see that these vectors will be the cluster's centers.

Hence, we define $N \times K$ binary variables r_{ij} which will allow us to establish a mapping between data points $\mathbf{x}^{(i)}$ and clusters S_j. In particular, we will define:

$$r_{ij} = \begin{cases} 1 & \text{if } \mathbf{x}^{(i)} \in S_j \\ 0 & \text{otherwise} \end{cases}$$

The notion that **clusters should be as compact as possible** is formalized by defining the following **cost function**, which is sometimes called also **distortion function**:

$$J = \sum_{j=1}^{K} \sum_{i=1}^{N} r_{ij} ||\mathbf{x}_i - \mu_j||^2$$

Note that this function J is the total **Within-Cluster Sum of Squares (WCSS)**.

For a single cluster S_j, the term $\sum_{i=1}^{N} r_{ij} ||\mathbf{x}_i - \mu_j||^2$ is the sum of squared distances of all points in that cluster from its center. This is a measure of the cluster's **inertia** or "compactness."

The goal of K-Means is to find the set of centroids $\{\mu_j\}$ and assignments $\{r_{ij}\}$ that minimizes this total inertia J.

The K-Means problem is solved by finding a partition $\widehat{S}$ minimizing the cost function:

$$\{\hat{r}_{ij}\}_{ij}, \{\hat{\mu}_j\}_j = \arg\min \sum_{j=1}^{K} \sum_{i=1}^{N} r_{ij} \left\| \mathbf{x}^{(i)} - \mu_j \right\|^2$$

Optimization

The minimization above can be performed using an iterative algorithm. The first step of the algorithm is to choose K random centroids μ_i. After this initialization, the algorithm iterates the following two steps:

Assignment

Eeach element $\mathbf{x}$ is assigned to the set with the closest centroid

$$r_{ij} = \begin{cases} 1 & \text{if } j = \arg_k \min ||\mathbf{x}^{(i)} - \mu_j||^2 \\ 0 & \text{otherwise} \end{cases}, \forall i, j$$

Which will lead to the partition:

$$S_i = \left\{ \mathbf{x} \in \mathbf{X} : \ \|\mathbf{x} - \mu_i\|^2 \leq \|\mathbf{x} - \mu_j\|^2 \ \ \forall \, j \in \{1, \dots K\} \right\}$$

Update

The centroids μ_i are re-computed from the assigned sets

$$\mu_j = \frac{1}{|S_j|} \sum_{\mathbf{x} \in S_j} \mathbf{x} = \frac{\sum_{i=1}^{N} r_{ij} \mathbf{x}^{(i)}}{\sum_{i=1}^{N} r_{ij}}$$

The algorithm converges when the update does not change any centroid. In some cases, the algorithms may never actually converge, hence it is often common to introduce as a termination criterion a maximum number of iterations.

The algorithm is not guaranteed to find the global optimum and the solution reached may depend on the random initialization. However, in practice it usually leads to a reasonable solution.

Pseudocode

We can see the optimization algorithm in pseudo-code as follows:

Randomly initialize K cluster centroids $\mu_1, \mu_2, \dots, \mu_K \in \Re^n$

Repeat until termination criterion is reached {

for i = 1 to N

$$r_{ij} = \begin{cases} 1 & \text{if } j = \arg_j \min ||\mathbf{x}^{(i)} - \mu_j||^2 \\ 0 & \text{otherwise} \end{cases}, \forall i, j$$ //assignment

for j = 1 to K

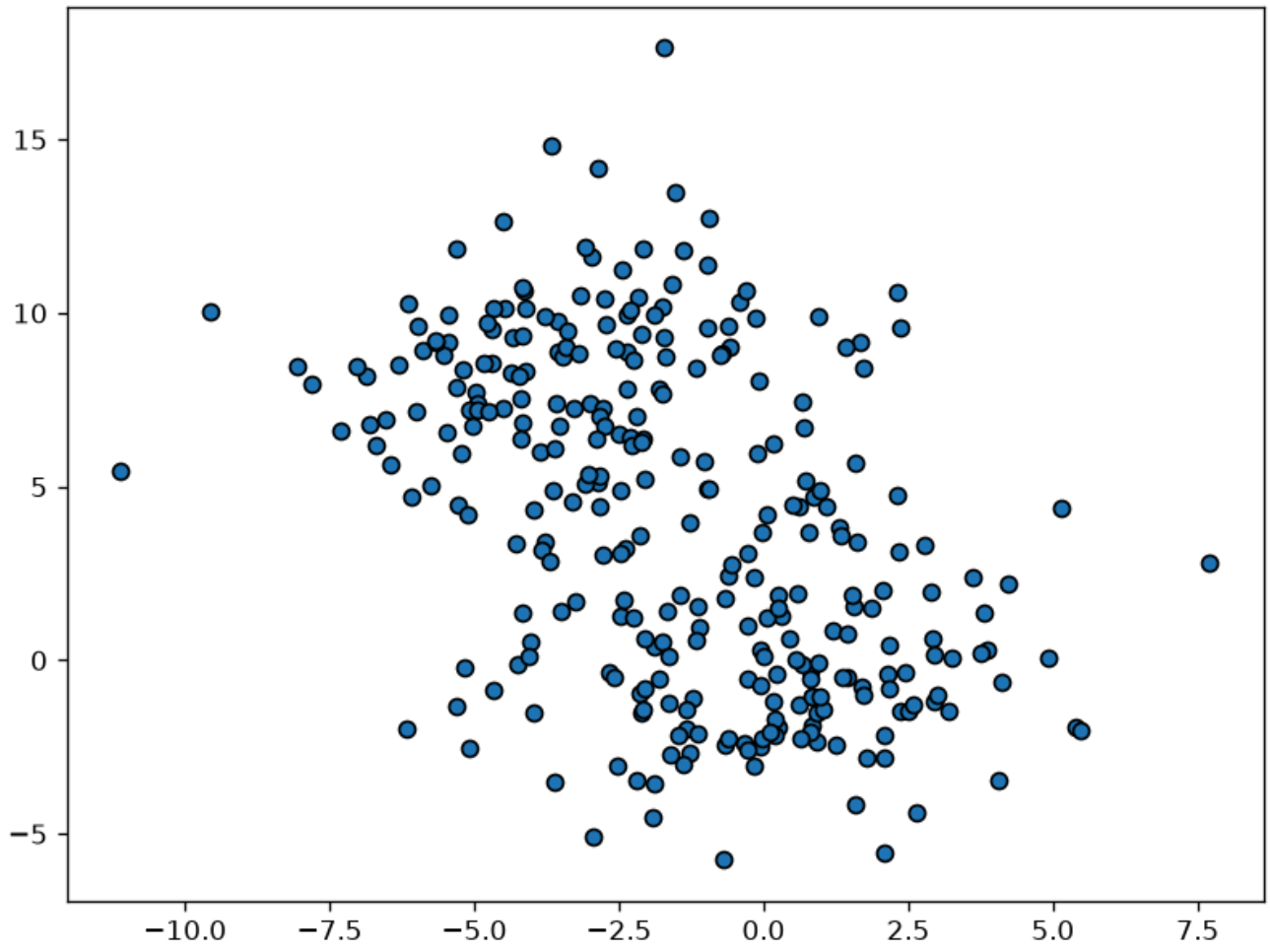
$$\mu_j = \frac{\sum_{i=1}^{N} r_{ij} \mathbf{x}^{(i)}}{\sum_{i=1}^{N} r_{ij}}$$ //update

}

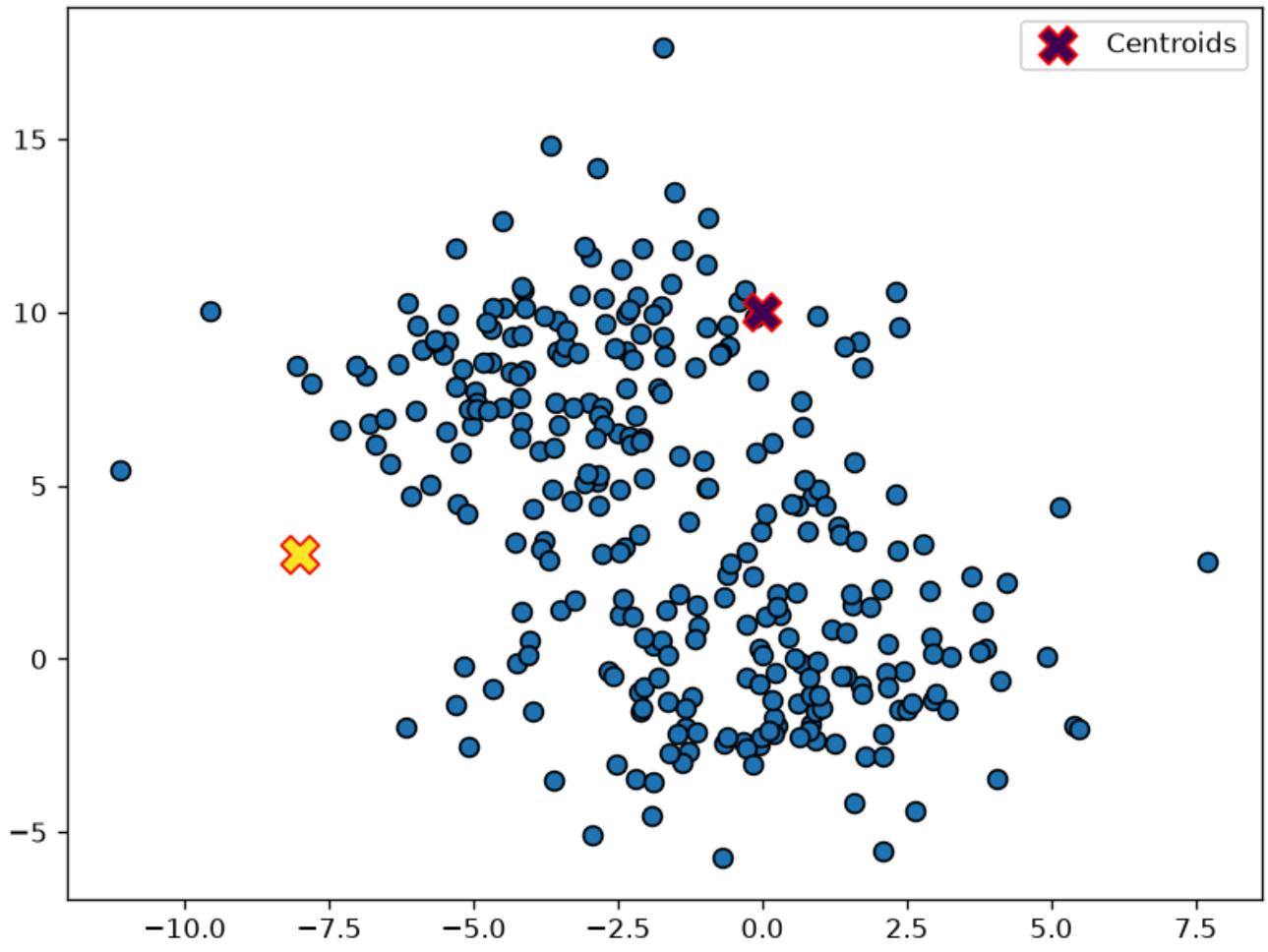
Example Execution

Let us see a graphical example of the process.

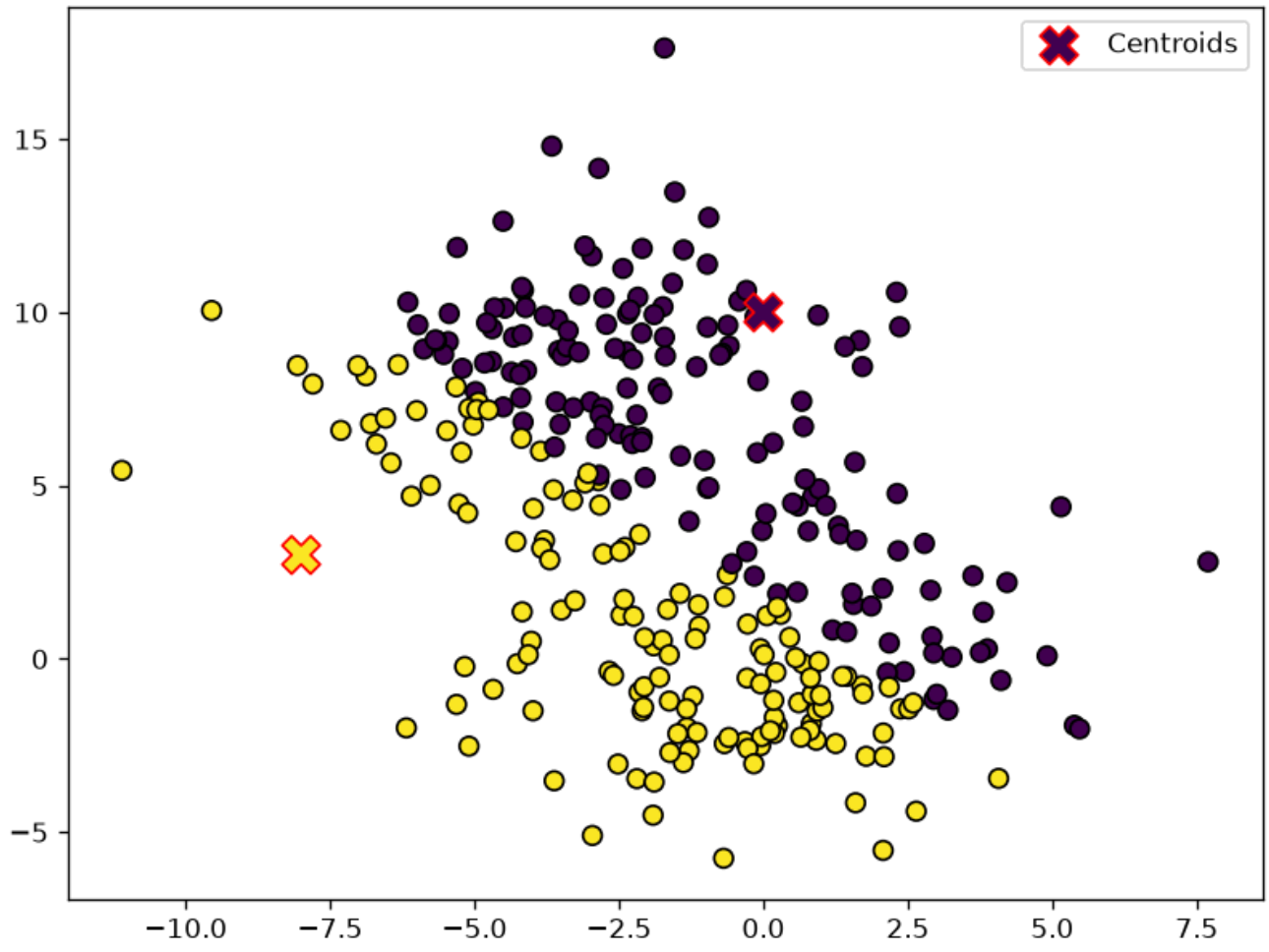
Dataset



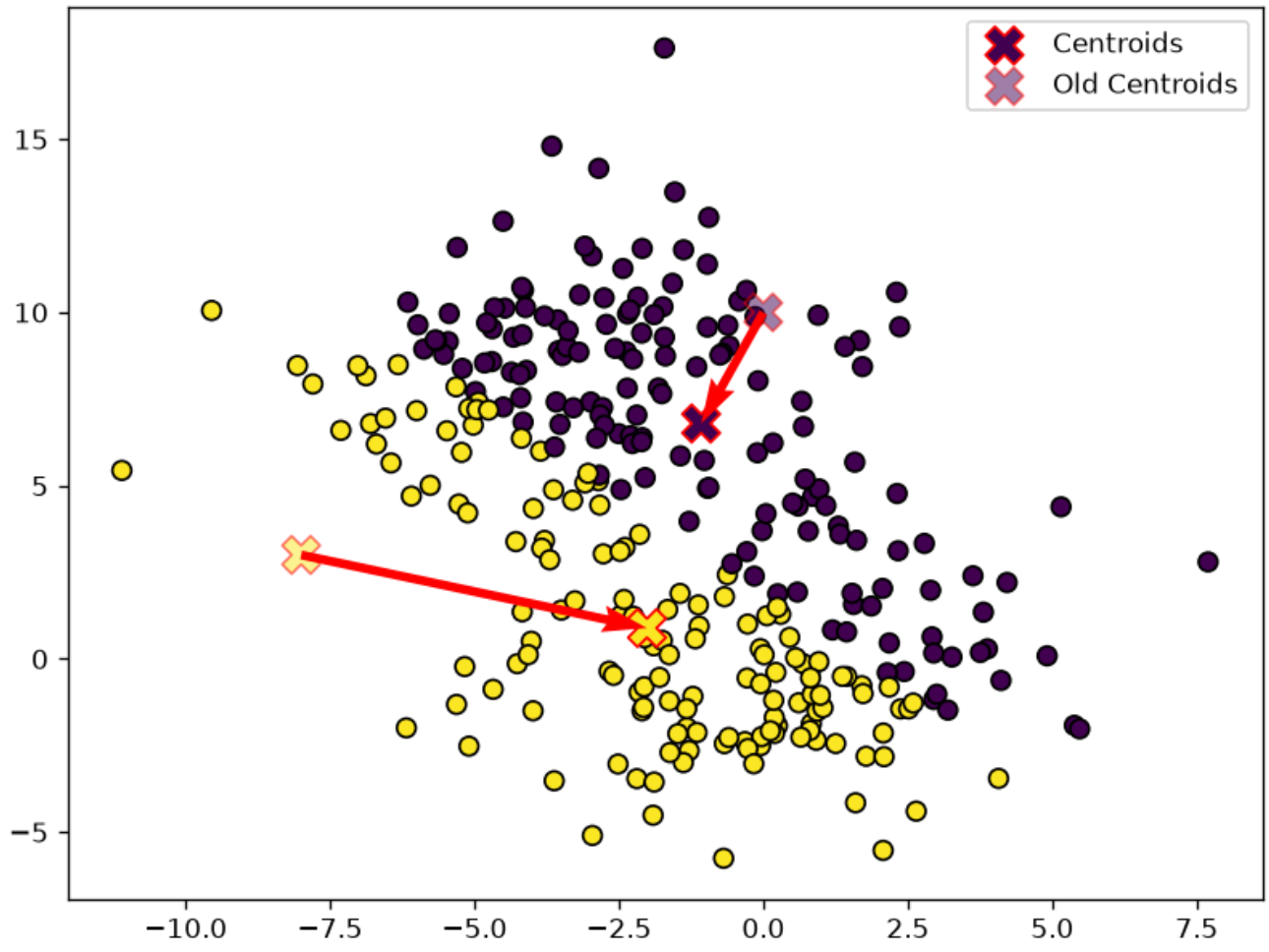
Initialization (Random)



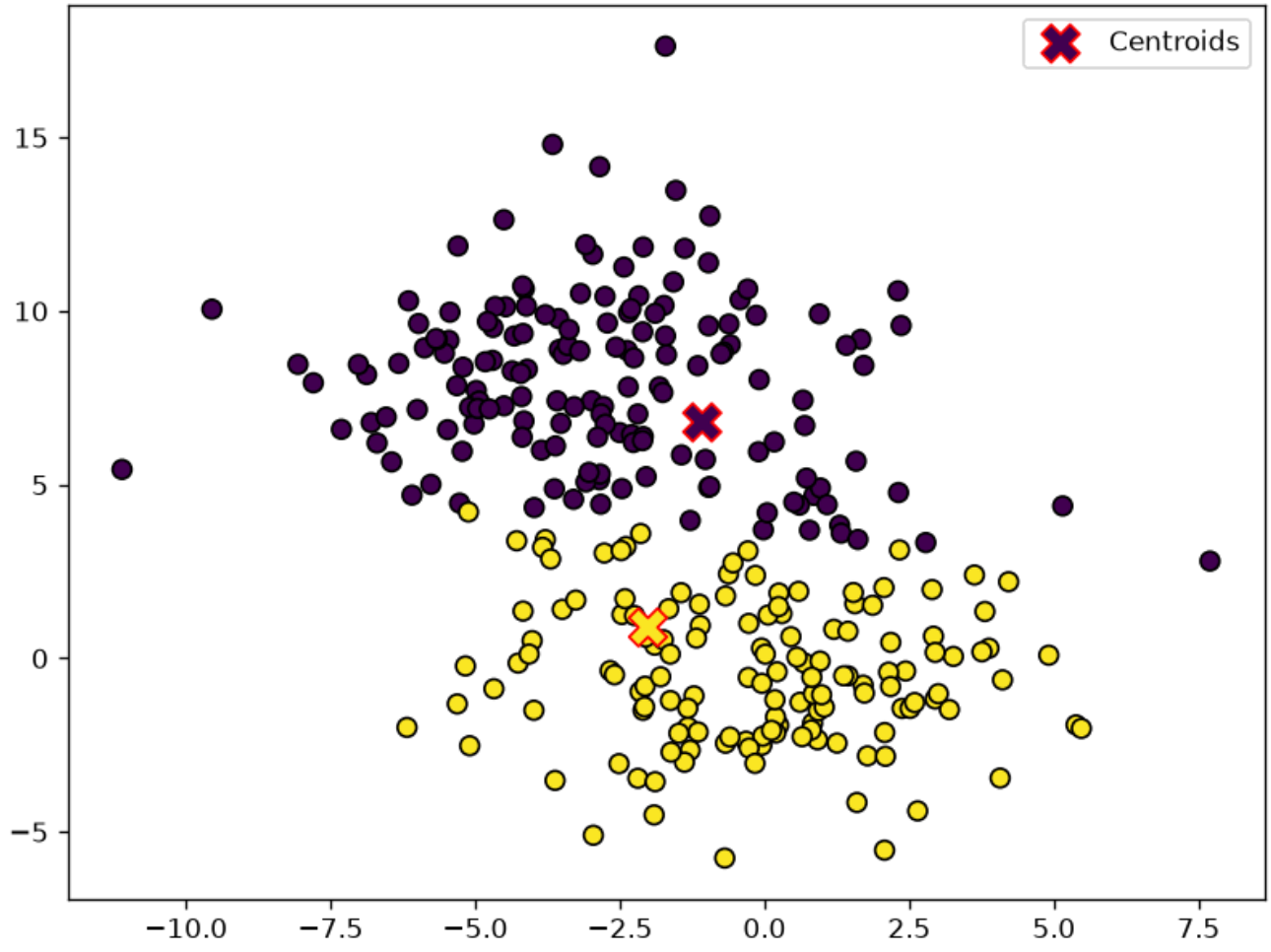
Assignment (1)



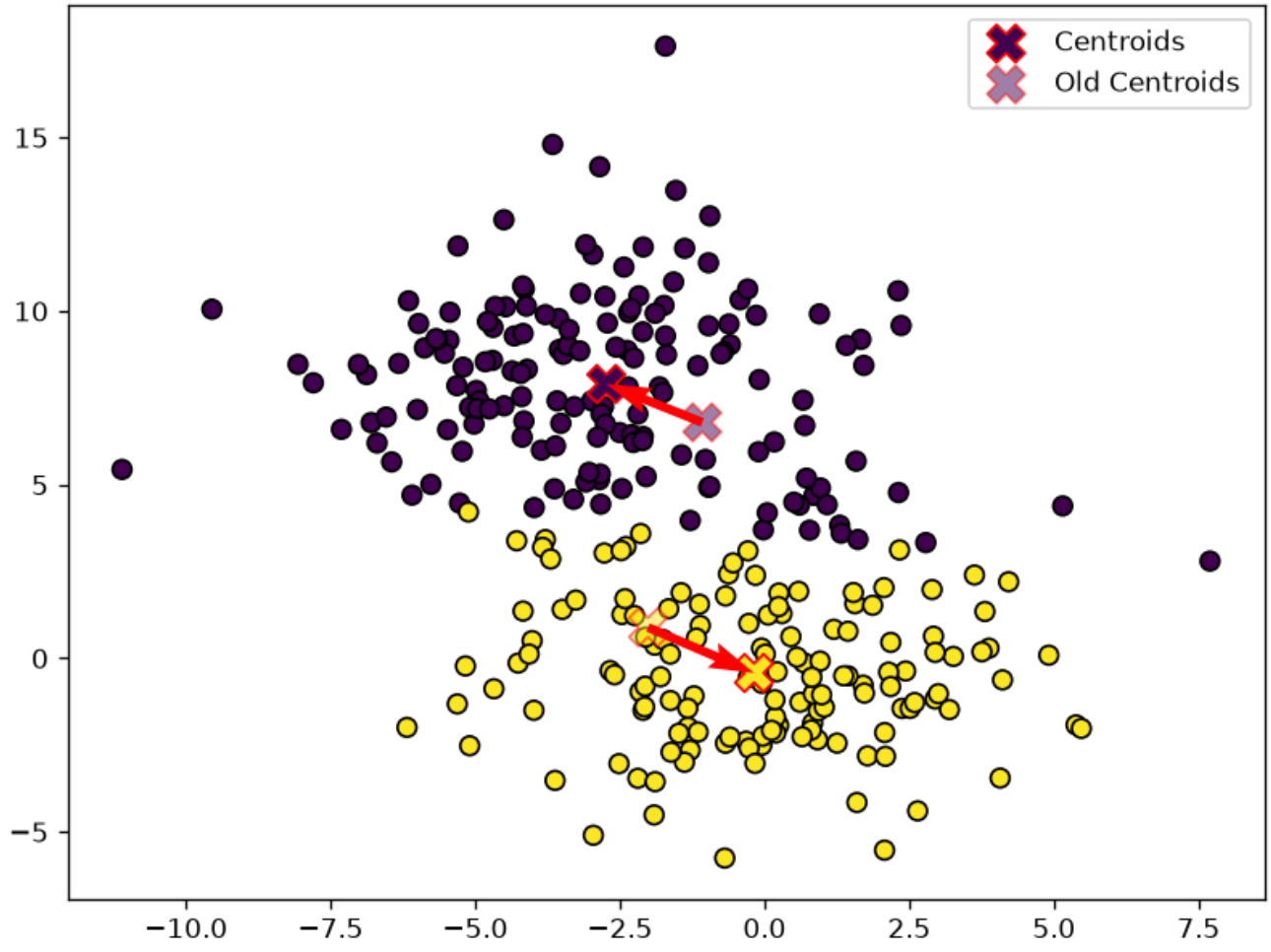
Update (1)



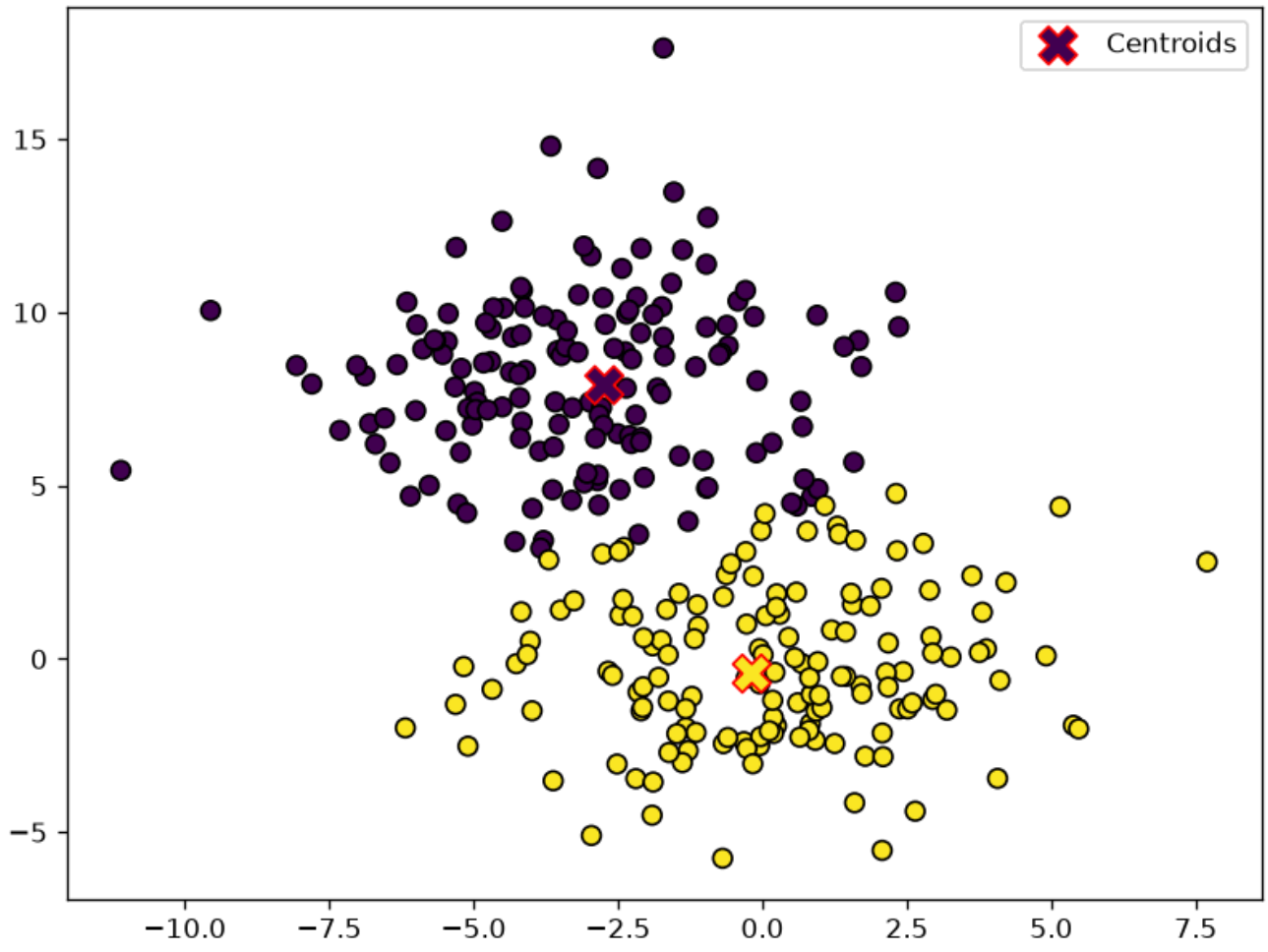
Assignment (2)



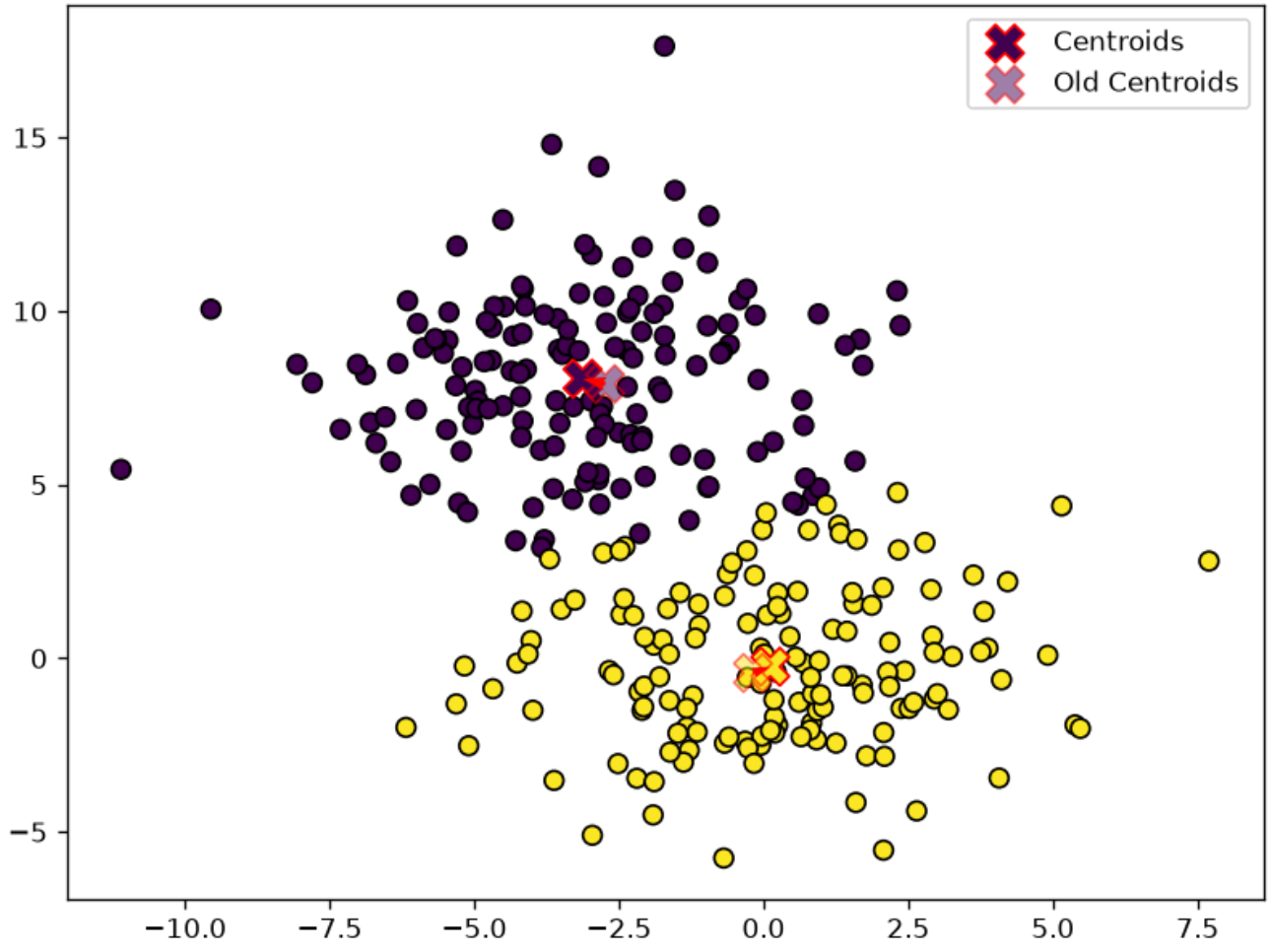
Update (2)



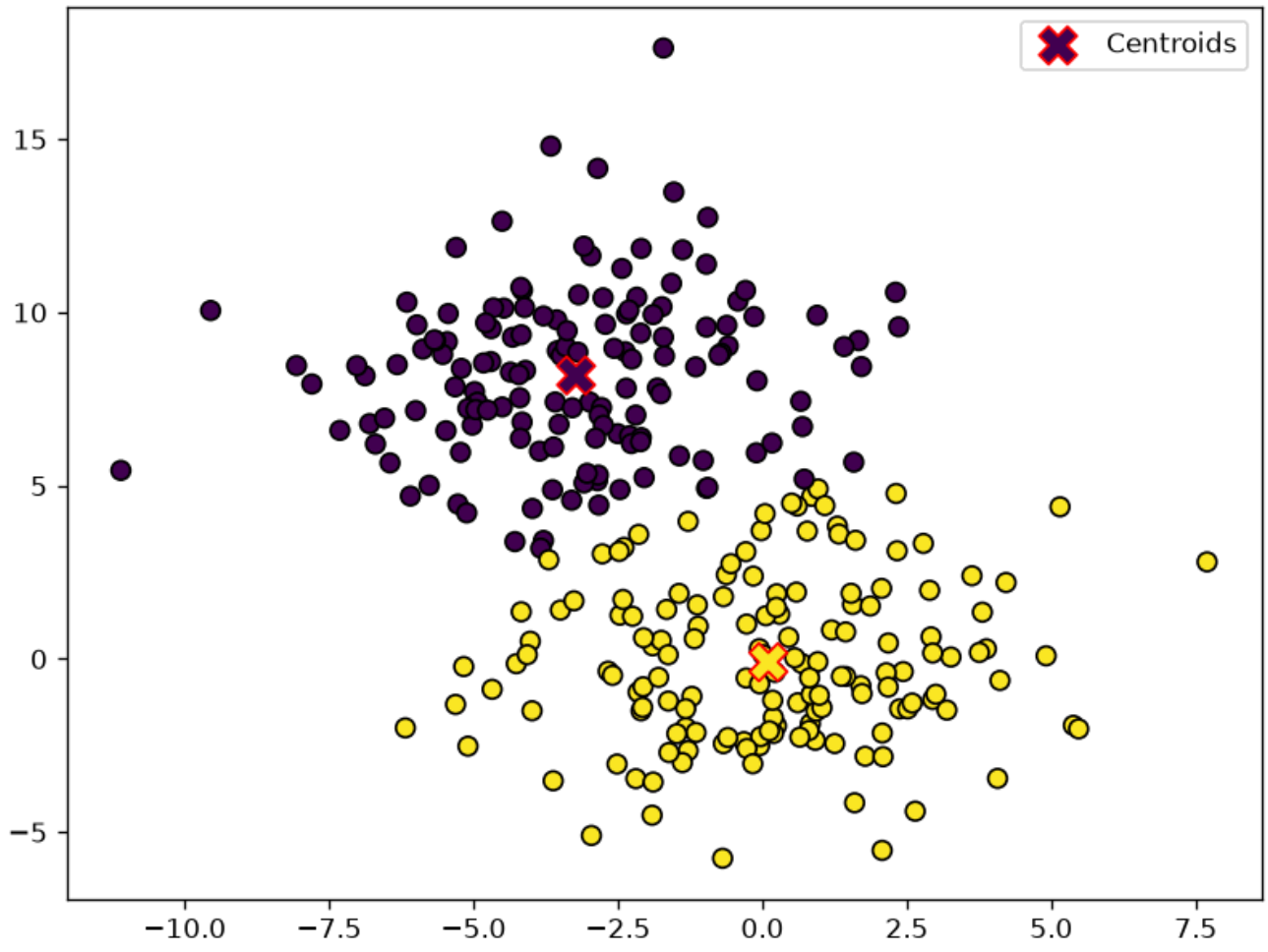
Assignment (3)



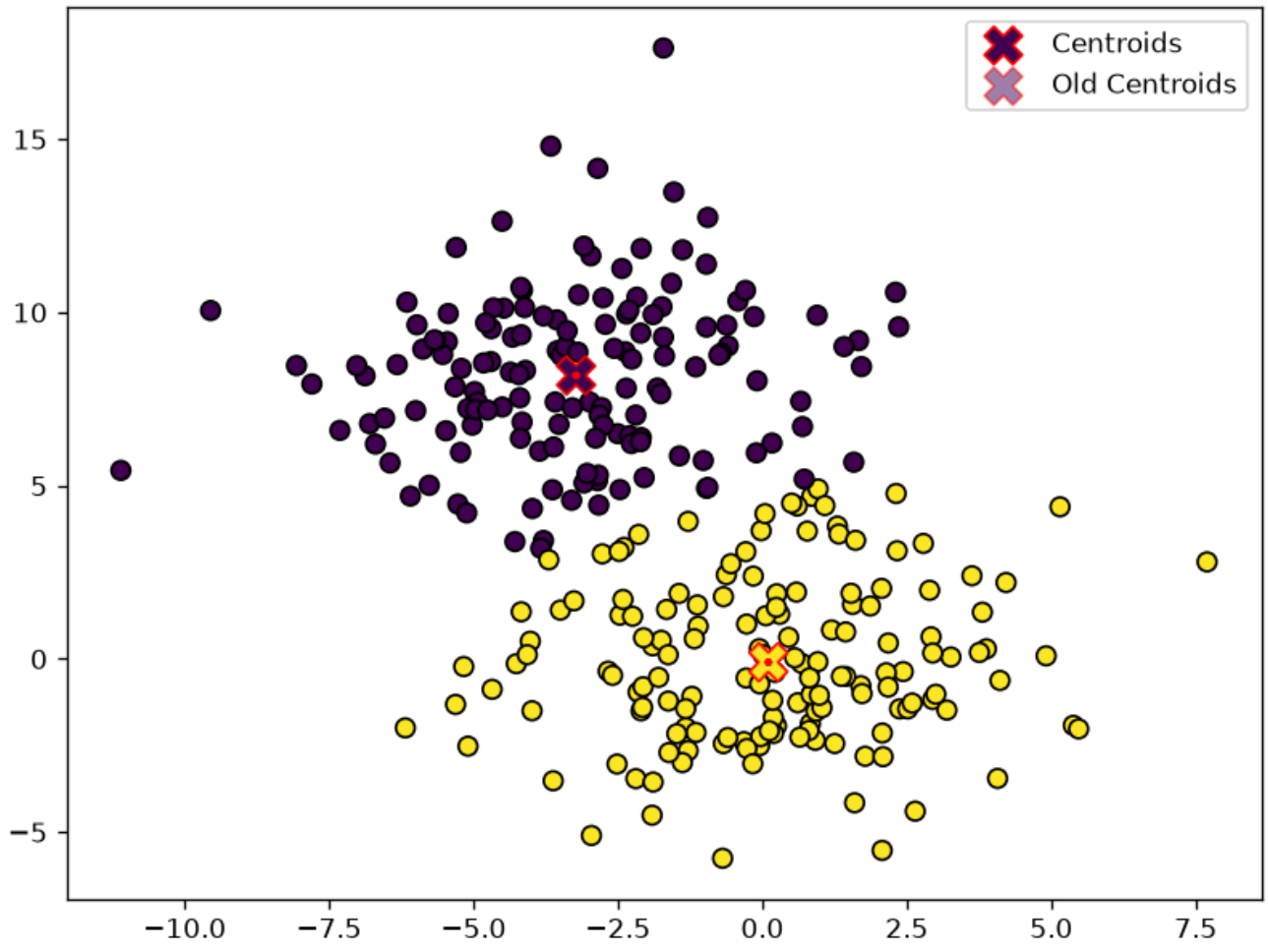
Update (3)



Assignment (4)

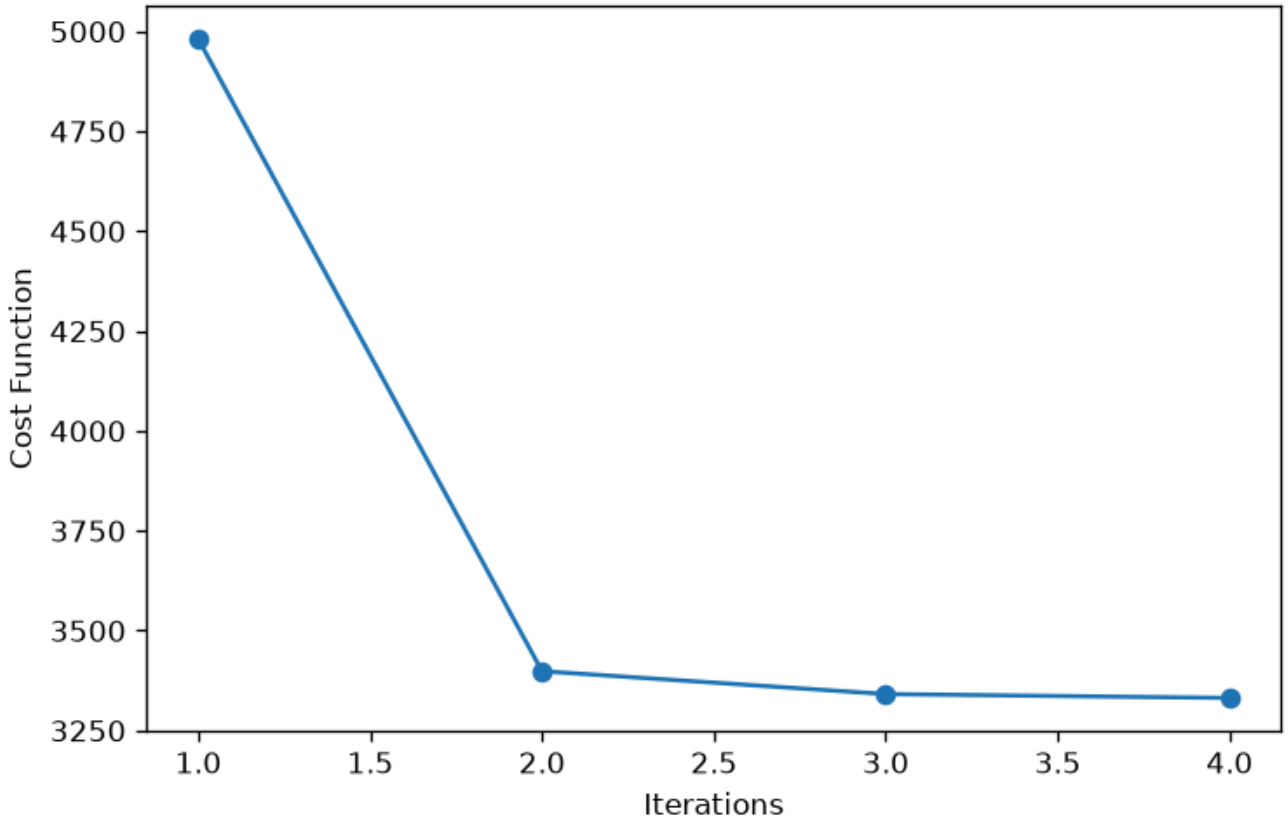


Update (4)



The optimization procedure terminates here as the centroids did not move in the last update step. The data is now clustered in two groups.

The plot below shows the value of the cost function at the end of each iteration:



Choosing the Right Value for K

In the K-Means clustering algorithm, K is a **hyper-parameter**. This means that **it is a parameter of the method affecting the final model we obtain after the optimization procedure**, but **its value is not automatically determined by the optimization procedure**.

One common approach to determine the values of hyper-parameters is to make some guesses and fit different models with the guessed values of the hyper-parameters. We can then choose the model which has the "best value" of the considered hyper-parameters.

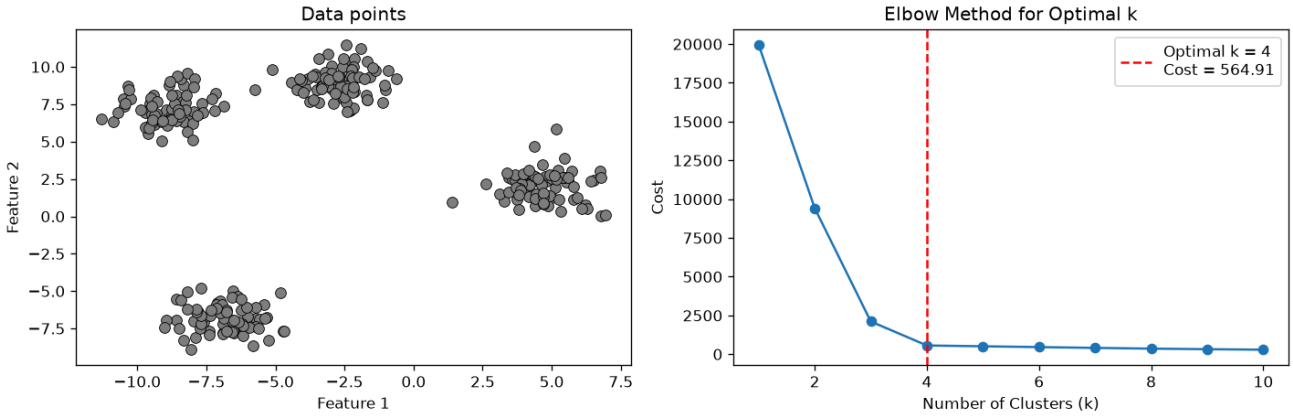
When it turns to determining the optimal K value for K-Means clustering, there are two main techniques which are commonly used: **the elbow method** and **the silhouette method**, which are discussed in the following.

Note that these are both **heuristic methods not giving many guarantees on the final selection of K**, but they can still help decide on the number of clusters, in particular if guided by an intuition of what a good value of K would look like.

Elbow Method

The elbow method consists in fitting K-Means models for different values of K. For each K, we then plot the final value of the cost function. We expect that **as K increases, the cost will decrease**. Indeed, if K is very large (even larger than it should be), we will anyway get a smaller cost due to the fact that the presence of more clusters will inevitably lower their individual variances.

However, we expect the cost function to **slow the speed of decrease in the presence of the optimal (or a good)** K. This is called the "elbow point" as the curve should look like an elbow. We can see it in the example below:



Silhouette Method

The silhouette method tries to assess how good a given clustering model is by measuring **how well the data is grouped in the identified clusters**. This is done by computing, **for each data point** a score indicating whether **how well the data point fits the assigned cluster with respect to all others**.

If the model is a good one (hence a good K value), we would expect each data point to fit well its assigned cluster and not to fit well any other cluster. If this does not happen, it means that the clusters are not very well separated and we **either have more or less clusters**.

To do so, for a given data point i assigned to cluster C_I, the silhouette method computes the following score:

$$a(i) = \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} d(i, j)$$

where $d(i, j)$ is the Euclidean distance between data point i and data point j. In practice, the score $a(i)$ is the **average distance of data point i with any other data point assigned to the same cluster**. We expect this number to be small if the cluster has a low variance.

We now want to compare this number to the value we would have **if i were to be assigned to another cluster** C_J. In particular, we will choose the cluster C_J which minimizes the average distance. This is done by computing the following score:

$$b(i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j)$$

Now we compare $a(i)$ with $b(i)$. Ideally, $a(i)$ should be much smaller than $b(i)$. If this happens, this means that the data point i fits in the cluster C_I much better than it would in any other cluster. To

do so, we compute the **silhouette score** of data point i as follows:

$$s(i) = \begin{cases} \frac{b(i)-a(i)}{\max\{a(i),b(i)\}} & \text{if } |C_I| > 1 \\ 0 & \text{otherwise} \end{cases}$$

The obtained score is such that:

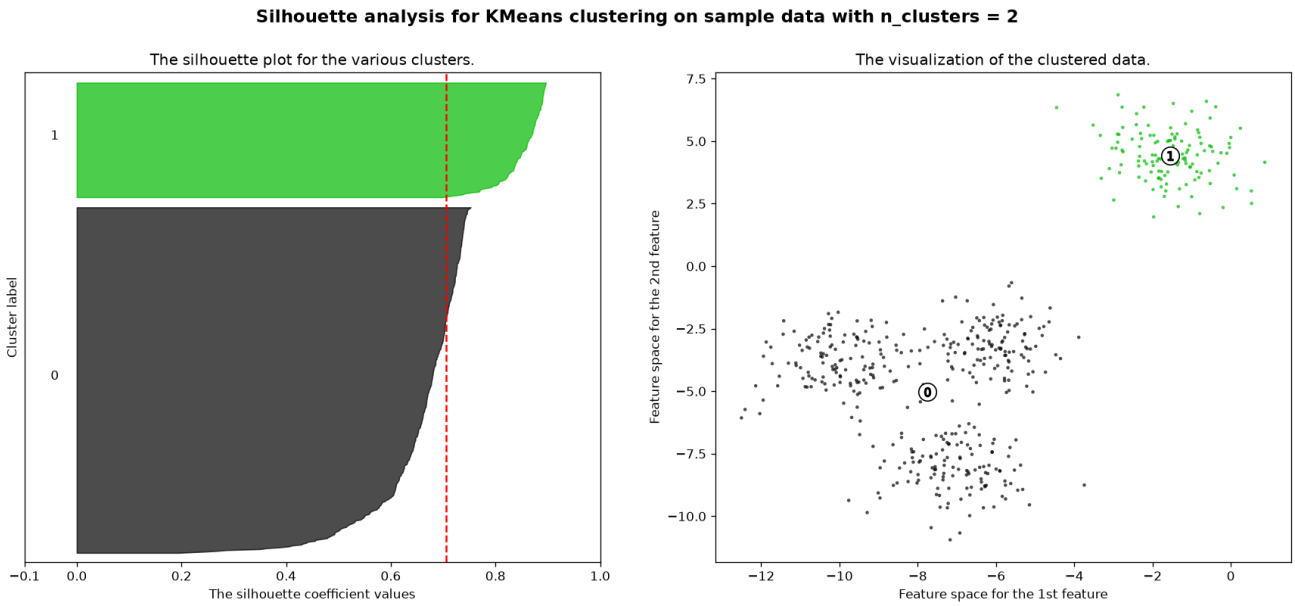
$$-1 \leq s(i) \leq 1$$

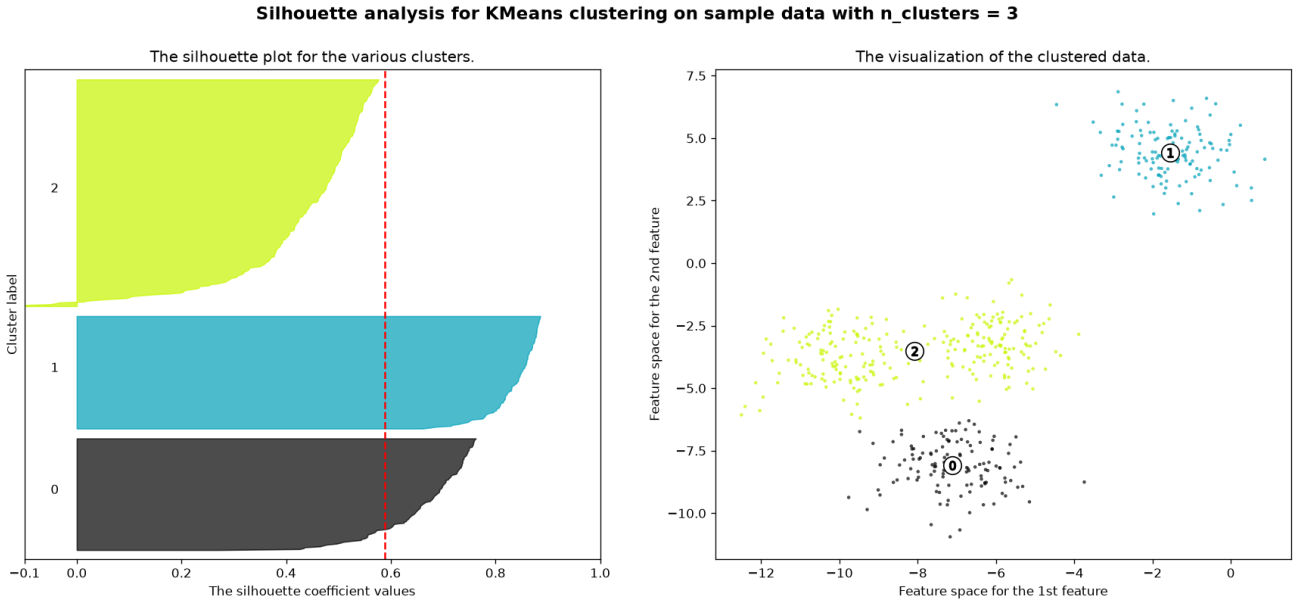
In practice:

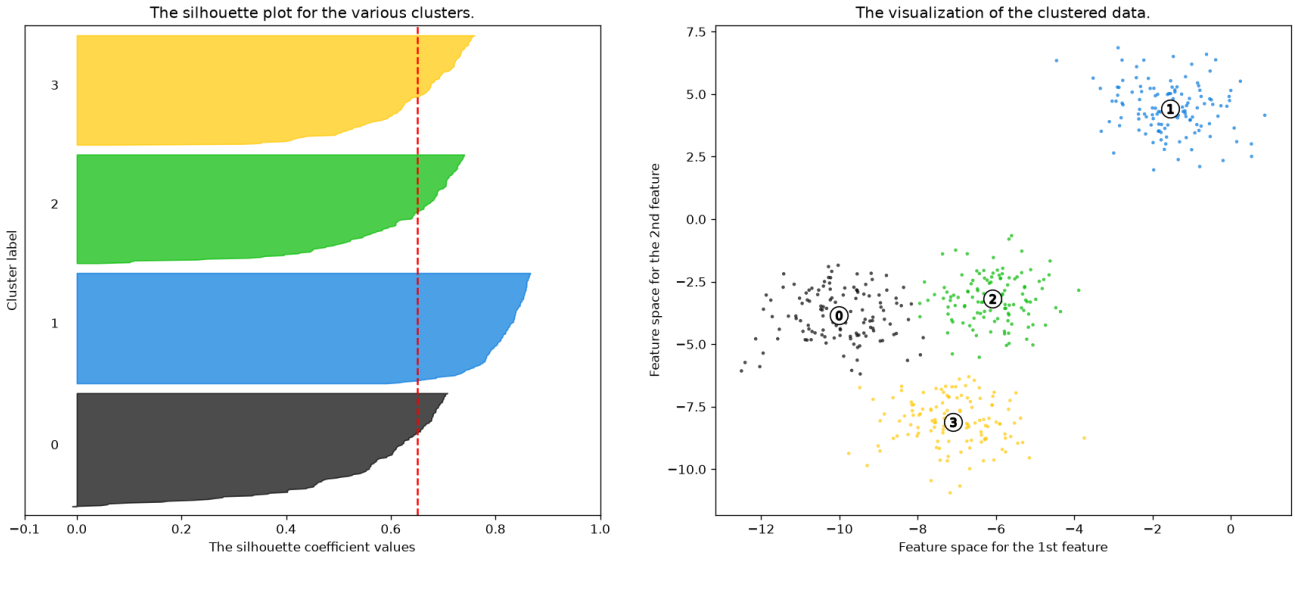
- A value of $s(i)$ close to 1 is obtained when $a(i) << b(i)$, meaning that i fits in C_I much better than it would fit in any other cluster;
- A value of $s(i)$ close to -1 is obtained when $a(i) >> b(i)$, meaning that another cluster C_J exists in which $a(i)$ fits better.
- A value of $s(i)$ close to 0 indicates that the data point is close to the **border between two natural clusters**.

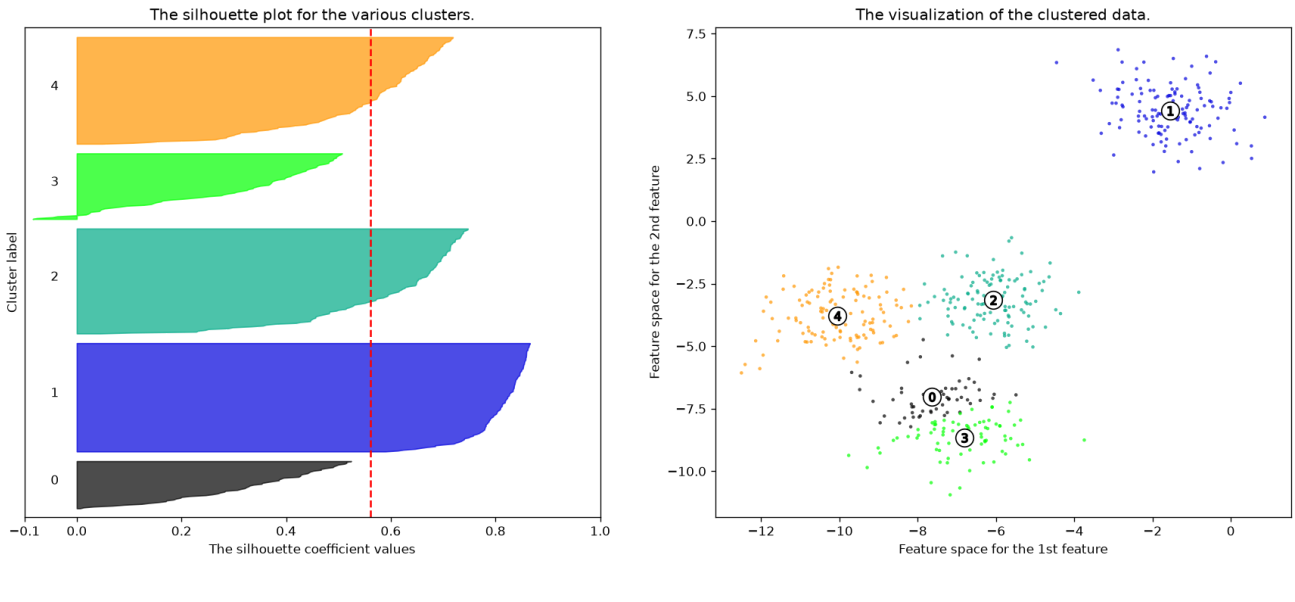
The score above is arbitrarily set to 0 if $|C_I| = 1$ as, in that case, the cluster would only contain the data point i. Note that this is **an assumption of neutrality**.

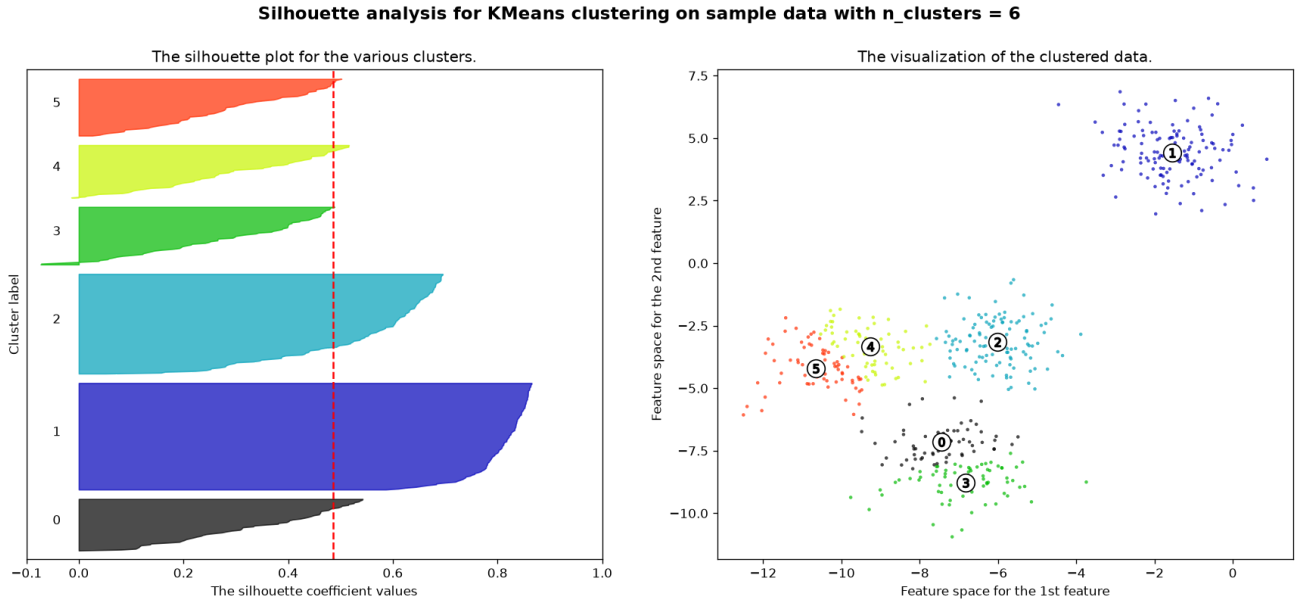
The correct number of clusters is usually chosen by showing the silhouette plot, which displays all the silhouette scores of data points within each clusters, arranged from largest to smallest. The plots below are taken from the scikit-learn documentation and show examples of silhouette plots for different choices of K in an example dataset:




Silhouette analysis for KMeans clustering on sample data with n_clusters = 3
The silhouette plot for the various clusters.
Cluster label
The silhouette coefficient values
The visualization of the clustered data.
Feature space for the 2nd feature
Feature space for the 1st feature


Silhouette analysis for KMeans clustering on sample data with n_clusters = 4
The silhouette plot for the various clusters.
Cluster label
The silhouette coefficient values
The visualization of the clustered data.
Feature space for the 2nd feature
Feature space for the 1st feature


Silhouette analysis for KMeans clustering on sample data with n_clusters = 5
The silhouette plot for the various clusters.
Cluster label
The silhouette coefficient values
The visualization of the clustered data.
Feature space for the 2nd feature
Feature space for the 1st feature

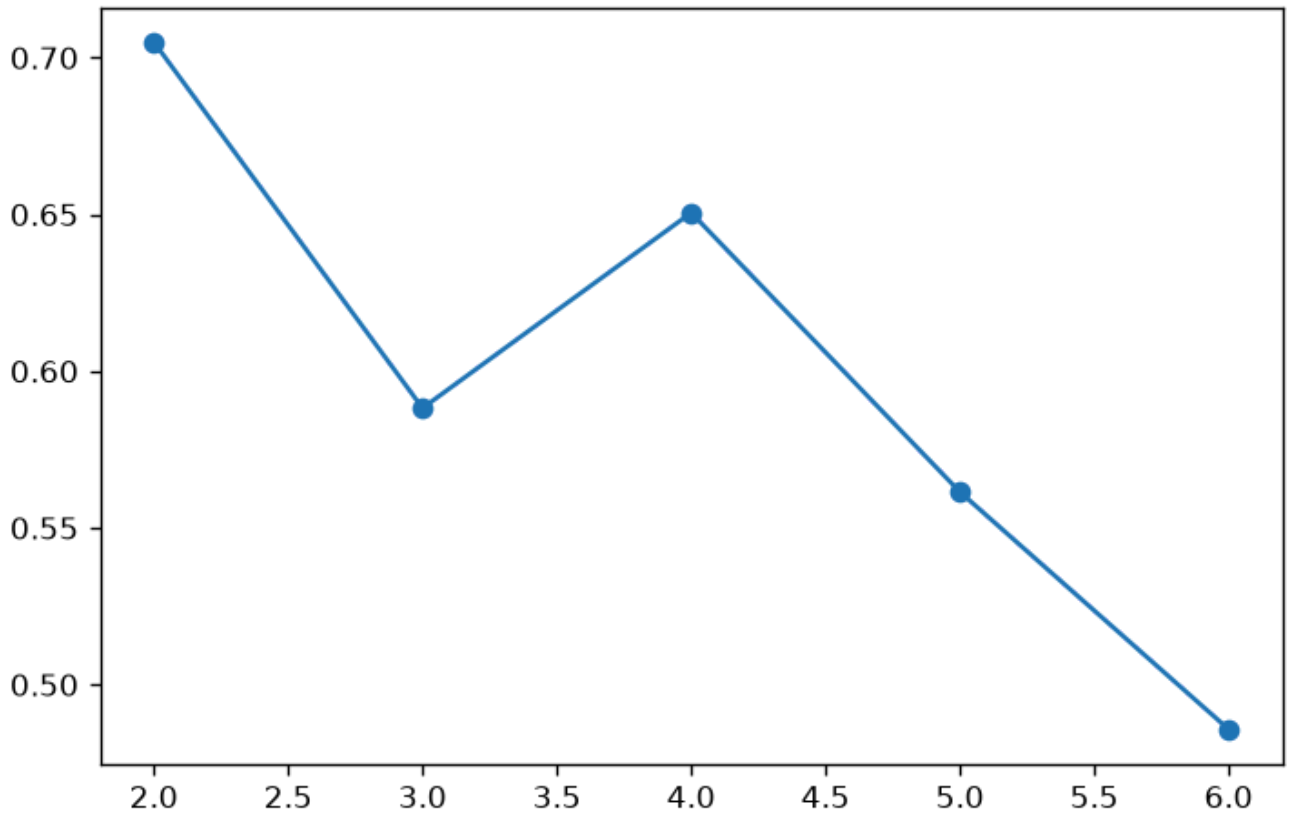


The vertical line in red displays the average silhouette score. Good choices of cluster numbers will have silhouette plots which achieve values similar to each other and close to the average one. For example $n = 2$ and $n = 4$ seem to be the best ones.

In practice, it is also useful to plot the average silhouette score versus the number of clusters and choose value of K which **maximizes** the average score.

Such plot would be as follows for the previous example:

```
plt.plot(range_n_clusters,silhouettes, '-o')
plt.show()
```



The plot below shows another example of such plot for a different dataset:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score

# Generate synthetic data
X, y = make_blobs(n_samples=300, centers=4, random_state=42)

# Function to compute K-means silhouette score for a given model and data
def compute_kmeans_silhouette(model, data):
    labels = model.predict(data)
    return silhouette_score(data, labels)

# Range of k values to test
k_values = range(2, 11)
silhouette_scores = []

# Compute silhouette score for each k
for k in k_values:
    kmeans_model = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans_model.fit(X)

    silhouette = compute_kmeans_silhouette(kmeans_model, X)
    silhouette_scores.append(silhouette)

# Find the optimal number of clusters using the silhouette method
optimal_k_silhouette = k_values[np.argmax(silhouette_scores)]
optimal_silhouette = np.max(silhouette_scores)

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

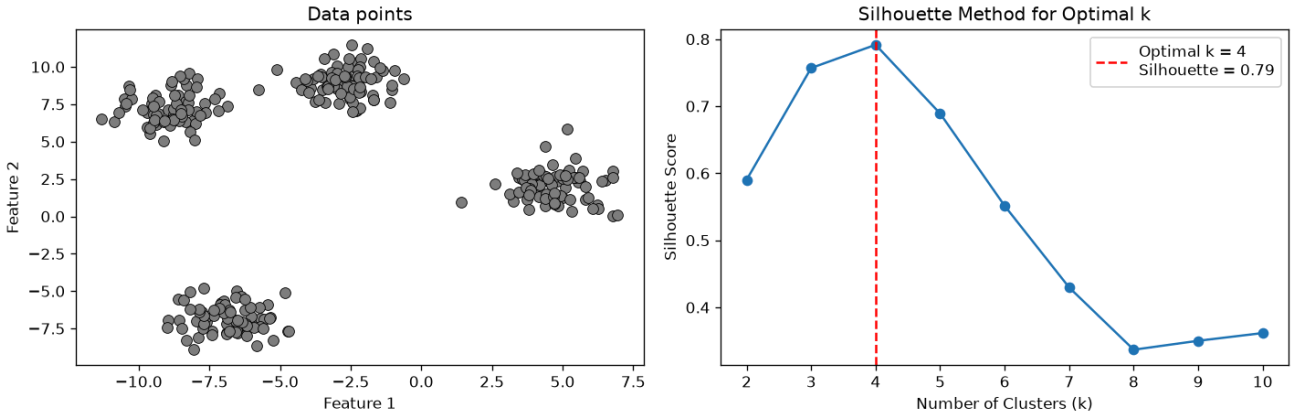
# Plot the data on the left
axs[0].scatter(X[:, 0], X[:, 1], c='gray', marker='o', edgecolors='k', s=50,
        linewidth=0.5)
axs[0].set_title('Data points')
axs[0].set_xlabel('Feature 1')
axs[0].set_ylabel('Feature 2')

# Plot silhouette scores on the right
axs[1].plot(k_values, silhouette_scores, marker='o')
axs[1].set_title('Silhouette Method for Optimal k')
axs[1].set_xlabel('Number of Clusters (k)')
axs[1].set_ylabel('Silhouette Score')

# Highlight the optimal number of clusters
axs[1].axvline(x=optimal_k_silhouette, color='red', linestyle='--', label=f'Optimal k
        = {optimal_k_silhouette}\nSilhouette = {optimal_silhouette:.2f}')
```

```
axs[1].legend()

plt.tight_layout()
plt.show()
```



Other Clustering Algorithms

It is important to note that various clustering algorithms exist beyond the commonly discussed k-means method. While k-means is a popular and widely used algorithm for partitioning data into distinct groups based on similarity, it is one approach among many. Numerous other clustering algorithms offer different perspectives and address specific challenges. Some examples include **hierarchical clustering**, **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise), and **spectral clustering**. These alternatives provide unique strengths and are tailored to different types of data and scenarios. While our focus has been on understanding clustering in general and the k-means algorithm in particular, it is important to know that other approaches may be more suitable depending on the specific problem at hand. In the next lectures, we will see **Gaussian Mixture Models (GMMs)**, a probabilistic model for density estimation which can also be interpreted as a clustering algorithm.

Lab: K-Means Clustering for Customer Segmentation

In this lab, we will apply the K-Means algorithm to a classic, real-world business problem: **Customer Segmentation**.

The Goal

Our goal is to use K-Means to "discover" natural groups within a mall's customer base. We'll use a dataset of mall customers and their `Annual Income` and `Spending Score`.

By finding these clusters, a marketing team could create targeted campaigns. For example, a "high income, low spending" group needs a different message than a "low income, high spending" group.

The Workflow

1. Load and explore the data (using a `pairplot` to spot relationships).
2. **Prepare the data:** Select our features and, most importantly, **scale them**.
3. **Find the best K:** Use both the **Elbow Method** and the **Silhouette Method** to determine the optimal number of clusters.
4. **Fit the final model:** Train our final K-Means model with the best K.
5. **Visualize and Interpret:** Plot the final clusters and, most importantly, *interpret what they mean* from a business perspective.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Import the tools for clustering
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score, silhouette_samples
```

Step 1: Load and Explore the Data

We'll load the "Mall Customers" dataset from a public URL. This dataset is perfect for clustering.

```
# Load the dataset. Not served from the site: the source grants no licence, so we
        read
# it from the original URL rather than redistribute it. See
        archive/datasets/README.md.
url = ("https://gist.githubusercontent.com/pravalliyaram/5c05f43d2351249927b8a3f3cc3e5e
       "raw/8bd6144a87988213693754baaa13fb204933282d/Mall_Customers.csv")
df = pd.read_csv(url)

# Rename columns for easier access
df = df.rename(columns={
    'Annual Income (k$)': 'Income',
    'Spending Score (1-100)': 'SpendingScore'
})

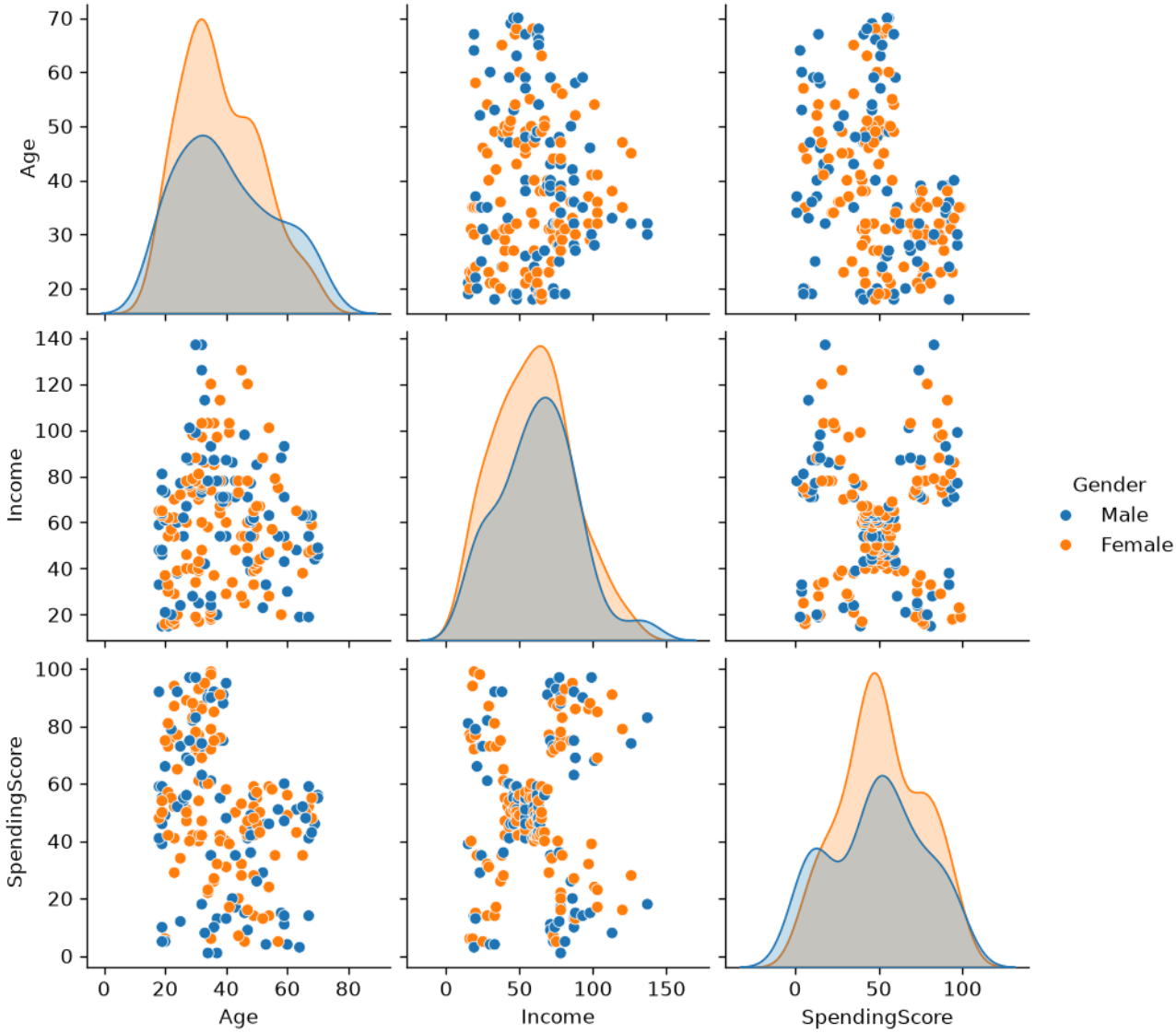
print(df.head())
print("\n--- Data Info ---")
df.info()
```

```
   CustomerID  Gender  Age  Income  SpendingScore
0           1    Male   19      15             39
1           2    Male   21      15             81
2           3  Female   20      16              6
3           4  Female   23      16             77
4           5  Female   31      17             40

--- Data Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   CustomerID     200 non-null    int64
 1   Gender         200 non-null    object
 2   Age            200 non-null    int64
 3   Income         200 non-null    int64
 4   SpendingScore  200 non-null    int64
dtypes: int64(4), object(1)
memory usage: 7.9+ KB
```

Let's use `seaborn.pairplot` to visualize all the relationships. Pay close attention to the `Income` vs. `SpendingScore` plot.

```python
# We'll drop CustomerID and make 'Gender' a color
sns.pairplot(df.drop('CustomerID', axis=1), hue='Gender', diag_kind='kde')
plt.show()
```



Step 2: Prepare and Scale the Data

The `pairplot` confirms our hypothesis: the `Income` **vs.** `SpendingScore` plot shows at least 5 distinct, non-linear groups of customers.

We will focus our clustering on just these two features.

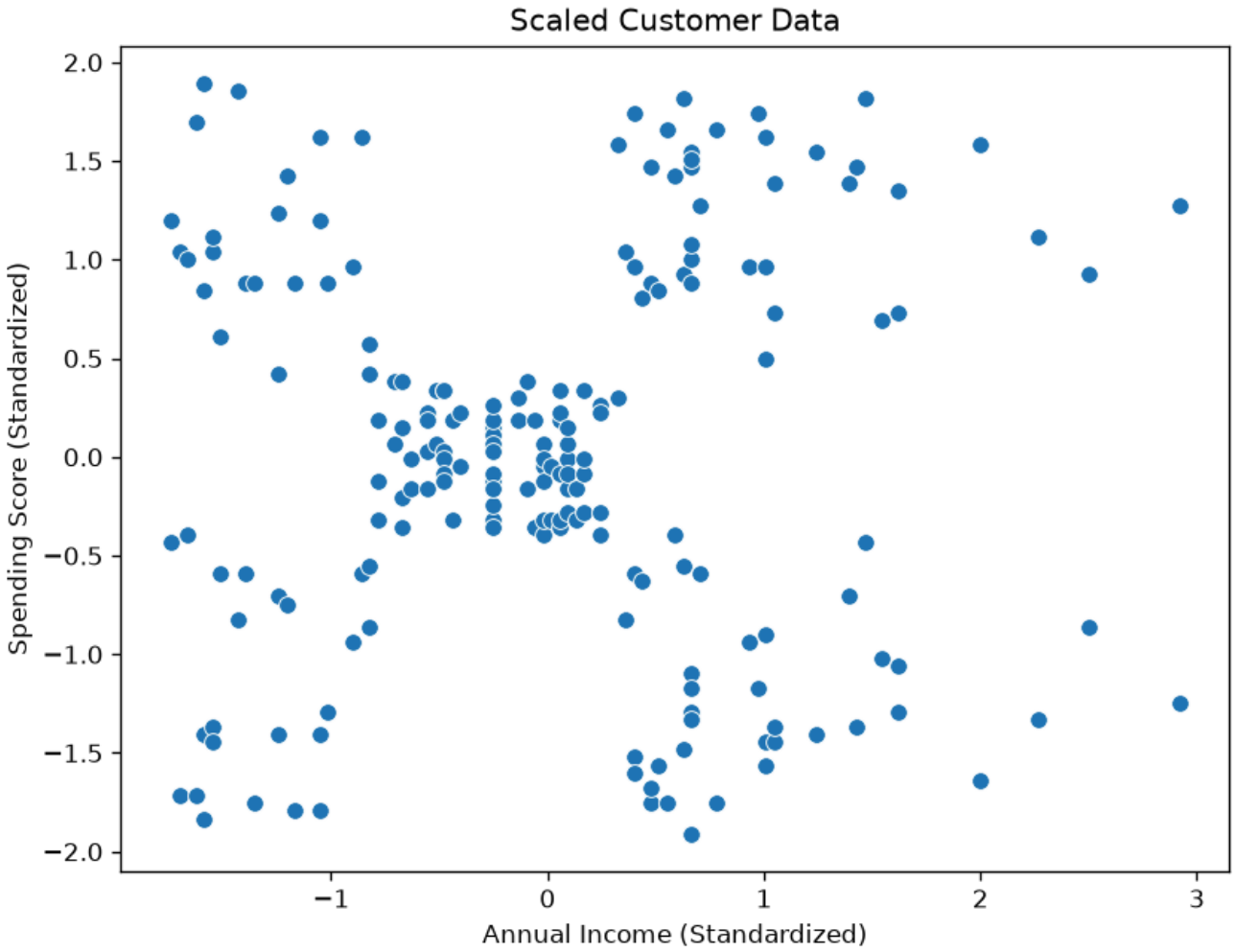
Crucially, `Income` (15-137) and `SpendingScore` (1-99) are on different scales. K-Means uses Euclidean distance, so if we don't scale, the `Income` feature will completely dominate the `SpendingScore`. We **must** use `StandardScaler`.

```
# 1. Select the features for clustering
X = df[['Income', 'SpendingScore']].values

# 2. Initialize the scaler
scaler = StandardScaler()

# 3. Fit the scaler and transform the data
X_scaled = scaler.fit_transform(X)

# Let's plot the scaled data
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_scaled[:, 0], y=X_scaled[:, 1], s=50)
plt.title('Scaled Customer Data')
plt.xlabel('Annual Income (Standardized)')
plt.ylabel('Spending Score (Standardized)')
plt.show()
```



Step 3: Find the Best K with the Elbow Method

Now we'll use the **Elbow Method**. We will run K-Means for $K = 1$ through $K = 10$ and plot the **Inertia** (the J cost function, or WCSS). We are looking for the "elbow" where the decrease in cost

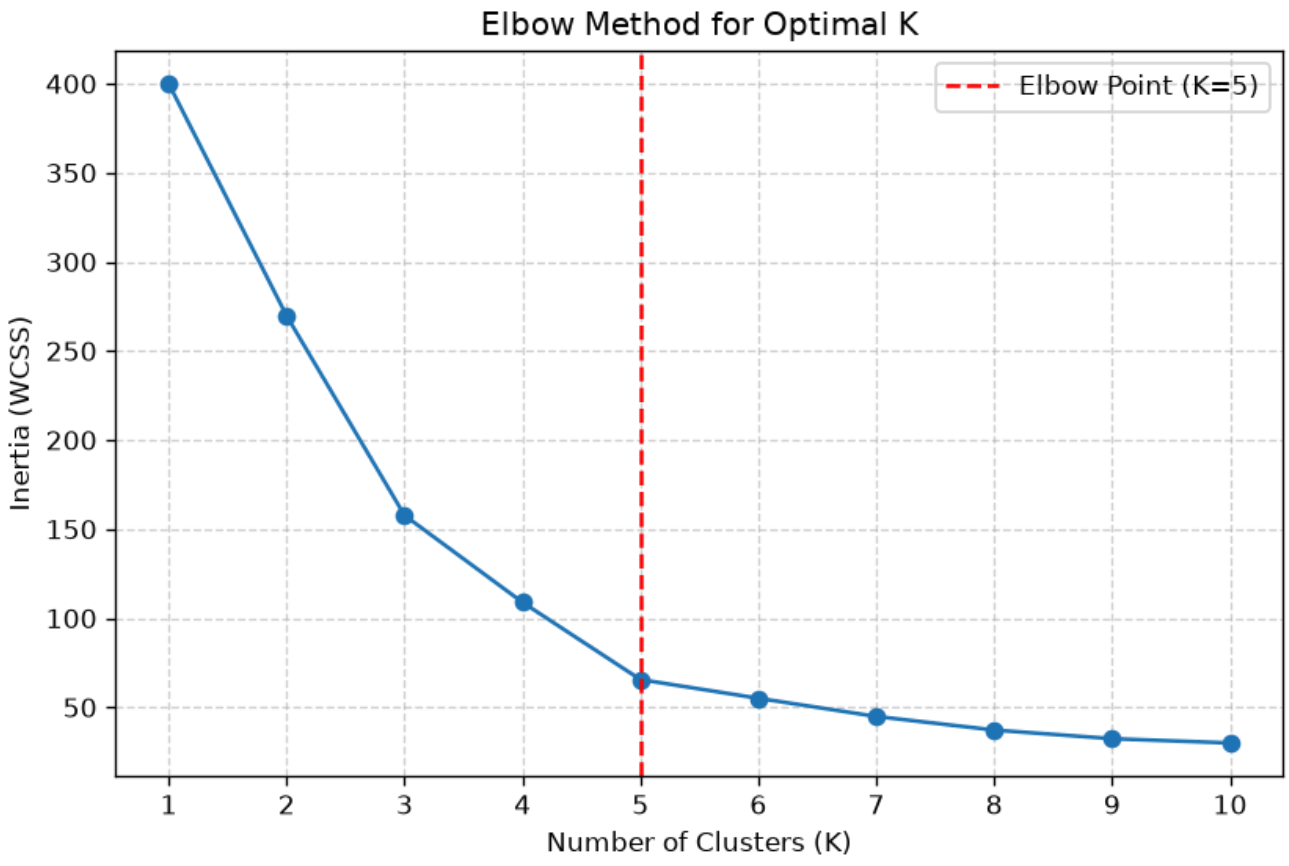
dramatically slows down.

```
# We'll use the scaled data from now on
k_values = range(1, 11)
inertias = []

for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_scaled)
    inertias.append(kmeans.inertia_) # .inertia_ is the WCSS (our cost function J)

# Plot the elbow curve
plt.figure(figsize=(8, 5))
plt.plot(k_values, inertias, 'o-')
plt.title('Elbow Method for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Inertia (WCSS)')
plt.xticks(k_values)
plt.grid(True, linestyle='--', alpha=0.6)

# Highlight the "elbow"
plt.axvline(x=5, color='red', linestyle='--', label='Elbow Point (K=5)')
plt.legend()
plt.show()
```



The elbow shows a clear "bend" appearing at $K = 5$. The drop in inertia from 5 to 6 is much smaller than the drop from 4 to 5.

Let's confirm this with the **Silhouette Method**.

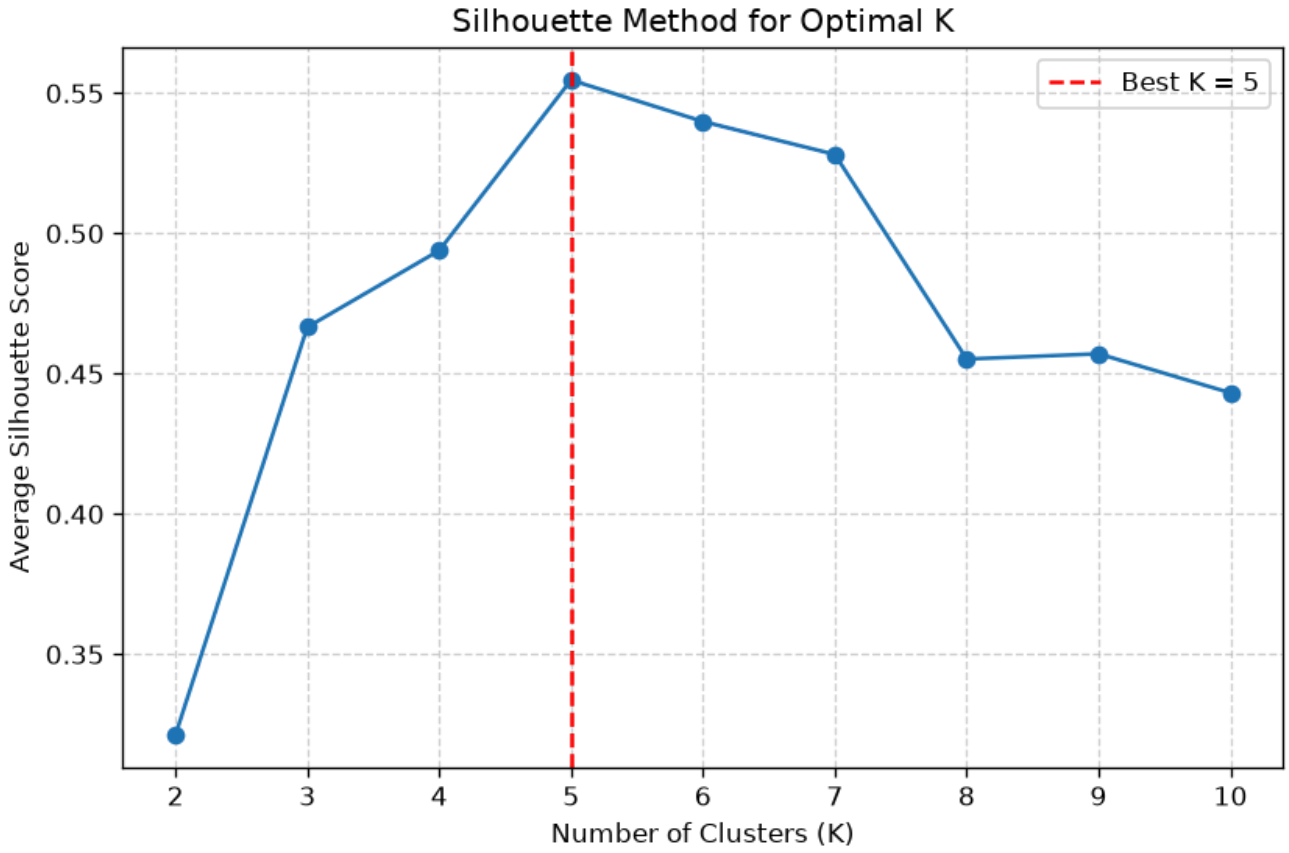
```
# We must start from K=2
k_values_sil = range(2, 11)
silhouette_scores = []

for k in k_values_sil:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X_scaled)

    # Calculate the average silhouette score
    score = silhouette_score(X_scaled, labels)
    silhouette_scores.append(score)

# Plot the silhouette scores
plt.figure(figsize=(8, 5))
plt.plot(k_values_sil, silhouette_scores, 'o-')
plt.title('Silhouette Method for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Average Silhouette Score')
plt.xticks(k_values_sil)
plt.grid(True, linestyle='--', alpha=0.6)

# Highlight the best K
best_k_sil = k_values_sil[np.argmax(silhouette_scores)]
plt.axvline(x=best_k_sil, color='red', linestyle='--', label=f'Best K =
        {best_k_sil}')
plt.legend()
plt.show()
```



Step 4: Fit and Interpret the Final Model

The Silhouette plot confirms it! The highest score (the best cluster separation) is achieved at $K = 5$.

Both methods agree. We will now build our final model using $K = 5$ and, most importantly, **interpret the results**.

```
# 1. Fit the final model with K=5
final_kmeans = KMeans(n_clusters=5, random_state=42, n_init=10)
final_labels = final_kmeans.fit_predict(X_scaled)

# 2. Get the cluster centers (on the *scaled* data)
final_centers_scaled = final_kmeans.cluster_centers_

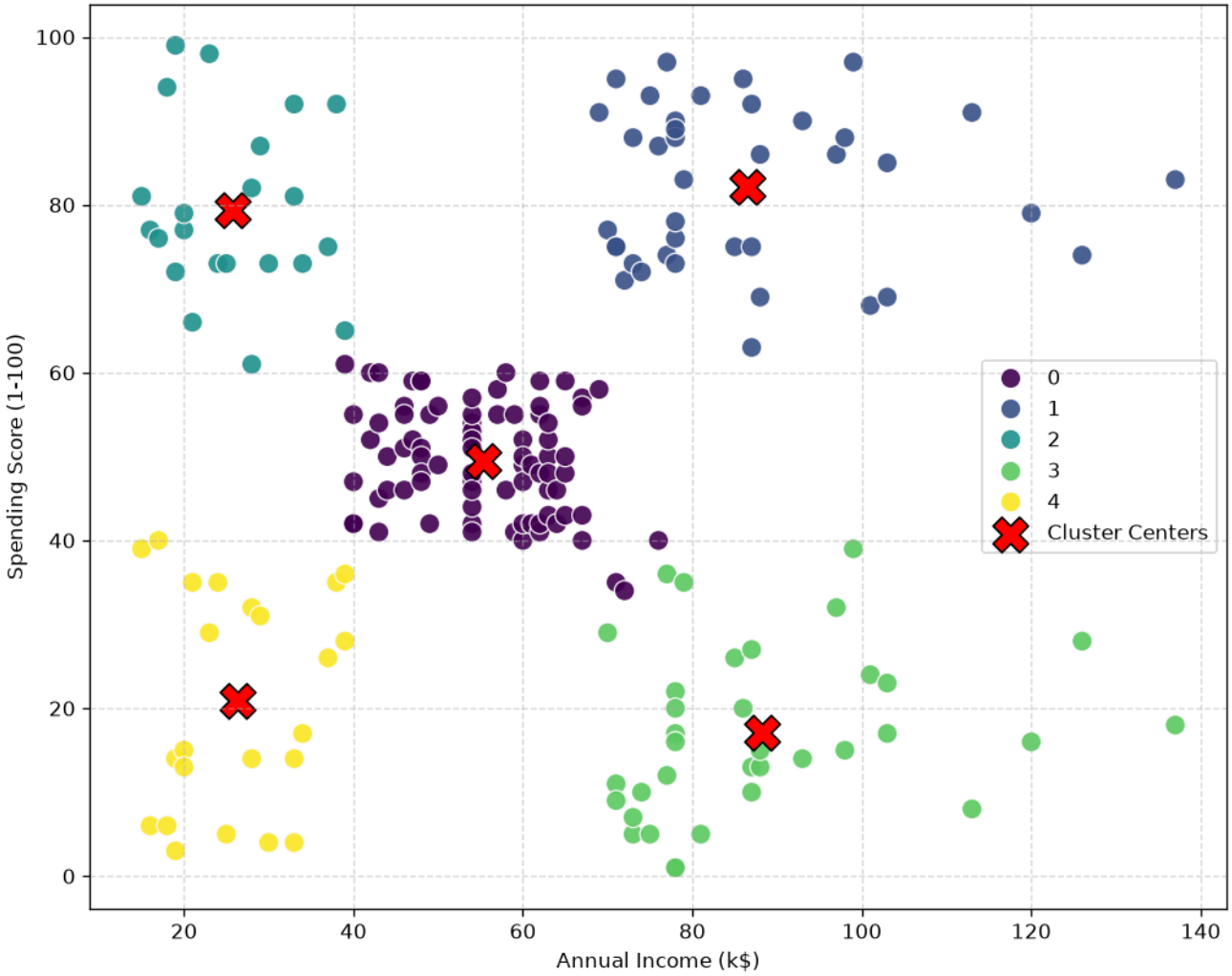
# 3. Add the cluster labels back to our *original* dataframe
df['Cluster'] = final_labels

# 4. Plot the final clustered data (using the *original*, unscaled data)
plt.figure(figsize=(10, 8))
sns.scatterplot(x='Income', y='SpendingScore', data=df,
                hue='Cluster', palette='viridis', s=100, alpha=0.9)

# 5. We must "inverse_transform" the centers to plot them on the original scale
final_centers_original = scaler.inverse_transform(final_centers_scaled)

# Plot the cluster centers
plt.scatter(final_centers_original[:, 0], final_centers_original[:, 1],
            marker='X', s=300, c='red',
            edgecolor='black', label='Cluster Centers')

plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```



Step 5: Business Interpretation

This is the most important step. What do these 5 clusters *mean*? Let's analyze the `Income` vs. `SpendingScore` for each group.

We can use `.groupby()` to see the "average" customer in each cluster.

```
# Calculate the mean Income and SpendingScore for each cluster
cluster_analysis = df.groupby('Cluster')[['Income', 'SpendingScore']].mean()
print(cluster_analysis)
```

```
             Income  SpendingScore
Cluster
0         55.296296      49.518519
1         86.538462      82.128205
2         25.727273      79.363636
3         88.200000      17.114286
4         26.304348      20.913043
```

Analysis of Segments

By looking at the plot and the table, we can give each cluster a clear, descriptive name:

- **Cluster 0: "The Standard"**
  - *Profile:* Average Income, Average Spending Score.
  - *Interpretation:* This is the large, "normal" group of customers.
- **Cluster 1: "The Targets" (or "VIPs")**
  - *Profile:* High Income, High Spending Score.
  - *Interpretation:* The ideal customer. They have money and they love to spend it.
- **Cluster 2: "The Risk" (or "Careless")**
  - *Profile:* Low Income, High Spending Score.
  - *Interpretation:* Spend a lot, but have low income. (e.g., college students).
- **Cluster 3: "The Careful Rich"**
  - *Profile:* High Income, Low Spending Score.
  - *Interpretation:* They have money, but don't spend it. A key group to target with different marketing.
- **Cluster 4: "The Savers"**
  - *Profile:* Low Income, Low Spending Score.
  - *Interpretation:* Cautious customers who don't spend much.

Conclusion: K-Means successfully partitioned the data into five distinct, meaningful groups that a business could now build specific marketing strategies for.

Going Further: Clustering in 3D (Age, Income, Spending)

The 2D visualization on `Income` and `SpendingScore` was a perfect way to understand *how* K-Means works. But in a real scenario, we'd want to use *all* the relevant information we have.

Let's try to cluster these customers again, but this time using **three** features:

- `Age`
- `Income`
- `SpendingScore`

The New Challenge: Preprocessing and Visualization

1. **Preprocessing:** All three features (`Age`, `Income`, `SpendingScore`) are on different scales. We **must** use `StandardScaler` on all of them.
2. **Visualization:** We can no longer use a simple 2D scatter plot. We will have to use a **3D scatter plot** to see our clusters.

```python
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# 1. Select the features for clustering
X_3D = df[['Age', 'Income', 'SpendingScore']].values

# 2. Initialize the scaler
scaler_3D = StandardScaler()

# 3. Fit the scaler and transform the data
X_scaled_3D = scaler_3D.fit_transform(X_3D)

print(f"New processed data shape: {X_scaled_3D.shape}")
```

```
New processed data shape: (200, 3)
```

Choosing K Again, in Three Dimensions

Now that we have our new, 3-dimensional `X_scaled_3D` data, we must re-run our Elbow and Silhouette methods to find the best K *for this new feature space*.

```
# --- 1. Elbow Method (on 3D data) ---
k_values = range(1, 11)
inertias_3D = []
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_scaled_3D)
    inertias_3D.append(kmeans.inertia_)

# --- 2. Silhouette Method (on 3D data) ---
k_values_sil = range(2, 11)
silhouette_scores_3D = []
for k in k_values_sil:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X_scaled_3D)
    score = silhouette_score(X_scaled_3D, labels)
    silhouette_scores_3D.append(score)

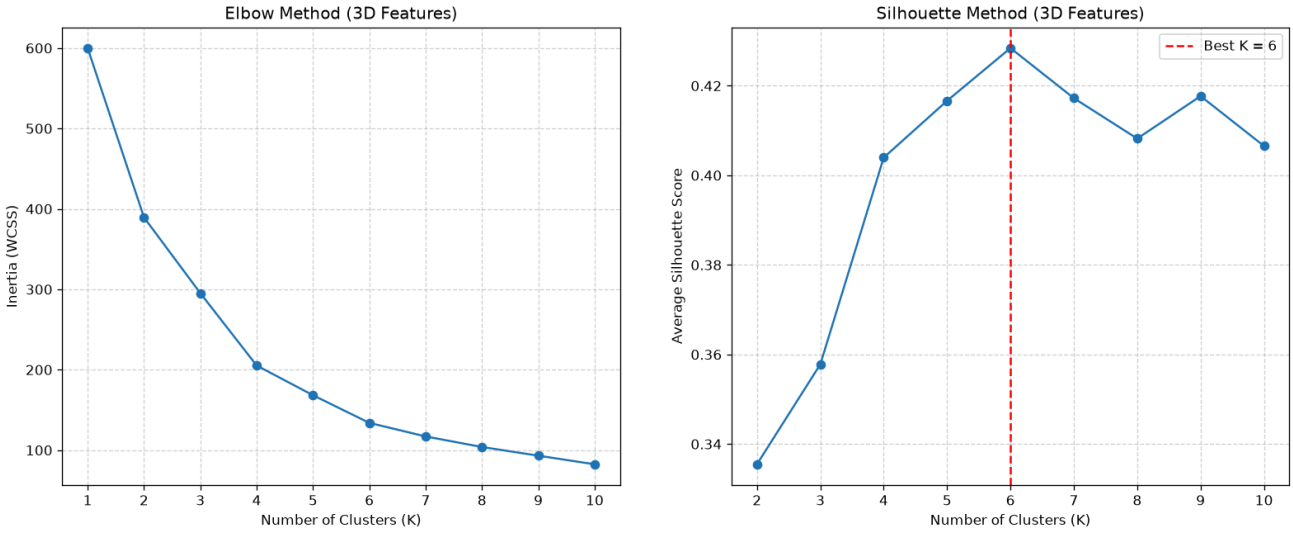
# --- 3. Plot the results ---
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# Elbow plot
ax1.plot(k_values, inertias_3D, 'o-')
ax1.set_title('Elbow Method (3D Features)')
ax1.set_xlabel('Number of Clusters (K)')
ax1.set_ylabel('Inertia (WCSS)')
ax1.set_xticks(k_values)
ax1.grid(True, linestyle='--', alpha=0.6)

# Silhouette plot
ax2.plot(k_values_sil, silhouette_scores_3D, 'o-')
ax2.set_title('Silhouette Method (3D Features)')
ax2.set_xlabel('Number of Clusters (K)')
ax2.set_ylabel('Average Silhouette Score')
ax2.set_xticks(k_values_sil)
ax2.grid(True, linestyle='--', alpha=0.6)

# Highlight the best K
best_k_sil_3D = k_values_sil[np.argmax(silhouette_scores_3D)]
ax2.axvline(x=best_k_sil_3D, color='red', linestyle='--', label=f'Best K =
        {best_k_sil_3D}')
ax2.legend()

plt.show()
```



Fitting and Interpreting the 3D Model

The Elbow method is again a bit ambiguous (maybe $K = 5$ or $K = 6$?), but the Silhouette plot often shows a clearer peak. Let's assume for this run the best score was at $K = 6$.

Now, how do we interpret these 6 clusters? We can't use a 2D plot.

The solution is to:

1. **Visualize** the clusters with a **3D scatter plot**.
2. **Analyze** the "profile" of each cluster using `groupby()`.

```
# 1. Fit the final model with the best K (e.g., K=6)
best_k = 6
final_kmeans_3D = KMeans(n_clusters=best_k, random_state=42, n_init=10)
final_labels_3D = final_kmeans_3D.fit_predict(X_scaled_3D)

# 2. Add the new cluster labels back to the ORIGINAL DataFrame
df['Cluster_3D'] = final_labels_3D

# 3. Bring the centers back to the original scale, as in the 2D case, so they
# can be plotted on top of the unscaled data
centers_original_3D = scaler_3D.inverse_transform(final_kmeans_3D.cluster_centers_)

# 4. Analyze the cluster profiles using groupby()
# We can now see the average 'Age', 'Income', etc. for each new cluster
cluster_analysis_3D = df.groupby('Cluster_3D')[['Age', 'Income',
        'SpendingScore']].mean()
print("--- Average Cluster Profiles (3D Features) ---")
print(cluster_analysis_3D.sort_values(by='Income'))
```

```
--- Average Cluster Profiles (3D Features) ---
                 Age     Income  SpendingScore
Cluster_3D
4          25.000000  25.260870      77.608696
5          45.523810  26.285714      19.380952
0          56.333333  54.266667      49.066667
1          26.794872  57.102564      48.128205
3          32.692308  86.538462      82.128205
2          41.939394  88.939394      16.969697
```

Visualizing the 3D Clusters

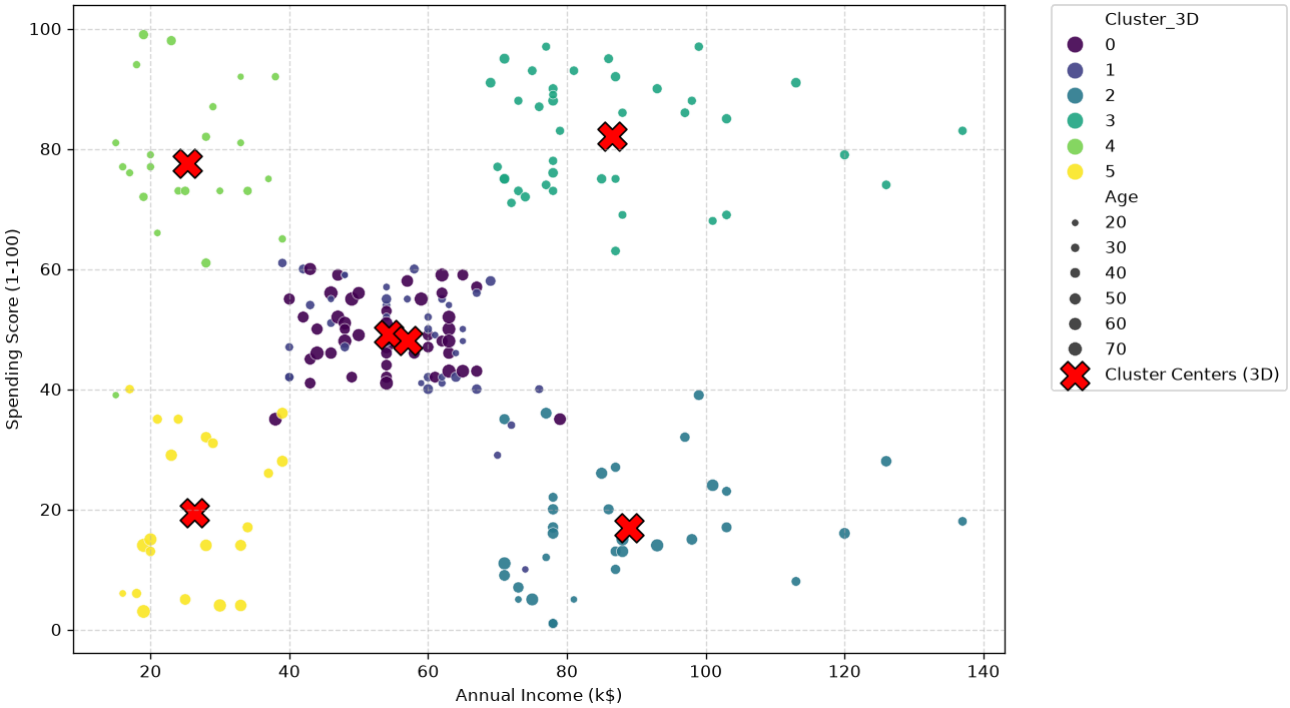
Now for the best part. We can use `matplotlib`'s 3D plotting tools to see the clusters in our new 3D feature space.

```
plt.figure(figsize=(10, 7)) # Increased figure size slightly for the legend

# Create the "4D" scatter plot
sns.scatterplot(
    data=df,
    x='Income',
    y='SpendingScore',
    hue='Cluster_3D',  # Color by cluster (categorical)
    size='Age',        # Vary dot size by Age (continuous)
    palette='viridis', # Using viridis to match the 3D plot
    alpha=0.9,         # Set alpha to 0.6 to see points overlap
    s=100              # Base size
)

# --- ADDED THIS PART ---
# Plot the cluster centers
# We use the 'Income' (column 1) and 'SpendingScore' (column 2)
# from our 3D centers array
plt.scatter(
    centers_original_3D[:, 1], # x = Income
    centers_original_3D[:, 2], # y = SpendingScore
    marker='X',                # Use a large 'X'
    s=300,                     # Make it big
    c='red',                   # Make it red
    edgecolor='black',         # With a black border
    label='Cluster Centers (3D)' # Add to legend
)
# -----------------------

plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```



Analysis: What the 4D Plot Tells Us

The plot is now more nuanced, and you'll notice the optimal number of clusters has changed from **5** to **6**. This is not a mistake; it is the **key finding** of this 3D analysis.

It demonstrates a fundamental concept: **changing your feature space can change the optimal structure of the data.**

- **The K=5 Model (2D):** This analysis *only* knew about `Income` and `SpendingScore`. In that 2D space, the Silhouette method told us the best way to group the data was into **5 clusters**.
- **The K=6 Model (3D):** This analysis used `Income`, `SpendingScore`, AND `Age`. When we added the `Age` dimension, the "shape" of the data changed. The Silhouette method, when run on this new 3D data, found that **6 clusters** was a better, more distinct grouping.

Visual Evidence

You can see *why* in the plot.

- In the original K=5 plot, the large **"Standard"** group (blue, Cluster 0) was a single, dense blob in the middle.
- In the new K=6 plot, this *same middle area* has been split into **two new groups**: a dark purple cluster (Cluster 0) and a dark blue cluster (Cluster 1).

The 3D model used the new `Age` feature (which we encode as the **size** of the dots) to discover this new pattern. It realized that the single "Standard" cluster was actually **two different groups** that just happened to overlap in the 2D projection. Based on the dot sizes, the model likely found:

- **Cluster 0 (Purple):** "Standard" customers (average income/spending) who are **Younger** (small dots).

- **Cluster 1 (Blue):** “Standard” customers (average income/spending) who are **Older** (large dots).

This is a much more powerful and useful segmentation, and we only found it by adding more relevant features to our analysis.

Alternative interactive 3D visualization:

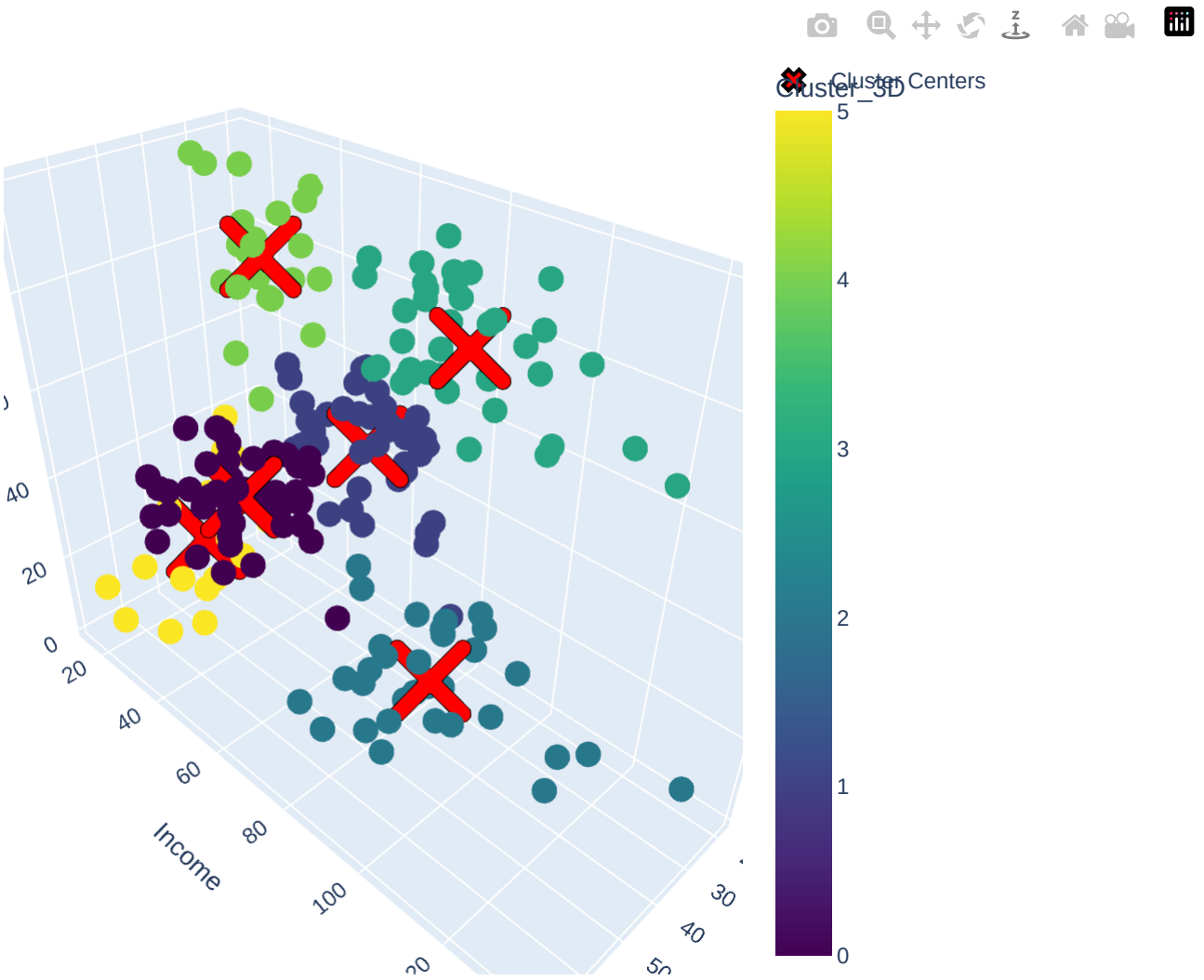
```
import plotly.graph_objects as go
import plotly.express as px

# --- We assume 'df' has the 'Cluster_3D' column ---
# --- and 'centers_original_3D' (unscaled) is in memory ---

# 1. Create the main scatter plot using Plotly Express
# This plots all the individual data points (our customers)
fig = px.scatter_3d(
    df,
    x='Age',
    y='Income',
    z='SpendingScore',
    color='Cluster_3D',  # Color the dots by their cluster
    color_continuous_scale='viridis', # Use the same color map
)

# 2. Add the Cluster Centers as a separate "trace"
# We plot these as large, red 'X' markers
fig.add_trace(go.Scatter3d(
    x=centers_original_3D[:, 0], # Age
    y=centers_original_3D[:, 1], # Income
    z=centers_original_3D[:, 2], # SpendingScore
    mode='markers',
    marker=dict(
        color='red',
        symbol='x',
        size=10,
        line=dict(color='black', width=2)
    ),
    name='Cluster Centers'
))

# 3. Show the interactive plot
fig.show()
```



Exercises

1. **Choosing K.** Run K-means for K from 2 to 10 on the mall-customer data. Plot both the elbow curve and the silhouette score. Do they agree? Which would you trust, and what would you do if they disagreed?
2. **Clusters that are not there.** Run K-means with $K = 4$ on data drawn from a single uniform distribution. Plot the result. It will look convincing — explain why, and say what test would have told you the clusters are spurious.
3. **Scaling.** Cluster the customers with and without standardisation. Which variable dominates the unscaled solution, and why?
4. **Shapes K-means cannot find.** Generate two concentric rings and cluster them with K-means. Explain the failure geometrically, then try DBSCAN on the same data.
5. **Interpretation.** For your best solution, describe each cluster in the language of the business: who are these customers? Then state one thing your description assumes that the clustering itself does not support.
6. **Stability.** Re-run K-means with ten different random seeds. How often do you get the same partition? What does that say about reporting a single clustering as *the* answer?

References

- Section 10.3.1 of [1]
- Section 2.3.9 of [2]
- Section 9.2 of [2]
- https://en.wikipedia.org/wiki/Silhouette_(clustering)

[1] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning. New York: springer.

[2] Bishop, Christopher M., and Nasser M. Nasrabadi. Pattern recognition and machine learning. Vol. 4. No. 4. New York: springer, 2006.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Density estimation

Chapter resources

Slides (PDF) Open in Colab Practice quiz

We have seen that clustering algorithms aim to mine the underlying structure of data by breaking it into coherent groups. For example k-means clustering **allows to identify parts of the feature space with high density (i.e., a large number of observations)** by partitioning all observations into a **discrete set of K data groups distributed around centroids**.

An alternative **smoother** approach would be to study the probability $P(X)$, where X is the data. Indeed, if we could determine the probability value of an given point in space $\mathbf{x}$, we could naturally discover clusters in the data (**zones with high probability values**). Knowing the distribution of the data is also useful for a number of things:

- Similar to clustering, if X is a series of observations of customers in a bank (e.g., age, sex, balance, salary, etc.), we may want to know $P(X)$ to understand which types of customers are more frequent (those that have large $P(X)$ values), or if there are different distinct groups of customers (e.g., if $P(X)$ has more than one mode);
- If X is a set of images of faces, knowing $P(X)$ would allow us to understand if an image x looks like an image of a face or not, which faces are more frequent, if there are more than one "groups" of faces, etc. If we could draw $x \sim P$, we could even generate new images of faces! (**by the way, this is what Generative Adversarial Networks do**);
- If X is a series of observations of bank transactions, by modeling $P(X)$ we can infer if a given observation $\mathbf{x}$ is a typical transaction (high $P(\mathbf{x})$) or whether it is an anomaly (low $P(X)$). If we can make assumptions on the nature of $P(X)$, we can even make inference on the properties of a given data point $\mathbf{x}$. For instance, if P is Gaussian, then datapoints which are further from the center are atypical and they can hence identify anomalous transactions.

A probability density is **a continuous function**, so estimating it is not as straightforward as estimating a **probability mass function**. There are two main classes of methods for density estimation:

- **Non-parametric methods**: these methods aim to estimate the density **directly from the data, without strong assumptions on the source distribution**. The main advantage of these methods is that they generally have few hyper-parameters to tune and can be used when no assumption can be made on the source of the data. A disadvantage is that, while we can numerically compute $f(\mathbf{x})$ with these methods (where f is the density function), we do not know the analytical form or property of f;
- **Parametric methods**: these methods aim to **fit a known distribution to the data**. An example of this approach is **fitting a Gaussian to the data**. The main advantage of these methods is that they

provide an analytical form for the density f, which can be useful in a number of contexts (for instance, the density may be part of a cost function we want to optimize). The downside is that these methods make strong assumptions on the nature of the data.

In the following, we will see the main representatives of these two classes of methods.

Non-parametric Density Estimation Techniques

We will now look at **non-parametric methods**, which estimate the density *directly from the data* without making strong assumptions (like that the data is Gaussian).

We will focus on the two main approaches:

1. **Fixed Windows (Histograms):** This method "discretizes" the feature space into a fixed grid of bins and counts how many points fall into each bin.
2. **Mobile Windows (Kernel Density Estimation):** This method centers a "window" on *every single data point* (or on a grid) and sums their influence, creating a much smoother estimate. This is also known as the **Parzen Window** method.

Method 1: Fixed Windows (ND Histograms)

This is the simplest method, extending the 1D histogram to multiple dimensions. We "discretize" the feature space into a fixed grid of "bins" (or "tiles") and count how many points fall into each bin.

A simple *count* is misleading. A bin that is 4x larger might have 4x more points but have the same *density*.

To get a true **probability density function** $f(\mathbf{x})$, our function must satisfy one rule: the **total "volume" under the function must integrate to 1**.

$$\int f(\mathbf{x})d\mathbf{x} = 1$$

To achieve this, we cannot just plot the counts. We must define the density $f(R_i)$ for a bin R_i as the *probability* of that bin divided by its *volume* (or area).

1. **Probability of the bin:** $P(R_i) = \frac{\text{Count of points in } R_i}{\text{Total Number of Points } (N)}$
2. **Volume of the bin:** $V(R_i)$ (e.g., for a 2x2 bin, the area is 4).

This gives us the formula for the density *height* of that bin:

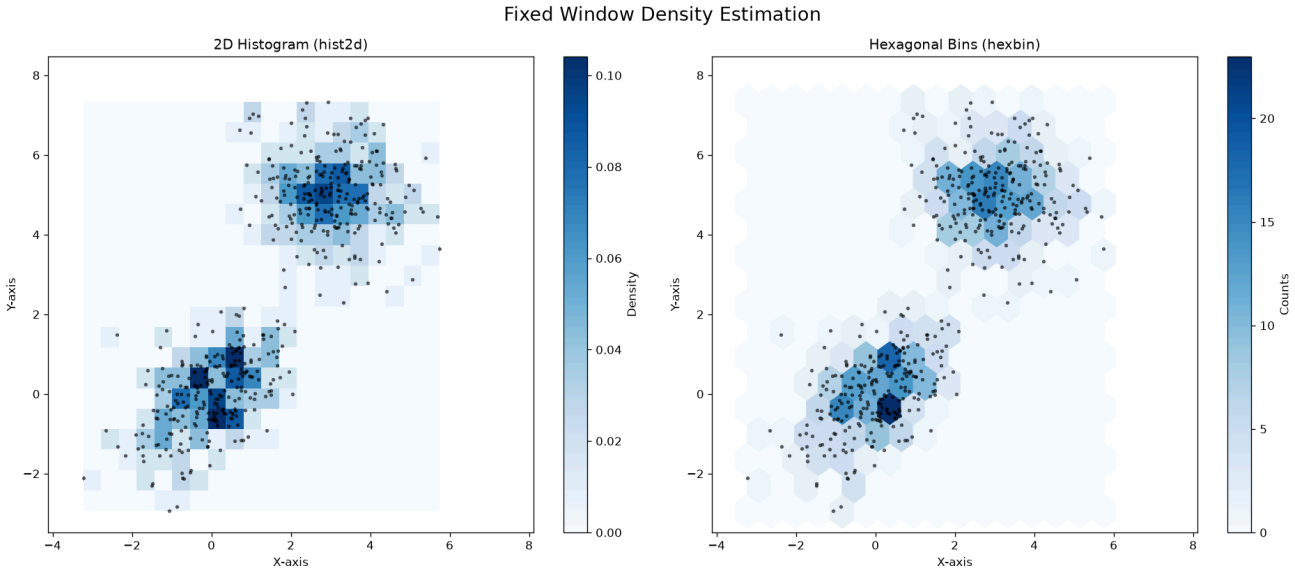
$$f(\mathbf{x} \in R_i) = \frac{P(R_i)}{V(R_i)} = \frac{|R_i|/N}{V(R_i)}$$

By dividing by the volume, we ensure that a small, crowded bin (small $V(R_i)$) will have a high density value, while a large, sparse bin (large $V(R_i)$) will have a low density value, even if they

contain the same number of points. This normalization is what ensures the total volume integrates to 1:

$$P(\mathbf{x} \in \cup_i R_i) = \sum_i P(R_i) = \sum_i f(\mathbf{x} \in R_i) V(R_i) = \sum_i \frac{|R_i|/N}{V(R_i)} V(R_i) = \frac{1}{N} \sum_i |R_i| = 1$$

This approach has two main hyperparameters: the **bin size** and the **grid's origin (position)**. A popular alternative to square bins is `hexbin`, which uses hexagonal bins that tile the space more naturally.



As you can see, both plots do a good job of finding the two high-density "blobs" in our data. The `hexbin` plot (right) often looks a bit smoother and more natural than the `hist2d` (left).

The main problem with "fixed window" methods is that the *bin size* is a difficult hyperparameter to choose. More importantly, the rigid, fixed grid is sensitive to where the points fall (the "bin edge" problem).

While we saw this in 2D, we can generalize it to N dimensions (e.g., squares become cubes or hypercubes).

Method 2: Mobile Windows (Kernel Density Estimation)

The "fixed window" (histogram) method has two main problems:

1. The "bin edges" are arbitrary and can change the shape of the density.
2. The resulting density is "blocky" and not smooth.

A more flexible and powerful non-parametric approach is to use a **"mobile window"**. This method is known as **Kernel Density Estimation (KDE)** or the **Parzen Window** method.

Instead of putting a fixed grid over the data, KDE places a "window" (or **kernel**) at each point in space $\mathbf{x}$ and asks, "How much data is inside this window?"

The “Naive” Approach: The Circular Kernel

A simple approach is to center a circle (or hypersphere) of a fixed radius h at a point $\mathbf{x}$ and calculate the density based on how many data points $N(\mathbf{x}, h)$ fall inside it.

This follows the same logic as the histogram: > Density = (Probability of being in the window) / (Volume of the window)

$$f(\mathbf{x}) = \frac{|N(\mathbf{x}, h)|/N}{V(h)}$$

where N is the total number of points, $|N(\mathbf{x}, h)|$ is the count of points inside the circle

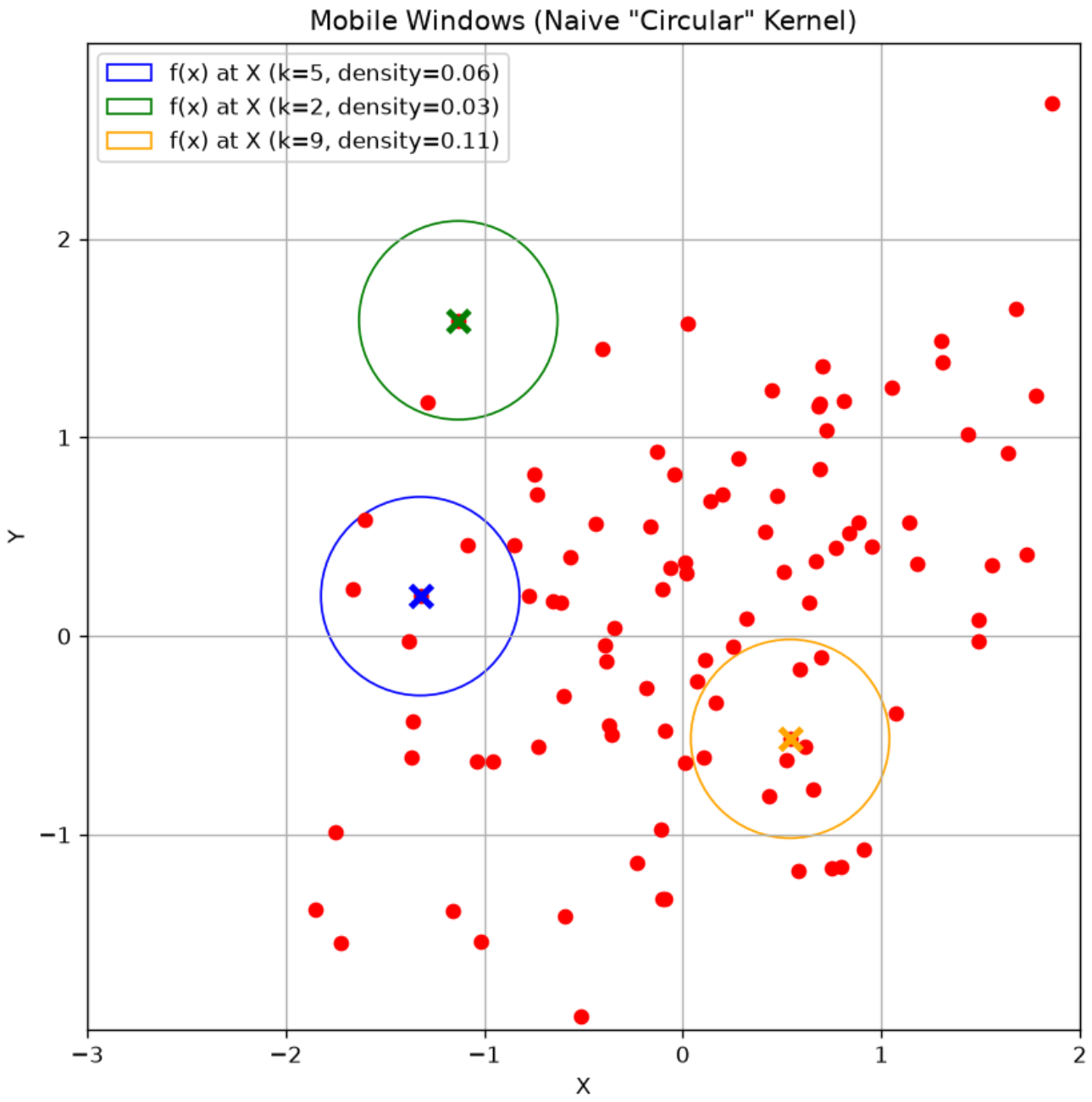
$$N(\mathbf{x}, h) = \{\mathbf{x}' \in \mathbf{X} \; s.t. \; ||\mathbf{x}' - \mathbf{x}||_2 \leq h\}$$

and $V(h)$ is the volume (or area) of the circle (e.g., $V(h) = \pi h^2$).

It is not hard to see that under the definition above, we have:

$$\int_{\mathbf{x} \in \Re^d} f(\mathbf{x}) d\mathbf{x} = 1$$

The plot below shows some density estimates using this method:



Kernel View

We can also write the expression above as:

$$f(\mathbf{x}) = \frac{1}{|\mathbf{X}|} \sum_{i=1}^{|\mathbf{X}|} K_h\left(\mathbf{x}_i - \mathbf{x}\right)$$

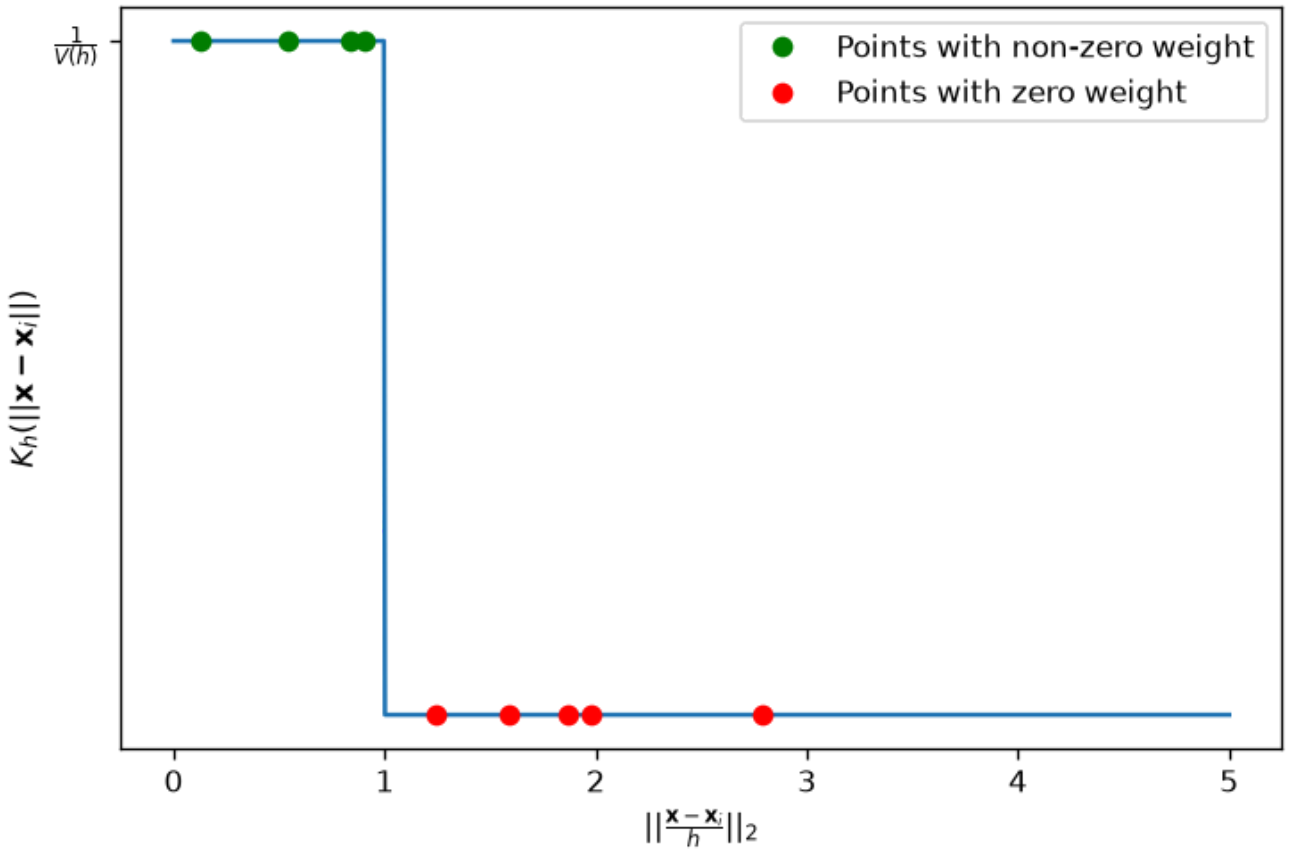
where K_h is defined as:

$$K_h\left(\mathbf{x}_i - \mathbf{x}\right) = \begin{cases} \frac{1}{V(h)} & \text{if } ||\frac{\mathbf{x}_i - \mathbf{x}}{h}||_2 \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

We will call K_h **a kernel function** depending on the bandwidth parameter h. This is a function which assigns a weight to a point $\mathbf{x}_i$ depending on its distance from $\mathbf{x}$. In the example above, we chose K_h as **a circular (or radial) kernel** which assigns a uniform score to all points falling in a circle of radius h centered at $\mathbf{x}$.

A main problem with this density estimation approach is that **it can be very sensitive to the location at which we are computing the density**. Indeed, we can find cases in which we obtain very different densities when we move the circle by a little bit.

We can note that **this is due by the kernel K making "hard decisions" on which elements to assign a non-zero score and which ones to assign a zero score**. Indeed, if we plot the kernel as a function of the distance between $\mathbf{x}$ and $\mathbf{x}_i$, rescaled by h, we obtain the following picture:



The figure above plots points putting their distance from $\mathbf{x}$ in the x axis and the assigned weight in the y axis. As can be noted, even very close points in the x axis (e.g., the last of the green ones and the first of the red ones) get assigned very different weights, which makes the overall process sensitive to small shifts in $\mathbf{x}$.

The Problem with the "Naive" Kernel

This "circular" (or "radial") kernel has a major flaw: it makes **"hard decisions"**. A point is either 100% "in" the circle (and gets a full vote) or 100% "out" (and gets a 0 vote).

This means that if you move the center $\mathbf{x}$ just a tiny bit, a data point might fall out of the circle, causing the density to "jump" abruptly. This creates a non-smooth, "bumpy" density estimate.

The Solution: "Smooth" Kernels

To fix this, we use a **smooth kernel** that assigns decreasing weight to points as their distance from the center $\mathbf{x}$ grows. The **Gaussian Kernel** is a reasonable choice:

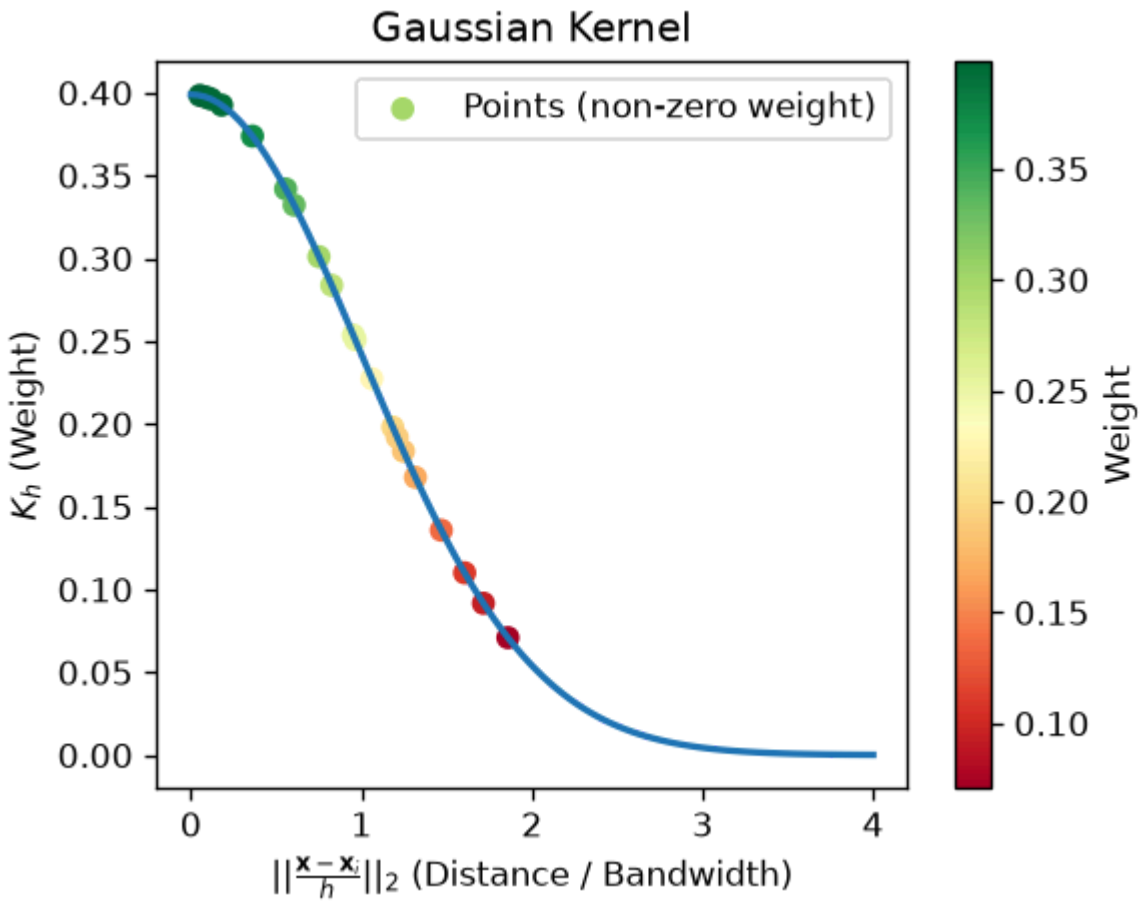
$$K_h(\mathbf{x}_i - \mathbf{x}) \propto \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}\|^2}{2h^2}\right)$$

With a smooth kernel, all points contribute to the density at $\mathbf{x}$, but nearby points contribute *much more* than faraway points.

The general formula for Kernel Density Estimation is a "sum of kernels," one for each data point:

$$f(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} K_h\left(\mathbf{x}_i - \mathbf{x}\right)$$

We can visualize this kernel as follows:



The Epanechnikov Kernel

The Gaussian Kernel solves the sensitivity problem. However, it is very expensive to compute. Indeed, for every new density estimate, we have to compute and sum Gaussian terms over the whole dataset. This can be expensive if we have a large dataset.

An alternative is to set to zero all elements which are far enough. This saves a lot of computation.

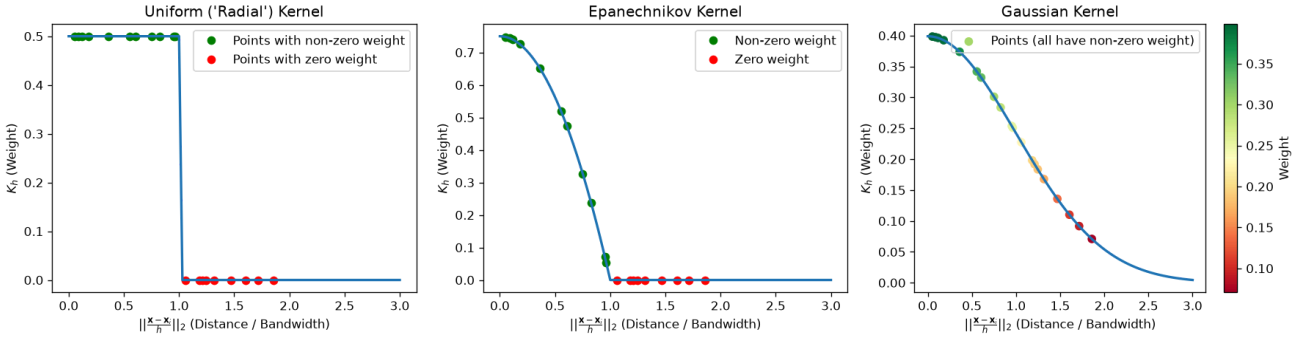
This is exactly what the Epanechnikov does. It is defined as follows:

$$K_h(\mathbf{x}_i - \mathbf{x}) = \frac{3}{4h^2}\left(1 - \frac{\|\mathbf{x}_i - \mathbf{x}\|^2}{h^2}\right)\mathbb{I}(\|\mathbf{x}_i - \mathbf{x}\| \leq h)$$

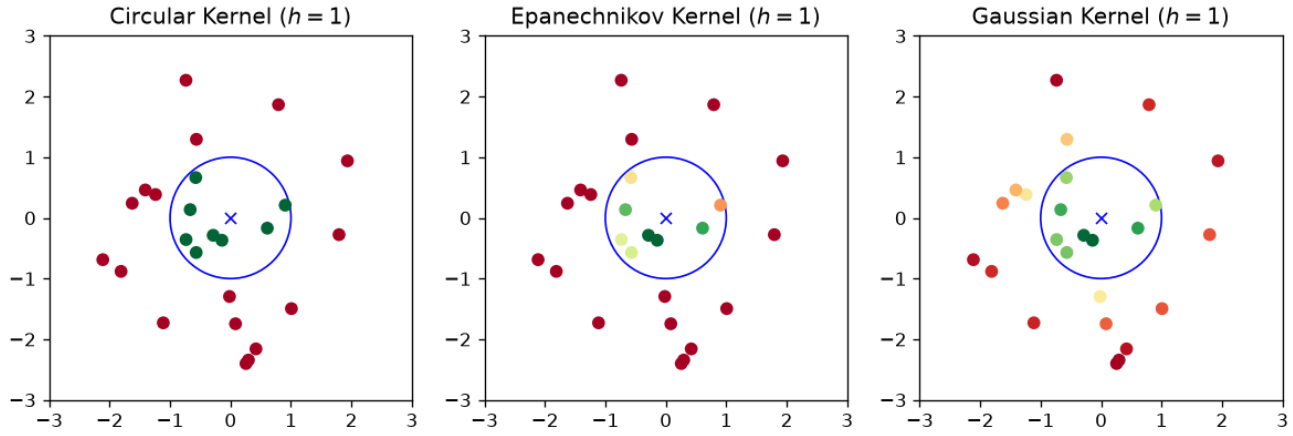
where $\mathbb{I}$ is the indicator function defined as:

$$\mathbb{I}(x) = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

The plot below compares the scores assigned to points depending on their distance from $\mathbf{x}$ according to the three kernels seen so far:



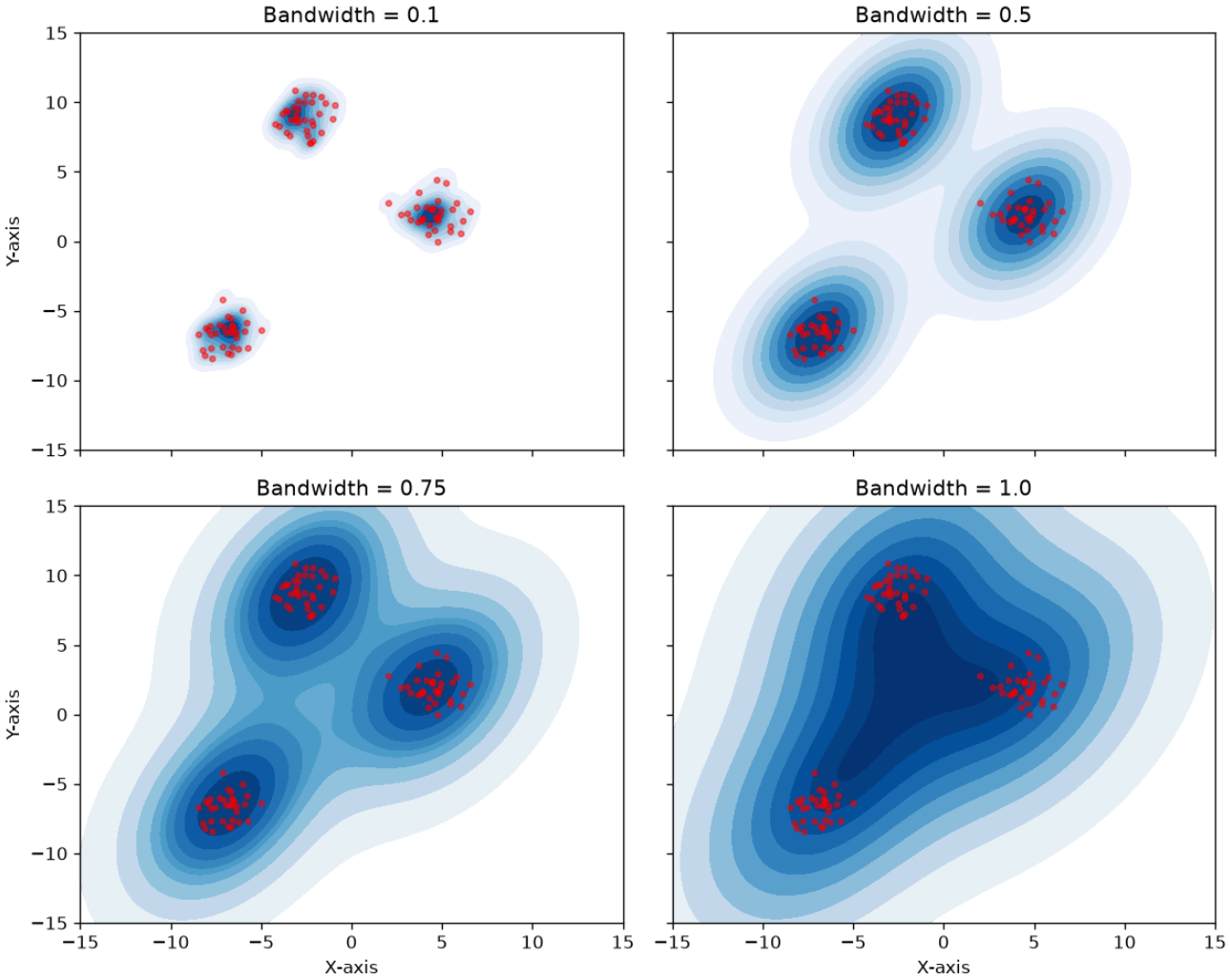
The effect of the kernels on the density estimations processes can be seen in the following plot:



The Bias-Variance Tradeoff: Choosing the Bandwidth h

The kernel shape (e.g., Gaussian) is not the most important choice. The **bandwidth (h)** is the *single most important hyperparameter* in density estimation. It controls the **bias-variance tradeoff**.

- **Small h (e.g., `bandwidth = 0.1`):**
  - **Low Bias, High Variance.**
  - The model "trusts" only a tiny local area. The resulting density is "spiky" and "noisy." It **overfits** to the training data.
- **Large h (e.g., `bandwidth = 1.0`):**
  - **High Bias, Low Variance.**
  - The model "blurs" the data by averaging over a huge area. The resulting density is "over-smoothed." It **underfits**, hiding the true structure (like the two separate clusters).



Final Result: KDE

As we can see, choosing a good bandwidth (like $h = 0.5$ or $h = 0.75$) is critical.

When we use a good kernel (like Gaussian) and a well-chosen bandwidth, we can produce a smooth, accurate, and non-parametric estimate of the true data density. The `seaborn.kdeplot` function does all of this for us automatically.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Generate random 2D points
np.random.seed(42)
data = np.random.multivariate_normal([0, 0], [[1, 0.5], [0.5, 1]], size=250)
data2 = np.random.multivariate_normal([3, 5], [[1, -0.2], [-0.2, 1]], size=250)

data = np.concatenate([data, data2])

# Set the size of the bins (adjust as needed)
bin_size = 0.5

# Calculate the range of the plot
x_min, x_max = data[:, 0].min() - 1, data[:, 0].max() + 1
y_min, y_max = data[:, 1].min() - 1, data[:, 1].max() + 1

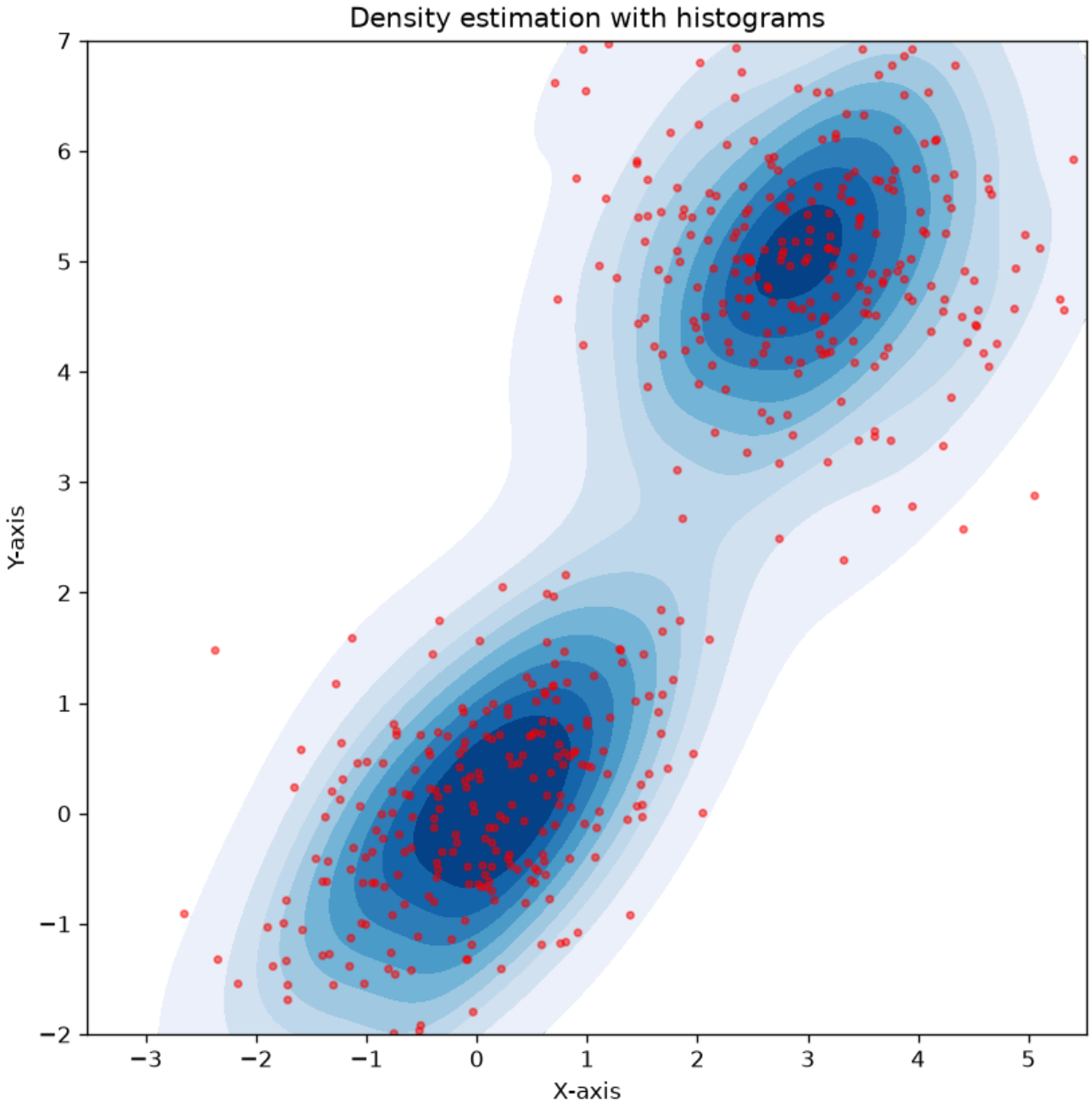
# Calculate the number of bins in each dimension
num_x_bins = int((x_max - x_min) / bin_size)
num_y_bins = int((y_max - y_min) / bin_size)

# Create a square bin plot
plt.figure(figsize=(8, 8))
plt.plot(data[:,0],data[:,1],'.r', alpha=0.5)
sns.kdeplot(x=data[:, 0], y=data[:, 1], cmap='Blues', fill=True)

# Set labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.axis('equal')
plt.title('Density estimation with histograms')

plt.xticks(np.arange(-3,8))
plt.yticks(np.arange(-3,8))
plt.xlim([-3,5])
plt.ylim([-2,7])
#plt.grid()

# Display the plot
plt.show()
```



Parametric Density Estimation

Parametric methods aim to estimate density by **fitting a parametric model to the data**. This has some advantages over non-parametric density estimation:

- **The chosen model has in general a known analytical formulation which we can reason on**, or that we can even plug into a cost function and differentiate to pursue some optimization objective;
- **The chosen model is compact.** While non-parametric models need to keep in memory the whole dataset to make density estimations at given points, once fitted, parametric models can be used to predict density values using the analytical formulation of the model;
- **If we choose an interpretable model, we can reason on the model to derive properties of the data.** For instance, if we fit a Gaussian on the data, then we know that the mean is the point with

highest density. We also know that when we go further from the mean, the density values decrease;
- **Parametric approaches impose constraints and hence work better when we do not have much data**. Non-parametric approaches, instead, make very little assumptions on the data, so they are better suited to the cases in which we have large datasets which are good representatives of the population.

Nevertheless, these methods also have disadvantages:

- **An optimization process is required** to fit the model to the data, i.e., find appropriate values which make the model "explain well the data". This can be a time-consuming process.
- Parametric models make **strong assumptions on the data**. For instance, if we fit a Gaussian to the data, we assume that the data is distributed in a Gaussian way. If this is not true, then the model will not be very accurate. This is a relevant point, as in many cases we cannot make strong assumptions on the data.

For the reasons above, **it is common to use non-parametric density estimations for visualizations** (e.g., the density plots we have seen and used in many cases), especially for exploration, when we do not know much about the data, and to use parametric model when we need to create efficient models able to make predictions (as we will see later in the course).

Fitting a Gaussian to the Data

The most common parametric method is to assume our data follows a **Multivariate Gaussian (Normal) Distribution**.

Recall the PDF (Probability Density Function) for a d-dimensional Gaussian:

$$N\left(\mathbf{x};\mu,\mathbf{\Sigma}\right)=\frac{1}{\sqrt{(2\pi)^d\det(\Sigma)}}e^{-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)}$$

This entire, complex distribution is defined by only two parameters:

- The mean vector μ (the center of the cloud).
- The covariance matrix $\mathbf{\Sigma}$ (the shape and orientation of the cloud).

Our task is to find the μ and $\mathbf{\Sigma}$ that **best fit our data** $\mathbf{X}$.

Maximum Likelihood Estimation (MLE)

Let $\mathbf{X}=\{\mathbf{x}_1,\dots,\mathbf{x}_N\}$ be a set of observations which we will assume to be **independent and drawn from a multivariate Gaussian distribution of unknown parameters**. We would like to find the parameters μ and $\mathbf{\Sigma}$ that identify the **most likely Gaussian distribution that may have generated the data**. To do so, we can use the **Maximum Likelihood (ML)** principle, which consists in finding the parameters which maximize the probability values that the Gaussian distribution would compute for the given data. The likelihood is defined as

$$P(\mathbf{X}|\mu, \mathbf{\Sigma}) = \prod_i \mathcal{N}(\mathbf{x}_i|\mu, \mathbf{\Sigma})$$

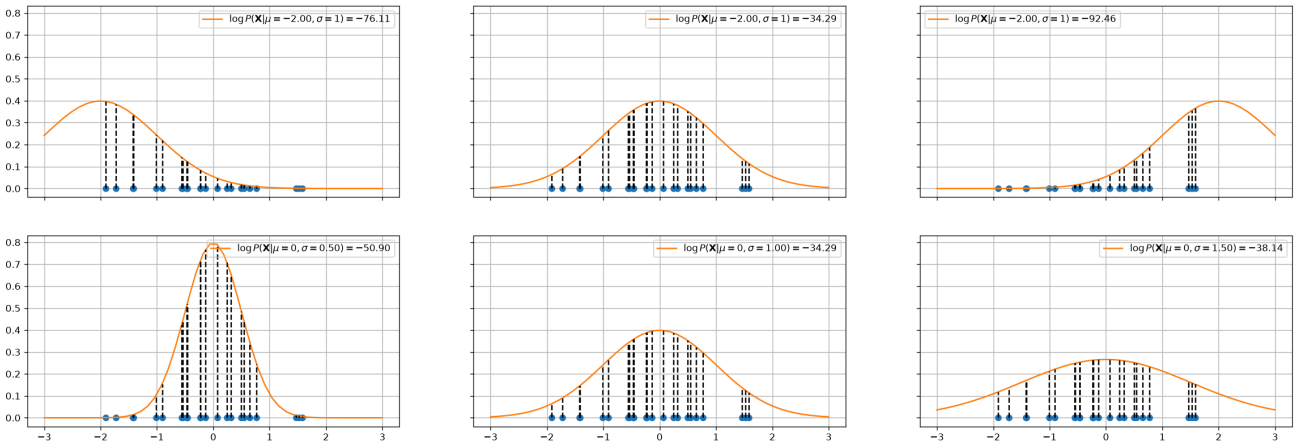
In practice, we maximize the log-likelihood instead.

Intuitively, this works as follows:

1. We "try" a specific Gaussian (a given μ and $\mathbf{\Sigma}$).
2. We calculate the probability density (the "likelihood") of *all* of our data points under this Gaussian.
3. We multiply these probabilities together to get the **Total Likelihood** of our entire dataset.
4. The goal is to find the *one* μ and $\mathbf{\Sigma}$ that **maximize this total likelihood**.

This sounds like a complex optimization problem, but for a Gaussian distribution, a closed-form solution exists. It can be shown by taking the derivative of the log-likelihood (the log of the likelihood function) and setting it to zero.

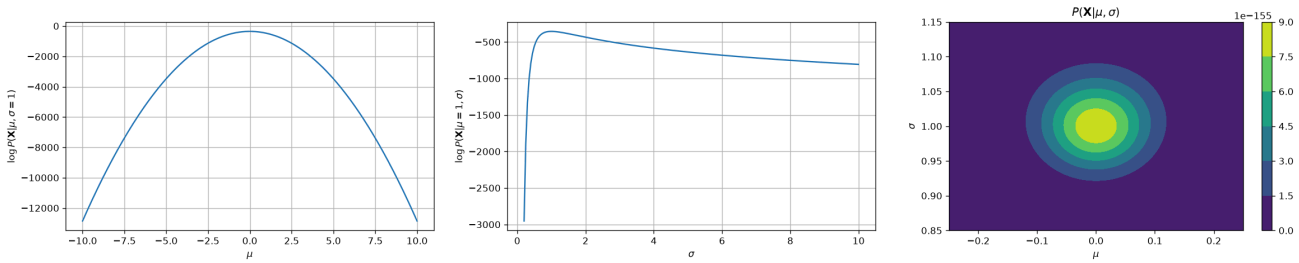
Let us consider a simple example in which we want to fit a Gaussian to a set of 1D data. The plot below shows the log likelihood for Gaussians with different means and standard deviations.



In the top plots, the standard deviation is fixed to 1, while the mean of the Gaussian is changed. The log likelihood is shown in the legend. As can be seen, the mean in the center plot leads to a larger log likelihood. **Intuitively, this happens when we choose a mean such that the majority of points fall in the central part of the Gaussian distribution**.

The plots in the bottom show a similar example in which the mean is fixed to 0 and the standard deviation is varied. As can be noted, when the standard deviation is too low (left) or to high (right), we obtain smaller log likelihoods.

The plots below show the log likelihood (first two plots) or the likelihood (last plot) as a function of different values of means and standard deviations.



The first two plots show how the log likelihood changes when μ and σ change, keeping the data fixed (same data as the previous plot). The last plot shows how the likelihood changes as a function of both μ and σ. As we can see, we have a maximum for $\mu = 0$ and $\sigma = 1$, which are the true parameters of the distribution which generated the data.

The Solution: Sample Mean and Covariance

The final result is incredibly intuitive. The Maximum Likelihood estimates for the Gaussian parameters are simply:

1. The ML estimate for the mean (μ_{ML}) is the sample mean:

$$\mu_{ML} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i$$

2. The ML estimate for the covariance ($\mathbf{\Sigma}_{ML}$) is the sample covariance matrix:

$$\mathbf{\Sigma}_{ML} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \mu_{ML})(\mathbf{x}_i - \mu_{ML})^T$$

(Note: In statistics, you often see a $1/(N-1)$ denominator for an "unbiased" estimate. The $1/N$ is the true Maximum Likelihood estimate, but in practice, they are nearly identical.)

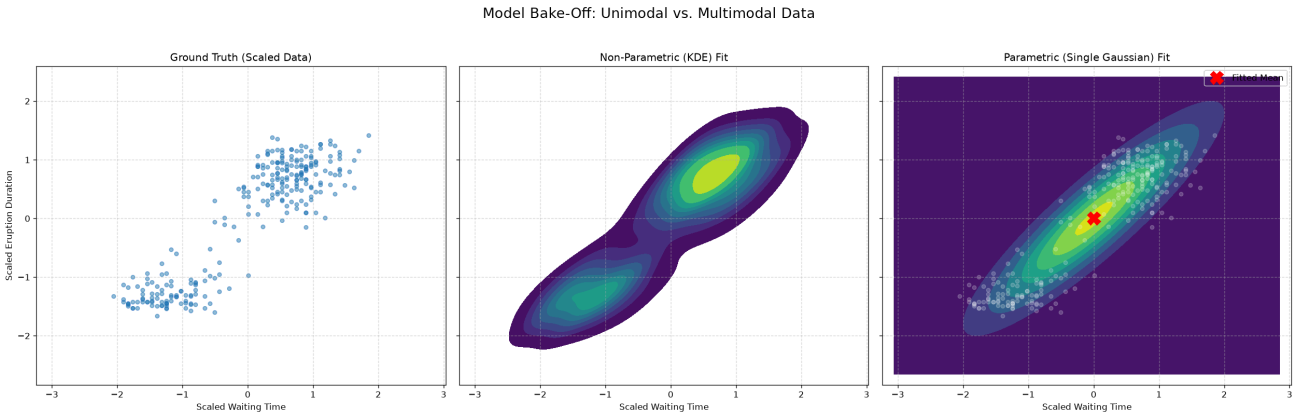
In short: "Fitting a Gaussian" is just a fancy way of saying "calculating the mean and covariance of your data."

The Weakness of a Single Gaussian: Bimodal Data

The single Gaussian model we just learned is powerful, but it's based on one very strong assumption: that our data comes from a *single cluster* (i.e., it is **unimodal**).

This assumption fails for many real-world datasets. A perfect example is the "Old Faithful" `geyser` dataset, which we know from our clustering lecture is **bimodal** (it has two distinct clusters).

What happens if we try to fit our simple parametric model (a single Gaussian) to this bimodal data? Let's compare it to the non-parametric KDE, which we know can find the true shape.



The result is clear:

- The **KDE plot (middle)** correctly finds the two high-density clusters. It is flexible and *non-parametric*, so it doesn't care that the data isn't a single "blob."
- The **Single Gaussian plot (right)** *fails*. It places its mean (the red 'X') in the empty, low-density space *between* the two clusters. It's a "unimodal" model trying to fit "bimodal" data.

This proves the limitation of our simple parametric model. To solve this, we need a more powerful parametric model that can *learn* to model multi-modal data.

Gaussian Mixture Models (GMM)

Our comparison showed a critical weakness: our simple parametric model, a single Gaussian, failed to capture the **bimodal** (two-cluster) structure of the `geyser` data. The non-parametric KDE *did* find the shape, but it doesn't give us a compact, mathematical model.

This leads to the obvious question: Can we create a *parametric* model that is flexible enough to handle multi-modal data?

Yes. The solution is a **Gaussian Mixture Model (GMM)**.

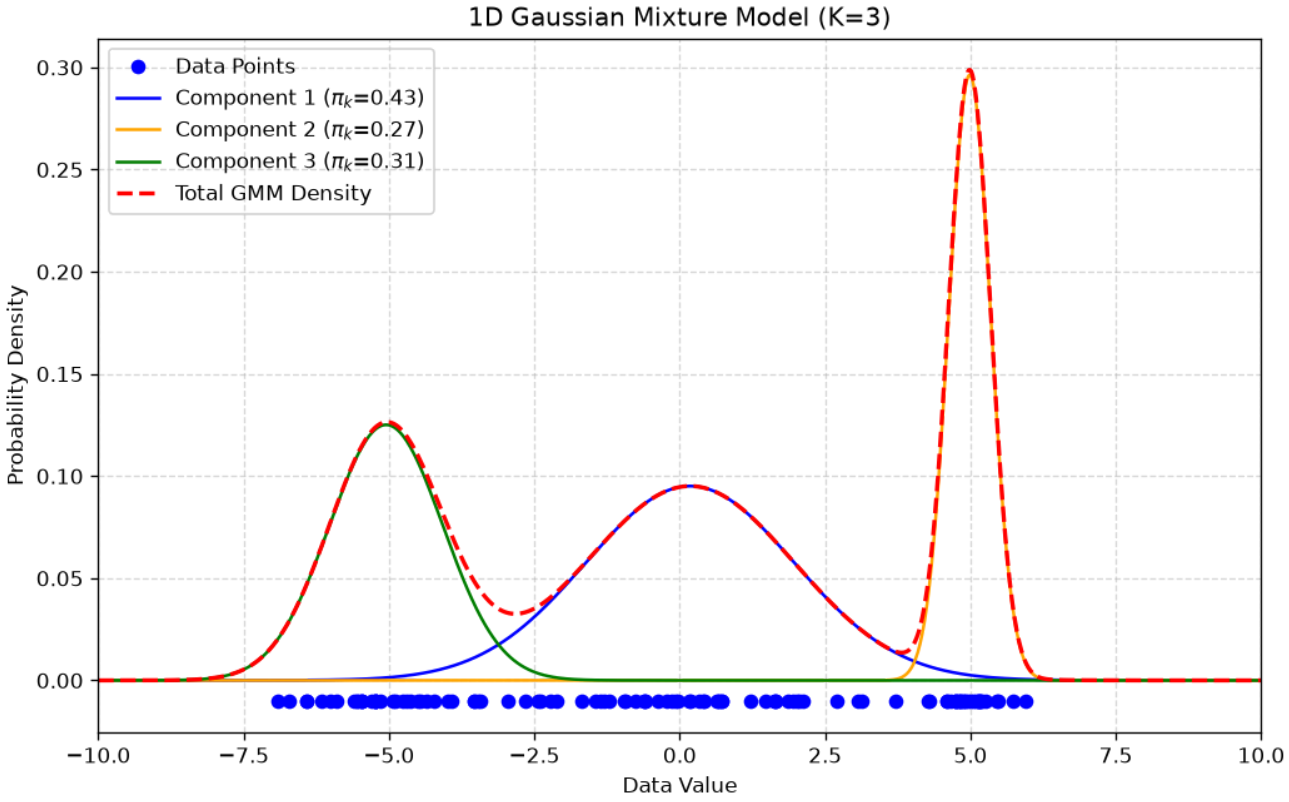
The idea is simple: instead of one Gaussian, we model the data as a **weighted average (a "mixture") of K different Gaussian components**.

$$P(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$$

This model is defined by three sets of parameters:

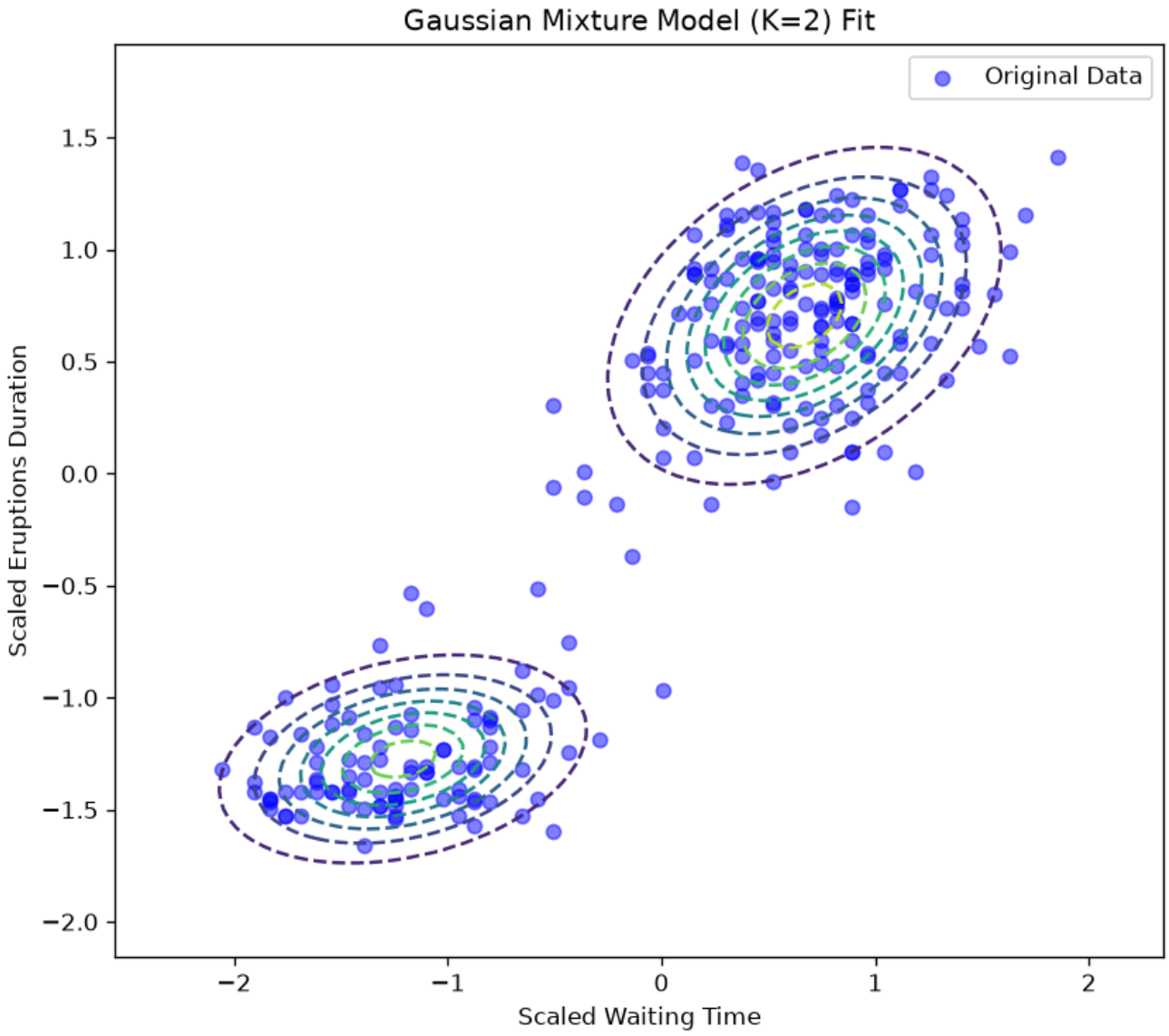
- π_k **(Mixing Coefficients):** The "weight" or prior probability of each component. (e.g., $\pi_1 = 0.6, \pi_2 = 0.4$). Must sum to 1.
- μ_k **(Means):** The center of each of the K Gaussian components.
- $\mathbf{\Sigma}_k$ **(Covariances):** The shape and orientation of each of the K components.

The 1D plot below shows a GMM with $K = 3$. The final density (red-dashed) is a *mixture* of three separate Gaussian components (blue, orange, green), allowing it to model complex, multi-modal data.



GMM Density Estimation Example

Now let's apply this to our `geyser` data. When we fit a GMM with $K = 2$, it should succeed where the single Gaussian failed.



Success! The GMM has learned two distinct Gaussian components that perfectly model the two clusters in our data.

GMM as “Soft Clustering”

GMM can be seen as a “soft” version of K-Means. If K-Means makes a hard assignment of data points to clusters, it can be shown that GMM does it in a “soft” way, computing instead the probability of assigning an element to a cluster.

It does this by introducing a **latent variable**, Z encoding the event:

> The example $\mathbf{x}$ belongs to a given cluster.

In practice, $Z = k$ denotes the case in which $X = \mathbf{x}$ belongs to cluster k.

A **latent variable** is a “hidden” variable for which we don’t observe ground-truth values, but which we assume is intrinsically part of the data. In this GMM, for every data point $\mathbf{x}_i$, there is a hidden z_i

that "indicates" which of the K Gaussian components *generated* that point.

To compute the soft assignment, we find the **posterior probability**, $P(Z = k|\mathbf{x})$: "What is the probability that this point $\mathbf{x}$ belongs to cluster k?"

We solve this using **Bayes' Theorem** to "flip" the question:

$$P(Z = k|\mathbf{x}) = \frac{P(\mathbf{x}|Z = k)P(Z = k)}{P(\mathbf{x})}$$

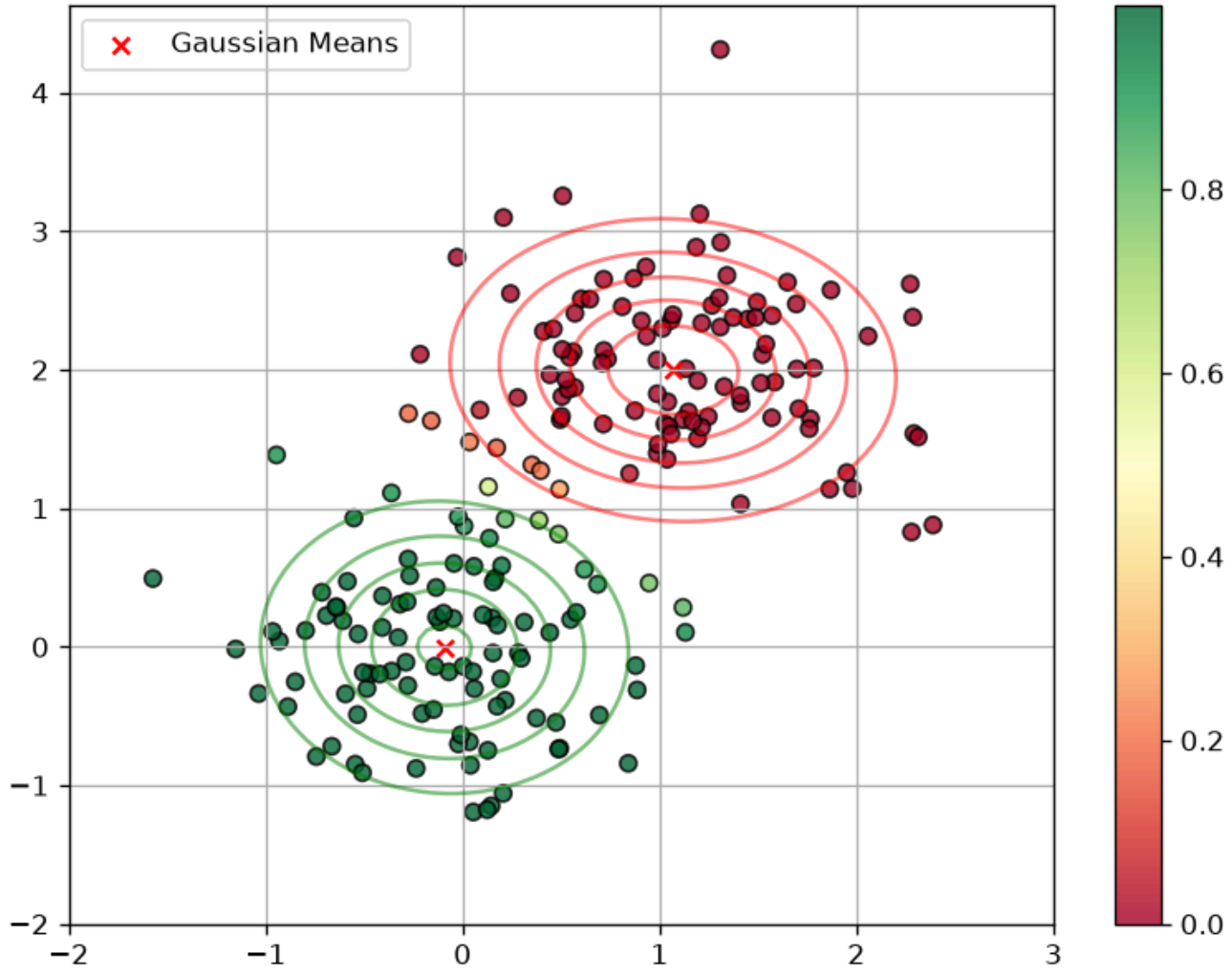
Once the GMM is trained (which we'll see how to do later with the E-M algorithm), we *have* all the pieces for this formula:

1. $P(\mathbf{x}|Z = k)$ **(The Likelihood):**
   - *In words:* "What is the probability of seeing $\mathbf{x}$ *if* it came from cluster k?"
   - *The value:* This is simply the value of our k-th Gaussian component: $\mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$.
2. $P(Z = k)$ **(The Prior):**
   - *In words:* "How common is cluster k in general?"
   - *The value:* This is our mixing coefficient, π_k.
3. $P(\mathbf{x})$ **(The Evidence):**
   - *In words:* "What is the total probability of seeing $\mathbf{x}$ under *any* cluster?"
   - *The value:* This is just the sum of all components, which is our GMM formula: $\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\mu_j, \mathbf{\Sigma}_j)$.

Putting it all together, the "soft assignment" (or **responsibility**, γ_k) for a point $\mathbf{x}$ belonging to cluster k is:

$$\gamma_k = P(Z = k|\mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\mu_j, \mathbf{\Sigma}_j)}$$

This formula is exactly what is being used to generate the colors in the plot below. Points in the middle get an "uncertain" color (like yellow, ≈ 0.5) because their probability is split between the two components.



How to “Learn” a GMM: Optimization by Maximum Likelihood

We have defined our GMM as a sum of weighted Gaussians:

$$P(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$$

The model’s parameters are all the means, covariances, and mixing coefficients:

$$\theta = (\pi, \mu, \mathbf{\Sigma}) = \{\pi_1, \dots, \pi_K, \mu_1, \dots, \mu_K, \mathbf{\Sigma}_1, \dots, \mathbf{\Sigma}_K\}$$

To “fit” the model, we use **Maximum Likelihood Estimation (MLE)**, just as we did for a single Gaussian. We want to find the parameters θ that maximize the log-likelihood of our observed data $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$:

$$\log P(\mathbf{X}|\theta) = \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x_i}|\mu_k, \mathbf{\Sigma}_k) \right]$$

This problem can not be optimized directly. Instead, a dedicated algorithm called Expectation-Maximization is used. We will not see this in details.

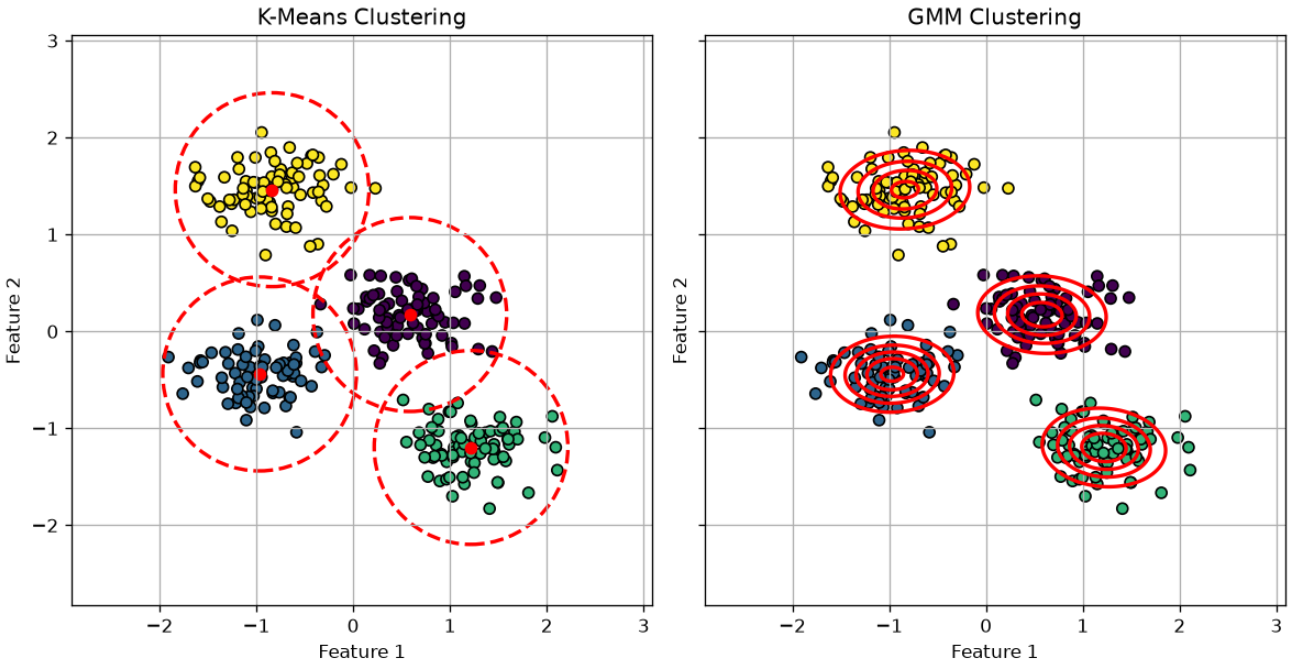
GMM vs K-Means

The EM algorithm is very similar to the optimization algorithm of K-Means. It can be shown that K-Means can be seen as a particular case of Gaussian Mixture Models in which:

- Points are assigned to clusters in a "*hard*" way, as compared to the **soft assignment** made by the responsibilities $\gamma(z_{ik})$ in Gaussian Mixture Models;
- Clusters are assumed to have diagonal covariance matrices $\epsilon\mathbf{I}$ where $\mathbf{I}$ is the $D \times D$ identity matrix and ϵ is a variance parameter which is **shared by all components**. This means that the clusters are assumed to be all **symmetrical and with the same shape**.

Despite these differences highlight the limitations of K-Means, it should be considered that K-Means is a significantly faster algorithm than GMM. Hence, if studying the probability density $P(\mathbf{X})$ is not needed and we can assume symmetrical clusters with similar shapes, K-Means is in practice a more popular choice, due to its faster convergence.

The plot below illustrates such differences:



Lab: Density Estimation for Anomaly Detection

In this lab, we will use our density estimation techniques to build a more complex **Anomaly Detector**.

The Goal

Our task is to build a model that can identify the digit **'0'** as an "anomaly."

1. **"Normal" Training Data:** We will train our models on a large dataset of "normal" digits (all images of '1's, '2's, '3's, ... '9's). This "normal" data is highly **multi-modal** (it has 9 distinct clusters).

2. **Test Data:** We will then create a test set containing a mix of "normal" digits (1-9) and "anomalous" digits ('0's).
3. **The "Bake-Off":** We will compare three models to see which is best at this task:
    - **Model 1 (Single Gaussian):** A "naive" parametric model. We expect this to fail, as it will try to fit one "average" blob over all 9 normal digits.
    - **Model 2 (GMM):** A "smart" parametric model. We will tell it to fit $K = 4$ components, to capture some diversity in the data.
    - **Model 3 (KDE):** A "flexible" non-parametric model.

A good model will give a **high density score** to the "normal" digits and a **very low score** to the "anomalous" '0's.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.mixture import GaussianMixture
from sklearn.neighbors import KernelDensity
from sklearn.metrics import roc_curve, auc
```

Step 1: Load and Prepare the Data

We'll load the `digits` dataset and split it into three groups:

1. `X_train_normal`: A training set of 70% of all "normal" digits (1-9).
2. `X_test_normal`: A test set of 30% of all "normal" digits (1-9).
3. `X_test_anomaly`: *All* of the "anomalous" digit '0's.

```
# 1. Load Data
X_all, y_all = load_digits(return_X_y=True)

# 2. Separate "normal" (1-9) from "anomaly" (0)
X_normal_all = X_all[y_all ≠ 0]
y_normal_all = y_all[y_all ≠ 0]

X_anomaly_all = X_all[y_all == 0]
y_anomaly_all = np.ones(len(X_anomaly_all)) # Label 1 = Anomaly

# 3. Create our Train/Test split
# We split the "normal" data into a train and test set
X_train_normal, X_test_normal, _, _ = train_test_split(
    X_normal_all, y_normal_all, test_size=0.3, random_state=42, stratify=y_normal_all
)
y_test_normal = np.zeros(len(X_test_normal)) # Label 0 = Normal

# 4. Create the final test set
X_test = np.concatenate((X_test_normal, X_anomaly_all))
y_test = np.concatenate((y_test_normal, y_anomaly_all))

print(f"Training data (Normal, 1-9): {X_train_normal.shape}")
print(f"Test data (Mixed): {X_test.shape}")
print(f"  → Normal '1-9's in test: {len(X_test_normal)}")
print(f"  → Anomalous '0's in test:  {len(X_anomaly_all)}")
```

```
Training data (Normal, 1-9): (1133, 64)
Test data (Mixed): (664, 64)
  → Normal '1-9's in test: 486
  → Anomalous '0's in test:  178
```

Step 2: Preprocessing (Scaling)

We will fit a `StandardScaler` *only* on our `X_train_normal` data and use it to transform both our train and test sets.

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_normal)
X_test_scaled = scaler.transform(X_test)
```

Step 3: Fit the Density Models

Now we will fit our three competitors to the `X_train_scaled` data (which contains 9 different clusters of digits).

```
# 1. Fit the Single Gaussian (Unimodal Parametric)
# This model will try to find the "average" of all 9 digits.
print("Fitting Single Gaussian (K=1)...")
model_single_gauss = GaussianMixture(n_components=1, random_state=42)
model_single_gauss.fit(X_train_scaled)

# 2. Fit the GMM (Multi-modal Parametric)
print("Fitting GMM (K=4)...")
model_gmm = GaussianMixture(n_components=4, random_state=42)
model_gmm.fit(X_train_scaled)

# 3. Fit the KDE (Non-Parametric)
# We'll use a bandwidth found from experimentation
print("Fitting KDE...")
model_kde = KernelDensity(kernel='gaussian')
model_kde.fit(X_train_scaled)

print("All three models are trained on the '1-9' digits.")
```

```
Fitting Single Gaussian (K=1)...
Fitting GMM (K=4)...
Fitting KDE...
All three models are trained on the '1-9' digits.
```

Step 4: Get Anomaly Scores

Now we'll use all three models to score our `X_test_scaled` data. We will use the **Negative Log-Likelihood (NLL)** as our "anomaly score."

- `model.score_samples()` returns the log-likelihood (high = normal).
- `-model.score_samples()` returns the NLL.

With NLL, a **low score** means "this point looks normal," and a **high score** means "this point looks weird (anomalous)." This is a more intuitive way to think of an "anomaly score."

```
# Get the NLL scores for every point in the test set
# (NLL = -LogLikelihood)
scores_single = -model_single_gauss.score_samples(X_test_scaled)
scores_gmm = -model_gmm.score_samples(X_test_scaled)
scores_kde = -model_kde.score_samples(X_test_scaled)

# Store results in a DataFrame for easy plotting
results_df = pd.DataFrame({
    'y_true': y_test,
    'SingleGaussian_NLL': scores_single,
    'GMM_K4_NLL': scores_gmm,
    'KDE_NLL': scores_kde
})

results_df['Label'] = results_df['y_true'].map({0: 'Normal (1-9)', 1: 'Anomaly (0)'})
```

Step 5: Quantitative Comparison (The ROC Curve)

The histograms show GMM and KDE are much better, but the **ROC Curve** gives us the definitive quantitative score.

We will plot the **True Positive Rate** (finding the '0's) vs. the **False Positive Rate** (falsely flagging the '1-9's) for all three models. The **Area Under the Curve (AUC)** will be our final metric.

```
from sklearn.metrics import roc_auc_score

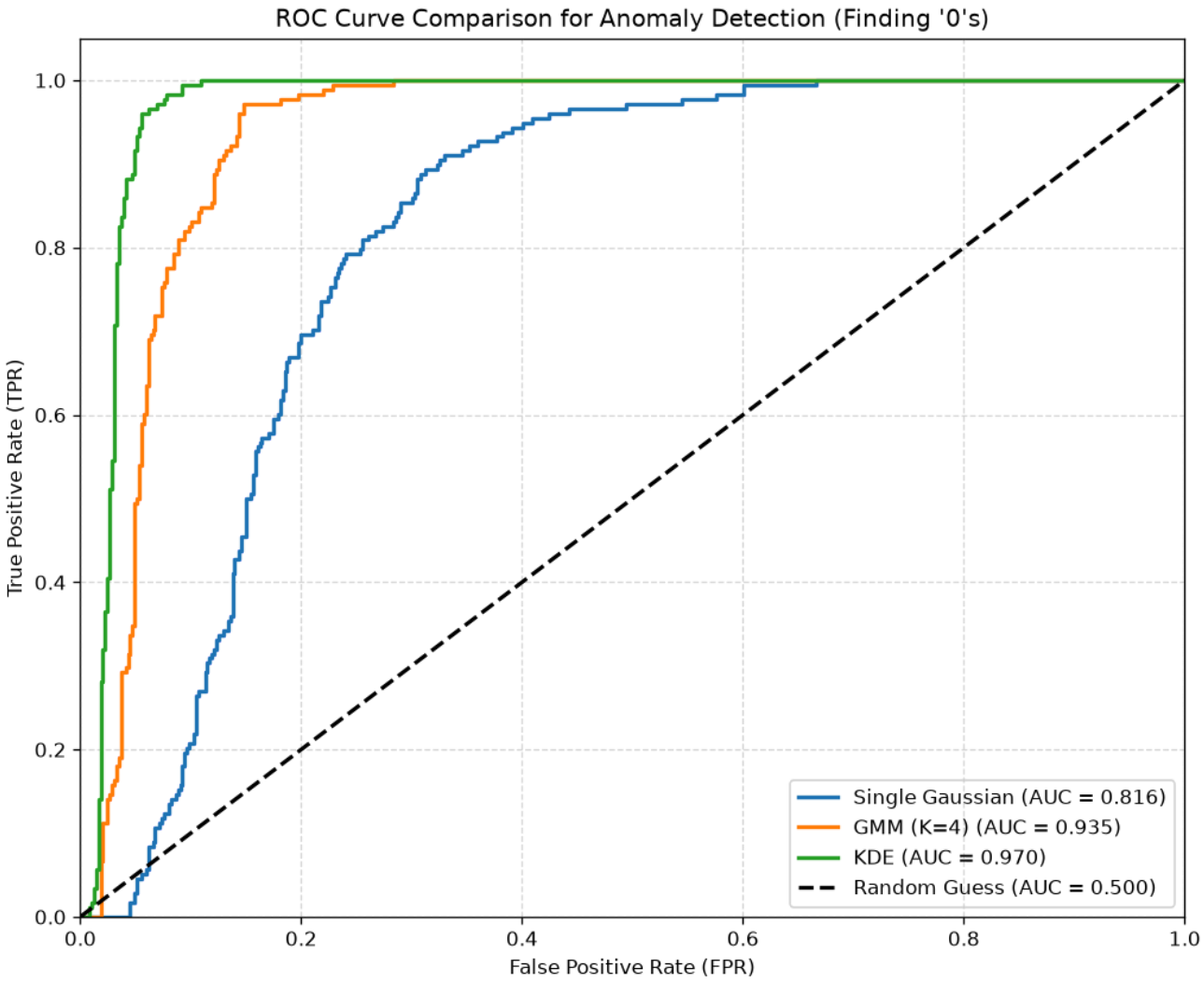
# --- Create the ROC Plot ---
plt.figure(figsize=(10, 8))

# --- 1. Single Gaussian ---
fpr, tpr, _ = roc_curve(results_df['y_true'], results_df['SingleGaussian_NLL'])
auc_single = auc(fpr, tpr)
plt.plot(fpr, tpr, lw=2, label=f'Single Gaussian (AUC = {auc_single:.3f})')

# --- 2. GMM (K=9) ---
fpr, tpr, _ = roc_curve(results_df['y_true'], results_df['GMM_K4_NLL'])
auc_gmm = auc(fpr, tpr)
plt.plot(fpr, tpr, lw=2, label=f'GMM (K=4) (AUC = {auc_gmm:.3f})')

# --- 3. KDE ---
fpr, tpr, _ = roc_curve(results_df['y_true'], results_df['KDE_NLL'])
auc_kde = auc(fpr, tpr)
plt.plot(fpr, tpr, lw=2, label=f'KDE (AUC = {auc_kde:.3f})')

# --- 4. Plot Styling ---
plt.plot([0, 1], [0, 1], 'k--', lw=2, label='Random Guess (AUC = 0.500)')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve Comparison for Anomaly Detection (Finding \'0\'s)')
plt.legend(loc='lower right')
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```



We note that the most flexible density estimation model is the KDE in this case. A single Gaussian oversimplifies the problem, while a GMM adds back some flexibility.

Exercises

1. **Bins lie.** Plot a histogram of the same variable with 5, 20 and 100 bins, and with the bin edges shifted by half a bin. How many different stories can one variable tell?
2. **Bandwidth.** Plot kernel density estimates of the same sample with a very small, a very large and a cross-validated bandwidth. Relate the three pictures to the bias-variance tradeoff.
3. **Kernel choice.** Compare Gaussian, Epanechnikov and tophat kernels at the same bandwidth. Which choice matters more, the kernel or the bandwidth?
4. **Parametric versus not.** Fit a single Gaussian and a KDE to a clearly bimodal sample. Plot both. What exactly does the parametric model get wrong, and what does it get right?
5. **Mixtures.** Fit Gaussian mixtures with 1 to 6 components and select the number by BIC. Does the selected model recover the structure you built into the data?
6. **Anomaly detection.** Use a fitted density to flag the 1% least likely points of a dataset. Inspect them. Are they errors, rare cases, or perfectly ordinary points that happen to sit in a sparse region?

References

- https://en.wikipedia.org/wiki/Multivariate_kernel_density_estimation
- https://en.wikipedia.org/wiki/Kernel_density_estimation
- http://faculty.washington.edu/yenchic/18W_425/Lec7_knn_basis.pdf
- Section 9.2 of [1]

[1] Bishop, Christopher M., and Nasser M. Nasrabadi. Pattern recognition and machine learning. Vol. 4. No. 4. New York: springer, 2006.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources

Dimensionality reduction: principal component analysis

Chapter resources

Slides (PDF) Open in Colab Practice quiz

So far, we have worked with datasets with a handful of interpretable features. In the real world, we often face datasets with *thousands* or even *millions* of features.

- An 800x600 pixel color image has $800 \times 600 \times 3 = 1,440,000$ features (pixels).
- A "bag of words" model for text can easily have 50,000 features (one for each word).

This creates two major problems:

1. **The Curse of Dimensionality:** In high-dimensional space, everything is "far apart," and distance-based models like KNN fail.
2. **Multicollinearity:** Many features are highly redundant (e.g., adjacent pixels in an image).

We cannot just **select** features (like we did with backward elimination) because every pixel in an image is *necessary*. Instead, we need to *summarize* or *compress* the data.

This is **Dimensionality Reduction**. Our goal is to transform our data from d (e.g., 1,440,000) dimensions to a new, smaller set of k (e.g., 100) "smart" features, while losing the least amount of information possible.

The most common way to do this is **Principal Component Analysis (PCA)**.

Feature Selection vs Feature Reduction

We have seen how, when many variables are available, it makes sense to select a subset of such variables. In regression analysis, in particular, we have seen how, when features are highly correlated, we should **discard some of them to reduce collinearity**. Indeed, if two variables x and y are highly correlated, one of the two is redundant to a given extent, and we can ignore it.

We have seen how variables can be selected in different ways:

- By looking at the correlation matrix, we can directly find those pairs of variables which are highly correlated;
- When defining a linear or logistic regressor, we can remove those variables with a high p-value. We have seen that these are variables which do not contribute significantly to the prediction of the dependent variable, given all other variables present in the regressor;

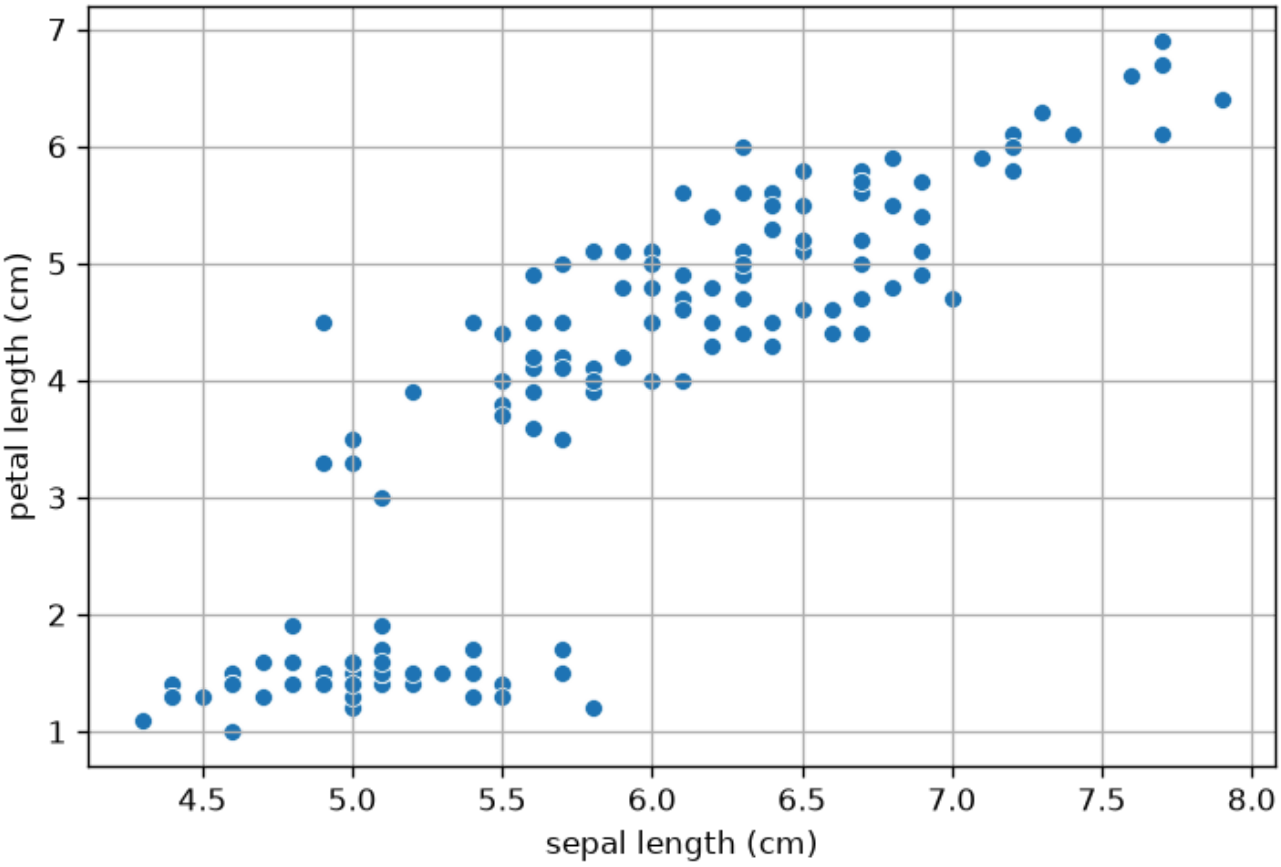
- Techniques such as **Ridge and Lasso regression** allow to perform some form of variable (or feature) selection, setting very low or zero coefficients for variables which do not contribute significantly to regression.

In general, however, when we discard a set of variables, we throw away some informative content, unless the variables we are removing can be perfectly reconstructed from the variables we are keeping, e.g., because they are linear combinations of other variables.

Instead of selecting a subset of features to work with, **feature reduction techniques** aim to **find a new set of features which summarize the original set of features losing a small amount of information, while maintaining a limited number of dimensions**.

Feature Reduction Example

Let us consider the Iris dataset. In particular, we will consider two features: `sepal length` and `petal length`:

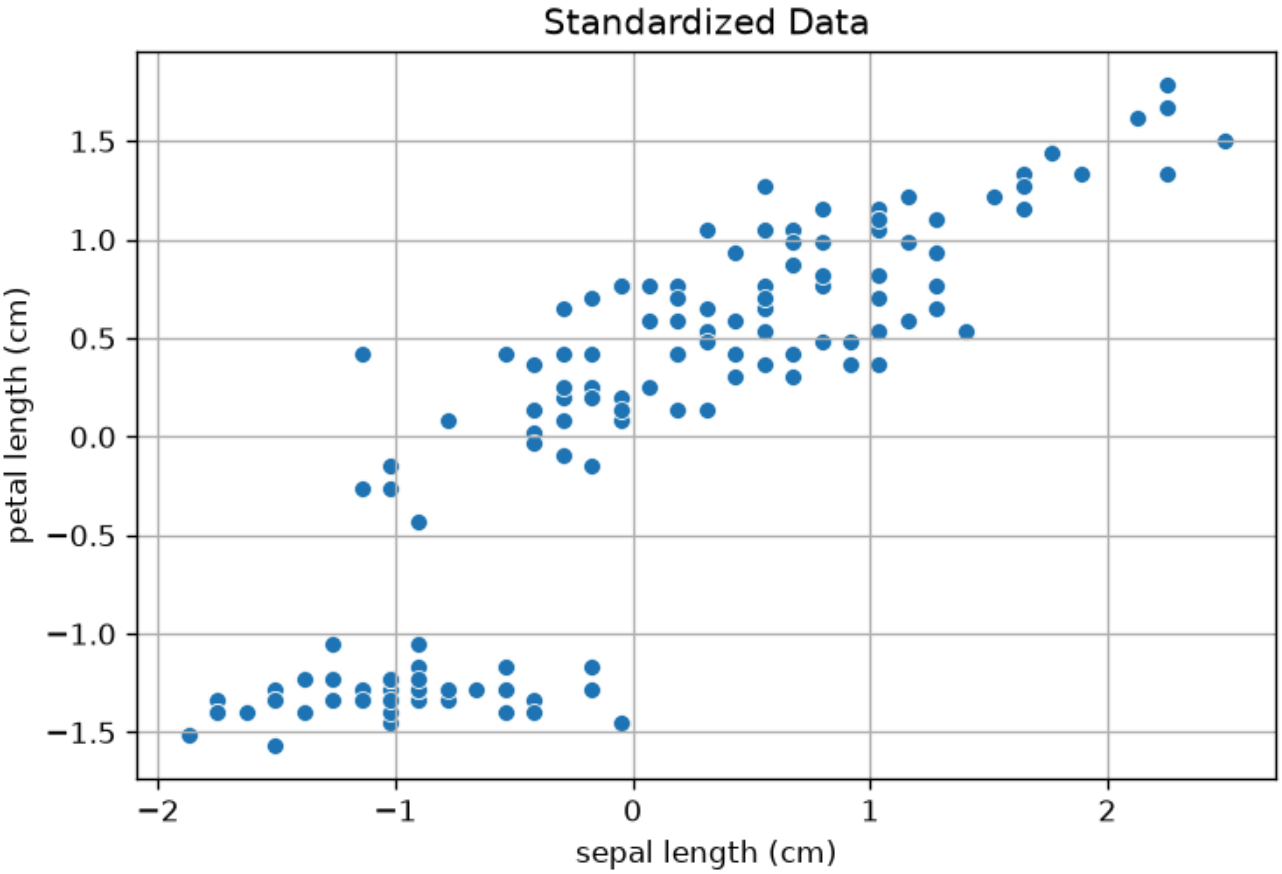


We can see from the plot above that the two features are highly correlated. Let us compute the Pearson coefficient and the related p-value:

```
PearsonRResult(statistic=np.float64(0.8717537758865832),
pvalue=np.float64(1.0386674194497531e-47))
```

Let's say we want to reduce the number of features (e.g., for computational reasons). We could think of applying **feature selection** and choose one fo the two, but we would surely **loose some information**. Instead, we could think of a **linear projection of the data into a single dimension** such that we loose the least possible amount of information.

To avoid any dependence on the units of measures and on the positioning of the points in the space, let's first standardize the data:

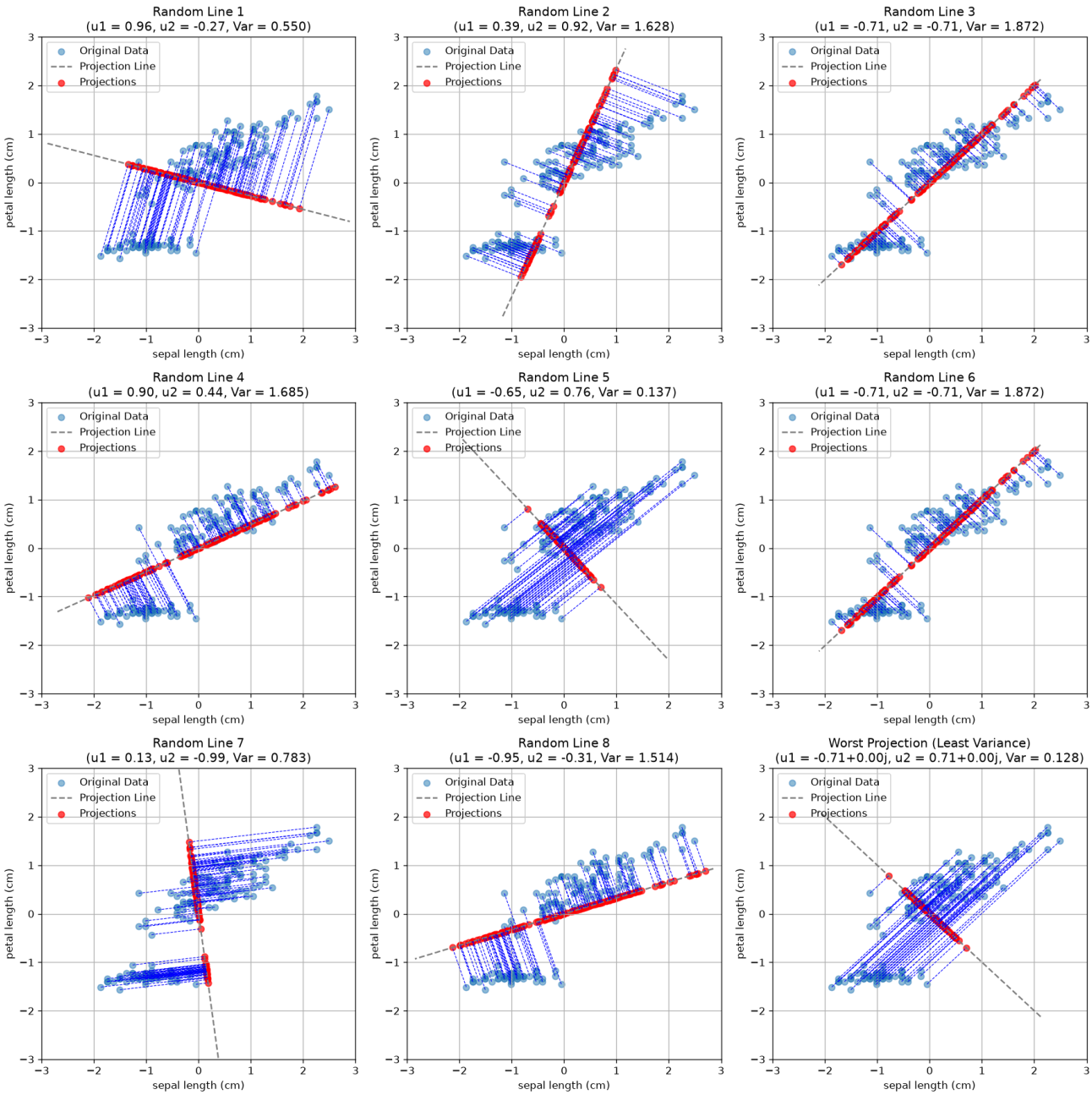


Transforming the 2D data to a single dimension with a linear transformation can be done with the following expression:

$$z = u_1 x_1 + u_2 x_2 = \mathbf{u}^T \mathbf{x}$$

Note that this is equivalent to projecting point (x_1, x_y) to a line passing through the origin and with the same direction on $\mathbf{u}$.

Once we choose a set of (u_1, u_2), we obtain a different projection, as in the example below, where we project the data to different random lines passing through the origin:



As can be seen, **not all projections are the same**. Indeed, some of them tend to project the points in a space in which they are very close to each other, while others allow for more “space” between the points.

Intuitively, we would like to project points in a way that **they keep their distinctiveness as much as possible**. In practice, we can quantify **distinctiveness** with **the variance of the projected data**.

General Formulation

We have seen a simple example in the case of two variables. Let us now discuss the general formulation in the case of D-dimensional variables.

Let $\{\mathbf{x}_n\}$ be a set of N observations, where $n = 1, \dots, N$, and $\mathbf{x}_n$ is a vector of dimensionality D. We will define Principal Component Analysis as a projection of the data to $M < D$ dimensions such that the variance of the projected data is maximized.

NOTE: to avoid any influence by the individual variances along the original axes, we will assume that **the data is standardized**.

We will first consider the case in which $M = 1$. In this case, we want to find a D-dimensional vector $\mathbf{u}_1$ such that, projecting the data to this vector, variance is maximized. Without loss of generality, we will choose $\mathbf{u}_1$ such that $\mathbf{u}_1^T \mathbf{u} = 1$. This corresponds to choosing $\mathbf{u}_1$ as a **unit vector**, which makes sense, as we are interested in the direction of the projection. We will find this assumption useful later.

Let $\overline{\mathbf{x}}$ be the sample mean:

$$\overline{\mathbf{x}} = \frac{1}{N} \sum_{n=1}^{N} x_n$$

The data covariance matrix $\mathbf{S}$ will be:

$$\mathbf{S} = \frac{1}{N} \sum_{n=1}^{N} (\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^T$$

We know that projecting a data point to $\mathbf{u}_1$ is done simply by the dot product:

$$\mathbf{u}_1^T \mathbf{x}$$

It is easy to see that the mean of the projected data is:

$$\mathbf{u}_1^T \overline{\mathbf{x}}$$

Its variance is:

$$\frac{1}{N} \sum_{n=1}^{N} (\mathbf{u}_1^T \mathbf{x}_n - \mathbf{u}_1^T \overline{\mathbf{x}})(\mathbf{u}_1^T \mathbf{x}_n - \mathbf{u}_1^T \overline{\mathbf{x}})^T = \mathbf{u}_1^T \frac{1}{N} \sum_{n=1}^{N} (\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^T \mathbf{u}_1 = \mathbf{u}_1^T \mathbf{S} \mathbf{u}_1$$

To find an appropriate $\mathbf{u}_1$, we need to maximize the variance of the projected data with respect to $\mathbf{u}_1$. The optimization problem can be formalized as follows:

$$\arg\max\{\mathbf{u}_1^T\mathbf{S}\mathbf{u}_1\} \text{ subject to } \mathbf{u}_1^T\mathbf{u}_1 = 1$$

Note that the constraint $\mathbf{u}_1^T\mathbf{u}_1 = 1$ is necessary. Without it, we could arbitrarily increase the variance by choosing vectors $\mathbf{u}_1$ with large modules, hence $\mathbf{u}_1^T\mathbf{u}_1 \rightarrow +\infty$.

We will not see how this optimization problem is solved in details, but it can be shown that the variance is maximized when:

$$\mathbf{u}_1^T\mathbf{S}\mathbf{u}_1 = \lambda_1$$

Note that this is equivalent to:

$$\mathbf{S}\mathbf{u}_1 = \lambda_1\mathbf{u}_1$$

From the result above, we find out that:

- The solution $\mathbf{u}_1$ is an eigenvector of $\mathbf{S}$. The related eigenvalue λ_1 is the variance along that dimension;
- Since we want to maximize the variance, among all eigenvectors, we should choose the one with the **largest eigenvalue** λ_1.

The vector $\mathbf{u}_1$ will be **the first principal component** of the data.

We can proceed in an iterative fashion to find the other components. To obtain a valid base (a new reference system), we will choose the next component $\mathbf{u}_2$ such that $\mathbf{u}_2 \perp \mathbf{u}_1$. We can hence identify the second principal component by choosing the vector $\mathbf{u}_2$ with maximizes the variance **among all vectors which are orthogonal to** $\mathbf{u}_1$.

In practice, it can be shown that the M principal components can be found by choosing the M eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$ of the covariance matrix $\mathbf{S}$ corresponding to the largest eigenvectors $\lambda_1, \ldots, \lambda_M$.

Let us define the matrix W as follows:

$$\mathbf{W} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \ldots \\ \mathbf{u}_M \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1D} \\ u_{21} & u_{22} & \ldots & u_{2D} \\ \ldots & & & \\ u_{M1} & u_{M2} & \ldots & u_{MD} \end{pmatrix}$$

where u_{ij} is the jth component of vector $\mathbf{u}_i$.

Let:

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \dots \\ \mathbf{x}_N \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1D} \\ x_{21} & x_{22} & \dots & x_{2D} \\ \dots & & & \\ x_{N1} & x_{N2} & \dots & x_{ND} \end{pmatrix}$$

be the $[N \times D]$ data matrix.

We can project the data matrix X to the principal components with the following formula:

$$\mathbf{Z} = \mathbf{X}\mathbf{W}^T$$

$\mathbf{Z}$ will be an $[N \times M]$ matrix in which the ith row denotes the ith element of the dataset projected to the PCA space.

We will call $\mathbf{Z}$ latent variables as they are not observed and encode properties of the data.

In geometrical terms, the PCA **performs a rotation of the D-dimensional data around the center of the data** in a way that:

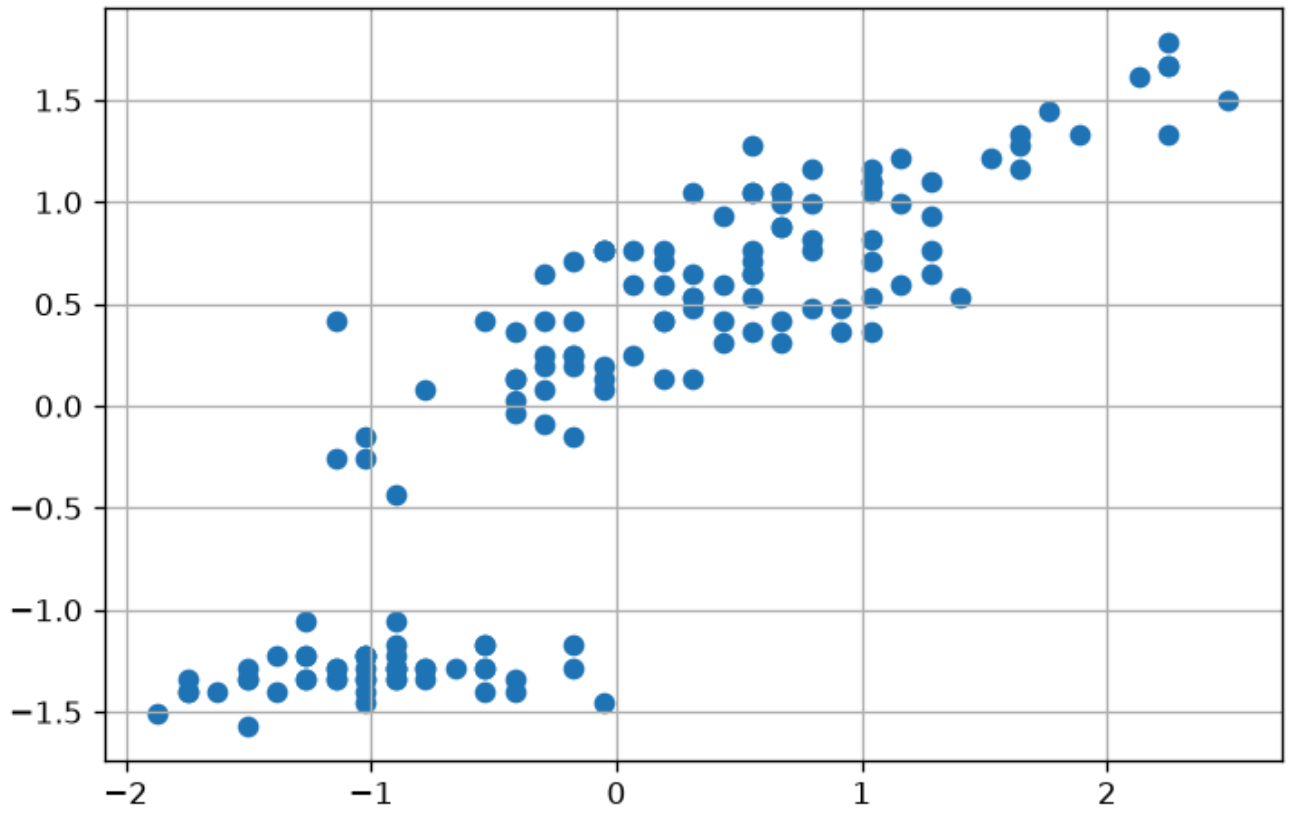
- The new axes are sorted by variance;
- The projected data is uncorrelated.

Back to Our Example

We now have the computational tools to fully understand our example. Let us plot the data again. We will consider the mean-centered data:

```
standardized_data.shape
```

```
(150, 4)
```



We have seen that the covariance matrix plays a central role in PCA. The covariance matrix of the original data will be as follows in our case:

```
array([[1.00671141, 0.87760447],
       [0.87760447, 1.00671141]])
```

The eigvenvalues and eigenvectors will be as follows:

```
Eigenvalues: [0.12910694 1.88431588]

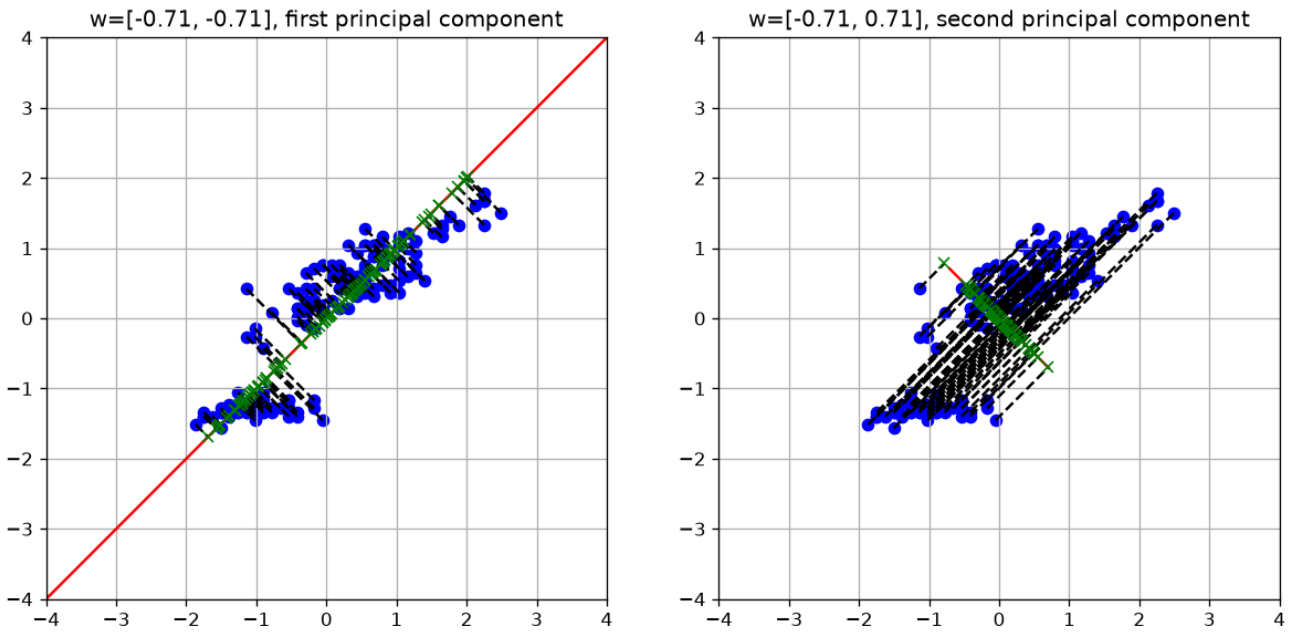
Eigenvectors:
 [[-0.70710678  0.70710678]
 [-0.70710678 -0.70710678]]
```

The rows of the eigenvector matrix identify two principal components. Let us re-order the principal components so that eigenvalues are descending (we want the first principal component to have the largest value):

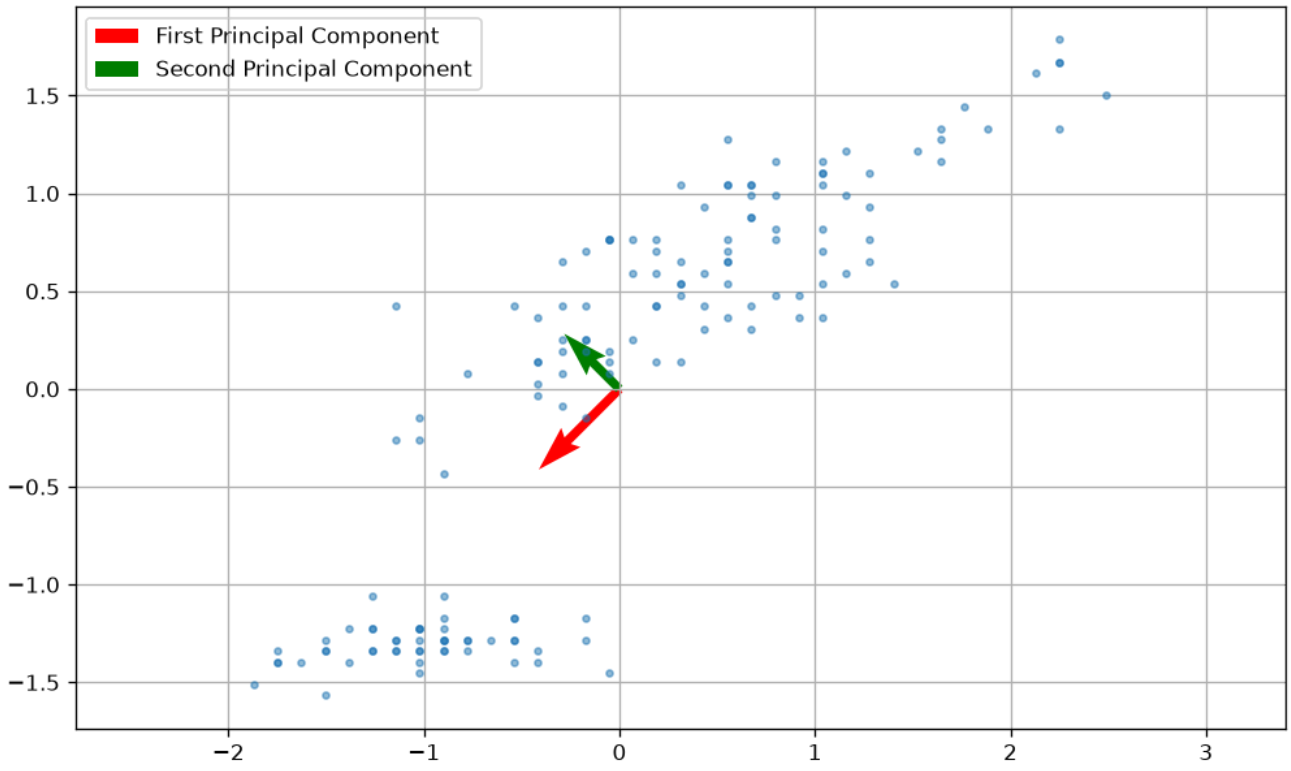
```
Eigenvalues: [1.88431588 0.12910694]

Eigenvectors:
 [[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]
```

The two components identify two directions along which we can project our data, as shown in the following plot:



The eigenvectors can also be interpreted as vectors in the original space. Getting back to our 2D example, they would be shown as follows:

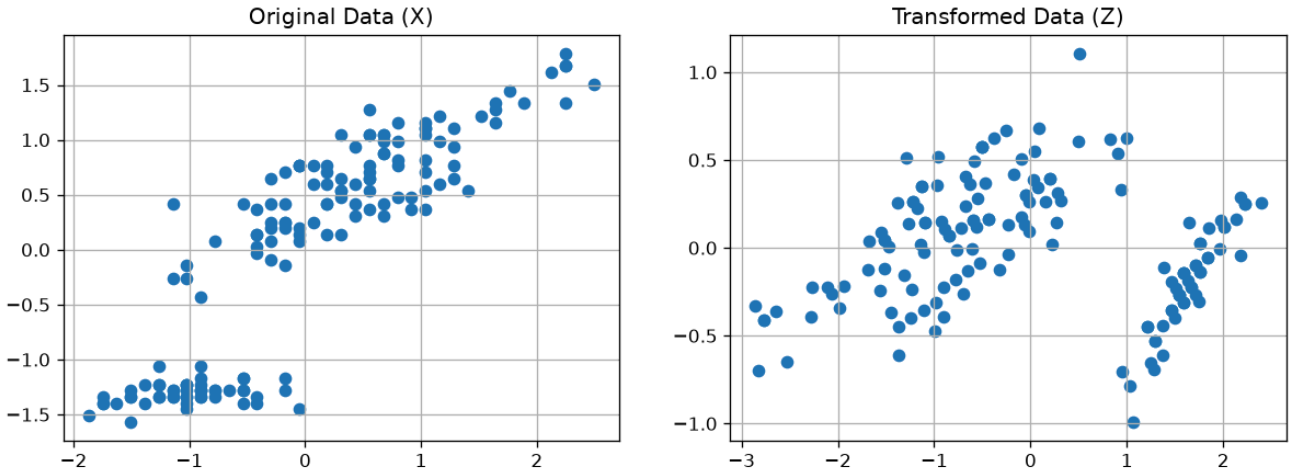


The two principal components denote the **main directions of variation within the data**. In the plot above, the vector sizes are proportional to the related eigenvalues (hence the variance).

As we can see, the two directions are orthogonal. We can also project the data maintaining the original dimensionality without truncating matrix $\mathbf{W}$ (in practice we set $M = D$) with:

$$\mathbf{Z} = \mathbf{X}\mathbf{W}^T$$

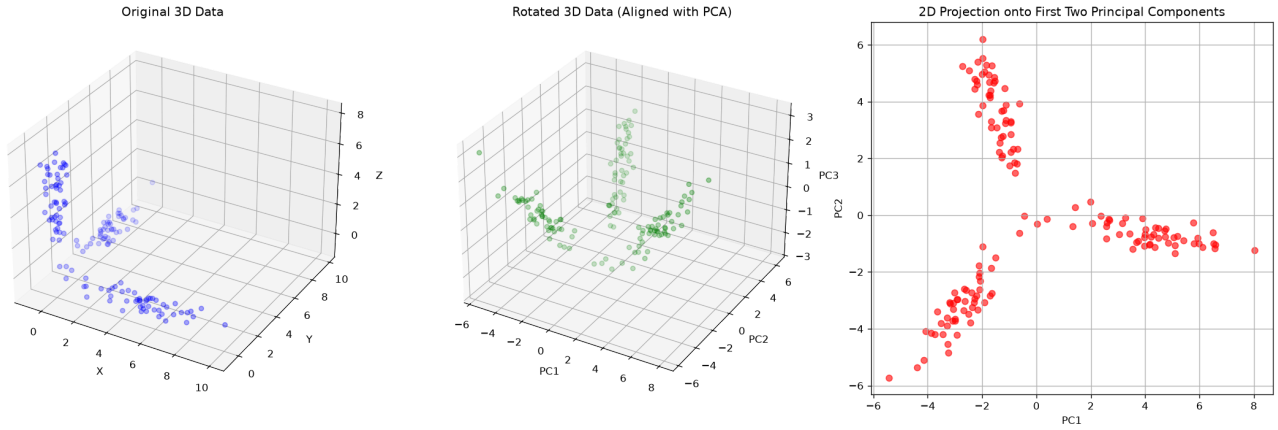
This process can also see as rotating the data in a way that the the first principal component becomes the new x axis, while the second principal component becomes the y axis. In this context, $\mathbf{W}$ can be seen as a **rotation matrix**. The plot below compares the original and rotated data:



We can see the projection to the first principal component in two stages:

1. Rotating the data;
2. Dropping the y axis.

Note that the same reasoning can be seen in 3D as shown in the following example:

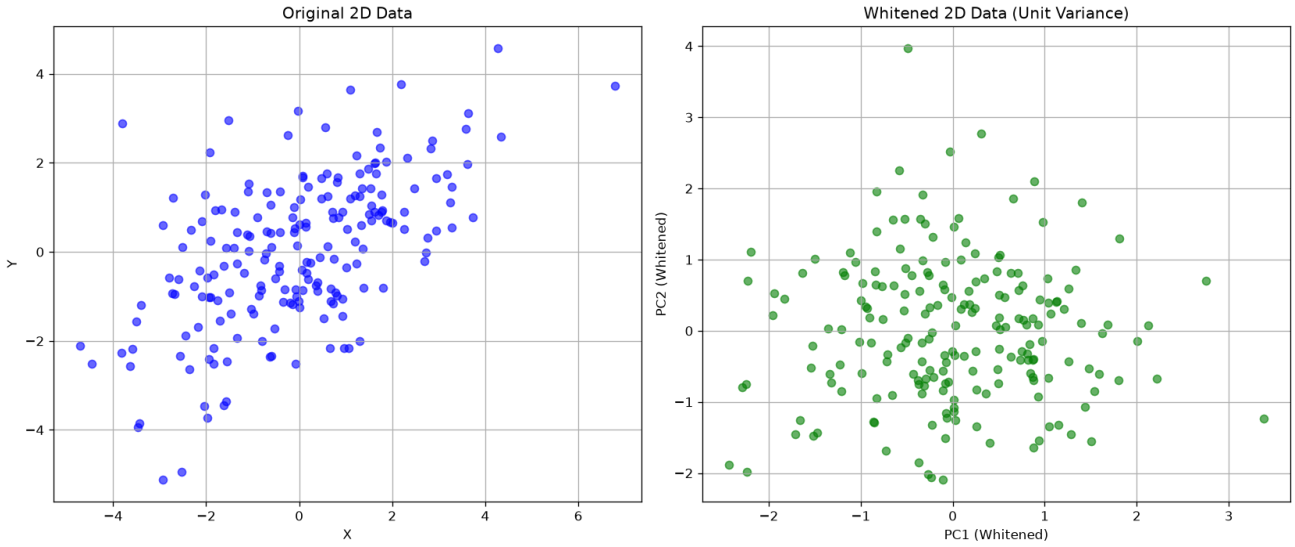


PCA for data whitening

It can also be seen that PCA transforms the original data with a given covariance matrix Σ to a new set of data (with new axes, the principal components), such that its covariance matrix Σ' is **a diagonal matrix**. This means that **the new variables** are now **decorrelated**.

Intuitively, this makes sense: **we rotated the data so that the new axes are aligned to main directions of variation and we constructed principal components to be orthogonal to each other**.

This process is also called **whitening transformation**. An example of this property is shown below:




```
(array([[3.60543087, 1.84308349],
        [1.84308349, 2.83596216]]),
 array([[ 1.00000000e+00, -6.92704504e-16],
        [-6.92704504e-16,  1.00000000e+00]]))
```

This whitening process can be useful when we need to make sure that the input variables are **decorrelated**. For instance, consider the case in which we need to fit a linear regression but we have multiple correlated variables. Pre-processing data with PCA would make sure that the input variables are decorrelated, hence removing any instability during fitting, but it should be also kept in mind that **the linear regression would become less interpretable because of the PCA transformation**.

Choosing an Appropriate Number of Components

We have seen how PCA allows to reduce the dimensionality of the data. In short, this can be done by computing the first M eigenvalues and the associated eigenvectors.

In some cases, it is clear what is the value of M we need to set. For example, in the case of sorting Fisher's Iris, we set $M = 1$ because we needed a scalar number. In other cases, we would like to reduce the dimensionality of the data, **while keeping a reasonable amount of information about the original data**.

We have seen that the variance is related to the MSE reprojection error and hence to the informative content. We can measure **how much information we are retaining by selecting M components by measuring the cumulative variance of the first M components**.

We will now consider as an example the whole Fisher Iris dataset, which has four features. The four Principal Components will be as follows:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
components				
z1	0.361387	0.656589	-0.582030	0.315487
z2	-0.084523	0.730161	0.597911	-0.319723
z3	0.856671	-0.173373	0.076236	-0.479839
z4	0.358289	-0.075481	0.545831	0.753657

The covariance matrix of the transformed data will be as follows:

	z1	z2	z3	z4
z1	4.23	0.00	0.00	0.00
z2	0.00	0.24	-0.00	0.00
z3	0.00	-0.00	0.08	-0.00
z4	0.00	0.00	-0.00	0.02

As we could expect, the covariance matrix is diagonal because the new features are decorrelated. We also note that the principal components are sorted by variance. We will define the variance of the data along the z_i component as:

$$var(z_i)$$

The total variance is given by the sum of the variances:

$$V = var(z_1) + var(z_2) + \ldots + var(z_D)$$

More in general, we will define:

$$V(n) = var(z_1) + var(z_2) + \ldots + var(z_n)$$

Hence $V = V(D)$.

In our example, the total variance V will be:

```
4.57
```

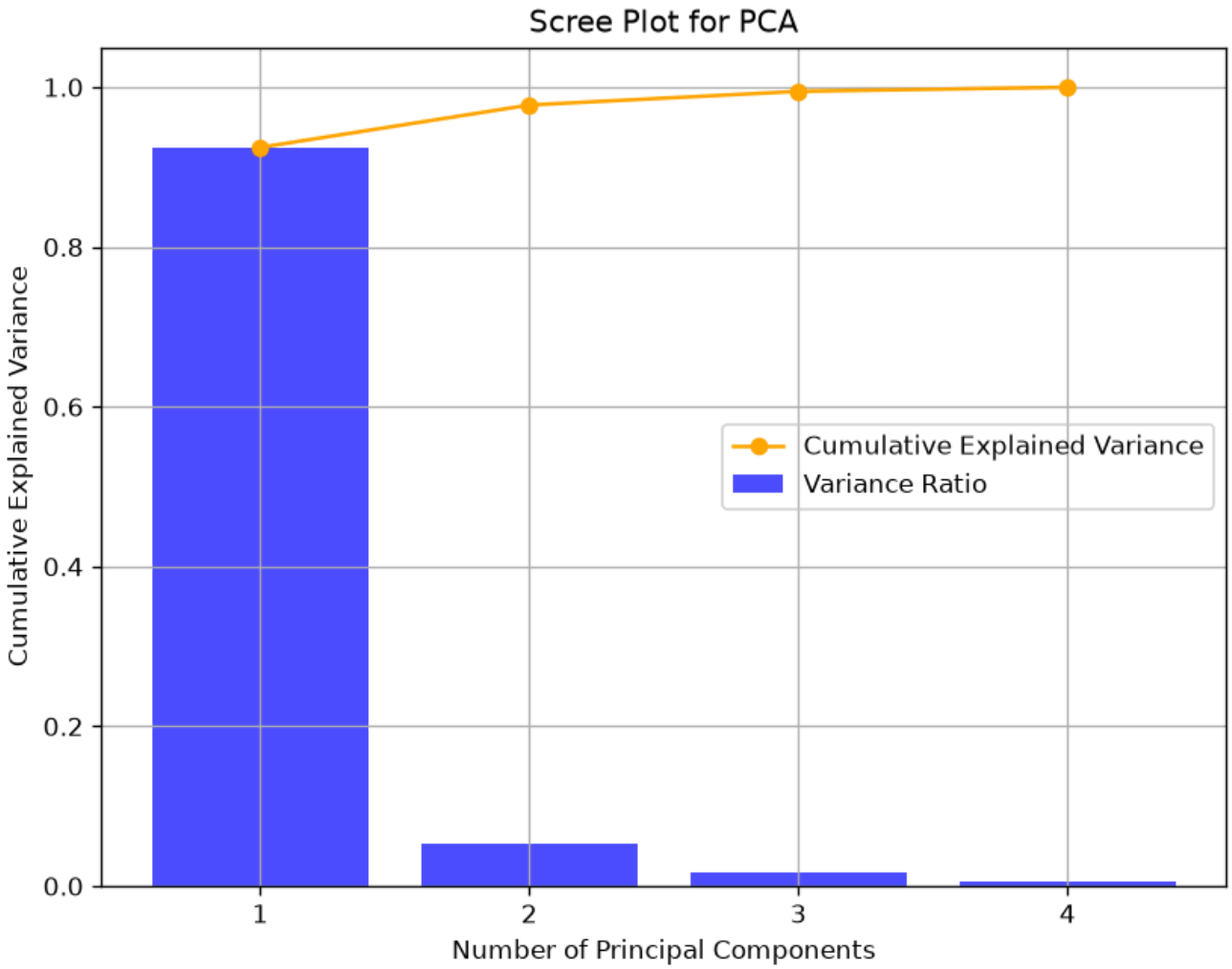
We can quantify the fraction of variance explained by each component n as follows:

$$\frac{var(z_n)}{V(D)}$$

If we compute such value for all n, we obtain the following vector in our example:

```
array([0.92461872, 0.05306648, 0.01710261, 0.00521218])
```

In practice, it is common to visualize the explained variance ratio in a **scree plot**:



As we can see, the first component retains 92.46% of the variance, while the second one retains only 5.3% of the variance. It is common to also look at the cumulative variance ratio, defined as:

$$\frac{V(n)}{V(D)}$$

In our case:

```
array([0.92461872, 0.97768521, 0.99478782, 1.        ])
```

The vector above tell us that considering the first two components accounts for about 97.77% of the variance.

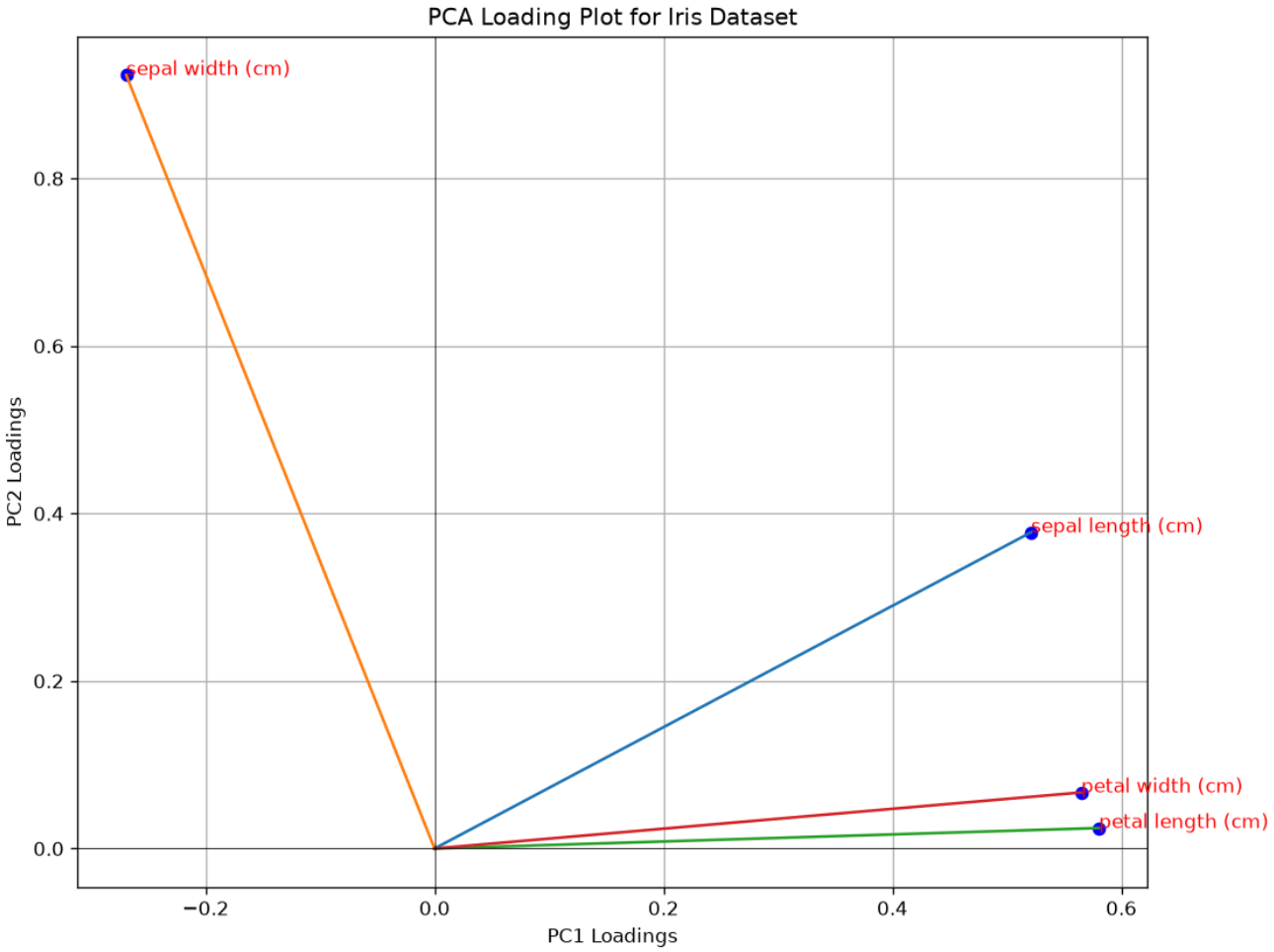
This approach can guide us in choosing the right number of dimensions to keep, selecting a budget of the variance we wish to be able to retain.

Interpretation of the Principal Components: Loading Plots

When we transform data with PCA, the new variables have a less direct interpretation. Indeed, if a given variable is a linear combination of other variables, it is not straightforward to assign a meaning to each component.

However, we know that the first components are the ones which contain most of the variance. Hence, inspecting the weights that each of these components given to the original features can give some insights into which of the original features are more relevant. In this context, **the weights of the principal components are also called loadings**. Loadings with **large absolute values are more influential in determining the value of the principal components**.

In practice, we can assess the relevance of features with a **loading plot**, which represents each original variable as a 2D point of coordinates given by the loadings corresponding to the first two principal components. Let us show the loading plot for our example:



In a loading plot:

- variables that cluster together are positively correlated, while variables on opposite sides may be negatively correlated;
- variables which are further from the origin have higher loadings, so they are more influential in the computation of the PCA.

A loading plot can also be useful to perform feature selection. For instance, from the plot above all variables seem to be influential, but `petal width` and `petal length` are correlated. We could think of using as a subset of variables `sepal width`, `sepal length` and one of the other two (e.g., `petal length`).

Laboratory: Applications of PCA

PCA has several applications in data analysis. We will summarize the main ones in the following.

Dimensionality Reduction and Visualization

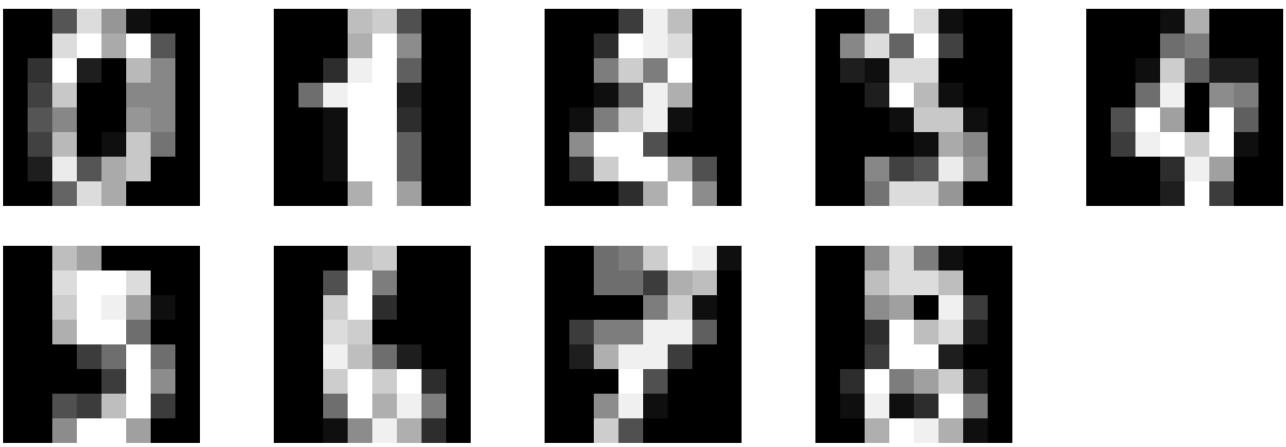
We have seen that data can be interpreted as a set of D-dimensional points. When $D = 2$, it is often useful to visualize the data through a scatterplot. This allows us to see how the data distributes in the space. When $D > 2$, we can show a series a scatterplot (a pairplot or scattermatrix). However, when D is very large, it is usually unfeasible to visualize data with all possible scatterplots.

In these cases, it can be useful to transform the data with PCA and then visualize the data points as 2D points in the space identified by the first two principal components.

We will show an example on the multidimensional dataset **DIGITS**. The dataset contains small images of resolution $8 \times 8 pixels$ representing handwritten digits from 0 to 9. We can see the dataset as containing 64 variables. Each variable indicates the pixel value at a specific location.

```
from sklearn.datasets import load_digits
digits=load_digits()

plt.figure(figsize=(12,4))
for i in range(9):
    plt.subplot(2,5,i+1)
    plt.imshow(digits.data[i].reshape((8,8)),cmap='gray')
    plt.axis('off')
plt.show()
```

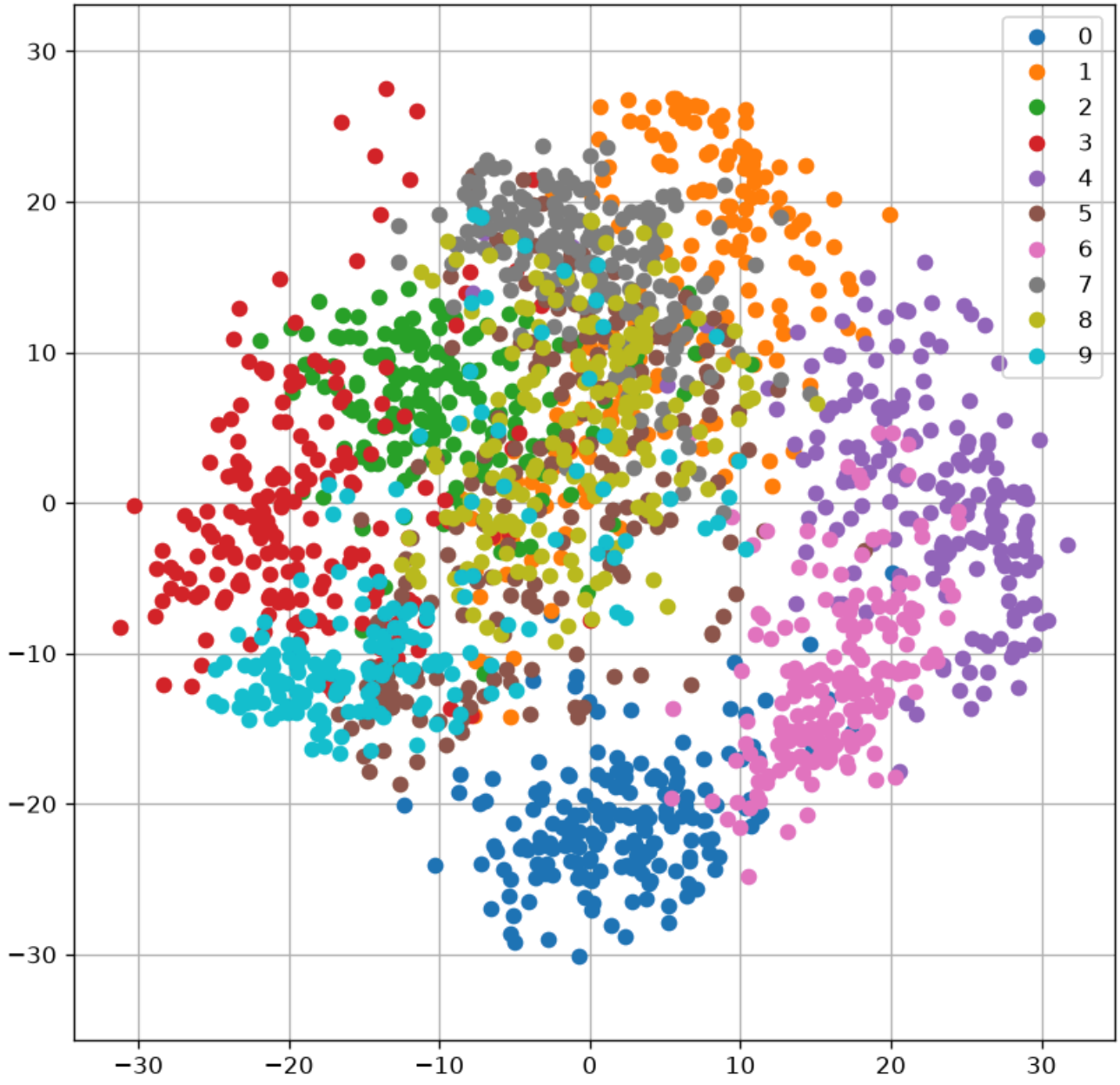


We can visualize the data with PCA by first projecting the data to $M = 2$ principal components, then plotting the transformed data as 2D points.

```
pca=PCA(n_components=2)
pca.fit(digits.data)
Y=pca.transform(digits.data)

plt.figure(figsize=(8,8))
legend = []
for c in np.unique(digits.target):
    plt.plot(Y[digits.target==c,0],Y[digits.target==c,1],'o')
    legend.append(c)

plt.legend(legend)
plt.grid()
plt.axis('equal')
plt.show()
```



Data Decorrelation - Principal Component Regression

PCA is also useful when we need to fit a linear regressor and we are in the presence of a dataset with multi-collinearity.

Let us consider the `auto_mpg` dataset:

```
from scipy.stats import zscore
from statsmodels.formula.api import ols
import pandas as pd
column_names = ["mpg", "cylinders", "displacement", "horsepower", "weight",
        "acceleration", "model_year", "origin", "car_name"]
data = pd.read_csv(DATA + "auto_mpg.data", sep=r"\s+", names=column_names,
        na_values="?")
data = data[["displacement", "cylinders", "horsepower", "weight", "acceleration",
        "model_year", "origin", "mpg"]]
data
```

	displacement	cylinders	horsepower	weight	acceleration	model_year	origin	mpg
0	307.0	8	130.0	3504	12.0	70	1	18.0
1	350.0	8	165.0	3693	11.5	70	1	15.0
2	318.0	8	150.0	3436	11.0	70	1	18.0
3	304.0	8	150.0	3433	12.0	70	1	16.0
4	302.0	8	140.0	3449	10.5	70	1	17.0
...	...	...	...	...	...	...	...	...
393	140.0	4	86.0	2790	15.6	82	1	27.0
394	97.0	4	52.0	2130	24.6	82	2	44.0
395	135.0	4	84.0	2295	11.6	82	1	32.0
396	120.0	4	79.0	2625	18.6	82	1	28.0
397	119.0	4	82.0	2720	19.4	82	1	31.0

398 rows × 8 columns

If we try to fit a linear regressor with all variables, we will find multicollinearity:

```
from statsmodels.formula.api import ols
ols("mpg ~ displacement + cylinders + horsepower + weight + acceleration + model_year
        + origin", data).fit().summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-17.2184	4.644	-3.707	0.000	-26.350	-8.087
displacement	0.0199	0.008	2.647	0.008	0.005	0.035
cylinders	-0.4934	0.323	-1.526	0.128	-1.129	0.142
horsepower	-0.0170	0.014	-1.230	0.220	-0.044	0.010
weight	-0.0065	0.001	-9.929	0.000	-0.008	-0.005
acceleration	0.0806	0.099	0.815	0.415	-0.114	0.275
model_year	0.7508	0.051	14.729	0.000	0.651	0.851
origin	1.4261	0.278	5.127	0.000	0.879	1.973

Indeed, some variables have a large p-value, probably because they are correlated with the others.

To perform Principal Component Regression, we can first compute PCA **on the independent variables**, then fit a linear regressor. We will choose $M = 4$ principal components:

```
pca = PCA(n_components=4)
dd = data.dropna()
pca.fit(dd.drop('mpg', axis=1))
zz = pca.transform(dd.drop('mpg', axis=1))

data2 = pd.DataFrame({
    f"z{i+1}": zz[:,i]
    for i in range(zz.shape[1])
})

data2=data2.join(dd['mpg'])

#print("Cumulative explained variance:", pca.explained_variance_ratio_.cumsum())

# Get explained variance ratios
explained_variance_ratio = pca.explained_variance_ratio_

# Calculate cumulative explained variance
cumulative_explained_variance = np.cumsum(explained_variance_ratio)

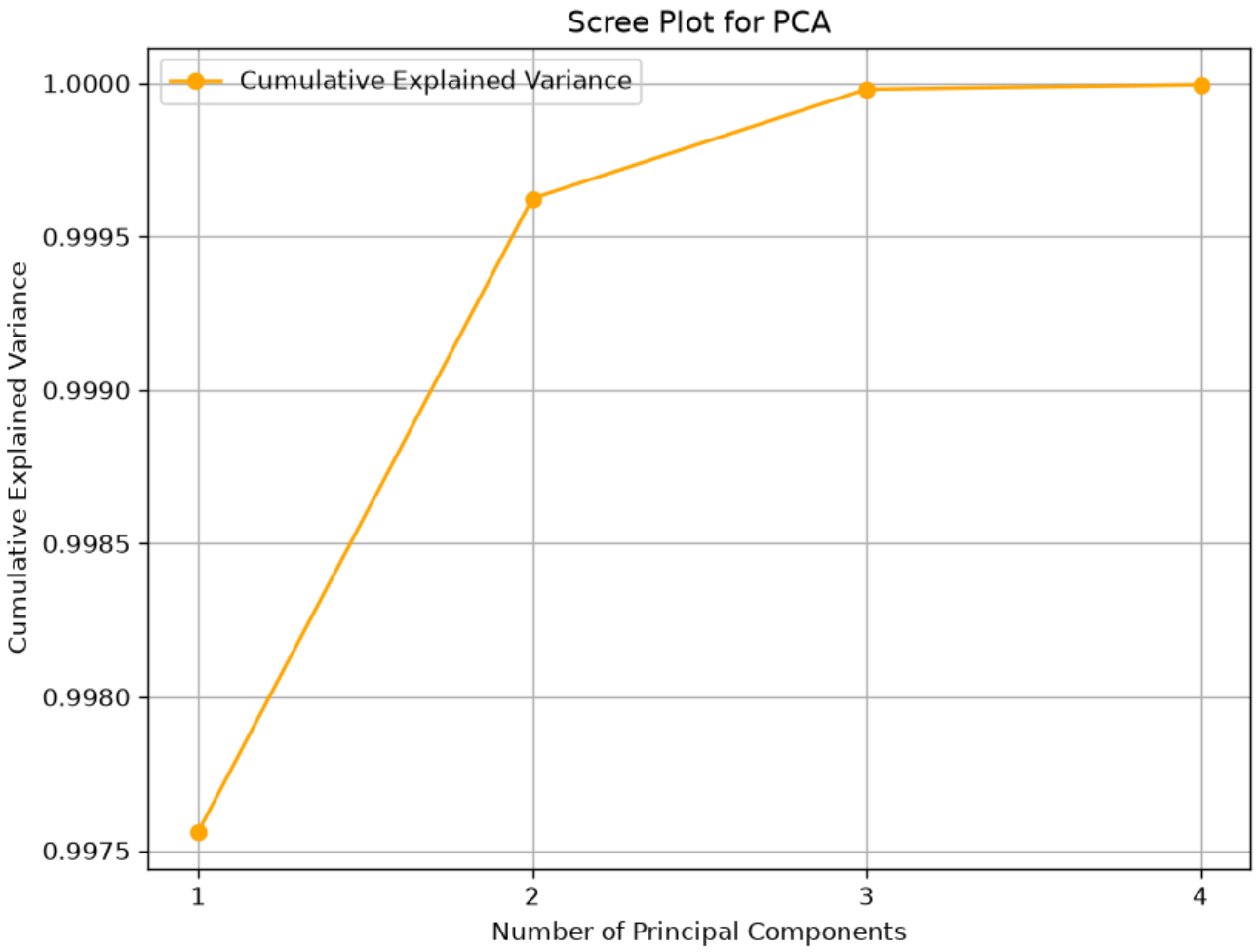
# Plot scree plot with both bar plot for variance and line plot for cumulative
        explained variance
plt.figure(figsize=(8, 6))

# Bar plot for explained variance ratio
plt.title('Scree Plot for PCA')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.xticks(np.arange(1, len(explained_variance_ratio) + 1))

# Line plot for cumulative explained variance
plt.plot(range(1, len(cumulative_explained_variance) + 1),
        cumulative_explained_variance, marker='o', color='orange', label='Cumulative
        Explained Variance')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')

# Show legend
plt.legend()
plt.grid()
plt.show()

ols("mpg ~ z1 + z2 + z3 + z4", data2).fit().summary()
```


Scree Plot for PCA
Cumulative Explained Variance
1.0000
0.9995
0.9990
0.9985
0.9980
0.9975
Cumulative Explained Variance
1
2
3
4
Number of Principal Components

OLS Regression Results

Dep. Variable:	mpg	R-squared:	0.492
Model:	OLS	Adj. R-squared:	0.487
Method:	Least Squares	F-statistic:	92.20
Date:	Sat, 26 Sep 2026	Prob (F-statistic):	9.24e-55
Time:	16:23:13	Log-Likelihood:	-1207.7
No. Observations:	386	AIC:	2425.
Df Residuals:	381	BIC:	2445.
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	23.3607	0.283	82.480	0.000	22.804	23.918
z1	-0.0051	0.000	-15.346	0.000	-0.006	-0.004
z2	-0.0320	0.007	-4.373	0.000	-0.046	-0.018
z3	-0.0552	0.018	-3.138	0.002	-0.090	-0.021
z4	0.8815	0.086	10.234	0.000	0.712	1.051

Omnibus:	3.702	Durbin-Watson:	1.125
Prob(Omnibus):	0.157	Jarque-Bera (JB):	3.435
Skew:	0.214	Prob(JB):	0.179
Kurtosis:	3.174	Cond. No.	860.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
While the result is not as interpretable as it was before, this technique may be useful in the case of predictive analysis.

Data Compression

Now let's see a simple example of data compression using PCA. In particular, we will consider the case of compressing images. An image can be seen as high-dimensional data, where the number of dimensions is equal to the number of pixels. For example, an RGB image of size 640×480 pixels has $3 \cdot 640 \cdot 480 = 921600$ dimensions.

We expect that there will be redundant information in all these dimensions. One method to compress images is to divide them into fixed-size blocks (e.g., 8×8). Each of these blocks will be an element belonging to a population (the population of 8×8 blocks of the image).

Assuming that the information in the blocks is highly correlated, we can try to compress it by applying PCA to the sample of blocks extracted from our image and choosing only a few principal components to represent the content of the blocks.

We will consider the following example image:

```
from sklearn.datasets import load_sample_image
flower = load_sample_image('flower.jpg')
print("Image Size:",flower.shape)
print("Number of dimensions:",np.prod(flower.shape))
plt.imshow(flower)
plt.axis('off')
plt.show()
```

```
Image Size: (427, 640, 3)
Number of dimensions: 819840
```



We will divide the image into RGB blocks of size $8 \times 8 \times 3$ (these are 8×8 RGB images). These will look as follows:

```
tiles = np.split(flower.ravel(),4270)
tiles=np.stack(tiles)
tiles.shape

plt.figure(figsize=(12,8))
plt.subplot(141)
plt.imshow(tiles[1855].reshape(8,8,3))
plt.axis('off')
plt.subplot(142)
plt.imshow(tiles[2900].reshape(8,8,3))
plt.axis('off')
plt.subplot(143)
plt.imshow(tiles[1856].reshape(8,8,3))
plt.axis('off')
plt.subplot(144)
plt.imshow(tiles[960].reshape(8,8,3))
plt.axis('off')
plt.show()
```



If we compute the PCA of all blocks, we will obtain the following cumulative variance ratio vector:

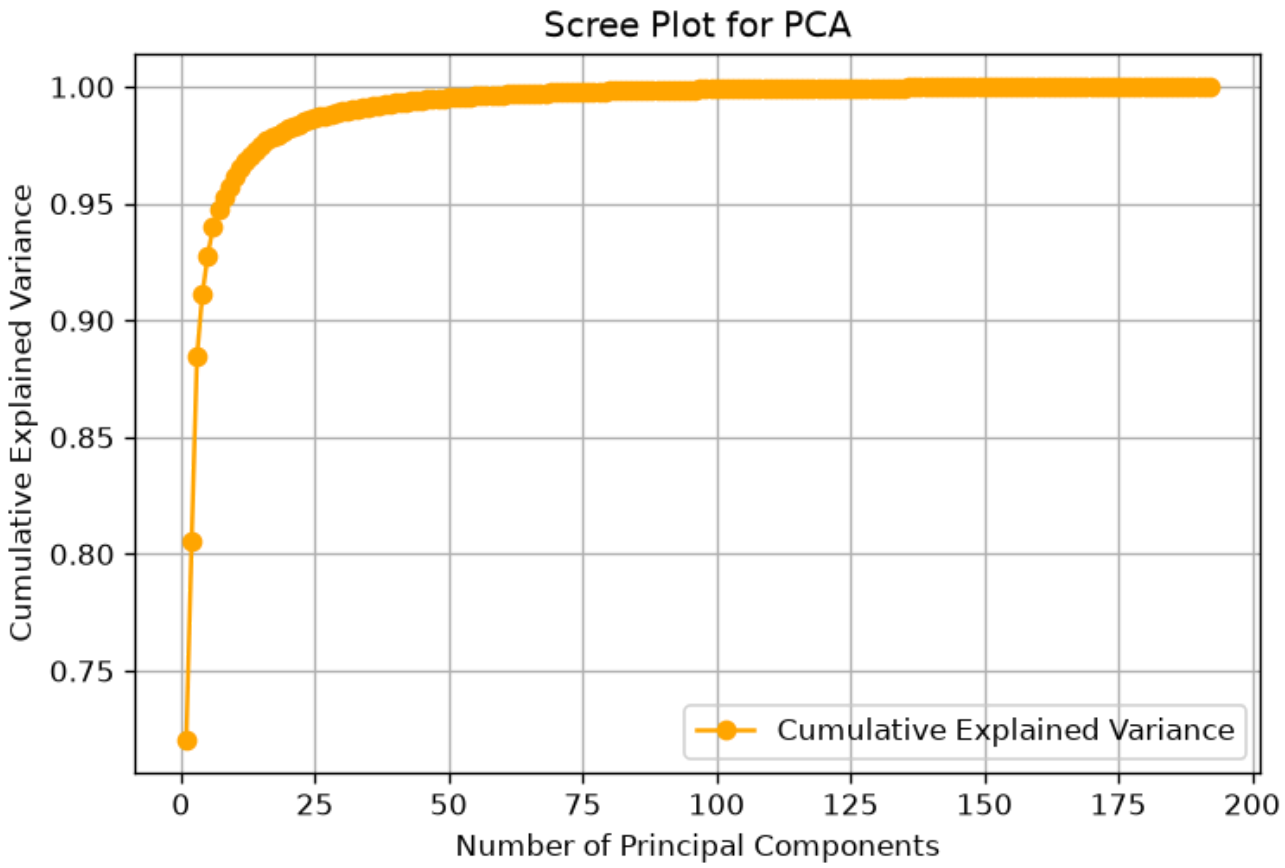
```
pca=PCA()
pca.fit(tiles)
np.cumsum(pca.explained_variance_ratio_)[:32]
```

```
array([0.72033381, 0.80549514, 0.88473859, 0.91106294, 0.92773699,
       0.94002703, 0.94742983, 0.95262255, 0.95724079, 0.96138202,
       0.96493482, 0.96813033, 0.97074766, 0.97304418, 0.97502489,
       0.9767685 , 0.97827391, 0.97961606, 0.98080081, 0.98195036,
       0.9829869 , 0.98399274, 0.98491241, 0.98570779, 0.98649071,
       0.98719462, 0.98779926, 0.98835149, 0.98888528, 0.98937603,
       0.98983658, 0.99027875])
```

```
# Bar plot for explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_explained_variance = explained_variance_ratio.cumsum()
plt.title('Scree Plot for PCA')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
#plt.xticks(np.arange(1, len(explained_variance_ratio) + 1))

# Line plot for cumulative explained variance
plt.plot(range(1, len(cumulative_explained_variance) + 1),
         cumulative_explained_variance, marker='o', color='orange', label='Cumulative
         Explained Variance')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')

# Show legend
plt.legend()
plt.grid()
plt.show()
```



As we can see, truncating at the first component allows us to retain about 72 of the information, truncating at the second allows us to retain about 80, and so on, up to 32 components, which allow us to retain about 99 of the information. Now let's see how to compress and reconstruct the image. We will choose the first 32 components, preserving 99 of the information.

If we do so, and then project the tiles to the compressed space each tile will be represented by only 32 numbers. This will lead to the following savings in space:

```
pca = PCA(n_components=32)
pca.fit(tiles)
compressed_tiles = pca.transform(tiles)
compressed_tiles.shape

print("Space saved in the compression: {:0.2f}%".format(100-
        100*compressed_tiles.size/tiles.size))
```

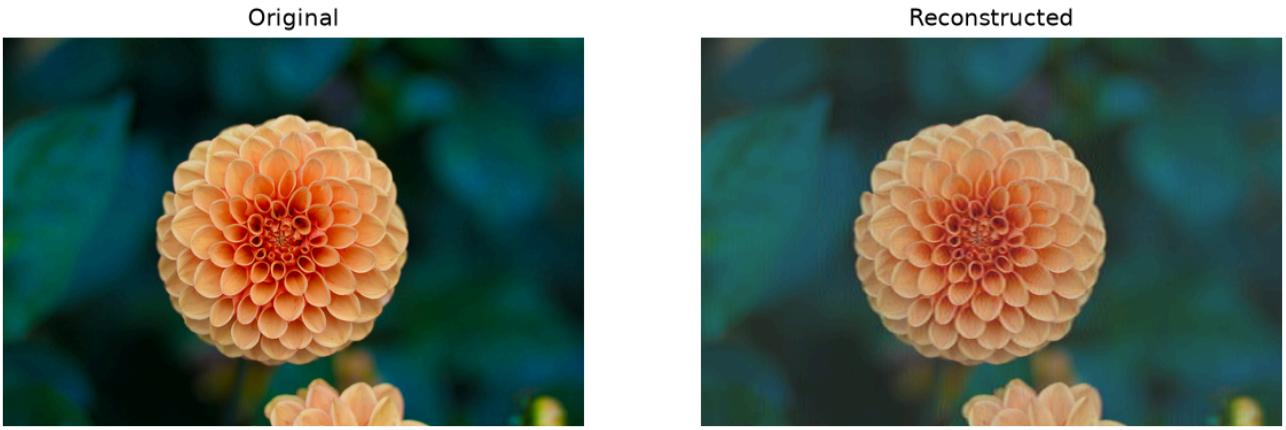
```
Space saved in the compression: 83.33%
```

We can reconstruct the original image by applying the inverse PCA transformation to the compressed patches. The result will be as follows:

```
reconstructed_tiles = pca.inverse_transform(compressed_tiles)
reconstructed_tiles.shape

reconstructed_tiles = (reconstructed_tiles-reconstructed_tiles.min())\
            /(reconstructed_tiles.max()-reconstructed_tiles.min())*255
reconstructed_tiles=reconstructed_tiles.astype('uint8')

reconstructed_flower=reconstructed_tiles.ravel().reshape(flower.shape)
plt.figure(figsize=(12,8))
plt.subplot(1,2,1)
plt.title('Original')
plt.imshow(flower)
plt.axis('off')
plt.subplot(1,2,2)
plt.title('Reconstructed')
plt.imshow(reconstructed_flower)
plt.axis('off')
plt.show()
```



The plot below shows how the reconstruction quality increases when more components are used:

```
def compress(n_components):
    pca = PCA(n_components=n_components)
    pca.fit(tiles)
    compressed_tiles = pca.transform(tiles)
    compressed_tiles.shape

    saved_sapce = 100-100*compressed_tiles.size/tiles.size

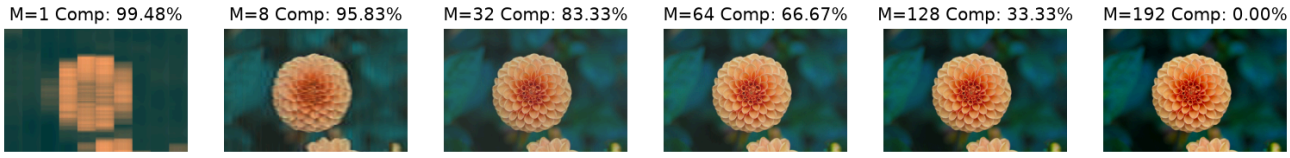
    reconstructed_tiles = pca.inverse_transform(compressed_tiles)
    reconstructed_tiles.shape

    reconstructed_tiles = (reconstructed_tiles-reconstructed_tiles.min())\
                /(reconstructed_tiles.max()-reconstructed_tiles.min())*255
    reconstructed_tiles=reconstructed_tiles.astype('uint8')

    reconstructed_flower=reconstructed_tiles.ravel().reshape(flower.shape)

    return reconstructed_flower, saved_sapce

plt.figure(figsize=(16,4))
for i,c in enumerate([1,8,32,64,128,192]):
    plt.subplot(1,6,i+1)
    reconstructed, saved_space = compress(c)
    plt.imshow(reconstructed)
    plt.axis('off')
    plt.title(f"M={c} Comp: {saved_space:0.2f}%")
```



Eigenfaces

When PCA is applied to a dataset of faces, the **Principal Components (the eigenvectors)** are no longer abstract directions; they can be visualized as "ghostly" faces themselves, known as **"Eigenfaces"**.

Each Eigenface represents a fundamental "component" or "feature" of human faces in the dataset (e.g., "lighting from the left," "presence of a nose," "smiling vs. frowning").

The Goal

1. **Load** a dataset of 400 human faces. Each image is 64x64, meaning we have **4,096 features** (pixels).
2. **Fit** PCA to the dataset to find the "Eigenfaces."
3. **Visualize** these new "features."

4. **Compress** an image by reducing it from 4,096 dimensions to just 20.
5. **Reconstruct** the face from the compressed 20-component version to see how much information we retained.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_olivetti_faces
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 1. Load the faces dataset
# 400 images, each 64x64 (4096 features)
faces = fetch_olivetti_faces()
X = faces.data
y = faces.target

print(f"Faces dataset shape: {X.shape}")

# 2. Fit PCA
# We'll ask for 20 components
n_components = 20
# whiten=True scales the components, often better for visualization
pca = PCA(n_components=n_components, whiten=True, random_state=42)
pca.fit(X)

# The "Eigenfaces" are just the principal components (the eigenvectors)
eigenfaces = pca.components_
# The "average face" that PCA centers the data on
mean_face = pca.mean_

print(f"Eigenfaces (components) shape: {eigenfaces.shape}")
```

```
Faces dataset shape: (400, 4096)
Eigenfaces (components) shape: (20, 4096)
```

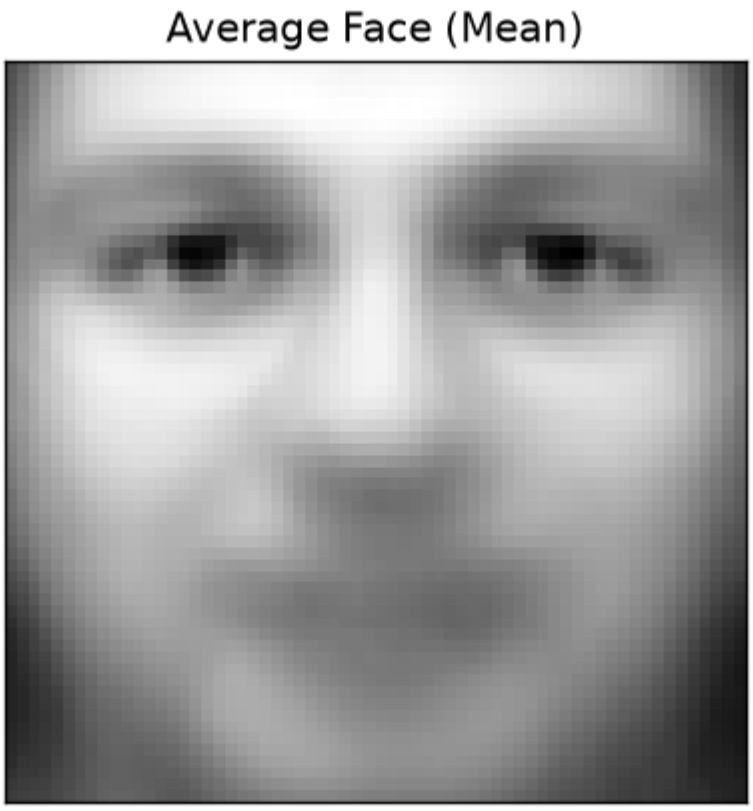
Visualizing the Principal Components (Eigenfaces)

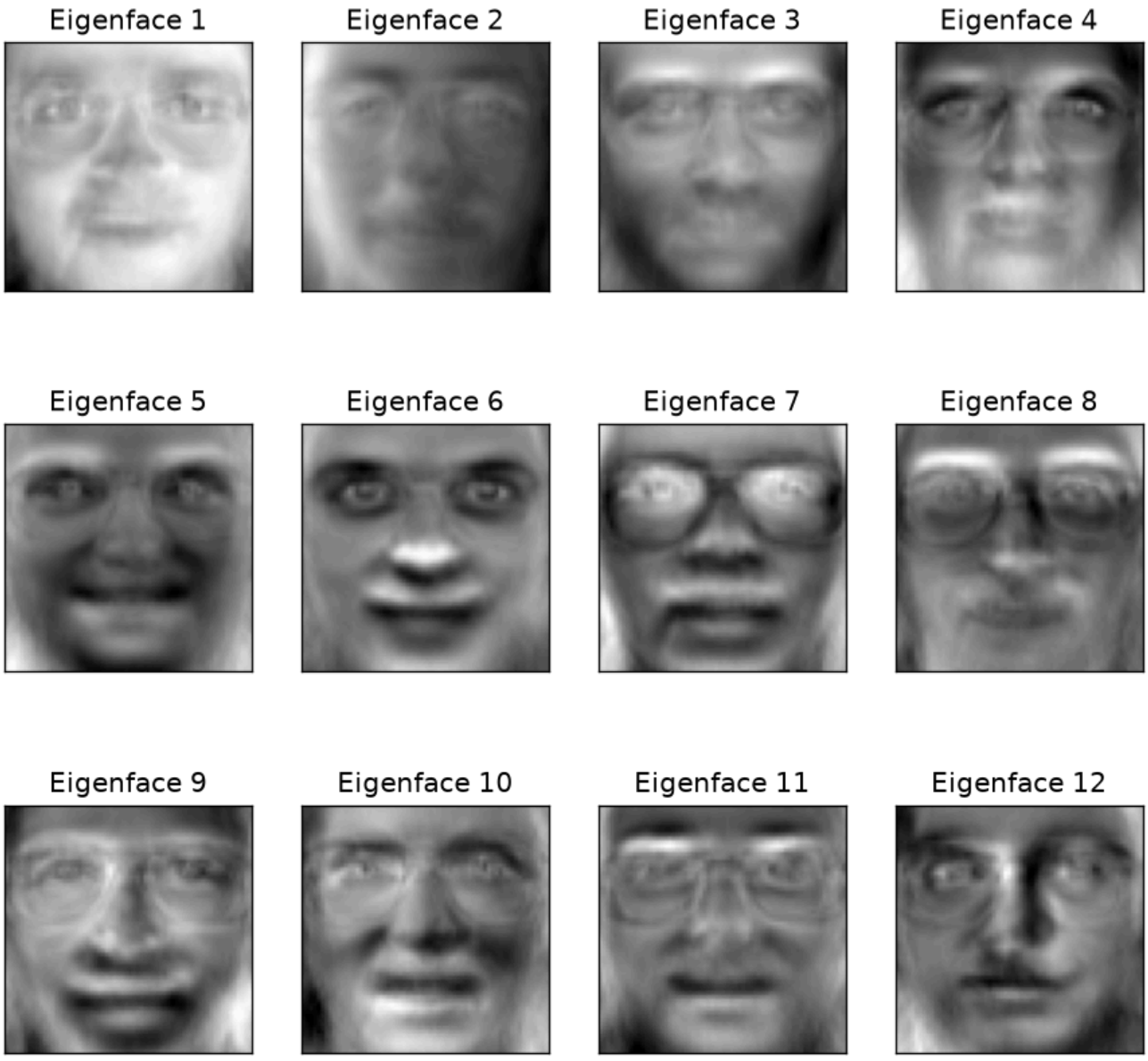
Let's plot the "average face" (the mean) and the first 12 "Eigenfaces" (PC1, PC2, etc.). These are the fundamental "building blocks" of all the faces in the dataset.

```python
def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    """Helper function to plot a gallery of portraits"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

# 1. Plot the "average face" (the mean that PCA subtracted)
plt.figure(figsize=(4,4))
plt.imshow(mean_face.reshape(64, 64), cmap=plt.cm.gray)
plt.title("Average Face (Mean)")
plt.xticks(())
plt.yticks(())
plt.show()

# 2. Plot the gallery of the first 12 eigenfaces
eigenface_titles = [f"Eigenface {i+1}" for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, 64, 64)
plt.show()
```





Interpretation

As you can see, the “Eigenfaces” are not faces, but “ghostly” components.

- **PC1 (“Eigenface 1”)** captures the main component of variance: the direction of the lighting (light from the left vs. light from the right).
- **Other components** learn to represent other key features: “wearing glasses vs. not,” “smiling vs. frowning,” “wide nose vs. narrow nose,” etc.

Data Compression and Reconstruction

Now for the magic. We can take any face and “compress” it by describing it as a *combination* of these eigenfaces.

We will take one test face and show its reconstruction using just 5, 10, 15 and 20 components.

```
# Take one face from the dataset
test_face = X[0]

# 1. Transform the face into its "component weights"
# This "compresses" the 4096-pixel vector into a 20-number vector
face_weights = pca.transform(test_face.reshape(1, -1))

# 2. Reconstruct the face using different numbers of components
n_comps_to_try = [5, 10, 15, 20]
reconstructed_faces = []

for n in n_comps_to_try:
    # Get the weights for just the first 'n' components
    weights_n = face_weights[0, :n]

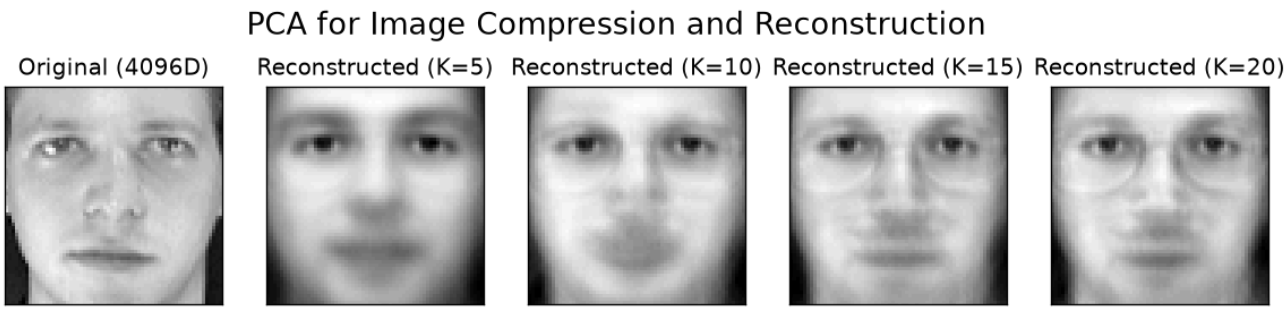
    # Get the "eigenfaces" (components) for just the first 'n'
    components_n = pca.components_[:n, :]

    # "Rebuild" the face by summing the (weight * eigenface)
    # We must also add back the "average face" we subtracted
    reconstructed = np.dot(weights_n, components_n) + pca.mean_
    reconstructed_faces.append(reconstructed)

# --- Plot the results ---
plt.figure(figsize=(12, 2.8))
plt.subplot(1, 5, 1)
plt.imshow(test_face.reshape(64, 64), cmap=plt.cm.gray)
plt.title("Original (4096D)")
plt.xticks(())
plt.yticks(())

for i, face in enumerate(reconstructed_faces):
    plt.subplot(1, 5, i + 2)
    plt.imshow(face.reshape(64, 64), cmap=plt.cm.gray)
    plt.title(f"Reconstructed (K={n_comps_to_try[i]})")
    plt.xticks(())
    plt.yticks(())

plt.suptitle("PCA for Image Compression and Reconstruction", fontsize=16)
plt.show()
```



Are These “Eigenface” Features Actually Better?

So far, we’ve seen that PCA can *compress* a 4096-pixel image into a 20-number vector, and then reconstruct it. This is great for storage.

But is this new, 20-dimensional vector a *better feature set* for a machine learning model?

Let’s run a “bake-off” to find out. We will build a **face recognition** classifier (using KNN) and test it on two different feature sets:

1. **Baseline Model:** KNN trained on the **raw 4096-pixel** vectors.
2. **PCA Model:** KNN trained on our **“smart” 20-component** vectors.

Our Hypothesis: The PCA model will be *faster* to train (fewer features) and *more accurate* (less noise).

```python
import numpy as np
import matplotlib.pyplot as plt
import time
from sklearn.datasets import fetch_olivetti_faces
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import classification_report, accuracy_score

# --- 1. Load the data ---
# This was the missing step. We must load the data first.
faces = fetch_olivetti_faces()
X_faces = faces.data
y_faces = faces.target
print(f"Loaded faces dataset. Shape: {X_faces.shape}")

# --- 2. Create our single Train/Test split ---
X_train, X_test, y_train, y_test = train_test_split(
    X_faces, y_faces, test_size=0.3, random_state=42, stratify=y_faces
)

print(f"Training on {X_train.shape[0]} samples, testing on {X_test.shape[0]}
        samples.")
```

```
Loaded faces dataset. Shape: (400, 4096)
Training on 280 samples, testing on 120 samples.
```

Baseline: KNN on 4096 Raw Pixels

First, our baseline. We build a pipeline that scales the 4096 pixel values and then runs the KNN classifier.

```python
# 1. Create a simple pipeline
pipe_knn_raw = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier(n_neighbors=5))
])

# 2. Fit and time the model
print("Training KNN on 4096 RAW pixel features...")
start_time = time.time()
pipe_knn_raw.fit(X_train, y_train)
end_time = time.time()

# 3. Evaluate on the test set
y_pred_raw = pipe_knn_raw.predict(X_test)
acc_raw = accuracy_score(y_test, y_pred_raw)
time_raw = end_time - start_time

print(f"  Accuracy: {acc_raw*100:.2f}%")
```

```
Training KNN on 4096 RAW pixel features...
  Accuracy: 79.17%
```

Model 2: KNN on 20 “Smart” PCA Features

Now, we’ll build a complete pipeline that does all the steps at once:

1. Scales the data.
2. Performs PCA to reduce 4096D $\rightarrow$ 20D.
3. Trains the KNN classifier on the new 20D features.

```python
# 1. Create the full PCA pipeline
pipe_knn_pca = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=20, whiten=True, random_state=42)),
    ('knn', KNeighborsClassifier(n_neighbors=5))
])

# 2. Fit and time the model
print("Training KNN on 20 PCA features...")
start_time = time.time()
pipe_knn_pca.fit(X_train, y_train)
end_time = time.time()

# 3. Evaluate on the test set
y_pred_pca = pipe_knn_pca.predict(X_test)
acc_pca = accuracy_score(y_test, y_pred_pca)
time_pca = end_time - start_time

print(f"  Accuracy: {acc_pca*100:.2f}%")
```

```
Training KNN on 20 PCA features...
  Accuracy: 83.33%
```

Conclusion

Let's look at the results.

Model	Features	Accuracy
Baseline	4096 raw pixels	79.17%
PCA Model	20 PCA components	**83.33%**

PCA didn't just *compress* the data; it **improved** it. By creating a cleaner, low-dimensional feature space, it allowed our simple KNN classifier to find the true patterns and make far more accurate predictions.

Exercises

1. **How many components?** Plot the cumulative explained variance for a dataset of your choice. How many components reach 90%? Is the scree plot's elbow in the same place?
2. **Scaling is not optional.** Run PCA with and without standardising, on data whose variables have very different units. Compare the first component's loadings and explain the difference.
3. **Read the loadings.** Produce the loading plot for the first two components and give each component a name in the language of the problem. Then state how confident you are in those names.

4. **Reconstruction.** Project a dataset onto k components and back. Plot the reconstruction error against k. Where does it stop falling quickly, and what is lost below that point?
5. **PCA before a model.** Compare a classifier on the raw features against the same classifier on the first k principal components, by cross-validation. Does reducing help, hurt, or neither — and does the answer depend on k?
6. **When PCA misleads.** Construct data where the direction of greatest variance is irrelevant to the question you care about. Show that the first component is useless, and say what you would use instead.

References

- Parts of Chapter 12 of [1]

[1] Bishop, Christopher M., and Nasser M. Nasrabadi. Pattern recognition and machine learning. Vol. 4. No. 4. New York: springer, 2006.

© 2026 Antonino Furnari · Department of Mathematics and Computer Science, University of Catania Built with Quarto

Credits and data sources