AI & Machine Learning for Genomic Data Science

Master in Genomic Data Science – Università di Pavia

Barbara Tarantino

AA 2025-2026

Course Outline

- Day 1 → Python Foundations & AI in Medicine
- Day 2 → Core Machine Learning for Genomics
- Day 3 → Deep Learning Foundations (PyTorch)
- Day 4 → Computer Vision for Medicine
- Day 5 → Large Language Models & Clinical Text

Objective: to acquire theoretical and practical bases to apply AI to genomics, clinical images, medical text.

Program - Day 1

Python Foundations & AI in Medicine

- AI in medicine and genomics (tabular, textual and image data)
- Python fundamentals (variables, control structures, functions)
- NumPy and Pandas for data management
- Pandas, Matplotlib and Seaborn for data exploration and visualization

Practical organization – Day 1

Morning (9:30 – 12:30)

- Introduction + AI in Medicine/Genomics
- Python essentials: variables, lists, loops, functions
- NumPy: Arrays and Vectorized Operations
- Python and NumPy Notebooks (Guided Tutorial)

Afternoon (14:30 – 17:00)

- Pandas: management of tabular datasets
- Visualizations with Matplotlib and Seaborn
- Notebook on Pandas, Matplotlib and Seaborn (guided tutorial)

Recap + Q&A

Section 1

Introduction to AI

Artificial Intelligence (AI)

Definition

Computer science discipline that develops systems capable of performing tasks that require human cognitive abilities (reasoning, pattern recognition, decisions).

Fundamentals

- Data (information to be analyzed)
- Mathematical algorithms (calculation rules)
- Computing power (the computer's ability to process data)

Principles

- Simulating human cognitive processes
- Learning from examples and experience
- Adapting to new data

Artificial Intelligence (AI)

Schematic Flow

Clinical/Biological Problem → Data Collection → AI Algorithm → Model → Interpretation of Results

Applications in medicine/genomics

- Diagnostic support systems
- Extracting information from reports
- Analysis of genomic datasets

Key Feature

AI is the overarching umbrella → includes ML and DL → **AI ⊃ ML ⊃ DL**

Machine Learning (ML)

Definition

A subset of AI that develops algorithms capable of learning from data without being explicitly programmed with fixed rules.

Principles

The algorithm builds a model from examples:

- **Input (X)** = training data (gene expression, clinical parameters).
- **Output (Y)** = labels or values to be predicted (e.g. tumor subtype, survival).

Machine Learning (ML)

Objective

Learning from data to:

- Supervised
- Uncovering hidden patterns (Unsupervised)
- Learn from interactions (Reinforcement).

Types of learning

- **Supervised Learning**: Labeled (X→Y).
  Example: predicting the tumor subtype by gene expression.
- **Unsupervised Learning**: without labels.
  Example: Clusters of patients based on molecular profiles.
- **Reinforcement Learning**: learning by trials/rewards.
  Example: surgical robots.

Machine Learning (ML)

Schematic Flow

- Supervised
  Data (X) + Labels (Y) → ML Algorithm → Model → Predictions on New Data
- Unsupervised
  Data (X) → ML Algorithm → Patterns/Clusters/Representations
- Reinforcement
  Agent → Environment → Actions → Rewards/Penalties → Optimized Policy

Output Types (Y)

- Classification: *Y* is a category.
  Example: healthy patient vs patient with diabetes.
- Regression: *Y* is a numeric value.
  Example: Expected blood glucose or blood pressure level.

Machine Learning (ML)

Applications in medicine/genomics

- Supervised:
  Predict response to therapy from clinical data; classify tumor subtypes by gene expression profiles.

- Unsupervised:
  Discover subgroups of patients with similar molecular profiles; size reduction to explore genomic data.

- Reinforcement:
  Optimization in surgical robotics or dynamic therapeutic strategies.

Key feature:

Requires feature engineering → the expert selects and transforms the relevant variables.

Deep Learning (DL)

Definition

A subset of ML that uses artificial neural networks with multiple layers (deep), capable of learning hierarchical representations of data.

Principles

- Each layer of the network processes the data and builds a more abstract representation of it.
- The model automatically learns relevant features from the raw data.

Objective

Analyze complex and unstructured data (images, texts, genomic sequences).

Deep Learning (DL)

Types of neural networks

- **Feedforward Neural Networks (FNN):** per dati tabulari.
- **Convolutional Neural Networks (CNN):** for biomedical imaging.
- **Recurrent Neural Networks (RNN, LSTM, Transformer):** by sequences (genomic or textual).

Schematic Flow

- Feedforward networks (FNN)
  Tabular data (gene expression) → Network layers → Prediction
- Convolutional networks (CNN)
  Image (radiology/histology) → Convolutional filters → Representation → Prediction
- Recurrent networks (RNN, LSTM, Transformer)
  Sequence (genomics or text) → Recurrent layers → Representation → Prediction

Deep Learning (DL)

Applications in medicine/genomics

- Tabular (FNN): Predict disease risk from clinical/genomic data.
- Imaging (CNN): Detect tumors in radiology or classify histological images.
- Sequences (RNN/LSTM/Transformer): analysis of genomic sequences or textual clinical reports.

Key Feature

- The DL automatically extracts relevant features from the raw data.
- It allows you to process complex data (images, sequences, texts) without having to do manual feature engineering.

Types of Data in Medicine and Genomics

Tabular → treated with classic ML and FNN (Day 2-3).

- Numeric tables
- Example: clinical values, gene expression (patient matrices × genes).

Images → treated with CNN (Day 4).

- Radiology (X-RAY, CT, MRI), digital pathology, microscopy.
- Example: Detecting abnormalities in a biopsy.

Textual → treated with LLM/Transformer (Day 5).

- Clinical notes, electronic records, scientific abstracts.
- Example: Automatically extract diagnoses from reports.

ML vs DL: Data Types and Features

Aspect	Machine Learning (ML, Day 2)	Deep Learning (DL, Day 3-5)
Tabular data	Directly usable (clinical, gene expression).	Usable, but often not necessary complex models.
Imagery	They require manual extraction of features (e.g. textures, shapes).	CNN models learn directly from pixels.
Text	Need to transform into vectors (e.g. bag-of-words).	RNN/Transformer models learn directly from sequences.
Genome sequences	They require numerical representations constructed by the expert.	DL models learn patterns directly from nucleotide bases.

Typical workflow of an AI project

- **Definition of the problem** → E.g. distinguishing subtypes of leukemia.
- **Data collection** → Clinical, genomic, image, text.
- **Data preparation** → Cleaning, normalization, train/test split.
- **Model training** → ML or DL algorithms.
- **Performance Evaluation** → Accuracy, Sensitivity, AUC, etc.
- **Interpretation and clinical validation** → Understand "why" the model makes certain decisions.

Section 2

Python Essentials

Why Python?

- Language with high readability and low level of syntactic complexity.

- De facto standard for Data Science and AI.

- Established ecosystem of science libraries:
  - **NumPy** (numerical calculation)
  - **Pandas** (tabular data management)
  - **Matplotlib / Seaborn** (visualization)
  - **PyTorch** (deep learning)

Python tools

- The Python ecosystem is used through interactive environments that combine **code**, **explanatory text,** and **output** in the same workspace.

- These environments, called **notebooks**, are ideal for step-by-step analysis, visualizations, and educational activities.

- **Google Colab** offers a cloud version of Python notebooks:
  it allows you to run code directly online, without local installations, with built-in saving and sharing in Google Drive.

Programming language

- A programming language allows you to formalize procedures in a form that can be executed by a computer.
- Python adopts a linear and readable syntax, suitable for the introduction to programming.

Example:

```
print("Hello, world!")
```

Output:

```
Hello, world!
```

Objects in Python

- In Python everything is an object: numbers, strings, lists, functions, complex structures.

- An object is an instance of a class:
  - The class defines structure (what data it contains)
  - behaviour (what operations it can perform)

- Each item has:
  - Attributes → properties that describe the internal state.
  - Methods → functions embedded in the class, applicable to the object.

Attributes and Methods

- Attributes: Internal characteristics of the object.
  - Syntax: `object.attribute`
  - They do not require brackets.

- Methods: functions embedded in the class, invoked by the object.
  - Syntax: `object.method(arguments)`
  - They can return information or change the status.

Example: str object

- "TP53" is an object of the `str class`
- The `str class` defines attributes and methods that are common to all strings.

```
gene="TP53"
gene.isalpha# Attribute: Returns False (also numbers)
gene.lower() # Method: returns "tp53"
```

- The object has attributes that describe its properties (`isalpha`)
- It has methods that allow operations on the content (`lower()`).
- All strings in Python share these attributes and methods.

Variables

- A variable is a reference to an object, a symbolic container that stores a value.
- Syntax: name = value.

```
gene = "TP53"        # stringa
expression = 2.5# decimal number
patients = 50        # intero
```

Variables are essential for structuring data and intermediate results.

Data types

- `int` → integers (`42`)
- `Float` → Real Numbers (`3.14`)
- `str` → stringhe (`"BRCA1"`)
- `bool` → logical values (`True, False`)
- `list` → ordered and editable sequences `[1, 2, 3]`
- `Tuples` → ordered and immutable sequences `(1, 2, 3)`
- `Set` → unordered sets of unique items `{1, 2, 3}`
- `dict` → key collections → value `{"gene": "TP53"}`

Data Types = basic conceptual structure in Python.

Casting (type conversion)

- Allows you to transform one type of data into another.

```
x="42"#string
int(x) + 1 # 43
float(x) + 0.5# 42.5
```

Useful for managing textual data imported from files.

Arithmetic operators

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Full Division:
- Module: %
- Power: **

```
5**2  # 25
10%3  # 1
```

Type operations

- Numbers (int, float)

```
a, b = 10, 3
A + B # 13
a/b #3.33
```

- Strings (str)

```
gene="TP53"
gene + "_mut"  # "TP53_mut"
gene[0:2] # "TP"
```

Booleans and logical operators

- Logical Type: `True, False.`
- Operatori: `==, !=, <, >, <=, >=, and, or, not.`

```
age = 18
age >= 18   # True
age < 18    # False

(age >= 18) and (age < 65) # True if both conditions are true
(age >= 18) or (age > 65) # True if at least one condition is true
not (age >= 18) # Reverses the value → False
```

Comments

- Comments describe the code and are not executed.
- Improve readability and documentation.
- Syntax: # comment.

```
# Gene Name
print("TP53") # TP53

# Difference in expression
print(8.5 - 3.2) # 5.3
```

Input/Output in Python

- `print()` → show results on the screen
- `input()` → enter data manually (user input simulation)
- `f-string` → formatted string, variables between {} are automatically replaced by their value.

```
gene = input("Enter the name of a gene: ") # |
print(f"Analysis started on gene {gene}.")
```

Suitable for simulating the entry of clinical or biological data (e.g. gene name to be analyzed).

Lists

- Ordered collections enclosed in square brackets [ ].
- Items are indexed starting at 0.

```
genes = ["TP53", "BRCA1", "EGFR"]
print(genes[0])   # "TP53"
print(len(genes)) # 3
```

Useful for representing sets of biological values (e.g. genes, samples).

List operations

```
genes = ["TP53", "BRCA1"]

genes.append("EGFR") # add
print(genes)           # ["TP53", "BRCA1", "EGFR"]

genes.remove("BRCA1")  # rimuovere
print(genes)           # ["TP53", "EGFR"]
```

Lists = dynamic structures for data collections.

Slicing

- Allows the extraction of subsets from lists or strings.
- Syntax: list[start:end] → start included, end excluded.

```
genes = ["TP53","BRCA1","EGFR","MYC","KRAS"]
genes[0:2]   # ["TP53", "BRCA1"]
genes[2:]    # ["EGFR", "MYC", "KRAS"]
genes[-1]    # "KRAS“
```

Visual Schema:
```
["TP53"(0), "BRCA1"(1), "EGFR"(2), "MYC"(3), "KRAS"(4)]
```

Critical for manipulating high-dimensional datasets.

List comprehension

- Compact creation of new lists from an iterable
- Useful for numerical transformations on data

```
# Normalized expression values to log2
values = [10, 100, 1000]
log2_values = [np.log2(x) for x in values]
print(log2_values)  # [3.32, 6.64, 9.97]
```

Suitable for rapidly transforming biological data sets.

Tuple

- Ordered sequences of elements, enclosed in ()
- Immutable → cannot be changed after creation

```
# Genomic coordinates (chromosome, start, end)
region = ("Chr17", 43044295, 43125482)
print(region[0]) # "chr17"
print(region[1:])  # (43044295, 43125482)
```

Suitable for representing fixed information (e.g. position of a gene on the genome).

Set

- Sets of unique elements, enclosed in {}
- Unsorted, duplicates are automatically deleted

```
# List of genes expressed in a sample
genes = {"TP53", "BRCA1", "BRCA1", "MYC"}
print(genes)   # {"TP53", "BRCA1", "MYC"}
```

Suitable for representing sets of characteristics without duplicates (e.g. genes unique in a sample).

Dictionaries

- Data structures based on key → value pairs, enclosed in { }.

```
patient = {
    "id": "P1",
    "diagnosis": "Breast Cancer",
    "age": 32,
    "TP53_expression": 2.5
}

print(patient["age"])   # 32
print(patient["TP53_expression"])   # 2.5
```

Suitable for representing complex information (e.g. patient metadata).

Dictionary operations

```
patient = {"id": "P1", "diagnosis": "AML"}

patient.keys()          # dict_keys(['id', 'diagnosis'])
patient.values()        # dict_values(['P1', 'AML'])

patient["diagnosis"] = "ALL" # update value
print(patient)                  # {'id': 'P1', 'diagnosis': 'ALL'}
```

High flexibility for heterogeneous clinical/biological data.

Control structures: if/else

- They allow conditional execution.

```
age = 70
if age >= 65:
    print("Elder")
else:
    print("Adult")

# Elderly
```

Loops

- They iterate after sequences.

```
genes = ["TP53","BRCA1","EGFR"]

for g in genes:
    print(g)
```

```
#TP53
#BRCA1
#EGFR
```

Essential for analyzing datasets line by line.

While loops

- Conditional iteration.

```
x = 0
while x < 3:
    print(x)
    x += 1

# 0
# 1
# 2
```

Functions

- A function is a block of self-contained code that performs a defined task.
- It can receive input data (arguments) and return a result (output).
- Syntax:

```
def somma(arg1, arg2):
    Result = arg1 + arg2
    return result
```

- Built-in functions: already available in Python (e.g. len(), print(), type()).
- User-defined functions: created for specific tasks.

```
len([1,2,3]) # built-in function → 3
```

Built-in functions

- Some ready-made tools:

```
values = [5, 7, 9]


len(values) # length → 3
sum(values) # somma → 21
max(values) # maximum → 9
min(values) # minimum → 5
type(values)  # tipo → <class 'list'>
```

Functions vs Methods vs Attributes

- Functions: Autonomous blocks that receive objects as arguments

```
len([1,2,3]) # 3
```

- Methods: Operations linked to an object, invoked with .

```
list = [1,2,3]
list.append(4) # [1,2,3,4]
```

- Attributes: Internal properties of an object, without parentheses

```
text="TP53"
text.isupper # True
```

Libraries

- In Python, libraries are also objects.
- A library collects functions, constants, and ready-made classes.
- They are used to extend Python without rewriting calculations or tools from scratch.
- Items are accessed with the period. (as with any object).

```
import math
math.pi# attribute → 3.1415...
math.sqrt(16) # → 4.0 function
```

Install a library

- Some libraries are already included in Python (e.g. `math`).
- Others must be installed separately.
- You use `pip`, Python's package manager.

```
pip install numpy
pip install pandas
```

- Once installed, you normally import into your code:

```
import numpy as np
```

Import a library

- Full Import

```
import math
math.sqrt(25)   # 5.0
```

- Import with aliases (more practical)

```
import numpy as np
np.mean([1, 2, 3]) # 2.0
```

- Targeted import (only what you need)

```
from math import sqrt
sqrt(36) # 6.0
```

Basic libraries: what they are used for

- `math` → basic mathematical operations (roots, logarithms, constants such as π).
- `Numpy (NP)` → advanced numerical calculations, efficient array management.
- `Pandas (PD)` → analysis and manipulation of tabular data (tables).

```
import math, numpy as np, pandas as pd
math.sqrt(9)              # 3.0
np.array([1, 2, 3])       # array numeric
Pd. Series([10, 20, 30]) # Data Column
```

Common Mistakes

```
# NameError: Undefined variable
print(x)

# IndexError: index beyond limits
list = [1, 2, 3]
print(list[5])

# KeyError: Missing key
diz = {"a": 1}
print(says["b"])
```

Corrected versions

```
x = 10
print(x) #10

list = [1, 2, 3]
print(list[2]) # 3

diz = {"a": 1}
print(says["a"]) # 1
```

Error Handling

- When an error occurs, the program crashes.
- With `try/except` we can catch the error and avoid breaking the code.

```
try:
    age = int(input("Enter the patient's age: "))
    print("Age registered correctly:", age)
except ValueError:
    print("Error: You have to enter an integer!")
```

Example of use:

```
Enter the age of the patient: twenty
Error: You need to enter a whole number!
```

Useful for avoiding crashes when reading data from inputs or clinical files (e.g. patient age).

Section 3

Numpy

Introduction to NumPy

- Fundamental library for numerical calculation in Python.
- It provides the `ndarray structure`: multidimensional arrays.
- Advantages:
  - Efficient management of large amounts of data.
  - Vector operations much faster than Python lists.

```
pip install numpy
```

```
import numpy as np
```

Utility in biomedicine: analysis of gene expression matrices, medical images, biological signals.

NumPy Array

- NumPy arrays (1D, 2D, Nd) are Python objects
- They are created with `np.array` by passing a list or list of lists as input.
- Like all objects, they have properties and methods
  - Properties = characteristics (e.g. `.shape`)
  - Methods = actions (e.g. `.reshape()`)

```
genes = np.array([[2.1, 3.4, 1.8],
                  [1.2, 0.7, 4.5]])
print(genes.shape)   # (2, 3)
```

Suitable for describing the structure of a biomedical dataset.

Array properties

- `.shape` → dimensions (rows × columns)
- `.dtype` → data type contained
- `.ndim` → number of dimensions

```
a = np.array([[1,2,3],[4,5,6]])
print(a.shape)   # (2, 3)
print(a.dtype)   # int64
print(a.ndim) #2 → two-dimensional array
```

Suitable for describing the structure of a dataset (e.g. patients × genes).

Creating Arrays with NumPy

- So far, we've seen how to create arrays manually with np.array([...]).
- NumPy also allows you to generate them automatically with dedicated functions.
- If we want a multidimensional array (e.g. matrix), we need to specify a tuple with rows and columns → `np.zeros((m,n))`
- If we want a one-dimensional array, we only need one number → `np.zeros(n)`

Main creation functions

- `np.zeros((m,n))` → matrix of zeros (tuple `(m,n)` with `m`=rows and `n`=columns)
- `np.ones((m,n))` → matrix of one
- `np.arange(start, stop, step)` → sequence with constant pitch
- `np.linspace(start, stop, num)` → equally spaced values

Useful for initializing matrices or simulating numerical datasets (e.g. patients × genes).

Creating Arrays with NumPy

```
import numpy as np

# 2×3 matrix of zeros
print(np.zeros((2, 3)))
# [[0. 0. 0.]
#  [0. 0. 0.]]

# Sequence 0 to 8 with Step 2
print(np.arange(0, 10, 2))
# [0 2 4 6 8]

# Equally spaced values from 0 to 1
print(np.linspace(0, 1, 5))
# [0.   0.25 0.5  0.75 1. ]
```

Processing methods

- NumPy methods allow you to change the shape of arrays without changing the data.
- These tools are critical for reorganizing numerical datasets or gene/patient matrices.
  - `.reshape()` → changes the shape (rows × columns) without altering the data
  - `.ravel()` → "flattens" the array into 1D
  - `tag. T` → transposed, swap rows with columns

```
import numpy as np
# Create a 1D array with values from 0 to 11
data = np.arange(0, 12, 1)
print(data)
# [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Processing methods

```
# We turn it into a 3×4 matrix
data = data.reshape(3, 4)
print(data)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# We can also check the new shape
print(data.shape)
# (3, 4)

# Let's go back to a 1D array with ravel()
print(data.ravel())
# [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Array Concatenation and Stacking

- With NumPy we can concatenate (join multiple arrays together) in two main ways:
  - `np.vstack()` → merges vertically (adds rows)
  - `np.hstack()` → merges horizontally (adds columns)

```
import numpy as np
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[5, 6]])

# I add b as a new line below a
print(np.vstack([a, b]))
# [[1 2]
#  [3 4]
#  [5 6]]
```

Useful for adding new patients (rows) to an existing dataset.

Array Concatenation and Stacking

- `np.hstack()` instead adds new columns to the arrays,
  so it can be used to add new variables or genes to patients already present.

```
a = np.array([[1, 2],
              [3, 4]])

b = np.array([[5],
              [6]])

# I add b as a new column to the right of a
print(np.hstack([a, b]))
# [[1 2 5]
#  [3 4 6]]
```

Useful for combining data from different analyses (e.g. gene expression and methylation).

List Python vs NumPy Array

Python List

- Generic structures, heterogeneous elements.
- Not optimized for numerical calculation.

```
[1,2,3] * 2
# [1,2,3,1,2,3]
```

NumPy Array

- Homogeneous structures, numerical data.
- Arithmetic operations = vector calculations.

```
np.array([1,2,3]) * 2
# [2 4 6]
```

Vector operations

- Element-wise operations: sum, product, powers.
- Advantage: Compact writing + high performance.

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(a + b)   # [5 7 9]
print(a * b)   # [ 4 10 18]
```

Broadcasting

- A mechanism that allows operations between arrays of different shapes, automatically replicating data.
- Avoid using explicit loops.

```
a = np.array([1,2,3])
print(a + 10)
# [11 12 13]

M = e.g.array([[1,2,3],
               [4,5,6]])
print(M + a)
# [[ 2 4 6]
# [ 5 7 9]]
```

Here a (form (3,)) is replicated on each row of M (form (2,3)).

Indexing and slicing

- Allows you to access specific elements or subsets of an array.
- Syntax similar to Python lists, but more flexible (multidimensional).

```
import numpy as np
patients = np.array([[10, 12, 15],
                     [ 8,  9, 11]])
print(patients[0, 1])     # elemento riga 0, colonna 1 → 12
print(patients[:, 0]) # all rows, first column → [10 8]
print(patients[0, :])     # prima riga, tutte le colonne → [10 12 15]
```

In a dataset patients × genes:

- `patients[:, 0]` → values of all patients for gene 1
- `patients[0, :]` → profile of a patient on all genes

Logical operations and Boolean masks

- NumPy arrays support logical conditions
- A condition returns a Boolean array (`True/False`)
- We can use it to filter values that meet the condition

```
expr = np.array([2.1, 3.5, 1.9, 4.2])
print(expr > 3.0)# [False  True False  True]
print(expr[expr > 3.0])# [3.5 4.2]
```

Suitable for selecting only genes with expression above a biological threshold.

Useful features

- `np.where(cond)` → indexes that satisfy a condition

```
expr = np.array([2.1, 3.5, 1.9, 4.2])
print(np.where(expr > 3.0))
# (array([1, 3]),)
```

- `np.unique(array)` → ordered unique values

```
genes = np.array(["TP53","BRCA1","TP53","EGFR"])
print(np.unique(genes))
# ['BRCA1' 'EGFR' 'TP53’]
```

Useful features

- `np.argmax(array)` → maximum value index
- `np.argmin(array)` → minimum value index
- `np.mean(array)` → average of values
- `np.std(array)` → standard deviation

```
expr = np.array([2.1, 3.5, 1.9, 4.2])

print(np.argmax(expr))  # 3 → valore massimo (4.2)
print(np.argmin(expr)) # 2 → minimum value (1.9)
print(np.mean(expr))    # 2.925 → media
print(np.std(expr)) # 0.88 → dispersion
```

Biomedical Application

- Omics or clinical data are represented as numerical matrices.
- NumPy is the basic tool for processing them.

```
data = np.array([[2.1, 3.5],
                 [1.8, 2.9],
                 [2.3, 3.7]])

# Mean expression of the first gene
print(np.mean(data[:,0]))   # 2.07
```

Utility: A common foundation on which to build machine learning and deep learning.

Summary exercises

- All material is available in the
  ML-GDS-2025 shared folder on Google Drive
  (https://drive.google.com/drive/folders/1SUKLxqGOZOP7IQY6fgchP9BHMT-1HqxH?usp=drive_link)

- Each day of the course has its own subfolder:
  D1 – D5, containing:
  - The Slides **/** folder (theoretical presentations)
  - the **notebooks/** folder (practical exercises).

- For **Day 1**, open:
  D1 – Python Foundations & AI in Medicine / notebooks/
  where you will find:
  - **`Python_NumPy_Notebook_Tasks.ipynb`**

Summary exercises

1. Open your notebook directly from Google Drive:
    1. right-click on the file → **Open with → Google Colaboratory**
       (if it doesn't appear, choose **Connect other apps → Colaboratory**).

2. The notebook will open in a ready-made Colab environment:
    1. no local installation,
    2. Python and the core libraries already available.

3. Each cell is executed with **Shift + Enter**.

4. You can save your work directly to Drive
   or download your notebook in `.ipynb format.`

Section 4

Pandas

Introduction to Pandas

- Library for analyzing tabular data (similar to Excel/SQL, but in Python).
- NumPy-based, efficient with large datasets.
- Main Facilities:
  - Series → Single Column (1D)
  - DataFrame → full table (2D).

```
pip install pandas
import pandas as pd
```

Utility: Exploring, cleaning, merging, and synthesizing data for ML/AI.

Create a Series

- 1D structure with index (labels) and values (data).
- `Pd. Series(values, index=labels)`

```
genes = pd. Series([2.5,7.1,5.3],
                   index=["TP53","BRCA1","EGFR"])

# TP53 2.5
# BRCA1 7.1
# EGFR 5.3
# dtype: float64
```

Useful for associating values with genes or patients.

Create a DataFrame

- 2D table with labeled rows and columns
- `Pd. DataFrame(dictionary)` → keys = columns, values = lists.

```
df = pd. DataFrame({
        "Gene":["TP53","BRCA1","EGFR"],
        "Expression":[2.5,7.1,5.3],
        "Patient":[1,1,2]
        })
#     Gene    Expression    Patient
# 0 TP53 2.5 1
# 1 BRCA1 7.1 1
#2 EGFR 5.3 2
```

Exploring the DataFrame

- It is used to understand the structure and content of the data.
- Attributes describe the properties of the DataFrame.
- Methods perform operations to display or summarize data.

```
DF.SHAPE # Size (n_righe, n_colonne)
df.head() # shows the first rows of the dataset
df.info() # About Columns and Data Types
df.describe() # Descriptive statistics for numeric columns
```

Practical example:

- `df.shape` → how many observations and variables the dataset contains
- `df.info()` → useful for checking missing values and data types
- `df.describe()` → useful for a first range and outlier check

Selecting Columns and Rows

- Columns: `df["ColumnName"]`
- Rows:
  - `df.loc[label]` → for index/label
  - `df.iloc[position]` → by numerical position.

```
df["Gene"]   # Colonna Gene

#0 TP53
#1 BRCA1
#2 EGFR
# Name: Gene, dtype: object
```

Selecting Columns and Rows

```
df.loc[0] # row labeled 0

# TP53 gene
# Expression     2.5
# Patient 1

df.iloc[1] # row in position 1

# Gene          BRCA1
# Expression      7.1
# Patient 1
```

Indices in Pandas

- Each DataFrame has an index that identifies the rows.
- `set_index(column)` → sets a column as the index.
- `reset_index()` → returns the index to progressive numbers

```
df_indexed = df.set_index("Gene")
print(df_indexed)
# Patient Expression
# Gene
# TP53 2.5 1
# BRCA1         7.1        1
# EGFR          5.3        2

# Reset the Numeric Index
df_reset = df_indexed.reset_index()
```

Suitable for organizing data with biological keys (e.g. genes or patients) and then easily returning to a numerical format.

Filter rows with conditions

- Returns only rows that meet the condition.
- Syntax: `df[df["Column"] condition value]`

```
df[df["Expression"] > 5] # Expression > 5
```

```
#     Gene    Expression    Patient
# 1   BRCA1       7.1           1
# 2    EGFR       5.3           2
```

```
df[(df["Patient"]==1) & (df["Expression"]>5)]
```

```
#     Gene    Expression    Patient
# 1   BRCA1       7.1           1
```

Edit columns

- Creation: `df["New"] = ...`
- Transform: Operations on existing column.
- Delete: `df.drop(columns=["Column"])`

```
df["High"] = df["Expression"] > 5     # nuova colonna

#       Gene    Expression    Patient     High
# 0     TP53         2.5          1         False
# 1    BRCA1         7.1          1         True
# 2     EGFR         5.3          2         True

df["Expression"] = df["Expression"]*2 # transformed column
df = df.drop(columns=["High"])        # elimina colonna
```

Sorting data

- Sort rows by one or more columns
- `df.sort_values(by="Colonna", ascending=True/False)`

```
print(df.sort_values(by="Expression", ascending=False))
```

```
      Gene   Expression  Patient
1    BRCA1          7.1        1
2     EGFR          5.3        2
0 TP53 2.5 1
```

Add and remove rows

- Add: `df.loc[nuovo_indice] = [valori_colonne]`
- Remove: `df.drop(index)`

```
df.loc[3] = ["MYC", 6.7, 3]   # nuova riga

#     Gene    Expression    Patient
#0 TP53 2.5 1
# 1   BRCA1       7.1           1
# 2    EGFR       5.3           2
# 3     MYC       6.7           3

df = df.drop(0) # delete row with index 0
```

Operations and functions

- Arithmetic operations apply to the entire column.
- External functions (NumPy): `np.function(df["Column"])`
- Custom transformations: `.apply(lambda x: ...)`

```
df["Expression"] * 2
```

```
import numpy as np
np.log(df["Expression"])
```

```
df["Expression"].apply(lambda x: x**2)
```

- `lambda x: x**2` means "a function that takes a value x and returns x squared".
- It's useful when you want to apply quick logic without having to define a complete function with `def`.

GroupBy and aggregations

- `groupby(...). mean() (o .max(), .sum() ecc.)`
- Apply a single aggregate function to the specified column

```
df.groupby("Patient")["Expression"].mean()

# Patient
#       1     4.8
#       2     5.3
#       3     6.7
# Name: Expression, dtype: float64
```

Here you only get the average for each group (a single result column).

GroupBy and aggregations

- `groupby(...). agg([...])`
- Apply multiple aggregate functions to the same column or multiple columns at the same time.

```
df.groupby("Patient")["Expression"].agg(["mean","std","max"])
```

```
#           mean   std  max
# Patient
#  1         4.8   3.25  7.1
# 2 5.3 NaN 5.3
# 3 6.7 NaN 6.7
```

Missing data (NaN)

- Identify: `.isna()`
- Replace: `.fillna(value)`
- Delete: `.dropna()`

```
df["Expression"].isna()
df["Expression"].fillna(df["Expression"].mean())
df.dropna()
```

Merge

- It is used to combine two DataFrames based on a common column (key).
- Each DataFrame must contain the column used as the merge key.
- With the `on option`, the name of the column is indicated.
- With `how`, you choose *which rows to keep*:
  - `"inner"` → only holds the values present in both DataFrames
  - `"left"` → holds all the lines of the left DataFrame
  - `"right"` → holds all the rows of the right DataFrame
  - `"outer"` → holds all rows of both DataFrames, even if they don't match
- The result is a new combined DataFrame, where the columns are merged.

Merge

```
df = pd. DataFrame({
       "Gene":["TP53","BRCA1","EGFR"],
       "Expression":[2.5,7.1,5.3],
       "Patient":[1,1,2]
       })
#     Gene    Expression    Patient
#0 TP53 2.5 1
# 1   BRCA1       7.1           1
# 2    EGFR       5.3           2

patients = pd. DataFrame({
  "Patient":[1,2,3],
  "Age":[55,63,49]
})
```

```
pd.merge(df, patients, on="Patient", how="left")
```

```
 Gene  Expression  Patient  Age
0 TP53 2.5 1 55
1  BRCA1       7.1        1   55
2  EGFR        5.3        2   63
```

DataFrame Concatenation

- `pd.concat()` → merges multiple DataFrames together
- `axis=0` → adds rows (one below the other)
- `axis=1` → adds columns (next to each other)
- `ignore_index=True` → automatically renumbers rows

```
df2 = pd. DataFrame({
    "Gene": ["MYC"],
    "Expression": [4.8],
    "Patient": [2]
})

df_combined = pd.concat([df, df2], axis=0, ignore_index=True)
print(df_combined)
   Gene  Expression  Patient
0 TP53 2.5 1
1  BRCA1       7.1        1
2  EGFR        5.3        2
3 MYC 4.8 2
```

Data reading and writing

- **Objective:** Import or export datasets (e.g. `.csv` files) quickly.
- **Main functions:**
  - `pd.read_csv()` → reads a file and creates a DataFrame
  - `df.to_csv()` → salva un DataFrame su file
- **Useful topics**
  - `index_col` → column to use as an index
  - `sep` → value separator (default `,`)
  - `na_values` → symbols to interpret as missing values

```
# Reading from file
df = pd.read_csv("data.csv", index_col=0, na_values=["NA", "?"])

# Write to file
df.to_csv("results.csv", index=False)
```

It is used to exchange data with Excel or other software in tabular format.

Section 5

Matplotlib/Seaborn

Introduction to Matplotlib

- Basic Python library for scientific visualization.
- It allows flexible and customizable charts (lines, bars, scatters, histograms, boxplots).
- Object structure:
  - Figure (`fig`) = the page
  - Axes (ax) = the graph inside the page

```
pip install matplotlib
import matplotlib.pyplot as plt
```

The sample dataset

- We use a mini-dataset of patients with gene expression.
- Columns:
  - Patient → identifier (P1, P2…).
  - Diagnosis → clinical group (ALL vs AML).
  - TP53, BRCA1 → expression values (arbitrary units).

```
import pandas as pd
df = pd. DataFrame({
        "Patient": ["P1","P2","P3","P4","P5","P6"],
        "Diagnosis": ["ALL","ALL","OFTEN","OFTEN","ALL","OFTEN"],
        "TP53": [2.1, 3.5, 4.0, 3.8, 2.7, 4.4],
        "BRCA1": [7.0, 6.5, 5.1, 5.9, 7.4, 5.3]
})
```

Create your first chart

```
fig, ax = plt.subplots()
ax.plot(df["Patient"], df["TP53"])
```

- `Fig, ax = plt.subplots()` → creates the page and chart.
- `ax.plot(x,y)` → draws a line.
- All changes go through `ax methods.`

Add title and labels

```
ax.set_title("TP53 Expression")
ax.set_xlabel("Patient")
ax.set_ylabel("Expression Value")
```

- Each chart must have: clear title and labeled axes.
- Methods `set_*` = tools to add context.

Subplots multipli

```
fig, axs = plt.subplots(1, 2, figsize=(8,4))

axs[0].plot(df["Patient"], df["TP53"])
axs[0].set_title("TP53")
axs[1].plot(df["Paziente"], df["BRCA1"])
axs[1].set_title("BRCA1")
```

- `nrows, ncols` = rows and columns of charts.
- `figsize=(L,H)` = dimensions in inches.
- Each `axs[i]` is an independent graph.

Main Types of Charts (Matplotlib)

```
ax.plot(df["Patient"], df["TP53"]) # Line
ax.bar(df["Patient"], df["TP53"]) # Bars
ax.scatter(df["TP53"], df["BRCA1"]) # Points
ax.hist(df["TP53"], bins=5)                # Istogramma
ax.boxplot(df["TP53"])                     # Boxplot
```

- Same data → different views.
- Each method produces a specific chart type.

Arguments in Drawing Functions

- For sign style:
  - `color="red"` → color (line/dots/bars).
  - `linestyle="--"` → line style (continuous, dashed).
  - `marker="o"` → dot symbol.
  - `alpha=0.5` → transparency (useful with overlapping data).
  - `bins=10` (hist only) → number of classes in the histogram.

```
ax.plot(df["Paziente"], df["TP53"], color="red", marker="o")
```

Arguments in Axes Methods

- For the context of the chart:
  - `ax.set_title("...")` → title.
  - `ax.set_xlabel("...")`, `ax.set_ylabel("...")` → axis labels.
  - `ax.set_xlim(min,max)` → range asse X.
  - `ax.set_ylim(min,max)` → range asse Y.
  - `ax.grid(True)` → adds supporting grid.

```
fig, ax = plt.subplots()
ax.scatter(df["TP53"], df["BRCA1"])
ax.set_title("TP53 vs BRCA1 Expression")
ax.set_xlabel("TP53 expression")
plt.show()
```

Saving Charts

```
fig.savefig("tp53.png", dpi=300, bbox_inches="tight")
```

- File name → choose format: `.png` (slide), `.pdf/.svg` (publications).
- `DPI` = Resolution:
  - 100 → screen draft.
  - 300 → high quality for items.
- `bbox_inches="tight"` → cuts the white margins.
- `pad_inches=0.1` → (optional) adds border.

Why Seaborn?

- Matplotlib: Very flexible, but requires a lot of code.

- Seaborn:
  - work directly with `DataFrame Pandas`,
  - create statistical graphs in a few lines,
  - style already optimized for readability.

- Objective: less code, clearer graphs.

Introduction to Seaborn

- Statistical visualization library based on Matplotlib's infrastructure.
- Objective: simpler and more elegant graphics.
- Works directly with `DataFrame Pandas.`
- Often used in biomedicine for: distributions, comparisons between groups, correlations.

```
pip install seaborn
import seaborn as sns
```

Difference in approach

```
# Matplotlib
fig, ax = plt.subplots()
ax.boxplot([df["TP53"][df["Diagnosis"]=="ALL"],
            df["TP53"][df["Diagnosis"]=="AML"]])
ax.set_title("TP53 for Diagnosis")

# Seaborn
sns.boxplot(x="Diagnosi", y="TP53", data=df)
```

- With Matplotlib → more code and manual data management.
- With Seaborn, → simply indicate the columns to be compared.
- Seaborn automatically manages data, colors, and labels.

Axis-level functions

- They create a single chart.
- Return an Axes object (such as Matplotlib).
- They are used alone or inside `plt.subplots()` for more precise layouts.

```
fig, ax = plt.subplots()
sns.boxplot(x="Diagnosi", y="TP53", data=df, ax=ax)
ax.set_title("TP53 expression for diagnosis")
ax.set_ylabel("Normalized Value")
```

Common examples:

```
sns.boxplot(), sns.scatterplot(), sns.barplot(),
sns.histplot(), sns.heatmap()
```

Figure-level functions

- They create a complete figure, even with multiple automatic subgraphs.
- Return a FacetGrid (for complex layouts).
- No need for `plt.subplots()` → Seaborn handles everything.

```
# Separate histogram for each diagnosis
sns.displot(data=df, x="TP53", col="Diagnosis", bins=10)

# Bar Chart for Genes and Diagnoses
sns.catplot(data=df_long, x="Gene", y="Expr", col="Diagnosi", kind="bar")
```

Perfect for comparing subgroups (e.g. tumor types, patients, conditions).

Axis-level vs Figure-level

Type	Product Chart	Item Returned	Example	When to use it
Axis-level	Single plot	`Axes`	`sns.boxplot(x="Diagnosi", y="TP53", data=df)`	When you want to customize with ax.set_*
Figure-level	Full Figure (Multiple Subgraphs)	`FacetGrid`	`sns.displot(data=df, x="TP53", col="Diagnosis")`	When you want to compare categories or create automatic layouts

Axis = *more manual control*
Figure = *more automation and group comparison*

Main chart types (Axis-level)

Type	Function	Objective	Useful parameters
Boxplot	`sns.boxplot()`	Compare distributions between groups	`hue, palette, order`
Bars	`sns.barplot()`	Show average by category	`errorbar, hue`
Dispersion	`sns.scatterplot()`	Relationship between two variables	`hue, style, size`
Histogram	`sns.histplot()`	Distribution of a variable	`bins, kde, multiple`
Heatmap	`sns.heatmap()`	Correlation matrix	`annot, cmap, vmin/vmax`

Personalization in Seaborn

- Colors and styles to make charts more readable and informative.

- Most common parameters:
  - `hue="column"` → color by category
  - `style="column"` → symbol or line per category
  - `size="column"` → marker size
  - `palette="Set2"` → color scale
  - `alpha=0.7` → transparency

```
sns.scatterplot(data=df, x="TP53", y="BRCA1",
                hue="Diagnosis", style="Diagnosis",
                palette="Set2", alpha=0.8)
```

Used to distinguish biological groups (e.g. ALL vs AML) clearly.

Practical Comparison — Matplotlib and Seaborn

Bookshop	Advantages	When to use it
Matplotlib	Full control of axes, figures, layouts	For complex graphics or publications
Seaborn	Simplicity, integration with Pandas, automation	For exploratory analysis and group comparison

```
# Matplotlib
plt.scatter(df["TP53"], df["BRCA1"])

# Seaborn
sns.scatterplot(data=df, x="TP53", y="BRCA1", hue="Diagnosi")
```

Summary exercises

- Open the
  D1 – Python Foundations & AI folder in Medicine → notebooks/.
- Select
  `Pandas_Matplotlib_Seaborn_Notebook_Tasks.ipynb`.
- Open the file with **Google Colab**.
- Run the cells with **Shift + Enter** to view results and graphs.

Colab allows you to integrate **code, text and output** in an interactive and collaborative environment directly online.

Conclusion Day 1

We have introduced the fundamentals of Python and the main libraries for data analysis and exploration (NumPy, Pandas, Matplotlib, Seaborn).

On the next day (Day 2) we will focus on the principles of Machine Learning applied to genomics.

See you tomorrow!