Data Science and Machine Learning Lab
Lab 01 - NumPy Introduction

Politecnico di Torino

Intro

By the end of this lab, you will be able to:

- Create and manipulate **NumPy arrays**: shapes, dtypes, attributes (`ndim`, `shape`, `size`), views vs copies, slicing, masking, fancy indexing.
- Use **universal functions** (element-wise unary/binary ops), **aggregations** (`sum`, `mean`, `std`, `argmax`, ...), **sorting/argsort**, **broadcasting**, **reshaping**.
- Apply these primitives to real data: **Iris** and **MNIST**.

1 Datasets Download

You will reuse the same datasets from Lab #0, now accessed via NumPy. You are encouraged to open these files and examine their structure, noting how commas separate values and how each line represents one record.

1.1 Iris

150 flowers × 5 columns (4 numeric features + species label). Example with `wget`:

```
wget "https://archive.ics.uci.edu/ml/machine-learning-
    databases/iris/iris.data" -O iris.csv
```

1.2 MNIST

A 10k-row subset (test split) with the first column as label (0–9) and 784 pixel columns (28×28):

```
wget "https://raw.githubusercontent.com/dbdmg/data-science-
    lab/master/datasets/mnist_test.csv" -O mnist_test.csv
```

2 Warm-Up with NumPy

This first part focuses on learning how to work with NumPy arrays and their attributes.

2.1 Array creation and attributes

In this part, you will learn how to create NumPy arrays in different ways and explore their fundamental attributes.

- **Manual array creation:** Define a small two-dimensional array manually (for example, using a nested list). Inspect its key attributes: the number of dimensions (`ndim`), shape (`shape`), total number of elements (`size`), and data type (`dtype`).
- **Generating arrays with built-in functions:** Create arrays using constructors such as `np.zeros`, `np.ones`, and `np.full`. Vary their shapes and data types (for example, `float32`, `int64`) and observe how the choice of `dtype` changes the stored values and memory usage.
- **Creating sequences of values:** Explore functions for generating numeric ranges: use `np.arange` to create evenly spaced integers and `np.linspace` to create a fixed number of equally spaced points between two limits.

2.2 Universal functions, aggregations, and sorting

In this section, you will explore NumPy's core operations for element-wise computation and data aggregation. For the following tasks, create two small 2×2 arrays of floats to test your code.

- **Element-wise operations:** Perform basic arithmetic operations between your arrays, such as addition, subtraction, multiplication, and exponentiation. Compare the outputs to confirm that each operation is applied to the corresponding elements. Then, apply a few universal functions (for example, `np.exp` or `np.sqrt`) and observe how they affect all entries simultaneously.
- **Aggregations and axis behavior:** Compute the `sum`, `mean`, `std`, and `argmax` of your array both globally and along specific axes. Take a look at the shapes of the results to understand how the choice of the `axis` parameter modifies the output.
- **Sorting and ranking:** Create a small integer matrix and apply `np.sort` along rows and columns. Then use `np.argsort` to see how the result changes.

2.3 Broadcasting

Broadcasting allows you to perform operations between arrays of different shapes without explicit loops. As an exercise, perform a column-wise normalization of a matrix by subtracting the minimum and dividing by the range for each feature.

$$(D - D_{\min})/(D_{\max} - D_{\min})$$

2.4 Indexing & views vs copies

Create small matrices and complete the following exercises:

- **Slicing submatrices:** Build a 3×4 matrix containing consecutive integers (e.g., from 1 to 12). Extract the upper-left 2×2 block and then the last two columns. Verify the shapes of the resulting slices.
- **Masked arrays:** Using the same matrix, create a Boolean mask that selects all elements greater than a chosen threshold (for example, values > 6). Use this mask to assign a new value (e.g., `-1`) to those positions.
- **Views vs copies:** Slice a subset of rows from your matrix into a new variable and modify one of its elements.
- **Fancy indexing:** Select non-contiguous rows (e.g., 0 and 2) and specific columns (e.g., 1 and 3) using index lists. Display the resulting submatrix and confirm its shape.

3 Iris Dataset with NumPy

Goal: In this part of the lab, you will load the Iris dataset, extract its features and labels, and analyze it using NumPy's statistical tools.

3.1 Feature extraction and statistical analysis

In this exercise, you will work with the Iris dataset to practice basic data handling and statistical analysis using NumPy. After downloading and saving the `iris.csv` file in your working directory, load it into a NumPy array using the following command:

```
X = np.genfromtxt('iris.csv', delimiter=',', usecols=(0, 1,
    2, 3), dtype=float)
y = np.genfromtxt('iris.csv', delimiter=',', usecols=4,
    dtype=str)
```

This command reads the CSV file and stores its contents in two structured NumPy arrays, one containing numerical measurements (X) and the other containing text labels (y).

- **Building the feature and label arrays:** Once you have loaded the dataset into memory, verify the data types and shapes of both arrays.
- **Computing global statistics:** Using NumPy's aggregation functions, compute the mean and standard deviation for each of the four features across the entire dataset (sepal length, sepal width, petal length, petal width).
- **Analyzing statistics by species:** Identify the unique species present in the dataset and, for each one, create a Boolean mask that selects only the corresponding samples from `X`. Compute per-class means and standard deviations, and examine which features show the most apparent separation between species.

3.2 Feature standardization

In this exercise, you will transform the Iris feature matrix so that each column (feature) is standardized to have zero mean and unit variance. This ensures that all measurements contribute equally to subsequent analyses, regardless of their original scale.

- **Computing the standardized matrix:** Use NumPy's aggregation functions to compute the column-wise mean and standard deviation of the feature matrix `X`. Then, apply the standardization formula:

$$Z = \frac{X - \text{mean}(X)}{\text{std}(X)}$$

to create a new matrix `Z` of the same shape as `X`. Confirm that the transformation has been applied independently to each feature column.

3.3 (*) Naïve single-feature classifier

In this final exercise on the Iris dataset, you will build a simple classifier that uses only one feature to predict the species of a flower.

- **Finding the most discriminative feature:** Compute the mean value of each feature separately for every species (Use for this exercise the feature matrix `X`). Organize these means into a small matrix where each row corresponds to a class and each column to a feature (resulting in a 3×4 array). Then, for every feature, calculate the absolute differences between class means for all possible pairs of species. Identify the feature that maximizes the smallest of these pairwise differences (this is the one that best separates all three species simultaneously).
- **Defining decision thresholds:** For the selected feature, sort the three class means in ascending order. Compute two midpoints between the consecutive class means. You will use these as thresholds to partition the

feature's range into three regions. Each region should correspond roughly to one species.

- **Implementing the classification rule and evaluating it:** Write a simple prediction function that, given a vector of values for the selected feature, assigns a class label according to the two thresholds: values below the first threshold belong to the first class, values between thresholds to the second, and values above the second to the third. Apply this rule to the entire dataset to obtain predicted species labels. Then, compare the predicted labels to the true ones and compute the classification accuracy.

4 MNIST Dataset with NumPy

In this part of the lab, you will work with the MNIST test dataset, a collection of handwritten digits represented as numerical pixel values. You will manipulate and analyze this dataset using only NumPy arrays, practicing how to handle high-dimensional data efficiently.

4.1 Loading and prepare the data

In this exercise, you will load the MNIST dataset and prepare it for analysis.

- **Importing the dataset:** Load the `mnist_test.csv` file using NumPy's file-reading functions as you have done for the Iris dataset. The dataset contains 10,000 rows and 785 columns. The first column corresponds to the digit label (ranging from 0 to 9), while the remaining 784 columns represent pixel intensities (ranging from 0 to 255) of a 28×28 image. Each row of `X` corresponds to a flattened 28×28 grayscale image.

- **Separating features and labels:** Split the loaded data into two parts: the label vector `y` (shape $(10000,)$) and the pixel matrix `X` (shape $(10000, 784)$). Confirm their dimensions and data types to ensure the split has been performed correctly.

4.2 Visual inspection of samples

Next, you will visualize a few digits directly from their numerical arrays to build an intuition of how the pixel intensities form characters.

- **Reshaping images:** Select a few sample indices and reshape their corresponding pixel vectors into 28×28 matrices.

- **Pixel–character mapping:** Convert the numeric pixel values of different images into characters according to their brightness range. Use the following mapping scheme:

$[0, 64)$ $\rightarrow$ `" "` (space)
$[64, 128)$ $\rightarrow$ `"."`
$[128, 192)$ $\rightarrow$ `"*"`
$[192, 256)$ $\rightarrow$ `"#"`

Hint: Initialize an empty character array using `np.empty_like` with `dtype=str`, and then fill it using Boolean masks that identify which pixels fall into each intensity range. Once all ranges are assigned, print the resulting character grid line by line so that the digit's shape is clearly recognizable.

4.3 Pixel frequency comparison by class

This final exercise will help you analyze how individual pixels contribute to the identity of a digit.

- **Building class masks:** Create two Boolean masks that identify all samples belonging to two specific digits (for example, 0 and 1). Use these masks to separate the corresponding rows from `X`.
- **Counting pixel activations:** For each pixel position, count how many times it is "active" (above a brightness threshold, such as 128) within each class.
- **Comparing pixel distributions:** Compute the absolute difference between the two count vectors to find the pixels that differ the most between the chosen digits. Identify the pixel position with the most significant difference.

4.4 (*) Pairwise distance analysis

In this step, you will explore how to measure similarity between digit images using NumPy broadcasting.

- **Selecting samples:** Choose four examples from the dataset corresponding to different digits (for instance, 0, 1, 1, and 7). Extract their pixel vectors into a smaller array `V` of shape $(4, 784)$.
- **Computing distances:** Using NumPy's vectorized operations, compute the pairwise Euclidean distances between all selected samples. You can rely on the identity

$$\|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2(a \cdot b)$$

to calculate all distances without using explicit loops. Store the results in a 4×4 distance matrix `D`.

- **Interpreting the distance matrix:** Inspect the values of `D` to determine which pairs of samples are visually most similar. Check whether the closest pairs correspond to the same digit label.