**Python Basic**

Contents

[A. Python Basic 3](#_Toc195280508)

[1. NumPy Library 3](#_Toc195280509)

[*numpy.arange()* 3](#_Toc195280510)

[*numpy*.linspace() 3](#_Toc195280511)

[2. Matplotlib Library 5](#_Toc195280512)

[matplotlib.pyplot.plot 5](#_Toc195280513)

[matplotlib.pyplot.scatter 9](#_Toc195280514)

[matplotlib.pyplot.pie 13](#_Toc195280515)

[matplotlib.pyplot.*bar* 13](#_Toc195280516)

[3. JSON Format 14](#_Toc195280517)

[4. Read– Write JSON files 15](#_Toc195280518)

[1. Reading JSON Files 15](#_Toc195280519)

[2. WRTING JSON Files 18](#_Toc195280520)

[5. Custom Plot Color 21](#_Toc195280521)

[6. Registering Custom Colors 22](#_Toc195280522)

[B. Classification problem (machine learning) 24](#_Toc195280523)

[1. Introdution 24](#_Toc195280524)

[2. Training, Validation, and Test Data Sets 26](#_Toc195280525)

[General Introduction 26](#_Toc195280526)

[Training Data Set 26](#_Toc195280527)

[Validation Data Set 27](#_Toc195280528)

[Test Data Set 27](#_Toc195280529)

[3. Scale Features 28](#_Toc195280530)

[4. Train/Test 32](#_Toc195280531)

[5. Rule Based Classification 34](#_Toc195280532)

[5.1 Introdution 34](#_Toc195280533)

[5.2. Forms of Classification Rules 34](#_Toc195280534)

[5.3. Rule Structure 34](#_Toc195280535)

[5.4. Common Rule Induction Algorithms 35](#_Toc195280536)

[5.5. Rule Evaluation Metrics 36](#_Toc195280537)

[5.6. Illustrative Example 36](#_Toc195280538)

[5.8. Reference : 38](#_Toc195280539)

[6. K-Nearest Neighbors (KNN) Classification Using Euclidean Distance 39](#_Toc195280540)

[6.1 Introdution 39](#_Toc195280541)

[6.2 Imlementation of KNN in Python 39](#_Toc195280542)

[6.3 Reference : 39](#_Toc195280543)

[7. Confusion Matrix 39](#_Toc195280544)

[8. Decision Tree 44](#_Toc195280545)

# A. Python Basic

## 1. NumPy Library

NumPy cung cấp nhiều hàm tạo mảng giúp dễ dàng thao tác dữ liệu. Dưới đây là một số hàm quan trọng.

### *numpy.arange()*

*-> Function* creates an array of evenly spaced values within a given interval. It is similar to Python’s built-in range() function but returns a NumPy array instead of a list.

**Syntax of numpy.arange():**

|  |
| --- |
| *numpy***.***arange***([**start**,** **]**stop**,** **[**step**,** **]**dtype**=None,** **\*,** like**=None)** |

**Parameters of numpy():**

* **start (optional):**The starting value of the sequence. Default is 0.
* **stop (required):**The endpoint of the sequence, exclusive.
* **step (optional):** The spacing between consecutive values. Default is 1.
* **dtype (optional):**The desired data type of the output array.

**Return Type:**

* Array of evenly spaced values.

**Example**

Let’s understand with a simple example:

|  |
| --- |
| **import** numpy **as** np  #create an array  arr**=** np**.***arange***(**5 **,** 10**)**  **print(**arr**)** |

**Output :**

|  |
| --- |
| [5 6 7 8 9] |

### *numpy*.linspace()

**-> linspace() function** in NumPy returns an array of evenly spaced numbers over a specified range. Unlike the **range() function** in Python that generates numbers with a specific step size. linspace() allows you to specify the total number of points you want in the array, and NumPy will calculate the spacing between the numbers automatically.

**Syntax of linspace():**

|  |
| --- |
| *numpy***.***linspace***(**start**,** stop**,** num**=**50**,** endpoint**=True,** retstep**=False,** dtype**=None,** axis**=**0**)** |

**Parameters:**

* **start:**[optional] start of interval range. By default start = 0
* **stop**: end of interval range
* **num**: [int, optional] No. of samples to generate
* **retstep**: If True, Stop is the last sample By default restep = False
* **endpoint**: If True, stop is included as the last value. If False, stop is excluded. By default endpoint=True.
* **dtype**: type of output array
* **axis**: If start and stop are arrays, axis specifies on what axis will the values be added. If axis = 0, value is added to front, if axis = -1 value is added at the end.

**Return Type:**

* ndarray
* step : [float, optional], if restep = True

**Example :**

|  |
| --- |
| **import** numpy **as** np  # Generate 10 numbers between 0 and 1  array **=** np**.***linspace***(**0**,** 1**,** num**=**10**)**  **print(**array**)** |

**Output :**

|  |
| --- |
| [0. 0.11111111 0.22222222 0.33333333 0.44444444 0.55555556  0.66666667 0.77777778 0.88888889 1. ] |

## 2. Matplotlib Library

Matplotlib is a low level graph plotting library in python that serves as a visualization utility.

### matplotlib.pyplot.plot

**matplotlib.pyplot.plot** is a function in the Matplotlib library used to create line plots by representing data with x and y coordinates in the form of lines or markers.

**Syntax :**

|  |
| --- |
| *matplotlib***.***pyplot***.***plot***(\***args**,** scalex**=True,** scaley**=True,** data**=None,** **\*\***kwargs**)** |

**Common Usage :**

|  |
| --- |
| *plot***([**x**],** y**,** **[**fmt**],** **\*,** data**=None,** **\*\***kwargs**)**  plot**([**x**],** y**,** **[**fmt**],** **[**x2**],** y2**,** **[**fmt2**],** **...,** **\*\***kwargs**)** |

**Parameters :**

| **Parameter** | **Data Type** | **Description** |
| --- | --- | --- |
| x | array-like or float | X-axis coordinates. If not provided, defaults to range(len(y)). |
| y | array-like or float | Y-axis coordinates. This parameter is required. |
| fmt | str, optional | Format string specifying color, marker, and line style. |
| data | dict, pandas.DataFrame, or numpy array | Labeled data that can be accessed using x, y. |
| scalex | bool, default True | Automatically adjust the x-axis limits based on data. |
| scaley | bool, default True | Automatically adjust the y-axis limits based on data. |
| \*\*kwargs | Line2D properties | Additional properties like color, linewidth, marker style, etc. |

**Format String (fmt) :**

The fmt string can include:

1. Color: 'b' (blue), 'g' (green), 'r' (red), 'c' (cyan), 'm' (magenta), 'y' (yellow), 'k' (black), 'w' (white).
2. Marker style: '.', ',', 'o', 'v', '^', 's', 'p', 'h', '\*', 'x'.
3. Line style: '-' (solid), '--' (dashed), '-.' (dash-dot), ':' (dotted).

**Example:**

|  |
| --- |
| *plot***(**x**,** y**,** 'go--'**,** linewidth**=**2**,** markersize**=**12**)** # Green dashed line with 'o' markers |

**Examples**

a. Simple Line Plot

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  **import** numpy **as** np  x **=** np**.***linspace***(**0**,** 10**,** 100**)**  y **=** np**.***sin***(**x**)**  plt**.***plot***(**x**,** y**)**  plt**.***xlabel***(**"X-axis"**)**  plt**.***ylabel***(**"Y-axis"**)**  plt**.***title***(**"Simple Line Plot"**)**  plt**.***show***()** |

Output :

A graph of a simple line plot

AI-generated content may be incorrect.

b. Using *fmt* for Formatting

|  |
| --- |
| *plt***.***plot***(**x**,** y**,** 'r--'**)** # Red dashed line  plt**.***show***()** |

Output :

A graph with red lines on a black background

AI-generated content may be incorrect.

c. Plotting Multiple Lines

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  **import** numpy **as** np  x **=** np**.***linspace***(**0**,** 10**,** 100**)**  y1 **=** np**.***sin***(**x**)**  y2 **=** np**.***cos***(**x**)**  plt**.***plot***(**x**,** y1**,** 'b-'**,** label**=**'sin(x)'**)** # Blue solid line  plt**.***plot***(**x**,** y2**,** 'r--'**,** label**=**'cos(x)'**)** # Red dashed line  plt**.***legend***()**  plt**.***show***()** |

A graph of a function

AI-generated content may be incorrect.

d. Plotting Labeled Data with data

|  |
| --- |
| **import** pandas **as** pd  df **=** pd**.***DataFrame***({**'x'**:** **[**1**,** 2**,** 3**,** 4**],** 'y'**:** **[**10**,** 20**,** 25**,** 30**]})**  plt**.***plot***(**'x'**,** 'y'**,** data**=**df**)**  plt**.***show***()** |

A graph on a black background

AI-generated content may be incorrect.

### matplotlib.pyplot.scatter

***matplotlib.pyplot.scatter*** *is a function in Matplotlib that creates scatter plots by plotting individual points based on given x and y coordinates. It allows customization of marker size, color, shape, and transparency, making it useful for visualizing relationships in datasets.*

***Syntax :***

|  |
| --- |
| *matplotlib***.***pyplot***.***scatter***(**x**,** y**,** s**=None,** c**=None,** marker**=None,** cmap**=None,** norm**=None,**vmin**=None,** vmax**=None,** alpha**=None,** linewidths**=None,**edgecolors**=None,** plotnonfinite**=False,** data**=None,** **\*\***kwargs**)** |

***Parameters :***

| ***Parameter*** | ***Type*** | ***Description*** |
| --- | --- | --- |
| *x* | *array-like* | *X-coordinates of data points.* |
| *y* | *array-like* | *Y-coordinates of data points.* |
| *s* | *float or array-like, optional* | *Marker size in points². Default is rcParams['lines.markersize'] \*\* 2.* |
| *c* | *array-like or list of colors, optional* | *Colors of markers. Can be a sequence of numbers mapped to a colormap, RGB(A) values, or a single color format string.* |
| *marker* | *MarkerStyle, optional* | *Defines the shape of the marker. Default is 'o'.* |
| *cmap* | *Colormap, optional* | *Colormap used when c is a sequence of values.* |
| *norm* | *Normalize, optional* | *Scaling function for mapping c values to [0,1] before applying cmap.* |
| *vmin, vmax* | *float, optional* | *Define the color scale limits.* |
| *alpha* | *float, optional* | *Transparency level of markers (0 = fully transparent, 1 = fully opaque).* |
| *linewidths* | *float or array-like, optional* | *Width of marker edges. Default is 1.5.* |
| *edgecolors* | *{'face', 'none'} or list of colors, optional* | *Color of marker edges. If 'face', edges match marker color. If 'none', edges are not drawn.* |
| *plotnonfinite* | *bool, default False* | *Whether to plot points with nonfinite color values (inf, -inf, NaN).* |
| *data* | *dict, optional* | *Data container allowing column access by string keys.* |
| *zorder* | int, optional | Defines the drawing order of the scatter plot relative to other elements. Higher values are drawn on top. Default is 1. |
| *\*\*kwargs* | *PathCollection properties* | *Additional properties for customization.* |

***Examples :***

***a. Basic Scatter Plot***

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  x **=** **[**40**,** 30**,** 20**,** 10**]**  y **=** **[**40**,** 2**,** 20**,** 10**]**  plt**.***scatter***(**x**,** y**)**  plt**.***xlabel***(**"X-axis"**)**  plt**.***ylabel***(**"Y-axis"**)**  plt**.***title***(**"Basic Scatter Plot"**)**  plt**.***show***()** |

***Output :***

***A graph with yellow dots

AI-generated content may be incorrect.***

***b. Customizing Marker Size and Color***

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  colors **=** np**.***random***.***rand***(**50**)**  sizes **=** 1000 **\*** np**.***random***.***rand***(**50**)**  plt**.***scatter***(**x**,** y**,** c**=**colors**,** s**=**sizes**,** alpha**=**0.5**)**  plt**.***show***()** |

***Output :***

***A screen shot of a black background

AI-generated content may be incorrect.***

***c. Using Different Marker Styles***

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  x **=** np**.***random***.***rand***(**50**)**  y **=** np**.***random***.***rand***(**50**)**  plt**.***scatter***(**x**,** y**,** marker**=**'^'**,** color**=**'red'**,** s**=**100**)**  plt**.***show***()** |

***A graph showing red triangles

AI-generated content may be incorrect.***

***d. Custom Scatter Plot with Red Markers***

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  X **=** np**.***random***.***rand***(**50**)**  Y1 **=** np**.***random***.***rand***(**50**)**  plt**.***scatter***(**X**,** Y1**,** c**=**'red'**,** marker**=**'o'**,** s**=**10**,** label**=**'objRed'**,** zorder**=**2**)**  plt**.***xlabel***(**"X-axis"**)**  plt**.***ylabel***(**"Y-axis"**)**  plt**.***title***(**"Red Scatter Plot"**)**  plt**.***legend***()**  plt**.***show***()** |

*A screen shot of a graph

AI-generated content may be incorrect.*

### matplotlib.pyplot.pie

Plot a pie chart. Make a pie chart of array x. The fractional area of each wedge is given by x/sum(x). The wedges are plotted counterclockwise, by default starting from the x-axis.

**Example**

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  labels **=** **[**'Python'**,** 'Java'**,** 'C++'**,** 'JavaScript'**]**  sizes **=** **[**40**,** 30**,** 20**,** 10**]**  plt**.***pie***(**sizes**,** labels**=**labels**,** autopct**=**'%1.1f%%'**)**  plt**.***show***()** |

**Output:**

A pie chart with text on it

AI-generated content may be incorrect.

*Pie chart using pyplot*

### matplotlib.pyplot.*bar*

*Make a bar plot. The bars are positioned at x with the given alignment. Their dimensions are given by height and width. The vertical baseline is bottom (default 0).*

*Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.*

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  categories **=** **[**'A'**,** 'B'**,** 'C'**,** 'D'**]**  values **=** **[**3**,** 7**,** 2**,** 5**]**  plt**.***bar***(**categories**,** values**)**  plt**.***show***()** |

***Output:***

*A graph with blue rectangles

AI-generated content may be incorrect.*

*Bar Chart using Pyplot*

## 3. JSON Format

The JSON file (JavaScript Object Notation) is a text-based data storage format that is easy for humans to read and write while also being easy for machines to parse and generate. JSON data is organized in key-value pairs.

**Example of color data in the data.json file:**

|  |
| --- |
| **[{**  "objColor"**:**"RED"**,**  "objRed"**:**99**,**  "objGreen"**:**4**,**  "objBlue"**:**9**,**  "id"**:**"140327575m1"  **},**  **{**  "objColor"**:**"BLUE"**,**  "objRed"**:**3**,**  "objGreen"**:**6**,**  "objBlue"**:**20**,**  "id"**:**"530909218m63"  **},**  **{**  "objColor"**:**"GREEN"**,**  "objRed"**:**14**,**  "objGreen"**:**31**,**  "objBlue"**:**8**,**  "id"**:**"880816628m50"  **},**  **{**  "objColor"**:**"WHITE"**,**  "objRed"**:**21**,**  "objGreen"**:**22**,**  "objBlue"**:**47**,**  "id"**:**"699721538m144"  **}**  **]** |

**Parameters:**

* **objColor** (String) → The name of the color (e.g., "RED", "BLUE", "GREEN", "WHITE").
* **objRed** (Integer) → The intensity of the red component in the RGB color model (0–255).
* **objGreen** (Integer) → The intensity of the green component in the RGB color model (0–255).
* **objBlue** (Integer) → The intensity of the blue component in the RGB color model (0–255).
* **id** (String) → A unique identifier for each object, likely used for tracking or reference.

## 4. Read– Write JSON files

### 1. Reading JSON Files

Suppose data.json contains:

|  |
| --- |
| **[{**  "objColor"**:**"RED"**,**  "objRed"**:**99**,**  "objGreen"**:**4**,**  "objBlue"**:**9**,**  "id"**:**"140327575m1"  **},**  **{**  "objColor"**:**"BLUE"**,**  "objRed"**:**3**,**  "objGreen"**:**6**,**  "objBlue"**:**20**,**  "id"**:**"530909218m63"  **},**  **{**  "objColor"**:**"GREEN"**,**  "objRed"**:**14**,**  "objGreen"**:**31**,**  "objBlue"**:**8**,**  "id"**:**"880816628m50"  **},**  **{**  "objColor"**:**"WHITE"**,**  "objRed"**:**21**,**  "objGreen"**:**22**,**  "objBlue"**:**47**,**  "id"**:**"699721538m144"  **}**  **]** |

#### 1.1 Using pandas - pd.read\_json()

**Syntax:**

|  |
| --- |
| **import** pandas **as** pd  df **=** pd**.***read\_json***(**"data.json"**,** orient**=**"records"**)** |

**Description:**

Reads a JSON file and loads it into a pandas DataFrame for easy data manipulation.\

**Parameters:**

* data.json: The JSON file to be read.
* orient="records": Specifies that the JSON file contains a list of dictionaries (records).

**Example**:

|  |
| --- |
| import pandas as pd  df = pd.*read\_json*('data.json',orient="records")  print(df[['id', 'objRed', 'objGreen', 'objBlue', 'objColor']]) |

*OUTPUT :*

A white background with black text

AI-generated content may be incorrect.

#### 1.2 Without pandas : json.loads()

**Syntax:**

|  |
| --- |
| **import** json  **with** **open(**"data.json"**,** "r"**)** **as** file**:**  data **=** json**.***load***(**file**)** |

**Description:**

Reads a JSON file and loads it into a Python list of dictionaries.

**Parameters:**

* data.json: The JSON file to be read.

**json.loads() Function**

The json.loads() function in Python is used to parse a JSON string and convert it into a Python object. It takes a JSON string as input and returns a corresponding Python object. The syntax of json.loads() is as follows

|  |
| --- |
| *json***.***loads***(**json\_string**,** **\*,** object\_hook**=None,** parse\_float**=None,** parse\_int**=None,** parse\_constant**=None,** object\_pairs\_hook**=None,** **\*\***kw**)** |

The json.loads() function allows us to convert a JSON string into a Python object, such as a dictionary or a list. This is useful when we receive JSON data from an API or a file and want to work with it in Python.

**Example1 :**

|  |
| --- |
| **import** json  json\_string **=** '{"name": "John", "age": 30, "city": "New York"}'  data **=** json**.***loads***(**json\_string**)**  **print(**data**[**"name"**])**  **print(**data**[**"age"**])**  **print(**data**[**"city"**])** |

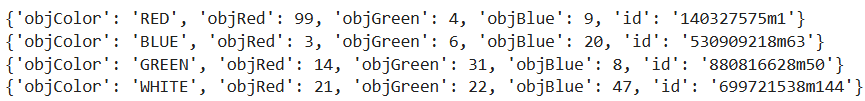
**Output:**

|  |
| --- |
| John  30  New York |

**Example2:**

|  |
| --- |
| **import** json  **with** **open(**"data.json"**,** "r"**)** **as** file**:**  data **=** json**.***load***(**file**)**  **for** record **in** data**:**  **print(**record**)** |

Output :



### 2. WRTING JSON Files

#### 2.1 Using pandas- df.to\_json

**Syntax:**

|  |
| --- |
| *df***.***to\_json***(**"output.json"**,** orient**=**"records"**,** indent**=**4**)** |

**Description:**

Writes a pandas DataFrame to a JSON file in a structured format.

**Parameters:**

* df.to\_json(...): Converts the DataFrame into a JSON file.
* "output.json": The output file name.
* orient="records": Specifies that each row of data should be represented as a separate JSON object.
* indent=4: Formats the JSON file with an indentation of 4 spaces for better readability.

**Example:**

|  |
| --- |
| **import** pandas **as** pd  # Read file data.json  df **=** pd**.***read\_json***(**"data.json"**,** orient**=**"records"**)**  # Modify DataFrame (Example: Changing objColor)  df**[**"objColor"**]** **=** df**.***apply***(lambda** row**:** "RED"**,** axis**=**1**)**  # OR df["objColor"] = “RED”  # Write to file output.json  df**.***to\_json***(**"output.json"**,** orient**=**"records"**,** indent**=**4**)**  **print(**df**)** |

**Explanation:**

* df.apply(...): Applies a function to each row in the DataFrame.
* lambda row: determine\_color(...): The lambda function calls determine\_color with the red, green, and blue values from each row.

After running this command, a new output.json file will be created, containing the updated data with the correct color names.

**Output :**

**A black text on a white background

AI-generated content may be incorrect.**

#### 2.2 Without pandas - json.dump()

**Syntax:**

|  |
| --- |
| **import** json  **with** **open(**"output.json"**,** "w"**)** **as** file**:**  json**.***dump***(**data**,** file**,** indent**=**4**)** |

**Description:**

Writes a Python list of dictionaries to a JSON file.

**json.dump() Function**

Syntax and Parameters

Python’s JSON.dump() function is used to serialize a Python object into a JSON-formatted string and write it to a file-like object. It inputs Python and file-like objects and writes the JSON data to the file. The syntax of json.dump() is as follows:

**Syntax :**

|  |
| --- |
| *json***.***dump***(**obj**,** fp**,** **\*,** skipkeys**=False,** ensure\_ascii**=True,** check\_circular**=True,** allow\_nan**=True,** cls**=None,** indent**=None,** separators**=None,** default**=None,** sort\_keys**=False,** **\*\***kw**)** |

*Converting Python Objects to JSON*

The json.dump() function allows us to convert a Python object into a JSON string, such as a dictionary or a list. This is useful when storing Python data in a JSON file or sending it over a network. Here’s an example:

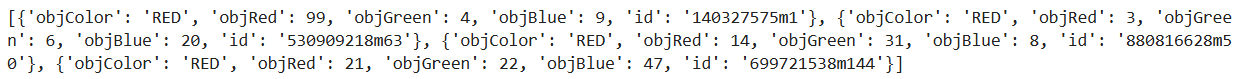
**Example1 :**

|  |
| --- |
| **import** json  data **=** **{**      "name"**:** "John"**,**      "age"**:** 30**,**      "city"**:** "New York"  **}**  **with** **open(**"data.json"**,** "w"**)** **as** file**:**      json**.***dump***(**data**,** file**)** |

**Example2:**

|  |
| --- |
| **import** json  # Read data from data.json  **with** **open(**"data.json"**,** "r"**)** **as** file**:**  data **=** json**.***load***(**file**)**  # Modify objColor to "RED" for all entries  **for** record **in** data**:**  record**[**"objColor"**]** **=** "RED"  # Write the modified data to output.json  **with** **open(**"output.json"**,** "w"**)** **as** file**:**  json**.***dump***(**data**,** file**,** indent**=**4**)**  **print(**data**)** |

**Output :**

****

## 5. Custom Plot Color

This API dynamically configures the color cycle in Matplotlib plots based on the objColor column of a DataFrame. This API ensures that all plots automatically use predefined colors from structured data, eliminating the need for manual color assignment.

**Syntax :**

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  **def** set\_plot\_colors**(**df**):**  #Set default colors for matplotlib plots based on objColor column.  plt**.***rcParams***[**'axes.prop\_cycle'**]** **=** plt**.***cycler***(**color**=**df**[**"objColor"**])** |

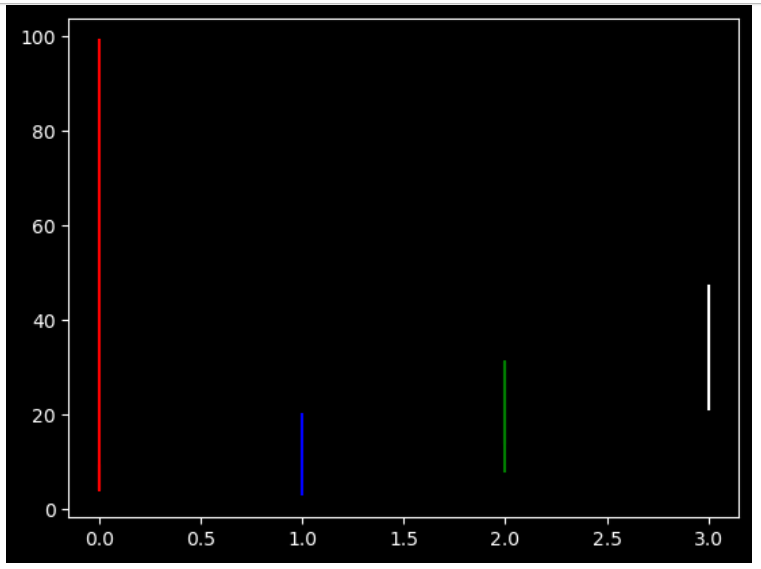
**Description :**

* plt. rcParams: A configuration dictionary in Matplotlib that controls default plot settings.
* plt.cycler: A tool for defining cycling properties like colors or styles in plots.

**Example :**

|  |
| --- |
| **import** matplotlib**.***pyplot* **as** plt  **import** pandas **as** pd  **import** numpy **as** np  **def** set\_plot\_colors**(**df**):**  """Set default colors for matplotlib plots based on objColor column."""  plt**.***rcParams***[**'axes.prop\_cycle'**]** **=** plt**.***cycler***(**color**=**df**[**"objColor"**])**  # Load JSON Data  df **=** pd**.***read\_json***(**'data.json'**,** orient**=**"records"**)**  X **=** np**.***arange***(**0**,** **len(**df**))** # X-axis indices  # Extract RGB values for plotting  Y1 **=** df**[**'objRed'**]**  Y2 **=** df**[**'objGreen'**]**  Y3 **=** df**[**'objBlue'**]**  Xline **=** **[**X**,** X**,** X**]**  Yline **=** **[**Y1**,** Y2**,** Y3**]**  plt**.***plot***(**Xline**,** Yline**,** linestyle**=**'solid'**,** linewidth**=**1.5**,** zorder**=**1**)**  # Apply dark theme  **%**matplotlib inline  plt**.***style***.***use***(**"dark\_background"**)**  # Apply color settings  set\_plot\_colors**(**df**)**  # Now any plot will use objColor as default colors  plt**.***show***()** |

**Output :**



## 6. Registering Custom Colors

*This API allows players to define and register custom colors using a unique name and a hex color code. The registered colors are stored in mpc.\_colors\_full\_map.*

***Syntax :***

|  |
| --- |
| **def** register\_color**(**color\_name**,** color\_spec**):**  mpc**.***\_colors\_full\_map***[**color\_name**]** **=** color\_spec  **return** mpc**.***\_colors\_full\_map* |

***Description :***

* *register\_color(color\_name, color\_spec): Adds a custom color to the internal color map.*
* *color\_name: A string representing the name of the color.*
* *color\_spec: A string representing the hex code of the color.*
* *mpc.\_colors\_full\_map: A dictionary that stores the color mappings.*

***Example:***

|  |
| --- |
| # Registering a new color  register\_color**(**"SUNSET"**,** "#FF5733"**)**  register\_color**(**"OCEAN"**,** "#1E90FF"**)**  # Printing updated color map  **print(**mpc**.***\_colors\_full\_map***)** |

***4.5 Expected Output***

|  |
| --- |
| **{**  "SUNSET"**:** "#FF5733"**,**  "OCEAN"**:** "#1E90FF"**,**  "BARE"**:** "#844E38"**,**  **...** **(**other predefined colors**)** **...**  **}** |

# B. Classification problem (machine learning)

## 1. Introdution

Machine learning (ML) is a branch of artificial intelligence that allows systems to learn from data and make predictions based on that knowledge. In ML, the classification problem refers to the task of categorizing or classifying input data into predefined classes or categories. This process involves analyzing data to detect patterns and relationships between features in the dataset, enabling the model to predict outcomes for new, unseen instances.

Classification is a supervised learning technique where the goal is to predict a class label for a given input based on labeled training data. This method has wide applications in various fields, such as medical diagnosis, spam email detection, and image recognition. By learning from a training set, a classifier can make predictions on new data, assigning each instance to one of the classes.

A dataset used for classification typically contains labeled data, meaning each data point is associated with a known class or category. For instance, in a medical dataset, each patient may have various attributes (such as age, symptoms, etc.) and a label indicating whether they have a specific disease. By training the model on such data, the algorithm learns to associate these features with their respective classes, making it capable of classifying new patient data.

**Data Set**

In machine learning, a dataset refers to a collection of data that is used to train, validate, and test models. It can vary from a simple array to a complex database with multiple attributes or features. The quality and quantity of the data directly influence the performance of the classification model.

For example, a simple dataset in an array format might look like this:

[99,86,87,88,111,86,103,87,94,78,77,85,86]

Or, it might be structured in a more detailed database format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Carname** | **Color** | **Age** | **Speed** | **AutoPass** |
| BMW | red | 5 | 99 | Y |
| Volvo | black | 7 | 86 | Y |
| VW | gray | 8 | 87 | N |
| VW | white | 7 | 88 | Y |
| Ford | white | 2 | 111 | Y |
| VW | white | 17 | 86 | Y |
| Tesla | red | 2 | 103 | Y |
| BMW | black | 9 | 87 | Y |
| Volvo | gray | 4 | 94 | N |
| Ford | white | 11 | 78 | N |
| Toyota | gray | 12 | 77 | N |
| VW | white | 9 | 85 | N |
| Toyota | blue | 6 | 86 | Y |

Looking at such a dataset, we can observe trends, such as the most popular car color being white, or the oldest car being 17 years old. But beyond these observations, machine learning enables us to predict outcomes, such as whether a car will have the "AutoPass" feature, by learning from the patterns in the available features (e.g., color, age, and speed).

**Data Types**

Understanding the types of data you are working with is essential when choosing the right technique or algorithm for your analysis. Data types in machine learning can be broadly categorized into three types:

1. **Numerical Data**: These are numerical values, and they can be further categorized into:
   * **Discrete Data**: These are countable data, typically represented as integers. For example, the number of cars passing by.
   * **Continuous Data**: These are measurable values that can take any numerical value. For example, the price of a product or the size of an item.
2. **Categorical Data**: These values represent categories or labels that do not have any inherent order. Examples include color labels (e.g., "red", "blue") or binary outcomes like "Yes" or "No".
3. **Ordinal Data**: This type of data has a natural order or ranking. For example, school grades (A > B > C) or satisfaction levels (Low, Medium, High).

Knowing the data type is crucial because it helps you select the appropriate algorithms and preprocessing techniques to use for effective model training and prediction

## 2. Training, Validation, and Test Data Sets

### General Introduction

In machine learning, one of the foundational tasks is building models that can learn from data and make predictions or decisions based on it. These models are typically trained and evaluated using data that is systematically divided into three main subsets: the training set, the validation set, and the test set. Each of these subsets plays a unique role in the model development process, ensuring that the final model performs well not only on the data it has seen but also on unseen data. This separation is crucial for reducing the risk of overfitting and for ensuring that the model can generalize to new inputs effectively.

### Training Data Set

The training data set is the primary set of data used during the learning process. It is employed to fit the parameters of the model, such as the weights in a neural network or the coefficients in a linear model. For supervised learning tasks, the training set typically contains pairs of input data and their corresponding outputs (labels or targets). The model learns by adjusting its parameters based on how well its predictions match the targets.

In practice, the training process involves feeding the model the input data, generating predictions, and then comparing these predictions to the actual labels. An optimization algorithm like gradient descent is used to minimize the difference between the predicted and actual outputs. This process is repeated iteratively to improve the model’s accuracy on the training set.

However, models that are too closely fitted to the training data may capture noise or patterns that do not generalize, a phenomenon known as overfitting. Therefore, relying solely on the training data set for model evaluation is insufficient.

In some applications, training data is continuously updated with new samples, a process known as incremental learning. This allows models to adapt over time as more data becomes available.

**Example: Training Dataset for Flower Classification**

Let’s consider a simple classification task using the popular **Iris dataset**. We want to train a model to classify flowers into one of three species: **Setosa**, **Versicolor**, or **Virginica**, based on features like sepal length and petal width.

**Sample Data (from the training dataset):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sepal Length (cm)** | **Sepal Width (cm)** | **Petal Length (cm)** | **Petal Width (cm)** | **Species** |
| 5.1 | 3.5 | 1.4 | 0.2 | Setosa |
| 6.7 | 3.1 | 4.7 | 1.5 | Versicolor |
| 7.2 | 3.6 | 6.1 | 2.5 | Virginica |

Each row is an **input-output pair** used during training:

* The input: the measurements of the flower
* The output: the species of the flower

### Validation Data Set

The validation data set is a separate subset of data used during the model development phase, primarily for tuning hyperparameters and selecting the best-performing model configuration. Hyperparameters are settings that are not learned from the training process but are set before training begins—such as the number of hidden layers in a neural network or the depth of a decision tree.

The validation set allows developers to evaluate how changes to the model’s architecture or training approach affect performance. By comparing different model configurations using the validation set, developers can identify which version of the model performs best on data that it has not seen during training.

This dataset also supports techniques like early stopping, where training is halted if performance on the validation set begins to degrade, indicating potential overfitting. It is essential that the validation data set is not used to train the model itself but only to guide decisions about its design and training duration.

### Test Data Set

The test data set is used to provide an unbiased evaluation of the final model after it has been trained and tuned. It is kept completely separate from both the training and validation sets to ensure that the performance metrics derived from it reflect the model’s ability to generalize to new, unseen data.

Once the best model configuration has been selected using the validation set, the model is evaluated on the test set. The results from this evaluation—such as accuracy, precision, recall, F1-score, etc.—are used to report the expected performance of the model in real-world scenarios.

Using a separate test set is especially important to detect overfitting that may have occurred during model selection or hyperparameter tuning. In some workflows, especially when using cross-validation, the need for a separate validation set may be reduced, but the final assessment should always rely on a test set that was not used at any prior stage.

## 3. Scale Features

When your data has different values, and even different measurement units, it can be difficult to compare them. What is kilograms compared to meters? Or altitude compared to time?

The answer to this problem is scaling. We can scale data into new values that are easier to compare.

Take a look at the table below :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Car** | **Model** | **Volume** | **Weight** | **CO2** |
| Toyota | Aygo | 1.0 | 790 | 99 |
| Mitsubishi | Space Star | 1.2 | 1160 | 95 |
| Skoda | Citigo | 1.0 | 929 | 95 |
| Fiat | 500 | 0.9 | 865 | 90 |
| Mini | Cooper | 1.5 | 1140 | 105 |
| VW | Up! | 1.0 | 929 | 105 |
| Skoda | Fabia | 1.4 | 1109 | 90 |
| Mercedes | A-Class | 1.5 | 1365 | 92 |
| Ford | Fiesta | 1.5 | 1112 | 98 |
| Audi | A1 | 1.6 | 1150 | 99 |
| Hyundai | I20 | 1.1 | 980 | 99 |
| Suzuki | Swift | 1.3 | 990 | 101 |
| Ford | Fiesta | 1.0 | 1112 | 99 |
| Honda | Civic | 1.6 | 1252 | 94 |
| Hundai | I30 | 1.6 | 1326 | 97 |
| Opel | Astra | 1.6 | 1330 | 97 |
| BMW | 1 | 1.6 | 1365 | 99 |
| Mazda | 3 | 2.2 | 1280 | 104 |
| Skoda | Rapid | 1.6 | 1119 | 104 |
| Ford | Focus | 2.0 | 1328 | 105 |
| Ford | Mondeo | 1.6 | 1584 | 94 |
| Opel | Insignia | 2.0 | 1428 | 99 |
| Mercedes | C-Class | 2.1 | 1365 | 99 |
| Skoda | Octavia | 1.6 | 1415 | 99 |
| Volvo | S60 | 2.0 | 1415 | 99 |
| Mercedes | CLA | 1.5 | 1465 | 102 |
| Audi | A4 | 2.0 | 1490 | 104 |
| Audi | A6 | 2.0 | 1725 | 114 |
| Volvo | V70 | 1.6 | 1523 | 109 |
| BMW | 5 | 2.0 | 1705 | 114 |
| Mercedes | E-Class | 2.1 | 1605 | 115 |
| Volvo | XC70 | 2.0 | 1746 | 117 |
| Ford | B-Max | 1.6 | 1235 | 104 |
| BMW | 2 | 1.6 | 1390 | 108 |
| Opel | Zafira | 1.6 | 1405 | 109 |
| Mercedes | SLK | 2.5 | 1395 | 120 |

It can be difficult to compare the volume 1.0 with the weight 790, but if we scale them both into comparable values, we can easily see how much one value is compared to the other.

There are different methods for scaling data, in this tutorial we will use a method called standardization.

The standardization method uses this formula:

z = (x - u) / s

Where z is the new value, x is the original value, u is the mean and s is the standard deviation.

If you take the **weight** column from the data set above, the first value is 790, and the scaled value will be:

(790 - [1292.23](https://www.w3schools.com/python/trypandas.asp?filename=demo_ml_scale_mean1)) / [238.74](https://www.w3schools.com/python/trypandas.asp?filename=demo_ml_scale_std1) = -2.1

If you take the **volume** column from the data set above, the first value is 1.0, and the scaled value will be:

(1.0 - 1.61) / 0.38 = -1.59

Now you can compare -2.1 with -1.59 instead of comparing 790 with 1.0.

You do not have to do this manually, the Python sklearn module has a method called StandardScaler() which returns a Scaler object with methods for transforming data sets.

**Example**

Scale all values in the Weight and Volume columns:

|  |
| --- |
| import pandas from sklearn import linear\_model from sklearn.preprocessing import StandardScaler scale = StandardScaler()  df = pandas.read\_csv("data.csv")  X = df[['Weight', 'Volume']]  scaledX = scale.fit\_transform(X)  print(scaledX) |

**Result:**

Note that the first two values are -2.1 and -1.59, which corresponds to our calculations:

|  |
| --- |
| [[-2.10389253 -1.59336644]  [-0.55407235 -1.07190106]  [-1.52166278 -1.59336644]  [-1.78973979 -1.85409913]  [-0.63784641 -0.28970299]  [-1.52166278 -1.59336644]  [-0.76769621 -0.55043568]  [ 0.3046118 -0.28970299]  [-0.7551301 -0.28970299]  [-0.59595938 -0.0289703 ]  [-1.30803892 -1.33263375]  [-1.26615189 -0.81116837]  [-0.7551301 -1.59336644]  [-0.16871166 -0.0289703 ]  [ 0.14125238 -0.0289703 ]  [ 0.15800719 -0.0289703 ]  [ 0.3046118 -0.0289703 ]  [-0.05142797 1.53542584]  [-0.72580918 -0.0289703 ]  [ 0.14962979 1.01396046]  [ 1.2219378 -0.0289703 ]  [ 0.5685001 1.01396046]  [ 0.3046118 1.27469315]  [ 0.51404696 -0.0289703 ]  [ 0.51404696 1.01396046]  [ 0.72348212 -0.28970299]  [ 0.8281997 1.01396046]  [ 1.81254495 1.01396046]  [ 0.96642691 -0.0289703 ]  [ 1.72877089 1.01396046]  [ 1.30990057 1.27469315]  [ 1.90050772 1.01396046]  [-0.23991961 -0.0289703 ]  [ 0.40932938 -0.0289703 ]  [ 0.47215993 -0.0289703 ]  [ 0.4302729 2.31762392]] |

**Predict CO2 Values**

When the data set is scaled, you will have to use the scale when you predict values:

**Example**

Predict the CO2 emission from a 1.3 liter car that weighs 2300 kilograms:

|  |
| --- |
| import pandas from sklearn import linear\_model from sklearn.preprocessing import StandardScaler scale = StandardScaler()  df = pandas.read\_csv("data.csv")  X = df[['Weight', 'Volume']] y = df['CO2']  scaledX = scale.fit\_transform(X)  regr = linear\_model.LinearRegression() regr.fit(scaledX, y)  scaled = scale.transform([[2300, 1.3]])  predictedCO2 = regr.predict([scaled[0]]) print(predictedCO2) |

**Result:**

|  |
| --- |
| [107.2087328] |

## 4. Train/Test

Evaluate Your Model

In Machine Learning we create models to predict the outcome of certain events, like in the previous chapter where we predicted the CO2 emission of a car when we knew the weight and engine size.

To measure if the model is good enough, we can use a method called Train/Test.

Train/Test is a method to measure the accuracy of your model.

It is called Train/Test because you split the data set into two sets: a training set and a testing set.

*80% for training, and 20% for testing.*

You *train* the model using the training set.

You *test* the model using the testing set.

*Train* the model means *create* the model.

*Test* the model means test the accuracy of the model.

Start With a Data Set

Start with a data set you want to test.

Our data set illustrates 100 customers in a shop, and their shopping habits.

**Example**

|  |
| --- |
| import numpy import matplotlib.pyplot as plt numpy.random.seed(2)  x = numpy.random.normal(3, 1, 100) y = numpy.random.normal(150, 40, 100) / x  plt.scatter(x, y) plt.show() |

**Result:**

The x axis represents the number of minutes before making a purchase.

The y axis represents the amount of money spent on the purchase.

A graph with blue dots

AI-generated content may be incorrect.

## 5. Rule Based Classification

### 5.1 Introdution

Rule-Based Classification is a method in data mining where the classification model is represented as a set of conditional rules, usually in the form **IF - THEN**

Each rule describes a specific condition on input attributes and assigns a class label accordingly, for example:

* IF weather = sunny AND humidity = high THEN play = no

This method stands out for its interpretability and ease of implementation, making it suitable for domains that require transparency in decision-making such as healthcare, finance, and education

Rules can be manually crafted based on expert knowledge or automatically generated from training data using machine learning algorithms

### 5.2. Forms of Classification Rules

**1. Single Rule**

A single rule defines one specific condition and leads to a unique class prediction

Example:

* IF age < 30 AND student = yes THEN buys\_computer = yes

The rule is applied to data instances, and if the condition matches, the class label is assigned

**2. Rule Set**

A rule set consists of multiple IF - THEN rules applied in sequence or based on priority

Example:

* IF age < 30 AND income = medium THEN buys\_computer = yes
* IF age > 30 AND student = no THEN buys\_computer = no
* IF student = yes THEN buys\_computer = yes

Using a rule set helps cover a broader and more complex data space

### 5.3. Rule Structure

Each rule generally has two parts:

* **IF part (Antecedent)**: Contains one or more attribute conditions, connected using logical AND or OR
* **THEN part (Consequent)**: Specifies the class label if the conditions are satisfied

Example:

* IF income = high AND credit\_rating = excellent THEN approve\_loan = no

### 5.4. Common Rule Induction Algorithms

**1. RIPPER Algorithm**

RIPPER stands for Repeated Incremental Pruning to Produce Error Reduction

Basic process:

* Select a target class
* Generate rules that cover examples of that class
* Apply pruning to simplify the rule and remove unnecessary conditions
* Remove correctly classified examples
* Repeat the process for remaining examples

RIPPER is efficient for large datasets and often produces accurate and readable rules

**2. CN2 Algorithm**

CN2 uses a beam search to discover high-quality rules

Steps include:

* Search the condition space using a beam width
* At each step, select promising conditions based on accuracy or reliability
* Form rules, prune them, and repeat

CN2 is flexible and works well with categorical attributes

**3. Rule Extraction from Decision Trees**

Each path from the root to a leaf in a decision tree can be turned into an IF - THEN rule

For example, from a tree:

* IF outlook = sunny AND humidity = high THEN play = no
* IF outlook = sunny AND humidity = normal THEN play = yes
* IF outlook = overcast THEN play = yes

This method produces understandable rules by nature of the tree structure

### 5.5. Rule Evaluation Metrics

Two key metrics are used to evaluate rules:

* **Accuracy**: The proportion of correctly classified examples among those covered by the rule
* **Coverage**: The proportion of data instances the rule applies to in the entire dataset

Formulas:

* Accuracy = Correctly classified instances / Instances covered by the rule
* Coverage = Instances covered by the rule / Total number of instances

A good rule typically balances high accuracy with sufficient coverage, though trade-offs may be necessary

### 5.6. Illustrative Example

Given the dataset:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Age** | **Income** | **Student** | **Credit\_rating** | **Buys\_computer** |
| <=30 | High | No | Fair | No |
| <=30 | High | No | Exellent | No |
| 31..40 | High | No | Fair | Yes |
| > 40 | Medium | No | Fair | Yes |
| > 40 | Low | Yes | Fair | Yes |

Generated rules:

* IF age = 31..40 AND income = high THEN buys\_computer = yes
* IF student = no AND credit\_rating = excellent THEN buys\_computer = no
* IF age > 40 AND credit\_rating = fair THEN buys\_computer = yes

Python Implementation Example

**Simple rule-based classifier in Python:**

|  |
| --- |
| import pandas as pd  # Step 1: Define a small dataset (customer attributes and whether they buy a computer)  data = [  {'age': 'young', 'income': 'high', 'student': 'no', 'credit\_rating': 'fair', 'buys\_computer': 'no'},  {'age': 'young', 'income': 'high', 'student': 'no', 'credit\_rating': 'excellent', 'buys\_computer': 'no'},  {'age': 'middle\_aged', 'income': 'high', 'student': 'no', 'credit\_rating': 'fair', 'buys\_computer': 'yes'},  {'age': 'senior', 'income': 'medium', 'student': 'no', 'credit\_rating': 'fair', 'buys\_computer': 'yes'},  {'age': 'young', 'income': 'medium', 'student': 'yes', 'credit\_rating': 'fair', 'buys\_computer': 'yes'}  ]  df = pd.DataFrame(data)  # Step 2: Define simple rules  rules = [  {  'conditions': {'age': 'middle\_aged'},  'class': 'yes'  },  {  'conditions': {'student': 'yes'},  'class': 'yes'  },  {  'conditions': {'age': 'young', 'income': 'high', 'student': 'no'},  'class': 'no'  }  ]  # Step 3: Function to classify a row based on rules  def classify(sample, rules):  for rule in rules:  if all(sample.get(attr) == value for attr, value in rule['conditions'].items()):  return rule['class']  return 'Unknown'  # Step 4: Apply the classifier to the dataset  df['predicted'] = df.apply(lambda row: classify(row, rules), axis=1)  # Step 5: Evaluate and display results  correct = (df['predicted'] == df['buys\_computer']).sum()  total = len(df)  accuracy = correct / total  print("=== Classified Data ===")  print(df[['age', 'income', 'student', 'credit\_rating', 'buys\_computer', 'predicted']])  print("\n=== Evaluation ===")  print(f"Accuracy: {accuracy:.2f}") |

**Result :**

|  |
| --- |
|  |

### 5.8 Reference :

1. [Rule Based Classification](https://www.kaggle.com/code/cansinkutlucan/rule-based-classification)

2. [Rule-Based Classification in Data Mining - Tpoint Tech](https://www.tpointtech.com/rule-based-classification-in-data-mining)

## 6. K-Nearest Neighbors (KNN) Classification Using Euclidean Distance

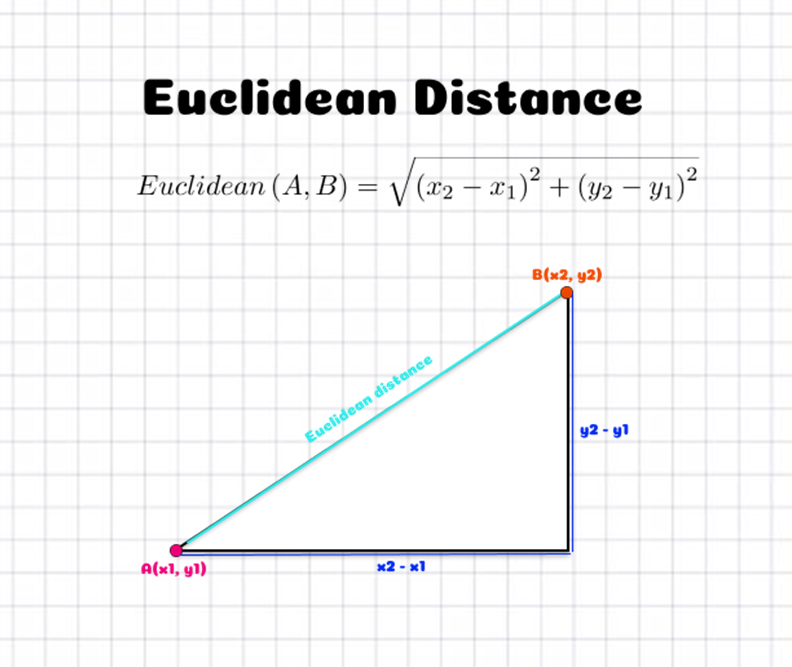
### 6.1 Introdution

Our behavior is shaped by the companions we grew up with. Our parents also shape our personalities in various ways. If you grow up among folks who enjoy sports, it is highly likely that you will end up loving sports. There are of course exceptions. KNN works similarly.

1. If you have a close buddy and spend most of your time with him/her, you will end up having similar interests and loving same things. That is kNN with k=1.
2. If you constantly hang out with a group of 5, each one in the group has an impact on your behavior and you will end up becoming the average of 5. That is kNN with k=5.

kNN classifier identifies the class of a data point using the majority voting principle. If k is set to 5, the classes of 5 nearest points are examined. Prediction is done according to the predominant class. Similarly, kNN regression takes the mean value of 5 nearest locations.

Do we witness folks who are close but how data points are considered to be close? The distance between data points is measured. There are various techniques to estimate the distance. Euclidean distance (Minkowski distance with p=2) is one of the most regularly used distance measurements. The graphic below explains how to compute the euclidean distance between two points in a 2-dimensional space. It is determined using the square of the difference between x and y coordinates of the locations.



### 6.2 Imlementation of KNN in Python

**Step 1 : Importing the modules**

To get started, we first import the necessary Python libraries. These libraries include NumPy for numerical operations, Pandas for handling tabular data, Matplotlib for data visualization, and several components from the scikit-learn library for model building and evaluation.

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.datasets import make\_blobs  from sklearn.neighbors import KNeighborsClassifier  from sklearn.model\_selection import train\_test\_split |

**Step2 : Importing/Creating a Dataset**

Instead of using a built-in dataset, we will manually create a small and simple dataset with two features and two classes. This helps us understand exactly how the KNN algorithm works without the complexity of large datasets.

|  |
| --- |
| data = {  'X1': [1, 2, 3, 6, 7, 8],  'X2': [1, 1, 2, 6, 7, 8],  'Label': [0, 0, 0, 1, 1, 1]  }  df = pd.DataFrame(data)  X = df[['X1', 'X2']]  y df['Label'] |

Here, the dataset has six data points. The first three belong to class 0, and the last three belong to class 1. The features X1 and X2 represent the coordinates of the data points in a 2D space.

**Step 3 – Visualizing the Dataset**

Before training the model, it is helpful to visualize the data points. We use a scatter plot to show how the two classes are distributed in the 2D space.

|  |
| --- |
| plt.figure(figsize=(6,6))  colors = ['red' if label == 0 else 'blue' for label in y]  plt.scatter(X['X1'], X['X2'], c=colors, s=100, edgecolor='k')  plt.title("Original Data Points")  plt.xlabel("X1")  plt.ylabel("X2")  plt.grid(True)  plt.show() |

The red points represent class 0, and the blue points represent class 1. This visualization helps us see the spatial separation between the two classes, which is exactly what the KNN algorithm relies on.

**Step 4 – Splitting the Data into Training and Test Sets**

To evaluate our model properly, we divide the dataset into a training set and a test set. The training set is used to fit the model, and the test set is used to evaluate how well the model performs on unseen data.

|  |
| --- |
| X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42) |

Here, we are using 67 percent of the data for training and 33 percent for testing.

**Step 5 – Training the KNN Model**

We now create an instance of the KNN classifier from scikit-learn. We choose k = 3, meaning the algorithm will consider the three nearest neighbors of a data point to determine its class.

|  |
| --- |
| knn = KNeighborsClassifier(n\_neighbors=3)  knn.fit(X\_train, y\_train) |

After this step, our model is trained and ready to make predictions.

**Step 6 – Making Predictions and Evaluating the Model**

We use the trained model to predict the class labels of the test data. Then we calculate the accuracy and confusion matrix to evaluate the performance.

|  |
| --- |
| y\_pred = knn.predict(X\_test)  accuracy = accuracy\_score(y\_test, y\_pred)  print("KNN Accuracy:", accuracy \* 100)  cm = confusion\_matrix(y\_test, y\_pred)  print("Confusion Matrix:\n", cm) |

The accuracy score tells us how many predictions the model got correct, and the confusion matrix provides detailed information about true positives, false positives, true negatives, and false negatives.

**Step 7 – Predicting a New Data Point (Optional)**

We can also test the model with a new data point that was not part of the original dataset. This shows how the KNN model generalizes to unseen data.

|  |
| --- |
| new\_point = np.array([[5, 5]])  pred = knn.predict(new\_point)  print(f"The predicted class for point {new\_point[0]} is: {pred[0]}") |

This prediction is based on the three nearest neighbors of the new point and the majority class among them.

**Step 8 – Visualizing KNN Decision Boundaries**

To better understand how KNN classifies new data, we can create a decision boundary plot. This plot shows the regions in the feature space that the model classifies as either class 0 or class 1.

|  |
| --- |
| h = 0.1  x\_min, x\_max = X['X1'].min() - 1, X['X1'].max() + 1  y\_min, y\_max = X['X2'].min() - 1, X['X2'].max() + 1  xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),  np.arange(y\_min, y\_max, h))  Z = knn.predict(np.c\_[xx.ravel(), yy.ravel()])  Z = Z.reshape(xx.shape)  plt.figure(figsize=(8,6))  plt.contourf(xx, yy, Z, cmap=plt.cm.RdBu, alpha=0.3)  plt.scatter(X['X1'], X['X2'], c=colors, s=100, edgecolor='k')  plt.title("KNN Decision Boundary")  plt.xlabel("X1")  plt.ylabel("X2")  plt.grid(True)  plt.show() |

This visualization gives a clear understanding of how the KNN model separates the classes in the feature space. The boundaries are not linear but depend on the position of nearby training points.

### 6.3 Reference :

1. <https://www.digitalocean.com/community/tutorials/k-nearest-neighbors-knn-in-python>

2. [KNN Classifier in Python: Implementation, Features and Application](https://www.analyticsvidhya.com/blog/2021/01/a-quick-introduction-to-k-nearest-neighbor-knn-classification-using-python/)

3. [KNN Algorithm – K-Nearest Neighbors Classifiers and Model Example](https://www.freecodecamp.org/news/k-nearest-neighbors-algorithm-classifiers-and-model-example/)

4. [The k-Nearest Neighbors (kNN) Algorithm in Python – Real Python](https://realpython.com/knn-python/)

## 7. Confusion Matrix

It is a table that is used in classification problems to assess where errors in the model were made.

The rows represent the actual classes the outcomes should have been. While the columns represent the predictions we have made. Using this table it is easy to see which predictions are wrong.

Creating a Confusion Matrix

Confusion matrixes can be created by predictions made from a logistic regression.

For now we will generate actual and predicted values by utilizing NumPy:

|  |
| --- |
| import numpy |

Next we will need to generate the numbers for "actual" and "predicted" values.

|  |
| --- |
| actual = numpy.random.binomial(1, 0.9, size = 1000) predicted = numpy.random.binomial(1, 0.9, size = 1000) |

In order to create the confusion matrix we need to import metrics from the sklearn module.

|  |
| --- |
| from sklearn import metrics |

Once metrics is imported we can use the confusion matrix function on our actual and predicted values.

|  |
| --- |
| confusion\_matrix = metrics.confusion\_matrix(actual, predicted) |

To create a more interpretable visual display we need to convert the table into a confusion matrix display.

|  |
| --- |
| cm\_display = metrics.ConfusionMatrixDisplay(confusion\_matrix = confusion\_matrix, display\_labels = [0, 1]) |

Vizualizing the display requires that we import pyplot from matplotlib.

|  |
| --- |
| import matplotlib.pyplot as plt |

Finally to display the plot we can use the functions plot() and show() from pyplot.

|  |
| --- |
| cm\_display.plot() plt.show() |

See the whole example in action:

Example

|  |
| --- |
| import matplotlib.pyplot as plt import numpy from sklearn import metrics  actual = numpy.random.binomial(1,.9,size = 1000) predicted = numpy.random.binomial(1,.9,size = 1000)  confusion\_matrix = metrics.confusion\_matrix(actual, predicted)  cm\_display = metrics.ConfusionMatrixDisplay(confusion\_matrix = confusion\_matrix, display\_labels = [0, 1])  cm\_display.plot() plt.show() |

Result

A chart with a few colored squares

AI-generated content may be incorrect.

**Results Explained**

The Confusion Matrix created has four different quadrants:

True Negative (Top-Left Quadrant)  
False Positive (Top-Right Quadrant)  
False Negative (Bottom-Left Quadrant)  
True Positive (Bottom-Right Quadrant)

True means that the values were accurately predicted, False means that there was an error or wrong prediction.

Now that we have made a Confusion Matrix, we can calculate different measures to quantify the quality of the model. First, lets look at Accuracy.

**Created Metrics**

The matrix provides us with many useful metrics that help us to evaluate our classification model.

The different measures include: Accuracy, Precision, Sensitivity (Recall), Specificity, and the F-score, explained below.

**Accuracy**

Accuracy measures how often the model is correct.

How to Calculate :

(True Positive + True Negative) / Total Predictions

Example

|  |
| --- |
| Accuracy = metrics.accuracy\_score(actual, predicted) |

**Precision**

Of the positives predicted, what percentage is truly positive?

How to Calculate

True Positive / (True Positive + False Positive)

Precision does not evaluate the correctly predicted negative cases:

Example

|  |
| --- |
| Precision = metrics.precision\_score(actual, predicted) |

**Sensitivity (Recall)**

Of all the positive cases, what percentage are predicted positive?

Sensitivity (sometimes called Recall) measures how good the model is at predicting positives.

This means it looks at true positives and false negatives (which are positives that have been incorrectly predicted as negative).

How to Calculate

True Positive / (True Positive + False Negative)

Sensitivity is good at understanding how well the model predicts something is positive:

Example

|  |
| --- |
| Sensitivity\_recall = metrics.recall\_score(actual, predicted) |

**Specificity**

How well the model is at prediciting negative results?

Specificity is similar to sensitivity, but looks at it from the persepctive of negative results.

How to Calculate

True Negative / (True Negative + False Positive)

Since it is just the opposite of Recall, we use the recall\_score function, taking the opposite position label:

Example

|  |
| --- |
| Specificity = metrics.recall\_score(actual, predicted, pos\_label=0) |

**F-score**

F-score is the "harmonic mean" of precision and sensitivity.

It considers both false positive and false negative cases and is good for imbalanced datasets.

How to Calculate

2 \* ((Precision \* Sensitivity) / (Precision + Sensitivity))

This score does not take into consideration the True Negative values:

Example

|  |
| --- |
| F1\_score = metrics.f1\_score(actual, predicted) |

All calulations in one:

Example

|  |
| --- |
| #metrics print({"Accuracy":Accuracy,"Precision":Precision,"Sensitivity\_recall":Sensitivity\_recall,"Specificity":Specificity,"F1\_score":F1\_score}) |

## 8. Decision Tree

A diagram of a scientific experiment

AI-generated content may be incorrect.

In this chapter we will show you how to make a "Decision Tree". A Decision Tree is a Flow Chart, and can help you make decisions based on previous experience.

In the example, a person will try to decide if he/she should go to a comedy show or not.

Luckily our example person has registered every time there was a comedy show in town, and registered some information about the comedian, and also registered if he/she went or not.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Age** | **Experience** | **Rank** | **Nationality** | **Go** |
| 36 | 10 | 9 | UK | NO |
| 42 | 12 | 4 | USA | NO |
| 23 | 4 | 6 | N | NO |
| 52 | 4 | 4 | USA | NO |
| 43 | 21 | 8 | USA | YES |
| 44 | 14 | 5 | UK | NO |
| 66 | 3 | 7 | N | YES |
| 35 | 14 | 9 | UK | YES |
| 52 | 13 | 7 | N | YES |
| 35 | 5 | 9 | N | YES |
| 24 | 3 | 5 | USA | NO |
| 18 | 3 | 7 | UK | YES |
| 45 | 9 | 9 | UK | YES |

Now, based on this data set, Python can create a decision tree that can be used to decide if any new shows are worth attending to.

First, read the dataset with pandas:

**Example**

Read and print the data set:

|  |
| --- |
| import pandas  df = pandas.read\_csv("data.csv")  print(df) |

To make a decision tree, all data has to be numerical.

We have to convert the non numerical columns 'Nationality' and 'Go' into numerical values.

Pandas has a map() method that takes a dictionary with information on how to convert the values.

{'UK': 0, 'USA': 1, 'N': 2}

Means convert the values 'UK' to 0, 'USA' to 1, and 'N' to 2.

**Example**

Change string values into numerical values:

|  |
| --- |
| d = {'UK': 0, 'USA': 1, 'N': 2} df['Nationality'] = df['Nationality'].map(d) d = {'YES': 1, 'NO': 0} df['Go'] = df['Go'].map(d)  print(df) |

Then we have to separate the *feature* columns from the *target* column.

The feature columns are the columns that we try to predict *from*, and the target column is the column with the values we try to predict.

**Example**

*X is the feature columns, y is the target column:*

|  |
| --- |
| features = ['Age', 'Experience', 'Rank', 'Nationality']  X = df[features] y = df['Go']  print(X) print(y) |

Now we can create the actual decision tree, fit it with our details. Start by importing the modules we need:

**Example**

*Create and display a Decision Tree:*

|  |
| --- |
| import pandas from sklearn import tree from sklearn.tree import DecisionTreeClassifier import matplotlib.pyplot as plt  df = pandas.read\_csv("data.csv")  d = {'UK': 0, 'USA': 1, 'N': 2} df['Nationality'] = df['Nationality'].map(d) d = {'YES': 1, 'NO': 0} df['Go'] = df['Go'].map(d)  features = ['Age', 'Experience', 'Rank', 'Nationality']  X = df[features] y = df['Go']  dtree = DecisionTreeClassifier() dtree = dtree.fit(X, y)  tree.plot\_tree(dtree, feature\_names=features) |

**Result Explained**

The decision tree uses your earlier decisions to calculate the odds for you to wanting to go see a comedian or not.

Let us read the different aspects of the decision tree:

A diagram of a mathematical equation

AI-generated content may be incorrect.

**Rank**

**Rank <= 6.5** means that every comedian with a rank of 6.5 or lower will follow the True arrow (to the left), and the rest will follow the False arrow (to the right).

**gini = 0.497** refers to the quality of the split, and is always a number between 0.0 and 0.5, where 0.0 would mean all of the samples got the same result, and 0.5 would mean that the split is done exactly in the middle.

**samples = 13** means that there are 13 comedians left at this point in the decision, which is all of them since this is the first step.

**value = [6, 7]** means that of these 13 comedians, 6 will get a "NO", and 7 will get a "GO".

**Gini**

There are many ways to split the samples, we use the GINI method in this tutorial.

The Gini method uses this formula:

Gini = 1 - (x/n)2 - (y/n)2

Where x is the number of positive answers("GO"), n is the number of samples, and y is the number of negative answers ("NO"), which gives us this calculation:

1 - (7 / 13)2 - (6 / 13)2 = 0.497

A diagram of a algorithm

AI-generated content may be incorrect.

The next step contains two boxes, one box for the comedians with a 'Rank' of 6.5 or lower, and one box with the rest.

**True - 5 Comedians End Here:**

**gini = 0.0** means all of the samples got the same result.

**samples = 5** means that there are 5 comedians left in this branch (5 comedian with a Rank of 6.5 or lower).

**value = [5, 0]** means that 5 will get a "NO" and 0 will get a "GO".

**False - 8 Comedians Continue:**

**Nationality**

**Nationality <= 0.5** means that the comedians with a nationality value of less than 0.5 will follow the arrow to the left (which means everyone from the UK, ), and the rest will follow the arrow to the right.

**gini = 0.219** means that about 22% of the samples would go in one direction.

**samples = 8** means that there are 8 comedians left in this branch (8 comedian with a Rank higher than 6.5).

**value = [1, 7]** means that of these 8 comedians, 1 will get a "NO" and 7 will get a "GO".

A diagram of a mathematical equation

AI-generated content may be incorrect.

**True - 4 Comedians Continue:**

**Age**

**Age <= 35.5** means that comedians at the age of 35.5 or younger will follow the arrow to the left, and the rest will follow the arrow to the right.

**gini = 0.375** means that about 37,5% of the samples would go in one direction.

**samples = 4** means that there are 4 comedians left in this branch (4 comedians from the UK).

**value = [1, 3]** means that of these 4 comedians, 1 will get a "NO" and 3 will get a "GO".

**False - 4 Comedians End Here:**

**gini = 0.0** means all of the samples got the same result.

**samples = 4** means that there are 4 comedians left in this branch (4 comedians not from the UK).

**value = [0, 4]** means that of these 4 comedians, 0 will get a "NO" and 4 will get a "GO".

A diagram of a algorithm

AI-generated content may be incorrect.

**True - 2 Comedians End Here:**

**gini = 0.0** means all of the samples got the same result.

**samples = 2** means that there are 2 comedians left in this branch (2 comedians at the age 35.5 or younger).

**value = [0, 2]** means that of these 2 comedians, 0 will get a "NO" and 2 will get a "GO".

**False - 2 Comedians Continue:**

**Experience**

**Experience <= 9.5** means that comedians with 9.5 years of experience, or less, will follow the arrow to the left, and the rest will follow the arrow to the right.

**gini = 0.5** means that 50% of the samples would go in one direction.

**samples = 2** means that there are 2 comedians left in this branch (2 comedians older than 35.5).

**value = [1, 1]** means that of these 2 comedians, 1 will get a "NO" and 1 will get a "GO".

A diagram of a sample

AI-generated content may be incorrect.

**True - 1 Comedian Ends Here:**

**gini = 0.0** means all of the samples got the same result.

**samples = 1** means that there is 1 comedian left in this branch (1 comedian with 9.5 years of experience or less).

**value = [0, 1]** means that 0 will get a "NO" and 1 will get a "GO".

**False - 1** Comedian Ends Here:

**gini = 0.0** means all of the samples got the same result.

**samples = 1** means that there is 1 comedians left in this branch (1 comedian with more than 9.5 years of experience).

**value = [1, 0]** means that 1 will get a "NO" and 0 will get a "GO".

#### Predict Values

We can use the Decision Tree to predict new values.

**Example:** Should I go see a show starring a 40 years old American comedian, with 10 years of experience, and a comedy ranking of 7?

**Example**

Use predict() method to predict new values:

|  |
| --- |
| print(dtree.predict([[40, 10, 7, 1]])) |

**Result:**

|  |
| --- |
| **[1]** |

**Example**

What would the answer be if the comedy rank was 6?

|  |
| --- |
| print(dtree.predict([[40, 10, 6, 1]])) |

**Result :**

|  |
| --- |
| **[0]** |