M-Lab CARTE AI Workshop 2025 - Lab A: Exploratory Data Analysis

In this introductory lab, we will get hands-on experience with Python, NumPy, and scikit-learn. This lab is designed to provide you with foundational skills for working in Python notebooks and using essential libraries for data science and machine learning.

Objectives

- Learn to use Python in a notebook environment.
- Get familiar with NumPy for numerical computations.
- Understand basic scikit-learn operations for machine learning tasks.

Overview

This lab will cover the following:

1. **Notebook Basics**:
   - Understand how to use notebook cells for code and text.
   - Learn shortcuts for running and modifying cells.
2. **NumPy Basics**:
   - Explore NumPy arrays, indexing, slicing, and basic operations.
   - Practice creating and manipulating multi-dimensional arrays.
   - Use vectorized operations for efficiency.
3. **Scikit-learn Basics**:
   - Load and explore the iris dataset.
   - Conduct exploratory data analysis (EDA) using visualizations.
   - Learn about the `n_samples x n_features` format for machine learning data.
   - Begin investigating the question: *Can we design a model to predict iris classifications?*

Dataset: Iris

We will work with the famous **iris dataset**, which includes:

- **Features**:
  - Sepal width (cm)
  - Sepal length (cm)
  - Petal width (cm)
  - Petal length (cm)
- **Target classes**:

- Setosa (class 0)
- Versicolor (class 1)
- Virginica (class 2)

The dataset is a great starting point for understanding basic machine learning workflows. You can learn more about it on its Wikipedia page.

An example of each iris species is shown below:

Setosa 

Versicolor 

Virginica 

Key Steps in Lab

1. **NumPy Operations**:
   - Create and manipulate arrays.
   - Perform basic indexing and slicing.
   - Use vectorized operations to compute metrics like accuracy.

2. **EDA with Scikit-learn**:
    - Load the iris dataset and inspect its structure.
    - Explore features and target classes.
    - Visualize data using matplotlib to identify patterns and relationships.
3. **Hands-on Exercises**:
    - Answer questions about the dataset using NumPy and scikit-learn.
    - Practice visualizing data and interpreting patterns.

Goals

By the end of this lab, you will:

- Understand how to use notebooks for interactive Python development.
- Gain proficiency in using NumPy for numerical operations.
- Begin exploring machine learning workflows using scikit-learn.
- Build confidence in visualizing and analyzing datasets.

Getting Started

1. Open this notebook and follow along with the provided code blocks and explanations.
2. Execute code cells as instructed to observe results.
3. For visualizations, ensure you have matplotlib installed and imported.
4. Focus on `TODOs` to solidify your understanding.

Let's begin by exploring the notebook environment and learning how to work with Python libraries like NumPy and scikit-learn. **Run the first code cell to import the necessary libraries!**

Importing Libraries: What and Why?

In Python, a **library** is a collection of pre-written code that provides tools and functionality to solve specific problems or perform common tasks, so you don't have to write everything from scratch. By importing a library, you gain access to its functions, classes, and modules, which you can use in your own code.

For example:

- **NumPy**: This library specializes in handling large, multi-dimensional arrays and matrices. It is optimized for numerical computations and provides powerful tools for mathematical operations, making it an essential library for data science and machine learning tasks. While Python is natively equipped to handle many of the same operations as NumPy, NumPy is much more efficient and is designed for numerical operations. Learn more

- **scikit-learn**: A widely used library for machine learning. It offers tools for data preparation, model training, and evaluation. With scikit-learn, you can build models for tasks like classification, regression, and clustering with ease. Learn more

Note: To run a cell on Google Colab, you can either click the Play icon to the left of the cell or use the keyboard shortcut "Shift+Enter".

In [1]:
```
import sklearn
import numpy as np
```

NumPy Basics

In Python, while you can create and manipulate lists, NumPy provides specialized tools for working with numerical data in arrays, making it faster and more efficient for large datasets.

The code below demonstrates:

1. **Creating an Array**: `np.arange(8)` generates a sequence of numbers from 0 to 7.
2. **Reshaping the Array**: `.reshape(2, 4)` reorganizes these numbers into a 2x4 (2 rows, 4 columns) array.

By running this code, you'll create a NumPy array called `array` and display it. We can then use the array to explore its properties and perform operations.

>> Run the code in the next cell to create and display the 2x4 array.

In [2]:
```
# Create a variable called "array" and fill it with a 2x4 NumPy array
array = np.arange(8).reshape(2, 4)
array  # Display the array
```

Out[2]:
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

We can access the **properties** of a NumPy array, such as its shape, number of dimensions, data type, and total number of elements, using a dot (`.`) followed by the property name.

In Python, an **object** (like a NumPy array) has **attributes** and **methods**:

- **Attributes** store information about the object, such as its shape or size. These are like "built-in variables" attached to the object.
- **Methods** are functions attached to the object that perform specific actions, like reshaping the array.

When accessing an attribute, you use the syntax `object.attribute`. For example:

- `array.shape` retrieves the dimensions of the array as a tuple (rows, columns).

- `array.ndim` retrieves the number of dimensions in the array.
- `array.size` retrieves the total number of elements in the array.

You don't need parentheses `()` for attributes because they are not functions you're calling—they simply hold information. In contrast, methods, like `array.reshape(2, 4)`, do require parentheses because they perform an operation.

Below is an example of accessing and printing several attributes of the array:

In [3]:
```
print("Shape:", array.shape)              # Tuple representing the array dim
print("Dimensions:", array.ndim)          # Number of dimensions (e.g., 2 fo
print("Data type:", array.dtype)          # Type of elements stored in the a
print("Number of elements:", array.size)  # Total number of elements in the
```

```
Shape: (2, 4)
Dimensions: 2
Data type: int64
Number of elements: 8
```

If we have a **Python list** (an ordered collection of elements), we can easily convert it into a NumPy array using the `np.array()` function. This is useful because NumPy arrays are more efficient and provide additional functionality compared to standard Python lists.

In [4]:
```
mylist = [0, 1, 1, 2, 3, 5, 8, 13, 21]  # A Python list
myarray = np.array(mylist)               # Convert the list to a NumPy array
myarray                                  # Displays the array
```

Out[4]:
```
array([ 0,  1,  1,  2,  3,  5,  8, 13, 21])
```

We can also work with **nested lists** (lists of lists) to create multidimensional NumPy arrays. Each nested list becomes a row in the resulting array, allowing us to represent tabular data or matrices. For example:

In [5]:
```
my2dlist = [[1, 2, 3], [4, 5, 6]]  # A nested list
my2darray = np.array(my2dlist)      # Convert it into a 2D NumPy array
my2darray                           # Displays the array
```

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

Indexing and Slicing NumPy Arrays

Indexing and slicing allow you to access specific parts of a NumPy array. These operations are essential for extracting and working with subsets of data.

- **Indexing**: Refers to accessing individual elements of an array by their position (index). In Python, indexing starts at 0, so the first element is at index `0`, the second at index `1`, and so on.

- **Slicing**: Refers to extracting a range of elements from an array. This is done using the colon (`:`) operator, which specifies a start, stop, and step size. For example, `array[start:stop]` will extract elements starting at `start` and stopping before `stop`.

In [6]:
```
array = np.arange(10)                                  # Create an array with r
print("Originally:", array)                            # Print the entire array
print("First four elements:", array[:4])               # Slice: first 4 element
print("After the first four elements:", array[4:])     # Slice: from index 4 to
print("The last element:", array[-1])                  # Indexing: access the l
```

```
Originally: [0 1 2 3 4 5 6 7 8 9]
First four elements: [0 1 2 3]
After the first four elements: [4 5 6 7 8 9]
The last element: 9
```

And we can index/slice multidimensional arrays, too.

In [7]:
```
array = np.array([[1, 2, 3], [4, 5, 6]])
print("Originally:", array)
print("First row only:", array[0])
print("First column only:", array[:, 0])
```

```
Originally: [[1 2 3]
 [4 5 6]]
First row only: [1 2 3]
First column only: [1 4]
```

Computing Accuracy with NumPy

In machine learning, a common task is to evaluate the performance of a classifier by comparing its **predictions** to the **true values** (or ground truth). One way to measure this is by calculating the classifier's **accuracy**, which is the proportion of correct predictions.

Accuracy is computed as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

In this example:

- `true_values` represents the ground truth labels for a dataset (0 = negative class, 1 = positive class).
- `predictions` represents the corresponding labels predicted by a machine learning model.

Using NumPy, we can:

1. **Convert Lists to Arrays**: First, we convert the Python lists into NumPy arrays for efficient element-wise operations.

2. **Compare Arrays Element-wise**: `true_values_array == predictions_array` returns a Boolean array where `True` indicates matching elements.
3. **Count Correct Predictions**: `np.sum()` sums the `True` values (treated as `1` in NumPy) to get the total number of correct predictions.
4. **Divide by Total Elements**: To calculate accuracy, divide the number of correct predictions by the total number of elements in the array, given by `.size`.

Here's the code:

In [8]:
```
true_values = [0, 0, 1, 1, 1, 1, 1, 0, 1, 0]  # Ground truth labels
predictions = [0, 0, 0, 1, 1, 1, 0, 1, 1, 0]  # Predicted labels

# Convert lists to NumPy arrays
true_values_array = np.array(true_values)
predictions_array = np.array(predictions)

# Compute accuracy
accuracy = np.sum(true_values_array == predictions_array) / true_values_arra
print("Accuracy:", accuracy * 100, "%")
```

```
Accuracy: 70.0 %
```

Scikit-learn Basics

Scikit-learn is a powerful Python library for building and evaluating machine learning models. It provides tools for a wide range of tasks, including:

- Data preparation
- Exploratory Data Analysis (EDA)
- Classification and regression
- Clustering and more

One key aspect of scikit-learn is that it expects data to be structured as a **2D matrix**, where:

- Rows represent individual **samples** (*n_samples*), like data points or observations.
- Columns represent **features** (*n_features*), which are the characteristics or measurements of the samples. The model also requires a separate column for the **target**, which contains the values you're trying to predict (e.g., labels for classification tasks).

To get started with scikit-learn, we'll use the famous **iris dataset**, which is a small and straightforward dataset commonly used for teaching and testing machine learning algorithms.

About the Iris Dataset

Each entry in the dataset represents a type of iris plant, categorized into one of three **classes** (targets):

- **Setosa** (class 0)
- **Versicolor** (class 1)
- **Virginica** (class 2)

Each iris sample includes the following **features**:

- Sepal length (cm)
- Sepal width (cm)
- Petal length (cm)
- Petal width (cm)

These features describe the physical characteristics of the plant, and the task is to use them to predict the correct class of iris.

Let's start by importing the dataset and conducting some basic exploratory data analysis (EDA) to understand its structure and content.

In [9]:
```
from sklearn.datasets import load_iris
iris_data = load_iris()
feature_data = iris_data.data
```

TODO: What are we looking at?

- Run the cell below and read the printed lines.
- In your own words: how many flowers are in this dataset, and how many measurements per flower?
- Discuss why the features are stored as floating point numbers (numbers with decimal places) and not integers (whole numbers).

In [10]:
```
print("Shape of feature data:", feature_data.shape)
print("Data type:", feature_data.dtype)
print("Number of samples:", feature_data.shape[0])
print("Number of features:", feature_data.shape[1])
```

```
Shape of feature data: (150, 4)
Data type: float64
Number of samples: 150
Number of features: 4
```

Next, we need to extract the **target data**, which represents the class labels for each sample in the dataset (e.g., Setosa, Versicolor, or Virginica). These labels indicate the category each iris sample belongs to and will be used as the ground truth for model training and evaluation.

In addition to the numerical labels (e.g., 0, 1, 2), we'll also retrieve the **target names**, which provide the human-readable class names (e.g., "Setosa"). This will help us

interpret the results later.

Run the code below to extract and save:

- `target_data` : The numeric class labels for each sample.
- `target_names` : The class names corresponding to the numeric labels.

In [11]:
```
target_data = iris_data.target       # Numeric class labels (0, 1, 2)
target_names = iris_data.target_names  # Human-readable class names ("Setosa
```

TODO: Understanding the labels

- Run the cell below and identify which number corresponds to each flower type.
- About how many 'setosa' samples are in the dataset? What does that suggest about class balance?
- Why do you think we store class names as words but use numbers for modeling?

In [12]:
```
print("Values in target_data:", target_data)
print("Data type of target_data:", target_data.dtype)
print("Values in target_names:", target_names)
print("Data type of target_names:", target_names.dtype)
print("-------------------------------------------------------")
print("Class distribution:")
print("Number of 'setosa' samples:", np.sum(target_data == 0))
print("Number of 'versicolor' samples:", np.sum(target_data == 1))
print("Number of 'virginica' samples:", np.sum(target_data == 2))
```

```
Values in target_data: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
Data type of target_data: int64
Values in target_names: ['setosa' 'versicolor' 'virginica']
Data type of target_names: <U10
-------------------------------------------------------
Class distribution:
Number of 'setosa' samples: 50
Number of 'versicolor' samples: 50
Number of 'virginica' samples: 50
```

Visual EDA: Exploring Feature Relationships

To better understand the dataset, we can perform **visual exploratory data analysis (EDA)** by plotting the samples based on a subset of their features. This helps us identify patterns or relationships between the features and the target classes.

In this example, we'll use **matplotlib**, a popular Python library for data visualization, to create a scatter plot of **sepal width** versus **sepal length**. Each data point will be color-

coded to represent its target class (Setosa, Versicolor, or Virginica). Learn more about matplotlib

Steps:

1. **Group Data by Class**:
   - Filter `feature_data` using Boolean indexing based on the `target_data` values to separate the samples into three groups: `setosa`, `versicolor`, and `virginica`.
2. **Create the Plot**:
   - Use `plt.scatter()` to plot each group with its respective features and a unique label.
3. **Add Labels and a Legend**:
   - Add axis labels, a title, and a legend to make the plot informative.

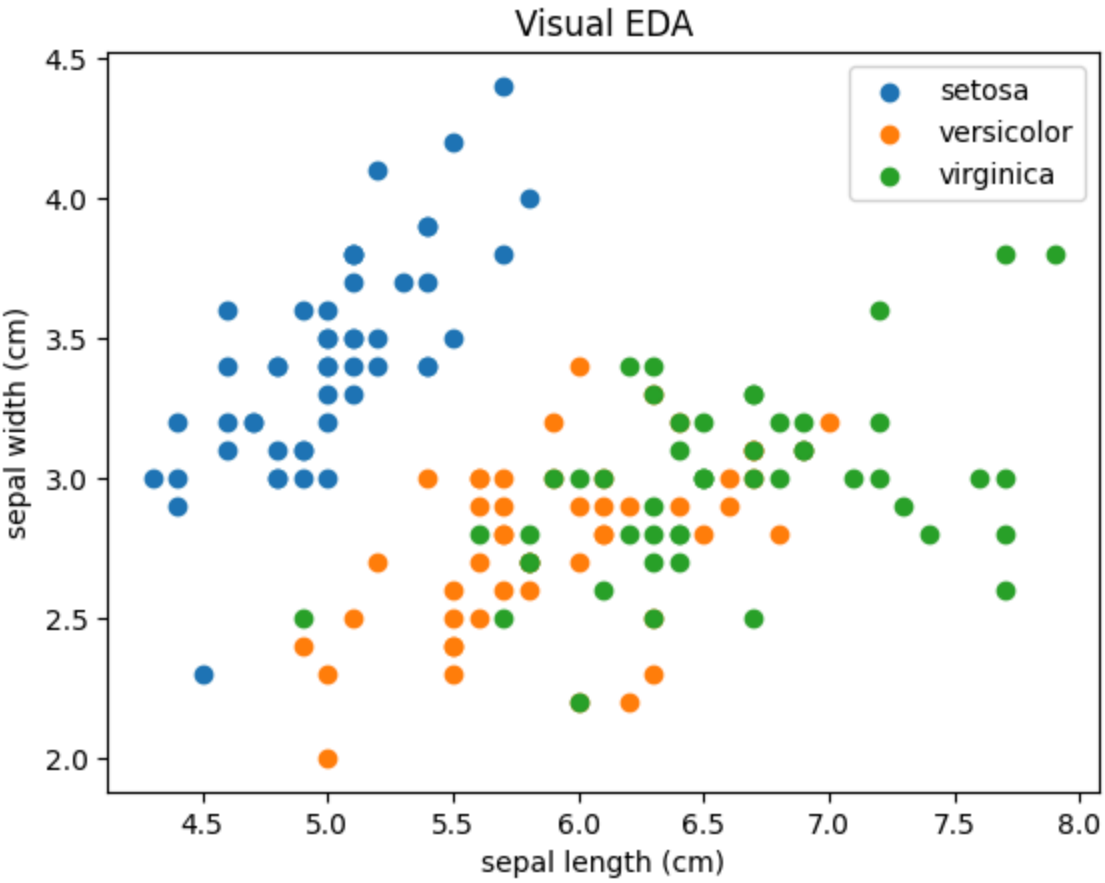
Run the code below to generate the plot:

In [13]:
```
import matplotlib.pyplot as plt

# Group samples by class
setosa = feature_data[target_data == 0]
versicolor = feature_data[target_data == 1]
virginica = feature_data[target_data == 2]

# Define Feature class to organize feature metadata
class Feature:
    def __init__(self, name, index):
        self.name = name
        self.index = index

# Create Feature objects for each measurement
SEPAL_LENGTH = Feature("sepal length", 0)
SEPAL_WIDTH = Feature("sepal width", 1)
PETAL_LENGTH = Feature("petal length", 2)
PETAL_WIDTH = Feature("petal width", 3)
```

In [14]:
```
def plot(x_feature, y_feature):
    # Create scatter plot
    plt.scatter(setosa[:, x_feature.index], setosa[:, y_feature.index], labe
    plt.scatter(versicolor[:, x_feature.index], versicolor[:, y_feature.inde
    plt.scatter(virginica[:, x_feature.index], virginica[:, y_feature.index]

    # Add labels and legend
    plt.legend()
    plt.xlabel(f"{x_feature.name} (cm)")
    plt.ylabel(f"{y_feature.name} (cm)")
    plt.title("Visual EDA")
    plt.show()
```

In [15]:
```
# Choose features to plot: SEPAL_LENGTH, SEPAL_WIDTH, PETAL_LENGTH, PETAL_Wl
x_feature = SEPAL_LENGTH  # Choose a feature for the x-axis
```

```
y_feature = SEPAL_WIDTH   # Choose a feature for the y-axis
plot(x_feature, y_feature)
```



TODO: What do you see?

- Look at the plot of sepal width versus sepal length and discuss which of the classes will be easier to separate visually.
  - Which classes seem to overlap, and where?
- Is it possible to visualize all the features for all samples in a single 2D plot? Why or why not?
- Change the features and see what happens: Currently, `x_feature=SEPAL_LENGTH` and `y_feature=SEPAL_WIDTH`. Change it to `x_feature=PETAL_LENGTH` and `y_feature=PETAL_WIDTH`. Similarly, plot other combinations and discuss how separable the classes are.

In the previous step, we used **Boolean indexing** to filter the feature data for each class based on the `target_data`. This allowed us to create a scatter plot where the iris classes were color-coded, making it easier to observe patterns and relationships between features.

Conclusion

In this lab, we explored the foundational tools and concepts necessary for working with Python, NumPy, and scikit-learn in the context of machine learning and data analysis. Here's a summary of what we covered:

1. **Python and Libraries**:
   - Learned the basics of importing and using libraries like NumPy and scikit-learn to streamline numerical and machine learning tasks.
2. **NumPy Fundamentals**:
   - Worked with arrays, explored their properties, and performed indexing, slicing, and vectorized operations for efficient data manipulation.
3. **Exploratory Data Analysis (EDA)**:
   - Loaded and inspected the iris dataset, visualized feature relationships using scatter plots, and analyzed patterns in the data.
4. **Scikit-learn Basics**:
   - Understood how machine learning datasets are structured and prepared, focusing on the `n_samples x n_features` format.
   - Explored class labels (targets) and their relationship to the feature data.

Key Takeaways

- NumPy and scikit-learn are essential tools for data science and machine learning, offering powerful abstractions for handling data and building models.
- Visual EDA provides critical insights into feature relationships and class separability, which inform model design.
- The iris dataset, while small, is an excellent starting point for learning about classification tasks and machine learning workflows.