



# **Data Analysis with Python**

## **Statistics and Machine Learning**

Volodymyr Kravchenko



# python

## Topic 1

### Exploratory Data Analysis, Cleaning and Visualization with Python



# Section 1.1

## **Python Programming Language as a Tool for Data Mining and Analytics**

# Python

## Overview of Built-in Data Structures

- **Lists** are one of the main built-in data types that can contain any other data type in Python.

- Creating a list:

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

- **Tuples** are a built-in data type used to store a sequence of values that cannot be changed.

- Creating a tuple:

```
tuple = (1, 2, 3, 4, 5)
```

- **Sets** are a built-in data type used to store unique values .

- Creating a set:

```
set = {1, 2, 3, 4, 5}
```

- **Dictionaries** are a built-in data type used to store key-value pairs .

- Creating a dictionary:

```
dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

- **Numeric data types** – int, float, complex

- Python has several built-in numeric data types, including integers (int), floating-point numbers (float), and complex numbers (complex).
- Creating a variable with a numeric value:

```
num = 5, float_num = 5.0, complex_num = 5 + 2j
```

- **Strings** are a built-in data type used to store textual information.

- Creating a string:

```
str = "Hello, World!"
```



# Python

## Working with Lists

- **Creating a list**

- Example of creating a list: `my_list = [1, 2, 3, 4, 5]`

- **Accessing elements of a list**

- Examples of accessing list elements by index: `my_list[0]`, `my_list[1]`
- Examples of accessing list elements by slice: `my_list[1:3]`, `my_list[:3]`, `my_list[3:]`

- **Changing list elements**

- Example of changing the value of a list element: `my_list[0] = 10`
- Example of adding an element to the list: `my_list.append(6)`
- Example of removing an element from the list: `my_list.remove(3)`

- **Sorting the list**

- Example of sorting the list in ascending order: `my_list.sort()`
- Example of sorting the list in descending order: `my_list.sort(reverse=True)`

- **Iterating over the list**

```
for item in my_list:  
    print(item)
```

- **Searching for elements in the list**

- Example of finding the index of an element in the list: `my_list.index(3)`
- Example of checking for the presence of an element in the list: `3 in my_list`

- **Adding elements to a list using the .extend() method**

```
my_list = [1, 2, 3]  
my_list.extend([4, 5, 6])  
print(my_list)
```

- **Reversing a list using the .reverse() method:**

```
my_list = [1, 2, 3, 4, 5]  
my_list.reverse()  
print(my_list)
```

- **Finding the index of an element in a list using the .index() method:**

```
my_list = [1, 2, 3, 4, 5]  
print(my_list.index(3))
```

- **Counting the occurrences of an element in a list using the .count() method:**

```
my_list = [1, 2, 3, 3, 4, 5]  
print(my_list.count(3))
```

- **Copying a list using the .copy() method:**

```
my_list = [1, 2, 3]  
my_list_copy = my_list.copy()  
print(my_list_copy)
```

- **zip()**



# Python

## Expression for Generating Lists and Two-Dimensional Lists (working with matrices)

- **Creating a list using “for”**

- Example of creating a list using for: `my_list = [i for i in range(10)]`

- **List comprehension for generating lists:**

- List comprehension is a concise way to write a loop that allows you to generate a new list based on an existing list. List comprehension can be useful when processing large volumes of data, when you need to quickly create a new list based on an old one.

```
new_list = [i*2 for i in my_list if i > 2]
```

- **Creating a list of squares of integers from 1 to 10:**

```
my_list = [i**2 for i in range(1, 11)]
```

- **Creating a list of only even numbers from 1 to 10:**

```
my_list = [i for i in range(1, 11) if i % 2 == 0]
```

- **Creating a list of uppercase letters from a string:**

```
my_list = [char.upper() for char in 'hello world']
```

- **Creating a list where each element is a list of three zeros:**

```
my_list = [[0, 0, 0] for i in range(3)]
```

- **Creating a list where each element is a string consisting of several asterisks, the number of which increases with each subsequent element:**

```
my_list = ['*' * i for i in range(1, 6)]
```

- **An array is a two-dimensional list commonly used to represent tables or images. Matrices can be created using nested lists.**

```
matrix = [[1, 2, 3],  
          [4, 5, 6],  
          [7, 8, 9]]
```

- **Accessing Array Elements: Array elements can be accessed using subscripts rows and columns.**

- `matrix = [[1, 2, 3],`
- `[4, 5, 6],`
- `[7, 8, 9]]`
- `# Access to the element in the first row and second column element = matrix[0][1]`

- **Unfolding is the process of converting nested lists into a one-dimensional list. It can be useful when processing large amounts of data or when comparing lists.**

- `nested_list = [[1, 2], [3, 4], [5, 6]]`
- `# Expand the nested list into a one-dimensional list`
- `flat_list = [item for sublist in nested_list for item in sublist]`

- **Nested lists can have standard methods such as `.append()`, `.extend()`, `.remove()`, and others.**

- `nested_list = [[1, 2], [3, 4], [5, 6]]`
- `#Add an element to the nested list nested_list[0].append(3)`
- `# Remove an element from the nested list nested_list[1].remove(4)`

# Python

## Advantages and Disadvantages of Lists

- **Advantages of lists:**
  - Versatility: Lists can contain any type of data, including other lists and tuples.
  - Flexibility: Lists allow you to change the content, size and order of elements at any time.
  - Convenience: Lists have many methods that allow you to perform various operations such as adding, removing, sorting, and filtering items.
- **Disadvantages of lists:**
  - Slow performance for large amounts of data: Because lists can take up a lot of memory and have a large number of elements, they can be slow to process large amounts of data.
  - Iteration: Iterating over a list can be slow, especially if you need to iterate over very large lists.
  - Poor performance for some operations: Some operations, such as searching for elements in a list, can be slow for large lists.

# Python

## Tuples: Definition, Advantages and Disadvantages

**Tuples** are collections that look like lists but are **unchangeable** and **immutable**.

- **Creation and initialization of tuples**

```
my_tuple = (1, 2, 3)
```

- **Access tuple elements**

```
print(my_tuple[0])
```

- **Operations on tuples**

```
my_tuple1 = (1, 2, 3)
```

```
my_tuple2 = (4, 5, 6)
```

```
my_tuple3 = my_tuple1 + my_tuple2
```

- **Example of using the .count() method**

```
my_tuple = (1, 2, 3, 3, 4, 5)
```

```
print(my_tuple.count(3))
```

### Advantages of tuples:

- **Immutability:** Tuples are immutable, meaning they cannot be changed once they are created. This can be useful in cases where data persistence is required.
- **Speed:** Since tuples cannot be changed, they can be stored in memory more efficiently, allowing large amounts of data to be processed more quickly.
- **Error-proof:** Since tuples cannot be changed, they protect against errors related to changing data in the wrong place.

### Disadvantages of tuples

- **Immutability:** Since tuples are immutable, it is not possible to change the contents of a tuple without creating a new one.
- **Limited functionality:** Since tuples do not have methods like `append()`, `extend()`, or `remove()`, their functionality is limited.
- **Data type limitations:** Since tuples are designed to store immutable data, they are not suitable for storing mutable data types like lists.

# Python

## Sets: Definition, Advantages and Disadvantages

- **Sets** are collections of unique and unordered items

```
my_set = {1, 2, 3}
```

- **Access to set elements**

```
print(2 in my_set):  
True
```

- An example of using the **.add()** method to **add an element to a set**

```
my_set.add(4)  
my_set.add('a')
```

- **Advantages of sets:**

- **Uniqueness:** Sets contain only unique elements, ensuring uniqueness of data and preventing duplication.
- **Speed:** Since sets have no ordered structure, they allow you to quickly check for the presence of elements.
- **Set Operations:** Sets allow you to perform set operations such as intersection, union, and difference.

- **Disadvantages of sets:**

- **Unordered:** Since sets do not have an ordered structure, it is not possible to use indexes to access elements.
- **Functionality limitations:** Because sets do not have methods such as `append()`, `extend()`, or `remove()`, their functionality is limited.
- **Incompatibility with lists:** Sets cannot contain mutable data types such as lists, which limits their use in some cases.



# Python

## Dictionaries – dict()

**Dictionaries** are collections of key-value pairs where each key is unique and is used to access the corresponding value

- **Creation and initialization of dictionaries**

```
my_dict = {'key1': 'value1',  
           'key2': 'value2'}
```

- **Access to dictionary items**

```
print(my_dict['key1'])
```

- **Operations on dictionaries**

```
my_dict['key3'] = 'value3'  
del my_dict['key1']
```

- **Using the .keys() to get a list of keys:**

```
print(my_dict.keys())  
for K in my_dict: print(K)
```

- **Using the .values() to get a list of values:**

```
print(my_dict.values())
```

- **Using the .items() to get a list of key-value pairs:**

```
print(my_dict.items())
```

- **Checking values in a dictionary**

```
K in D
```

- **.update(dict):** Adds key-value pairs from another dictionary to the current dictionary.
- **.items():** Returns a list of key-value pairs in a dictionary.
- **.clear():** Removes all key-value pairs in the dictionary.
- **.copy():** Creates a copy of the dictionary.
- **.setdefault(key[, default]):** Returns the value corresponding to the key key. If the key is not found, a new key is created with the value default (if specified) and returns that value.
- **.fromkeys(seq[, value]):** Creates a new dictionary with keys taken from seq and values set to value (if specified).

### Advantages:

- Easy access to value with key
- Quick search and access to items
- Support for a wide range of operations (adding, deleting, changing elements, copying, resizing, etc.)
- Support for different types of keys and values
- Easily store key-value pairs in sorted order (by key)

### Disadvantages:

- Dictionaries can take up a lot of memory to store key-value pairs
- Search speed may degrade as the dictionary size increases
- Keys must be unique, otherwise old values will be overwritten
- Dictionaries do not preserve the order in which elements are added
- Dictionaries do not have built-in functions for sorting by value



# Python

## Data Types. Mathematical and Assignment Operators

- **Integers (int):** Integers from -2147483648 to 2147483647 (for 32-bit platform) or from -9223372036854775808 to 9223372036854775807 (for 64-bit platform).
  - **Long integers (long):** integers of any size that exceeds the range of integers.
  - **Real numbers (float):** Floating-point numbers represented in IEEE 754 floating-point format.
  - **Complex numbers (complex):** numbers consisting of real and imaginary parts.
  - **String (object):** Examples of values: "A", 'abc', "comp 2011", ...
  - **Boolean (bool):** Examples of values: True, False
- 
- **abs(x):** Returns the absolute value of x.
  - **pow(x, y):** Returns x raised to the power of y.
  - **round(x, n):** Rounds a number x to n decimal places.
  - **log(x):** Returns the natural logarithm of x.
  - **sum(iterable):** returns the sum of all elements in an iterable object.
  - **max(iterable):** returns the largest element in the iterable object.
  - **min(iterable):** returns the smallest element in the iterable object.
  - **len(x):** returns the length of the sequence passed as an argument (list, tuple, string, etc.).

| Math Operator | Description           |
|---------------|-----------------------|
| +             | Addition              |
| –             | Subtraction           |
| *             | Multiplication        |
| /             | Division              |
| //            | // Integer Division   |
| %             | % Modulus (Remainder) |
| **            | Power                 |

| Assignment Operator | Description               |
|---------------------|---------------------------|
| =                   | Assignment                |
| +=                  | Addition Assignment       |
| -=                  | Subtraction Assignment    |
| *=                  | Multiplication Assignment |
| /=                  | Division Assignment       |

# Python

## Application of Functions to Specific Element. Boolean Values

- Python has a built-in **map()** method that allows you to apply a certain function to each element in a list and return an iterator with the results.
- For example, if you want to increment each value in the list by 1, you can use the **map()** method with a function variable that takes each element of the list as an argument and returns the increment by 1:

```
numbers = [1, 2, 3, 4, 5]
new_numbers = list( map(lambda x: x + 1,
                        numbers) )
```

- In this example, **lambda** is a function that takes one argument **x** and returns **x + 1**. The result of the code execution is an iterator with the numbers [2, 3, 4, 5, 6].
- The **map()** method can be used with any function that takes a single argument, and it can also be used with standard Python functions such as **len()**, **str()**, **int()**, **float()**, etc.
- The **map()** method can be used not only with lists, but also with other iterable objects such as tuples, sets, dictionaries, and others.

- In Python, the logical data type is called **bool** and can take two values: **True** or **False**. This data type is used to represent **Boolean values** and the results of logical operations.
- **Boolean values** are used in conditional statements such as **if**, **while**, **for**, etc. Also, they are the result of performing a comparison between two objects, for example, **a == b** will return **True** if **a** is equal to **b**.
- Python also has logical operations: **and** (logical "AND"), **or** (logical "OR") and **not** (logical "NOT"). With their help, you can combine logical values and obtain more complex logical expressions.
- For example, if you want to check whether the number **x** is less than 10 and greater than 0, you can use the following expression:

```
x = 5
if x > 0 and x < 10:
    print("x is between 0 and 10") else:
    print("x is not between 0 and 10")
```

Here, **x > 0** and **x < 10** are boolean expressions that return **True** if **x** satisfies these conditions. By combining them via **and** operation, you check whether **x** satisfies both conditions.

- **bool(x)** – converts the **x** object into a logical data type. If **x** is empty, zero, or **False**, the function will return **False**, otherwise, **True**.
- **all(iterable)** – returns **True** if all elements in the iterable object are true, and **False** otherwise. If iterable is empty, the function will return **True**.
- **any(iterable)** – returns **True** if at least one element in the iterable object is true, and **False** otherwise. If iterable is empty, the function will return **False**.



# Python

## Missing Values (NaN, NULL). Math Library

- Python has a special value called NaN, which means "not a number" and is often used in floating point calculations where infinity or uncertainty arises.
- A NaN value can be created using the `float('nan')` function. A NaN value can also be returned as the result of an arithmetic operation that fails, such as dividing zero by zero.
- In Python, you can check if a variable contains a NaN value using the `math.isnan()` function, which returns True if the value is NaN and False otherwise. For example:

```
import math
x = float('nan')
if math.isnan(x):
    print("x is NaN")
else:
    print("x is not NaN")
```

- In this example, we check if the variable `x` contains a NaN value using the `math.isnan()` function. The function returns True because `x` contains a NaN value.

**Math** a built-in Python library that provides functions for mathematical calculations.

### Functions from the library Math:

- **`math.sqrt(x)`** – calculates the square root of a number `x`.
- **`math.sin(x)`** – calculates the sine of the number `x`.
- **`math.cos(x)`** – calculates the cosine of the number `x`.
- **`math.tan(x)`** – calculates the tangent of the number `x`.
- **`math.log(x)`** – calculates the natural logarithm of a number `x`.
- **`math.log10(x)`** – calculates the decimal logarithm of a number `x`.
- **`math.exp(x)`** – calculates the exponent of a number `x`.
- **`math.pow(x, y)`** – calculates the number `x` raised to the power of `y`.
- **`math.pi`** – a constant containing the number  $\pi$ .
- **`math.e`** – a constant containing a number `e`.

**Disadvantages** of the Math library: not all possibilities of the math library are applicable in certain tasks; possible inaccuracies due to rounding of values.

# Python

## Strings. Encoding

**String** are a sequence of characters enclosed in quotation marks. In Python, strings can be enclosed in single (') or double (") quotes.

### Basic operations with strings:

- `len(s)` - returns the length of the string `s`.
- `s1 + s2` - joins two lines `s1` and `s2`.
- `s * n` - repeats the string `sn` times.
- `s[i]` - returns the string character `s` at index `i`.
- `s[i:j]` - returns the substring of string `s` from index `i` to `j` (not inclusive).
- `s.upper()` - returns the string `s` in upper case.
- `s.lower()` - returns the string `s` in lower case.
- `s.strip()` - returns the string `s` with leading and trailing whitespace removed.
- `s.replace(old, new)` - returns the string `s` with all occurrences of `old` replaced by `new`.
- `s.find(sub)` - returns the index of the first occurrence of substring in string `s` or `-1` if `sub` is not found.
- `s.startswith(prefix)` - returns `True` if the string `s` starts with `prefix`, otherwise - `False`.
- `s.endswith(suffix)` - returns `True` if the string `s` ends with a `suffix`, otherwise - `False`.
- Character encoding is the process of converting characters into a bit code that can be stored in memory or transmitted over a network. In Python, character strings have a default encoding (usually UTF-8), but can be converted to other encodings using the **`encode()`** method.
- UTF-8 is one of the most common character encodings that can be used to represent the characters of most languages in the world. UTF-8 encodes each character using a sequence of bytes of variable length, which allows efficient use of memory.
- In Python, character encodings can be important for processing text files, working with databases, and for transferring data over a network. To ensure proper processing of character strings and character encoding, it is recommended to use the appropriate functions and methods in Python, such as **`encode()`**, **`decode()`**, **`str()`**, **`bytes()`**, **`open()`**, **`io.StringIO()`**, and others.
- If your character strings contain Cyrillic characters and you are working with files, it is recommended to use UTF-8, which supports Cyrillic and other characters. If you save a character string to a file with a `.txt` extension, you can open the file for writing like this:

```
with open('myfile.txt', 'w', encoding='utf-8') as f:  
    f.write('Hello world!')
```

This will ensure that the Cyrillic alphabet is correctly encoded in the file and saved.

# Character strings

## function `print()`

- The `print()` function outputs the value to the console, but it does not allow you to format the output. To output values with certain formatting, use the built-in `format()` function.
- The syntax of the `format()` function is as follows:

```
formatted_string = "My name is {} and I am {} years old.".format(name, age)
```

where `{}` is the place to insert the value, and `name` and `age` are the values to insert.

- You can limit the number of digits after the decimal point for floating-point numbers:

```
pi = 3.14159265359
formatted_string = "pi is {:.2f}.".format(pi)
```

- You can also use named arguments, which allow you to refer to parameters by their name rather than their position.  
For example:

```
name = "John"; age = 30
formatted_string = "My name is {name} and I am {age} years old.".format(name=name, age=age)
```

# NumPy

## Creating NumPy Arrays

- NumPy (short for "Numerical Python") is a language library Python for performing mathematical and numerical operations on
- arrays and matrices.
- NumPy is a key component of many scientific computing and workflows with data analysis, including machine learning, data visualization, and more.

```
import numpy as np
```

### # Converting a list to a NumPy array

```
my_list = [1, 2, 3, 4]
my_array = np.array(my_list)
print(my_array)
```

### # Converting a tuple to a NumPy array

```
my_tuple = (5, 6, 7, 8)
my_array = np.array(my_tuple)
print(my_array)
```

### # Creating an array of zeros

```
my_zeros_array = np.zeros((3, 4))
print(my_zeros_array)
```

### # Creating an array of ones

```
my_ones_array = np.ones((2, 5))
print(my_ones_array)
```

### # Creating an array with a range of values

```
my_range_array = np.arange(1, 10, 2)
print(my_range_array)
```

### # Creating an array with evenly spaced values

```
my_linspace_array = np.linspace(0, 10, 6)
print(my_linspace_array)
```

```
arr = np.array([1, 2, 3, 4, 5])
print(arr[0])
    Output: 1
print(arr[-1])
    Output: 5
```

```
arr = np.array([1, 2, 3, 4, 5])
print(arr[1:4])
    Output: [2 3 4]
```

```
arr = np.array([1, 2, 3, 4, 5])
bool_arr = arr > 3
print(bool_arr)
    Output: [False False False
            True  True]
```

```
new_arr = arr[bool_arr]
print(new_arr)
    Output: [4 5]
```

### # creating a 2D array

```
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6]])
```

### # shape of the array

```
print(arr_2d.shape)
    output: (2, 3)
```

|       |   | Axis1 |     |     |
|-------|---|-------|-----|-----|
|       |   | 0     | 1   | 2   |
| Axis0 | 0 | 0,0   | 0,1 | 0,2 |
|       | 1 | 1,0   | 1,1 | 1,2 |
|       | 2 | 2,0   | 2,1 | 2,2 |

### # data type of the array

```
print(arr_2d.dtype)
    output: int64
```

### # number of dimensions of the array

```
print(arr_2d.ndim)
    output: 2
```

### # total number of elements in the array

```
print(arr_2d.size)
    output: 6
```

# NumPy

## Creating arrays with different data types

### Data types:

- **bool**: Boolean (true or false) values
- **int**: signed integers (int8, int16, int32, int64)
- **uint**: unsigned integers (uint8, uint16, uint32, uint64)
- **float**: floating point numbers (float16, float32, float64, float128)
- **complex**: complex numbers (complex64, complex128, complex256)
- **object**: Python objects
- **string\_**: data with a fixed string length (string\_16, string\_32, etc.)
- **unicode\_**: data with a fixed length Unicode string (unicode\_16, unicode\_32 and etc.)

NumPy also provides functions to convert between data types, such as **astype()** and **view()**.

```
import numpy as np
# Creating arrays with different data types
arr_bool = np.array([True, False, True])
arr_int = np.array([1, 2, 3], dtype=np.int16)
arr_float = np.array([1.0, 2.5, 3.7], dtype=np.float64)
arr_complex = np.array([1 + 2j, 2 + 3j, 3 + 4j],
dtype=np.complex128)
arr_string = np.array(['apple', 'banana', 'cherry'],
dtype=np.string_)
arr_unicode = np.array([u'apple', u'banana', u'cherry'],
dtype=np.unicode_)
# Printing the arrays and their data types
print(arr_bool, arr_bool.dtype)
print(arr_int, arr_int.dtype)
print(arr_float, arr_float.dtype)
print(arr_complex, arr_complex.dtype)
print(arr_string, arr_string.dtype)
print(arr_unicode, arr_unicode.dtype)
```

```
# Removing an element from an array
a = np.array([1, 2, 3, 4, 5])
b = np.delete(a, [0, 2]) # delete the
1st and 3d elements
print(b) # [2, 4, 5]
a = np.array([1, 2, 3, 4, 5])
a = np.delete(a, 2) # delete the 3d
element
print(a) # [1, 2, 4, 5]
a = np.array([1, 2, 3, 4, 5])
a.pop(2) # remove the 3d element
print(a) # [1, 2, 4, 5]
```

### # Combining arrays

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.concatenate((a, b)) #
concatenate arrays a and b
print(c) # [1, 2, 3, 4, 5, 6]
a = np.array([1, 2, 3])
b = np.array([3, 4, 5])
c = np.union1d(a, b) # union of arrays
a and b
print(c) # [1, 2, 3, 4, 5]
```

### # Crossing arrays

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

```
c = np.intersect1d(a, b)
print(c) # [3, 4]
```

### # Array difference

```
x = np.array([1, 2, 3])
y = np.array([3, 4, 5])
z = np.setdiff1d(x, y)
print(z) # [1, 2]
```

### # Check if is a subset

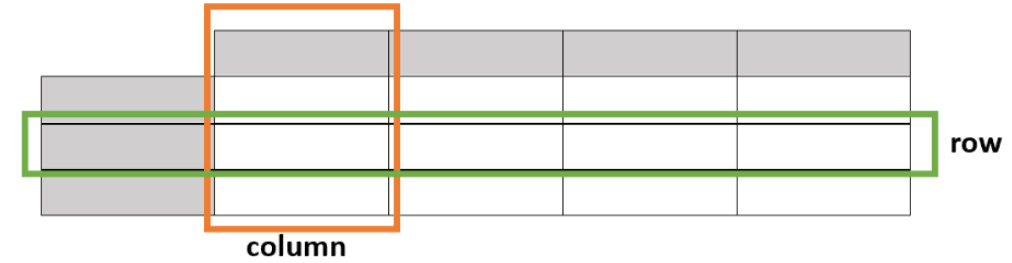
```
x = np.array([1, 2, 3])
y = np.array([1, 2, 3, 4, 5])
z = np.in1d(x, y)
print(z) # [True, True, True]
```

# Pandas

## The main aspects

- Library pandas contains structures data and means manipulation the data designed with purpose maximum to simplify and speed up cleaning and analyzing data with Python.
- <https://pandas.pydata.org/docs/index.html>
- Series** – one-dimensional array of data. It can hold data of any type: string, integer, float, dictionaries, lists, Booleans, and more
- Different formats data, which you can download to DataFrame, such as CSV, excel, SQL, JSON, etc.
- To load data to DataFrame use `read_csv()`, `read_excel()`, `read_sql()`, `read_json()` and so on. Define parameters, such as: `skiprows`, `header`, `separator` and others.

- DataFrame** – 2D data structure constructed with rows and columns, similar to a SQL database or Excel spreadsheet. It consists of a dictionary of lists in which the list each have their own identifiers or keys.



```
import pandas as pd
import numpy as np
# Create Series from a List
my_list = [10, 20, 30, 40, 50]
s = pd.Series(my_list,
name='my_series')
# Create DataFrame from a list of lists
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
df = pd.DataFrame(data,
columns=['col1', 'col2', 'col3'])
# Show the First and the Last Examples (Rows, Records)
print(s.head())
print(df.tail())
# Size of data structure
print(s.shape)
```

## NumPy vs Pandas

|                                 | NumPy                                                                             | Pandas                                                                                        |
|---------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Target Data                     | Numerical data (arrays and matrix)                                                | Tabular data                                                                                  |
| Main Data Object                | ndarray (multi-dimensional array)                                                 | Data frame (2d table) and series (1d array)                                                   |
| Usage                           | Numerical calculations                                                            | Data analysis and visualization                                                               |
| Machine learning, Deep learning | NumPy arrays can be directly fed as inputs to libraries, like, Scikit, Tensorflow | Data frame and series cannot be directly fed as inputs to libraries, like, Scikit, Tensorflow |
| Memory                          | Consume less memory                                                               | Consume more memory                                                                           |
| Performance                     | Has a better performance when number of rows is 50K or less                       | Has a better performance when number of rows is 500K or more                                  |
| Indexing                        | Indexing of NumPy arrays is very fast                                             | Indexing of the Pandas series is very slow                                                    |



# Pandas

## Loading text and binary data

```
# Loading CSV file with comma in quality separator
```

```
df_csv = pd.read_csv('file.csv', sep=',')
```

```
# Loading TSV file with tabulation in quality separator
```

```
df_tsv = pd.read_table('file.tsv', sep='\t')
```

```
# Loading Excel file with by letter with by name 'Sheet1'
```

```
df_excel = pd.read_excel('file.xlsx', sheet_name='Sheet1')
```

```
# Loading JSON file
```

```
df_json = pd.read_json('file.json')
```

```
# Loading binary file Pickle
```

```
df_pickle = pd.read_pickle('file.pickle')
```

```
# Loading binary file HDF5
```

```
df_hdf = pd.read_hdf('file.h5', 'my_data')
```

## Loading files from Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
os.listdir('/content/drive')
df = pd.read_csv('/content/drive/MyDrive/titanic.csv', sep=';', decimal=',')
df_x = pd.read_excel("/content/drive/MyDrive/titanic3.xls")
```

**Pickle binaries** are a file format that allows you to store any Python object in a binary format.

Using Pandas, we can save and load DataFrame in Pickle format. This is useful in cases where we need to save the DataFrame in the same format as we have it after performing some operations on it.

To save a DataFrame in Pickle format, we can use the **to\_pickle()** method, which takes a filename and saves the DataFrame in Pickle format.

To load a DataFrame from a Pickle file, we can use the **read\_pickle()** method. It takes a filename and loads a DataFrame from a Pickle file

Pickle binaries may contain viruses, so it is important not to download Pickle files from unknown sources. Also be aware that saving and loading a DataFrame in Pickle format can take up more space than in text formats such as CSV.

**Binary HDF5 files** are a file format that allows you to store large data objects and work with them efficiently.

**HDF5** is a structured data warehouse that allows you to store and organize large data objects in a format that is efficient for storage and processing. By means of **Pandas**, we can save and load DataFrame in HDF5 format.

To save the DataFrame in HDF5 format, we can use the **to\_hdf()**.

To load DataFrame from HDF5 file we can use **read\_hdf()**.

Binary HDF5 files are a more complex file format than text formats such as CSV. HDF5 supports the "chunking" function, which allows you to divide large data sets into smaller parts (chunks) that are stored separately.

```
df.to_hdf('file.h5', key='my_data', mode='w', format='table', chunksize=1000)
```

# Pandas

## Series and DataFrames

**Series object** is one of two basic types data used for data storage and processing. **Series** can be considered as one-dimensional an array of named rows and data types, similar to a column in a DataFrame or a column in a database. *The main attributes* of the Series :

- **values**: a one-dimensional NumPy array that contains the Series values.
- **index**: the Series index that identifies each element in the array.
- **dtype**: Series data type.

The Series object can be created using the **pd.Series()**.

```
data = {'John': 25, 'Jane': 30, 'Bob': 35}
s = pd.Series(data)
print(s)
```

```
data = {'John': 25, 'Jane': 30, 'Bob': 35}
s = pd.Series(data)
s['Mary'] = 28
print(s)
```

```
data = {'John': 25, 'Jane': 30, 'Bob': 35}
s = pd.Series(data)
print(s['Jane'])
```

```
data = {'John': 25, 'Jane': 30, 'Bob': 35}
s = pd.Series(data)
s.drop('Jane', inplace=True)
print(s)
```

```
data = {'John': 25, 'Jane': 30, 'Bob': 35}
s = pd.Series(data)
s_filtered = s[s > 28]
print(s_filtered)
```

**DataFrame object** is one of the two main data types used to store and process data. A **DataFrame** can be thought of as a two-dimensional array with named rows and columns, similar to a database table or spreadsheet. *The main attributes* of the DataFrame:

- **values**: a two-dimensional NumPy array that contains the values of the DataFrame.
- **index**: the DataFrame index identifying each row in the array.
- **columns**: DataFrame column names.
- **dtypes**: data types of each DataFrame column.

The DataFrame object can be created using the **pd.DataFrame()**.

**Key arguments of the DataFrame constructor:**

- **Dictionary or dictionary of dictionaries**. Dictionary keys are used to create column names, and values are used to populate

columns with data.

- **Dictionary with lists**. Dictionary keys are used to create column names, and values are used to populate columns with data.
- **A dictionary of dictionaries**, where the keys of the internal dictionary are used as the row index and the values are used to populate the data in the columns.
- **A list of dictionaries**, where each dictionary corresponds to one row in the DataFrame.
- **Tuples or lists**. Each tuple or list is used to create a row in the DataFrame.
- **Series objects**. If the list of Series objects have the same index, they can be combined into a DataFrame.



# Pandas

## Creating DataFrame

### # Creation DataFrame with dictionary with lists

```
data_dict = {'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]}
df = pd.DataFrame.from_dict(data_dict)
```

### # Creation DataFrame with dictionary dictionaries

```
data_dict = {'row1': {'a': 1, 'b': 2, 'c': 3}, 'row2': {'a': 4, 'b': 5, 'c': 6}, 'row3': {'a': 7, 'b': 8, 'c': 9}}
df = pd.DataFrame.from_dict(data_dict, orient='index')
```

### # Creation DataFrame with list dictionaries

```
data_list = [{'a': 1, 'b': 2, 'c': 3}, {'a': 4, 'b': 5, 'c': 6}, {'a': 7, 'b': 8, 'c': 9}]
df = pd.DataFrame(data_list)
```

### # Creation DataFrame with tuples

```
data_list = [(1, 'a', True), (2, 'b', False), (3, 'c', True)]
df = pd.DataFrame(data_list, columns=['num', 'letter', 'bool'])
```

### # Creation DataFrame with list lists

```
data_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
df = pd.DataFrame(data_list)
```

### # Creation DataFrame with list about 'objects' Series

```
s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], index=['a', 'b', 'c'])
s3 = pd.Series([7, 8, 9], index=['a', 'b', 'c'])
df = pd.DataFrame([s1, s2, s3])
```



# Pandas

## Subset Selection from DataFrame

In the Pandas library, two functions - **loc** and **iloc** - are available for selecting a subset of data in a DataFrame. Both functions allow you to access DataFrame elements by row and column index but have some differences in implementation.

The **loc** function allows you to access DataFrame elements by row and column names. The indices passed to the loc function include endpoints and can be used to select a single element or a subset of elements.

```
data = {'name': ['John', 'Jane', 'Bob'], 'age': [25, 30, 35]}
df = pd.DataFrame(data, index=['a', 'b', 'c'])
subset = df.loc[['a', 'b'], ['name', 'age']]
print(subset)
```

The **iloc** function allows you to access DataFrame elements by row and column positions. The indices passed to the **iloc** function do not include endpoints and can be used to select a single element or a subset of elements.

```
data = {'name': ['John', 'Jane', 'Bob'],
        'age': [25, 30, 35]}

df = pd.DataFrame(data)
subset = df.iloc[:2, :2]
print(subset)
```

| Type                | Notes                                                                                                                                                                                                 |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| df[column]          | Select single column or sequence of columns from the DataFrame; special case conveniences: Boolean array (filter rows), slice (slice rows), or Boolean DataFrame (set values based on some criterion) |
| df.loc[rows]        | Select single row or subset of rows from the DataFrame by label                                                                                                                                       |
| df.loc[:, cols]     | Select single column or subset of columns by label                                                                                                                                                    |
| df.loc[rows, cols]  | Select both row(s) and column(s) by label                                                                                                                                                             |
| df.iloc[rows]       | Select single row or subset of rows from the DataFrame by integer position                                                                                                                            |
| df.iloc[:, cols]    | Select single column or subset of columns by integer position                                                                                                                                         |
| df.iloc[rows, cols] | Select both row(s) and column(s) by integer position                                                                                                                                                  |
| df.at[row, col]     | Select a single scalar value by row and column label                                                                                                                                                  |
| df.iat[row, col]    | Select a single scalar value by row and column position (integers)                                                                                                                                    |
| reindex method      | Select either rows or columns by labels                                                                                                                                                               |



# Pandas

## DataFrame: indexing and filtering and merging

```
###
data = {'name': ['John', 'Jane', 'Bob'], 'age': [25, 30, 35]}
df = pd.DataFrame(data, columns=['name', 'age'])
print(df)
###
data = {'name': ['John', 'Jane', 'Bob'], 'age': [25, 30, 35]}
df = pd.DataFrame(data, columns=['name', 'age'], index=['a',
'b', 'c'])
print(df)
###
data = {'name': ['John', 'Jane', 'Bob'], 'age': [25, 30, 35]}
df = pd.DataFrame(data, columns=['name', 'age'], index=['a',
'b', 'c'])
print(df.loc['b', 'name'])
```

```
### choose columns with data about age and sex
df_x[['age', 'sex']]
### choose the first 5 speakers
df_x.iloc[:, :5]
### choose the last 5 speakers
df_x.iloc[:, -5:]
### choose columns with 5 on 10
df_x.iloc[:, 5:10]
### choose women with class 1
df_x[(df_x['sex'] == female) & (df_x['pclass'] == 1)]
### choose children from 10 years
df_x[df_x[age] >= 10]
### choose lines with certain categories values
df_x[df_x['embarked'].isin(['C', 'Q'])]
```

```
# create df1
data1 = {'id': [1, 2, 3, 4], 'name': ['John', 'Jane',
'Bob', 'Mary']}
df1 = pd.DataFrame(data1)
# create df2
data2 = {'id': [2, 3, 4, 5], 'age': [25, 30, 35, 40]}
df2 = pd.DataFrame(data2)
# merging two dataframe by id
result = pd.merge(df1, df2, on='id')
print(result)
```

The **how** argument determines which type of connection (join) to use when joining tables:

- **inner** (default): Stores only rows that have a matching key (shared values) in both tables.
- **outer**: stores all rows from both tables. Some values may be omitted if they are not in one of the tables.
- **left**: stores all rows from the left table and only those rows from the right table that have a corresponding key in the left table. Some values may be omitted if they are not in the right table.
- **right**: stores all rows from the right table and only those rows from the left table that have a corresponding key in the right table. Some values may be omitted if they are not in the left table.



# Pandas

## DataFrame: Append, Missing Values Handling, Duplicates

In the Pandas library, to add new rows of data to a DataFrame, you can use the **append()**, or indexing using the `[]` operator.

You can also use indexing with the `[]` operator and the `loc[]` method, which add new rows of data

Regardless of which method is used to add new rows of data, it is important to make sure that the data has the correct format and data types to ensure that it is correctly added to the DataFrame.

```
data = {'name': ['John', 'Jane',  
                'Bob'], age: [25, 30, 35]}  
df = pd.DataFrame(data)  
new_data = {'name': 'Mary', age:  
            40}  
df = df.append(new_data,  
               ignore_index=True)  
df.loc[3] = ['Mary', 40]
```

**Missing values** (NaN - Not a Number) are values that are missing from the data, but not replaced by zeros or any other values. Handling missing values is an important part of data analysis and the Pandas library has several ways to handle them.

- To find missing values in a DataFrame, you can use **isna()**.
- You can use the **fillna()** method to replace missing values.
- **dropna()** can be used to remove rows containing missing values.

```
data = {'name': ['John', 'Jane',  
                'Bob', 'Mary'], age: [25, 30,  
                None, 40]}  
df = pd.DataFrame(data)  
# finding absent values  
print(df.isna())  
# replacement absent values  
mean_age =  
df[age].mean() df[age].fillna(mean_age,  
                              inplace=True)  
# removal lines with absent values  
df.dropna
```

For definition **repeated records** in DataFrame you can use **duplicated()** and **drop\_duplicates()**: **duplicated()** returns True for everyone line, which repeated in DataFrame; **drop\_duplicates()** returns DataFrame without repeated records

```
data = {'name': ['John', 'Jane',  
                'Bob', 'John'], age: [25, 30, 35,  
                25]}  
df = pd.DataFrame(data)  
# finding repeated records  
print(df.duplicated())  
# removal repeated records  
df_unique = df.drop_duplicates()  
print(df_unique)  
# finding repeated records with  
taking into account column name  
print(df.duplicated(subset=['name']  
                    ))  
# removal repeated records with  
taking into account column name  
df_unique = df.drop_duplicates(  
subset=['name'])  
print(df_unique)
```



# Web Scrapping

## Examples: IBM data and Site coinmarketcap.com

```
# this url contains the data you need that scrape (from IMB data)
url = https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-DA0321EN-SkillsNetwork/labs/datasets/Programming Languages.html
```

```
# Your code here
```

```
from bs4 import BeautifulSoup # this module helps in web scrapping
import requests # this module helps us to download a web page
```

```
data= requests.get(url).text
```

```
soup = BeautifulSoup(data)
```

```
# find a html table in the web page
```

```
table = soup.find(table)
```

```
# Get all rows from the table
```

```
t = []
```

```
i = 0
```

```
for row in table.find_all('tr'): # in html table row is represented by the
tag <tr>
```

```
    # Get all columns in each row.
```

```
    cols = row.find_all('td') # in html a column is represented by the tag
<td>
```

```
    if i == 0:
```

```
        i = i+1
```

```
        continue
```

```
    name = col[1].getText() # store the value in column 3 as color_name
```

```
    code = col[3].getText() # store the value in column 4 as color_code
```

```
    t.append({'Language': name, 'Average Annual Salary': int(code.replace("$",
""))}.replace(", ", ""))})
```

```
pd.DataFrame(t)
```

```
import requests
```

```
url = 'https://coinmarketcap.com/'
```

```
response = requests.get(url)
```

```
html = response.content
```

```
from bs4 import BeautifulSoup
```

```
soup = BeautifulSoup(html, 'html.parser')
```

```
table = soup.find(table, class_='cmc-
table')
```

```
# receiving columns headers
```

```
headers = []
```

```
for th in table.find_all('th'):
```

```
    headers.append(th.text.strip())
```

```
# receiving data from table
```

```
data = []
```

```
for tr in table.find_all('tr')[1:]:
```

```
    row = []
```

```
    for td in tr.find_all('td'):
```

```
        row.append(td.text.strip())
```

```
    data.append(row)
```

```
# preservation data in DataFrame
```

```
pd.DataFrame(data, columns=headers)
```



# Pandas & Numpy

## Cheat Sheets and Key Links

- **Official documentation Pandas:** here you can find full documentation on to everyone functions and methods Pandas.
  - <https://pandas.pydata.org/docs/>
- **Pandas Cheat Sheet from Pandas:** it contains short review of basic functions and methods of Pandas.
  - [https://pandas.pydata.org/Pandas\\_Cheat\\_Sheet.pdf](https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf)
- **10 Minutes to Pandas:** here you can find fast review basic functions and methods pandas, which will help quickly get acquainted with basics
  - [https://pandas.pydata.org/docs/user\\_guide/10min.html](https://pandas.pydata.org/docs/user_guide/10min.html)
- **Python for Data Science Handbook:** this one book contains more detailed review Pandas and others libraries for work with data inPython.
  - <https://jakevdp.github.io/PythonDataScienceHandbook/>
- **10 Best NumPy Cheat Sheets.**
  - <https://www.kaggle.com/getting-started/255139>
- **McKinney, W. (2022). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.** O'Reilly Media.

## Section 1.2

# **Exploratory Data Analysis (EDA) and Data Visualization**

# Data Mining

**Data mining** is the process of discovering insightful, interesting, and novel patterns, as well as descriptive, understandable, and predictive models from large-scale data.

## Exploratory Data Analysis

Exploratory data analysis aims to explore the numeric and categorical attributes of the data individually or jointly to extract key characteristics of the data sample via statistics that give information about the centrality, dispersion, and so on.

# Data Mining Studies



- collect data in a way that does not directly interfere with how the data arise (“observe”)
- only establish an association
- **retrospective**: uses past data
- **prospective**: data are collected throughout the study
- randomly assign subjects to treatments
- establish causal connections



## Study: Breakfast cereal keeps girls slim

USA TODAY  
Sept 8, 2005

[...]

Girls who ate breakfast of any type had a lower average body mass index, a common obesity gauge, than those who said they didn't. The index was even lower for girls who said they ate cereal for breakfast, according to findings of the study conducted by the Maryland Medical Research Institute with funding from the National Institutes of Health (NIH) and cereal-maker General Mills.

[...]

The results were gleaned from a larger NIH survey of 2,379 girls in California, Ohio, and Maryland who were tracked between the ages of 9 and 19.

[...]

As part of the survey, the girls were asked once a year what they had eaten during the previous three days.

[...]

## 3 possible explanations

1. Eating breakfast causes girls to be slimmer
2. Being slim causes girls to eat breakfast
3. A third variable is responsible for both:

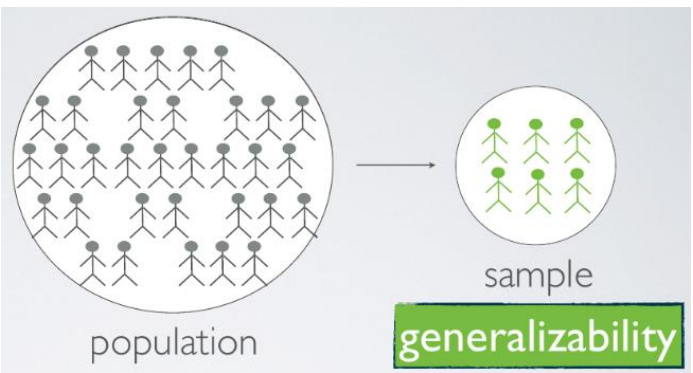
**confounding variables** – extraneous variables that affect both the explanatory and the response variable, and that make it seem like there is a relationship between them

**Correlation does not imply causation**

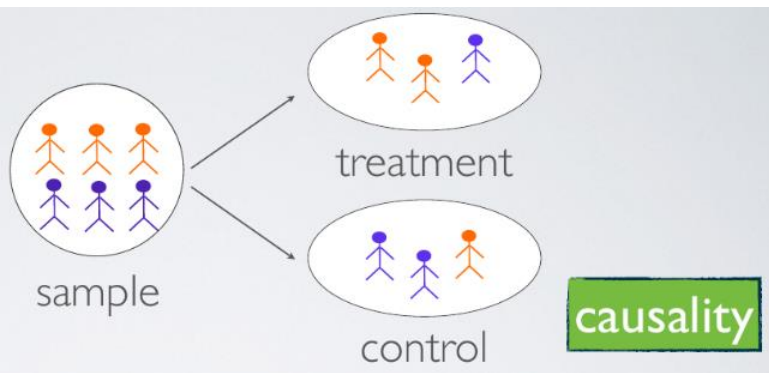
# Data Mining

## Random Sampling vs Random Assignment

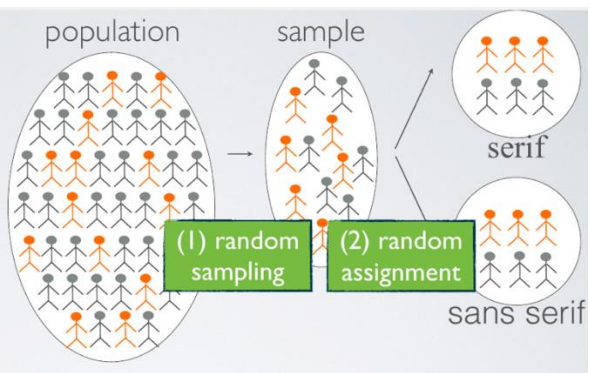
Random Sampling



Random Assignment



Example



|                    | Random assignment             | No random assignment             |                     |
|--------------------|-------------------------------|----------------------------------|---------------------|
| Random sampling    | causal and generalizable      | not causal, but generalizable    | Generalizability    |
| No random sampling | causal, but not generalizable | neither causal nor generalizable | No generalizability |
|                    | Causation                     | Association                      |                     |

*Handwritten notes:*

- ideal experiment* (points to Random assignment / Random sampling)
- most observational studies* (points to No random assignment / Random sampling)
- most experiments* (points to Random assignment / No random sampling)
- bad observational studies* (points to No random assignment / No random sampling)

# Data Mining

## Basics (1): Data Matrix

Datasets are in the form of

- a data matrix
- sequences
- text
- time-series
- images
- audio
- video
- other

|           | Sepal length | Sepal width | Petal length | Petal width | Class           |
|-----------|--------------|-------------|--------------|-------------|-----------------|
|           | $X_1$        | $X_2$       | $X_3$        | $X_4$       | $X_5$           |
| $x_1$     | 5.9          | 3.0         | 4.2          | 1.5         | Iris-versicolor |
| $x_2$     | 6.9          | 3.1         | 4.9          | 1.5         | Iris-versicolor |
| $x_3$     | 6.6          | 2.9         | 4.6          | 1.3         | Iris-versicolor |
| $x_4$     | 4.6          | 3.2         | 1.4          | 0.2         | Iris-setosa     |
| $x_5$     | 6.0          | 2.2         | 4.0          | 1.0         | Iris-versicolor |
| $x_6$     | 4.7          | 3.2         | 1.3          | 0.2         | Iris-setosa     |
| $x_7$     | 6.5          | 3.0         | 5.8          | 2.2         | Iris-virginica  |
| $x_8$     | 5.8          | 2.7         | 5.1          | 1.9         | Iris-virginica  |
| $\vdots$  | $\vdots$     | $\vdots$    | $\vdots$     | $\vdots$    | $\vdots$        |
| $x_{149}$ | 7.7          | 3.8         | 6.7          | 2.2         | Iris-virginica  |
| $x_{150}$ | 5.1          | 3.4         | 1.5          | 0.2         | Iris-setosa     |

### Data Matrix

$$\mathbf{D} = \begin{pmatrix} & X_1 & X_2 & \cdots & X_d \\ \mathbf{x}_1 & x_{11} & x_{12} & \cdots & x_{1d} \\ \mathbf{x}_2 & x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_n & x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix}$$

where  $\mathbf{x}_i$  denotes the  $i$ -th row, which is a  $d$ -tuple given as

$$\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{id})$$

and  $\mathbf{X}_j$  denotes the  $j$ -th column, which is an  $n$ -tuple given as

$$\mathbf{X}_j = (x_{1j}, x_{2j}, \dots, x_{nj})$$

Number of attributes  $d$  is called the **dimensionality** of the data.

**Rows** may also be referred to as

- *entities*,
- *instances*,
- *examples*,
- *records*,
- *transactions*,
- *objects*,
- *points*,
- *feature-vectors*,
- *tuples*.

**Columns** may also be called as

- *attributes*,
- *properties*,
- *features*,
- *dimensions*,
- *variables*,
- *fields*.

**Attributes**

- **Numeric** - have a real-valued or integer-valued domain
  - **Continuous** (can take on any number of infinite values given, within a given range)
  - **Discrete** (can take on one of a specific set of numeric values where we're able to count or enumerate all of the possibilities)
- **Categorical** – domain composed of a set of symbols
  - **Nominal** (values are unordered)
  - **Ordinal** (values are ordered)

If the  $d$  attributes or dimensions in  $\mathbf{D}$  are all numeric, then each row can be considered as a  **$d$ -dimensional point** or  **$d$ -dimensional column vector**:

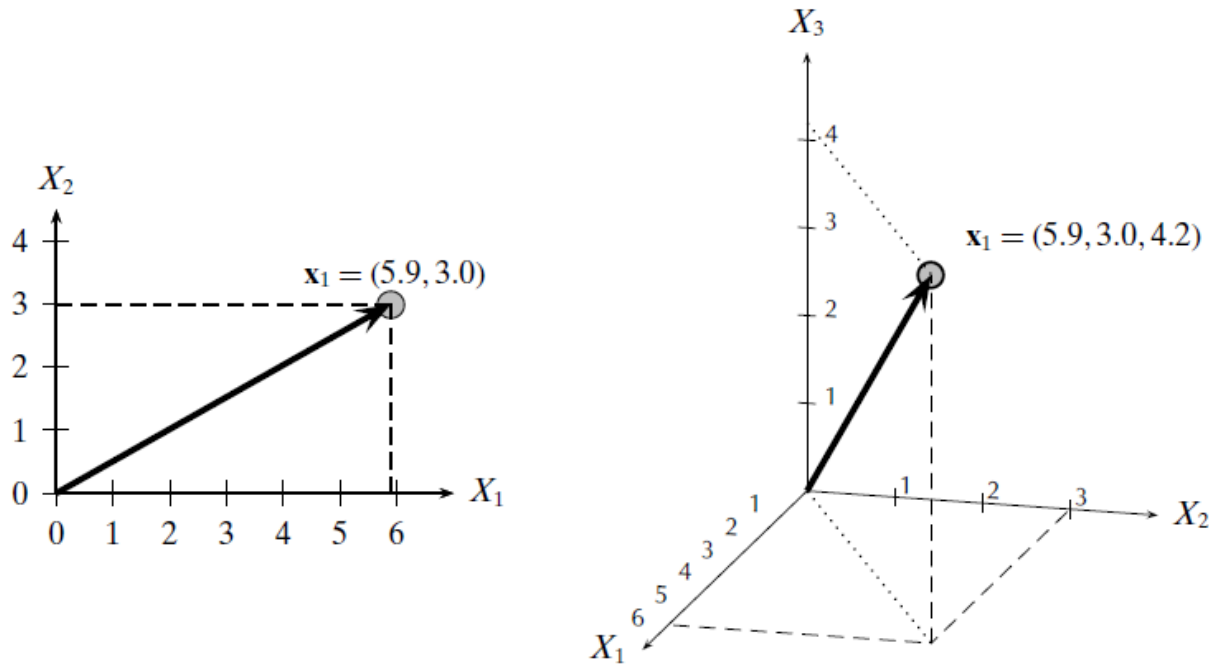
$$\mathbf{x}_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{id} \end{pmatrix} = (x_{i1} \ x_{i2} \ \cdots \ x_{id})^T \in \mathbb{R}^d$$

**Univariate analysis** – the analysis of a single attribute.  
**Bivariate analysis** – simultaneous analysis of two attributes  
**Multivariate analysis** – analysis of more than two attributes.

# Data Mining

## Basics (2): Matrix and Vectors

Row  $x_1$  as a point and vector in  $\mathbb{R}^2$  and  $\mathbb{R}^3$



Each numeric column or attribute can also be treated as a **vector** in an  $n$ -dimensional space  $\mathbb{R}^n$ :

$$X_j = \begin{pmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{nj} \end{pmatrix}$$

If all attributes are numeric, then the data matrix **D** is in fact an  $n \times d$  matrix, also written as  $\mathbf{D} \in \mathbb{R}^{n \times d}$ , given as

$$\mathbf{D} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} = \begin{pmatrix} -\mathbf{x}_1^T- \\ -\mathbf{x}_2^T- \\ \vdots \\ -\mathbf{x}_n^T- \end{pmatrix} = \begin{pmatrix} | & | & \cdots & | \\ X_1 & X_2 & \cdots & X_d \\ | & | & \cdots & | \end{pmatrix}$$

# Data Mining

## Basics (3): Matrix and Vectors

Let  $\mathbf{a}$  and  $\mathbf{b} \in \mathbb{R}^m$  be two  $m$ -dimensional vectors

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

### Length

$$\|\mathbf{a}\| = \sqrt{\mathbf{a}^T \mathbf{a}} = \sqrt{a_1^2 + a_2^2 + \dots + a_m^2} = \sqrt{\sum_{i=1}^m a_i^2}$$

### Angle

$$\cos \theta = \frac{\mathbf{a}^T \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} = \left( \frac{\mathbf{a}}{\|\mathbf{a}\|} \right)^T \left( \frac{\mathbf{b}}{\|\mathbf{b}\|} \right)$$

**Dot Product** (as the scalar value)

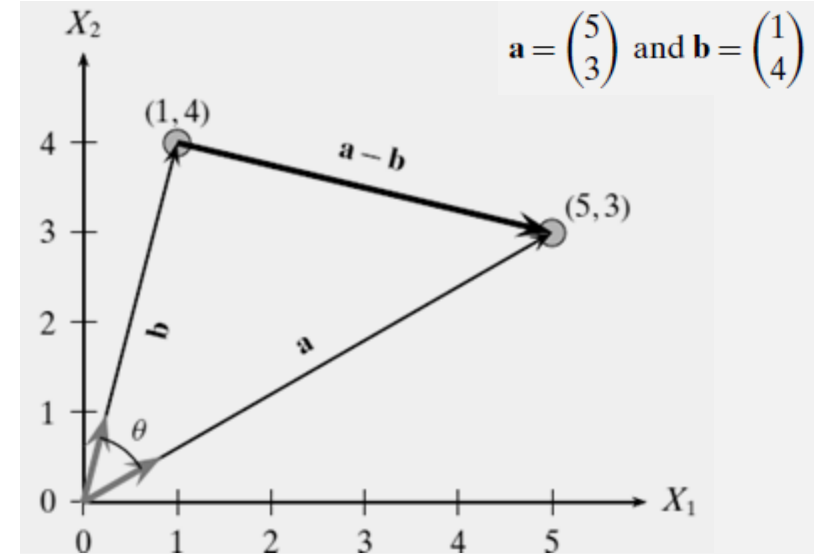
$$\begin{aligned} \mathbf{a}^T \mathbf{b} &= (a_1 \ a_2 \ \dots \ a_m) \times \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \\ &= a_1 b_1 + a_2 b_2 + \dots + a_m b_m \\ &= \sum_{i=1}^m a_i b_i \end{aligned}$$

### Distance

$$\begin{aligned} \delta(\mathbf{a}, \mathbf{b}) &= \|\mathbf{a} - \mathbf{b}\| = \\ &= \sqrt{(\mathbf{a} - \mathbf{b})^T (\mathbf{a} - \mathbf{b})} = \sqrt{\sum_{i=1}^m (a_i - b_i)^2} \end{aligned}$$

### Unit vector

$$\mathbf{u} = \frac{\mathbf{a}}{\|\mathbf{a}\|} = \left( \frac{1}{\|\mathbf{a}\|} \right) \mathbf{a}$$



$$\delta(\mathbf{a}, \mathbf{b}) = \sqrt{(5-1)^2 + (3-4)^2} = \sqrt{16+1} = \sqrt{17} = 4.12$$

$$\mathbf{a} - \mathbf{b} = \begin{pmatrix} 5 \\ 3 \end{pmatrix} - \begin{pmatrix} 1 \\ 4 \end{pmatrix} = \begin{pmatrix} 4 \\ -1 \end{pmatrix}$$

$$\|\mathbf{a} - \mathbf{b}\| = \sqrt{4^2 + (-1)^2} = \sqrt{17} = 4.12.$$

$$\mathbf{u}_a = \frac{\mathbf{a}}{\|\mathbf{a}\|} = \frac{1}{\sqrt{5^2 + 3^2}} \begin{pmatrix} 5 \\ 3 \end{pmatrix} = \frac{1}{\sqrt{34}} \begin{pmatrix} 5 \\ 3 \end{pmatrix} = \begin{pmatrix} 0.86 \\ 0.51 \end{pmatrix}$$

$$\mathbf{u}_b = \begin{pmatrix} 0.24 \\ 0.97 \end{pmatrix}$$

$$\cos \theta = \frac{\begin{pmatrix} 5 \\ 3 \end{pmatrix}^T \begin{pmatrix} 1 \\ 4 \end{pmatrix}}{\sqrt{5^2 + 3^2} \sqrt{1^2 + 4^2}} = \frac{17}{\sqrt{34} \times 17} = \frac{1}{\sqrt{2}}$$

# Missing Values

## Handling with Python

- "Handling Missing Data" (working with missing data) is the process of processing and solving the problem of missing values in the data. This is an important aspect of preparing data for analysis, as most analysis algorithms cannot handle data that contains gaps.
- Pandas has a set of functions that allow you to work with missing data. For example, you can delete rows or columns that contain missing values, fill them with the mean or median value, or use other methods to fill in the missing values.
- Some of the methods for handling missing data in Pandas:
  - **isnull()** - returns a Boolean mask that shows which values in the DataFrame are missing
  - **dropna()** - removes rows or columns containing missing values
  - **fillna()** - fills missing values in DataFrame
  - **interpolate()** - fills in missing values based on interpolation between existing values
  - **replace()** - replaces the values in the DataFrame with other values
- When working with missing data, it is important to consider the nature of the data and the purpose of the analysis. Incorrect handling of missing values can lead to distortion of results and incorrect conclusions.

```
# Filling NA in the column 'Age' with median
titanic['age'].fillna(titanic['age'].median(), inplace=True)

# Filling NA in the column 'Embarked' with the most frequent value
most_common = titanic['embarked'].value_counts().index[0]
titanic['embarked'].fillna(most_common, inplace=True)

# Interpolation in the column 'Age'
titanic['age'] = titanic['age'].interpolate()

# Interpolation in the column 'Fare'
titanic['fare'] = titanic['fare'].interpolate(method='linear',
limit_direction='forward')

# Replacing values in the column 'Sex'
titanic['sex'].replace({'male': 0, 'female': 1}, inplace=True)

# Replacing values in the column 'Embarked':
titanic['embarked'].replace({'S': 0, 'C': 1, 'Q': 2}, inplace=True)
```

The **interpolate()** method is used to fill in missing values in a DataFrame by interpolating between the existing values.

**Interpolation** is a method of finding new values between two known data points. In Pandas, **interpolate()** uses different interpolation methods, including:

- **linear** - linear interpolation that uses a straight line between the two nearest values to fill in the missing value
- **quadratic** - quadratic interpolation that uses a quadratic function between the two nearest values
- **cubic** - cubic interpolation that uses a cubic function between the two nearest values
- and others



# Data Manipulation

## Missing Values, Duplicate Values, One-Hot Encoding

- To visualize missing data, you can use heat maps or graphs that show the distribution of missing values across different columns of data.

```
import seaborn as sns
titanic_sns = sns.load_dataset('titanic')
sns.heatmap(titanic_sns.isnull(), cbar=False)
```

- DataFrame **deduplication** is the process of identifying and removing duplicate rows in a table. Duplicate rows can occur for a variety of reasons, such as data entry errors, incomplete or inaccurate data, and duplicate data when joining tables
- One way to remove duplicates in Pandas is to use the **drop\_duplicates** method.
- If we want to remove duplicates in the "Name" column, we can use the following code:

- ```
titanic.drop_duplicates(subset=['Name'], inplace=True)
```

- If we want to remove duplicates across the entire table, we can use the following code:

- ```
titanic.drop_duplicates(inplace=True)
```

- Redundant column removal** in Pandas is the process of eliminating unnecessary columns from a DataFrame to optimize data analysis and storage. By removing columns that don't contribute meaningful information, we can improve computational efficiency and reduce data size.

```
# Show the first 5 rows of DataFrame
```

```
print(titanic.head())
```

```
# Remove unnecessary columns
```

```
titanic = titanic.drop(['name', 'ticket'], axis=1)
```

- The process of converting categorical variables into numerical ones (One-Hot Encoding)** means creating columns for each possible value of the variable. This allows you to store categorical information in numerical form and use it in machine learning models.

```
embarked_dummies = pd.get_dummies(titanic['embarked'], prefix='embarked')
titanic = pd.concat([titanic, embarked_dummies], axis=1)
```



# Data Manipulation

## Missing Values, Duplicate Values, One-Hot Encoding

- To visualize missing data, you can use heat maps or graphs that show the distribution of missing values across different columns of data.

```
import seaborn as sns
titanic_sns = sns.load_dataset('titanic')
sns.heatmap(titanic_sns.isnull(), cbar=False)
```

- DataFrame **deduplication** is the process of identifying and removing duplicate rows in a table. Duplicate rows can occur for a variety of reasons, such as data entry errors, incomplete or inaccurate data, and duplicate data when joining tables
- One way to remove duplicates in Pandas is to use the **drop\_duplicates** method.
- If we want to remove duplicates in the "Name" column, we can use the following code:

- ```
titanic.drop_duplicates(subset=['Name'], inplace=True)
```

- If we want to remove duplicates across the entire table, we can use the following code:

- ```
titanic.drop_duplicates(inplace=True)
```

- Redundant column removal** in Pandas is the process of eliminating unnecessary columns from a DataFrame to optimize data analysis and storage. By removing columns that don't contribute meaningful information, we can improve computational efficiency and reduce data size.

```
# Show the first 5 rows of DataFrame
```

```
print(titanic.head())
```

```
# Remove unnecessary columns
```

```
titanic = titanic.drop(['name', 'ticket'], axis=1)
```

- The process of converting categorical variables into numerical ones (One-Hot Encoding)** means creating columns for each possible value of the variable. This allows you to store categorical information in numerical form and use it in machine learning models.

```
embarked_dummies = pd.get_dummies(titanic['embarked'], prefix='embarked')
titanic = pd.concat([titanic, embarked_dummies], axis=1)
```



# Data Manipulation

## Date-Time format. Sort values. Differences

### # Loading Data

```
url= 'https://stoq.com/q/d/1/?s=eurusd&i=d'  
data = pd.read_csv(url) ; print(data.info())
```

| # | Column | Non-Null | Count    | Dtype   |
|---|--------|----------|----------|---------|
| 0 | Date   | 13780    | non-null | object  |
| 1 | Open   | 13780    | non-null | float64 |
| 2 | High   | 13780    | non-null | float64 |
| 3 | Low    | 13780    | non-null | float64 |
| 4 | Close  | 13780    | non-null | float64 |

### # Convert into datetime

```
data['Date']= pd.to_datetime(data['Date'])  
print(data.info())
```

| # | Column | Non-Null | Count    | Dtype          |
|---|--------|----------|----------|----------------|
| 0 | Date   | 13780    | non-null | datetime64[ns] |
| 1 | Open   | 13780    | non-null | float64        |
| 2 | High   | 13780    | non-null | float64        |
| 3 | Low    | 13780    | non-null | float64        |
| 4 | Close  | 13780    | non-null | float64        |

```
df_eurusd = data.set_index('Date')
```

|            | Open    | High    | Low     | Close   |
|------------|---------|---------|---------|---------|
| Date       |         |         |         |         |
| 1971-01-04 | 0.53530 | 0.53530 | 0.53530 | 0.53530 |
| 1971-01-05 | 0.53500 | 0.53500 | 0.53500 | 0.53500 |

### # Sort by date

```
data = data.sort_values(by='Date')
```

### # Create a variable with price increments

```
data['price_diff'] = data['Close'].diff()
```

### # Create a variable with relative price increases

```
data['rel_diff'] = data['price_diff'] / data['Close'].shift(1)  
display(data.head())
```

|   | Date       | Open   | High   | Low    | Close  | price_diff | rel_diff  |
|---|------------|--------|--------|--------|--------|------------|-----------|
| 0 | 1971-01-04 | 0.5353 | 0.5353 | 0.5353 | 0.5353 | NaN        | NaN       |
| 1 | 1971-01-05 | 0.5350 | 0.5350 | 0.5350 | 0.5350 | -0.0003    | -0.000560 |

### #### Or with row indexing

### # Sort by date

```
df_eurusd = df_eurusd.sort_index()
```

### # Create a variable with price increments

```
df_eurusd['price_diff'] = df_eurusd['Close'].diff()
```

### # Create a variable with relative price increases

```
df_eurusd['rel_diff'] = df_eurusd['price_diff'] /  
df_eurusd['Close'].shift(1)  
display(df_eurusd.head())
```

|            | Open   | High   | Low    | Close  | price_diff | rel_diff  |
|------------|--------|--------|--------|--------|------------|-----------|
| Date       |        |        |        |        |            |           |
| 1971-01-04 | 0.5353 | 0.5353 | 0.5353 | 0.5353 | NaN        | NaN       |
| 1971-01-05 | 0.5350 | 0.5350 | 0.5350 | 0.5350 | -0.0003    | -0.000560 |



# Data Grouping

## Aggregated characteristics

```
# Data grouping by column 'Pclass'  
grouped = titanic.groupby('pclass')
```

```
# Calculating the mean age for each group  
average_age = grouped['age'].mean()  
print(average_age)
```

```
# Calculating the count of passengers by class and sex  
grouped = titanic.groupby(['pclass', 'sex']).count()  
print(grouped)
```

```
# Calculating the average age of passengers using external function  
def get_mean_age(group):  
    return group['age'].mean()  
  
grouped = titanic.groupby('pclass').apply(get_mean_age)  
print(grouped)
```

```
# Calculating the average and median age of passengers for each group by class and sex using NumPy's functions  
grouped = titanic.groupby(['pclass', 'sex'])['age'].agg([np.mean, np.median])  
print(grouped)
```

```
# Creating a contingency table  
pd.crosstab(index=titanic['sex'], columns=titanic['pclass'])
```

```
# Grouping by weeks  
w_prices = data.groupby(pd.Grouper(key='Date', freq='W')).mean()['Close']  
print(w_prices)
```

```
# Grouping by months  
m_prices = data.groupby(pd.Grouper(key='Date', freq='M')).mean()[['Close', 'rel_diff']]  
print(m_prices)
```

# Univariate Analysis

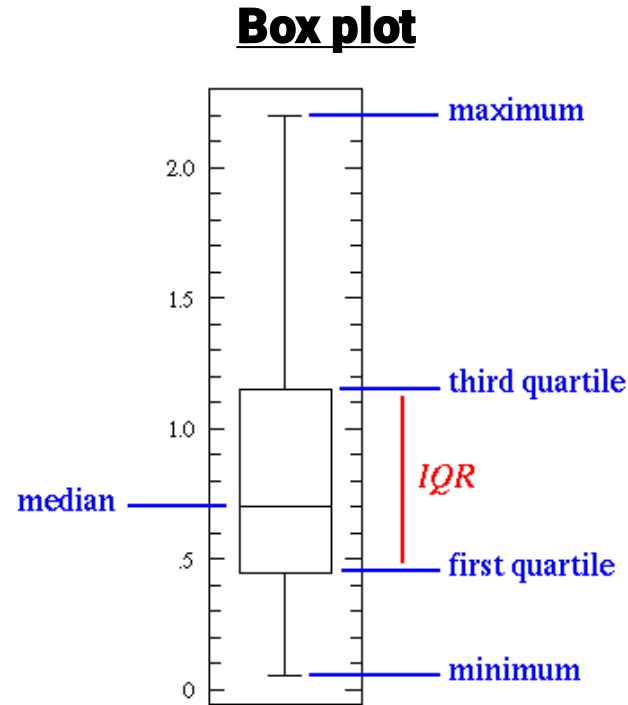
It is the analysis of a single attribute (feature, variable)

$$\mathbf{D} = \begin{pmatrix} X \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

# Statistics Features

## Informative view of your data (1)

- The line in the middle is the **median** value of the data. Median is used over the mean since it is more robust to outlier values.
- The **first quartile** is essentially the 25th percentile; i.e. 25% of the points in the data fall below that value.
- The **third quartile** is the 75th percentile; i.e. 75% of the points in the data fall below that value.
- The **min** and **max** values represent the upper and lower ends of our data range.



A **box plot** perfectly illustrates:

- When the box plot is **short** it implies that much of your data points are similar, since there are many values in a small range.
- When the box plot is **tall** it implies that much of your data points are quite different, since the values are spread over a wide range.
- If the median value is closer to the **bottom** then we know that most of the data has lower values. If the median value is closer to the **top** then we know that most of the data has higher values. Basically, if the median line is not in the middle of the box then it is an indication of **skewed** data.
- Are the whiskers **very long**? That means your data has a high **standard deviation** and **variance** i.e. the values are spread out and highly varying. If you have long whiskers on one side of the box but not the other, then your data may be highly varying only in one direction.

### Measures of Central Tendency

#### Sample Mean

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^n x_i$$

the average value of  $x_i$  's (of all the points)

#### Median

$$P(X \leq m) \geq \frac{1}{2} \text{ and } P(X \geq m) \geq \frac{1}{2}$$

The **median**  $m$  is the "middle-most" value; half of the values of  $X$  are less and half of the values of  $X$  are more than  $m$ .

#### Mode

$$\text{mode}(X) = \arg \max_x \hat{f}(x)$$

The **mode** of a random variable  $X$  is the value at which the probability mass/density function attains its maximum value

**Robustness.** A statistic is robust if it is not affected by extreme values (such as outliers) in the data. The sample mean is unfortunately not robust because a single large value (an outlier) can skew the average.

**"Plans based on average assumptions are wrong on average"**

# Statistics Features

## Informative view of your data (2)

### Total Variance

$$\text{var}(\mathbf{D}) = \frac{1}{n} \sum_{i=1}^n \delta(\mathbf{x}_i, \boldsymbol{\mu})^2 = \frac{1}{n} \sum_{i=1}^n \|\mathbf{x}_i - \boldsymbol{\mu}\|^2$$

The total variance of the data matrix  $\mathbf{D}$  is the average squared distance of each point from the mean. The variance of a random variable  $\mathbf{X}$  provides a measure of how much the values of  $\mathbf{X}$  deviate from the mean or expected value of  $\mathbf{X}$

### Sample Variance

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \hat{\mu})^2$$

It is the average squared deviation of the data values  $x_i$  from the sample mean  $\hat{\mu}$

### Sample Standard Deviation

$$\hat{\sigma} = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \hat{\mu})^2}$$

## Measures of Dispersion

### Range

$$r = \max\{X\} - \min\{X\}$$

The value range or simply range of a random variable  $\mathbf{X}$  is the difference between the maximum and minimum values of  $\mathbf{X}$

### Sample Range

$$\hat{r} = \max_{i=1}^n \{x_i\} - \min_{i=1}^n \{x_i\}$$

Range is sensitive to extreme values, and thus is not robust

### Interquartile Range

$$IQR = q_3 - q_1 = F^{-1}(0.75) - F^{-1}(0.25)$$

**Quartiles** are special values of the quantile function  $F^{-1}$  that divide the data into four equal parts.

1. The **first quartile** is the value  $q_1 = F^{-1}(0.25)$ , to the left of which **25%** of the points lie;
2. the **second quartile** is the same as the median value  $q_2 = F^{-1}(0.5)$ , to the left of which **50%** of the points lie;
3. the **third quartile**  $q_3 = F^{-1}(0.75)$  is the value to the left of which **75%** of the points lie;
4. the **fourth quartile** is the maximum value of  $X$ , to the left of which **100%** of the points lie.

IQR is robust by definition

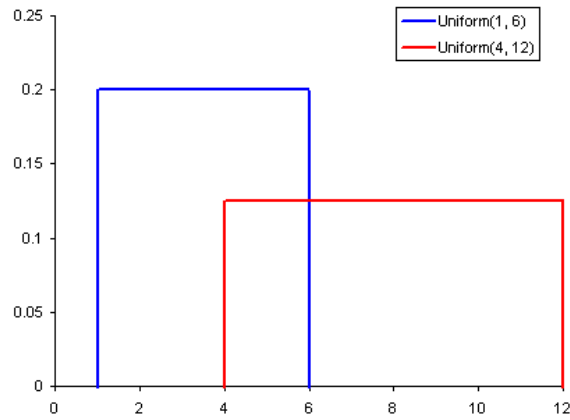
## Standard Score or the Z-Score

$$z_i = \frac{x_i - \hat{\mu}}{\hat{\sigma}}$$

The z-score of  $x_i$  measures the deviation of  $x_i$  from the mean value  $\hat{\mu}$ , in units of  $\hat{\sigma}$

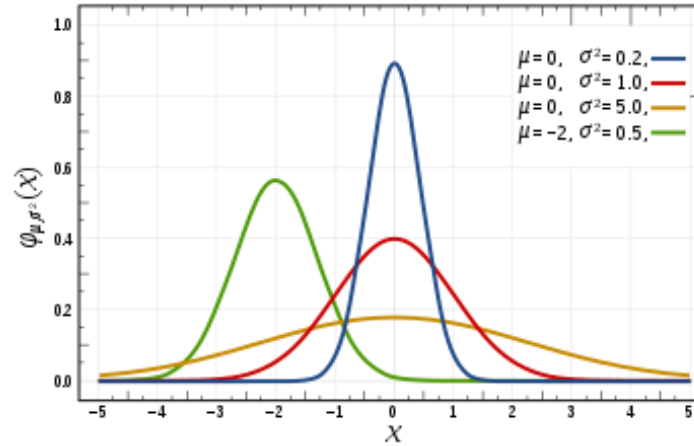
# Probability Distribution

- **Probability** can be defined as the percent chance that some event will occur.
- It is quantified in the range of 0 to 1 where
  - 0 means we are certain this **will not occur** and
  - 1 means we are certain it **will occur**.
- A **probability distribution** is then a function which represents the probabilities of all possible values in the experiment.



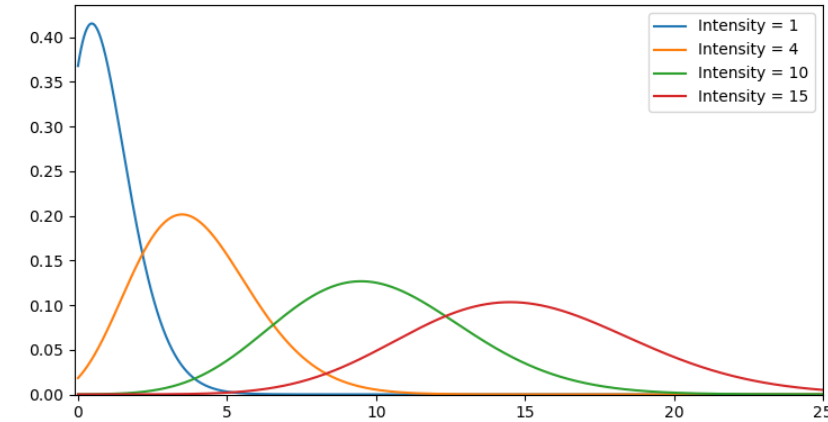
**Uniform**

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a, b] \\ 0 & \text{otherwise,} \end{cases}$$



**Normal**

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$



**Poisson**

## Empirical Cumulative Distribution Function

$$\hat{F}(x) = \frac{1}{n} \sum_{i=1}^n I(x_i \leq x)$$

where

$$I(x_i \leq x) = \begin{cases} 1 & \text{if } x_i \leq x \\ 0 & \text{if } x_i > x \end{cases}$$



# Data Visualization in Python: Matplotlib and Seaborn

# Visualization

## Matplotlib and Seaborn

- **Matplotlib** is the core data visualization library in Python. It provides basic graphing functions such as line graphs, bar charts, and pie charts. Matplotlib also allows detailed customization of the graphics appearance, including colors, fonts, and sizes.
- **Seaborn** is a library that is based on matplotlib, but provides advanced functionality for data visualization. It allows you to quickly and easily create more complex visualizations such as distribution plots, dependence plots, and multivariate plots. Seaborn has built-in styles and color palettes that allow light to change the appearance of graphs.

Work with by several graphs and axes using **plt.subplots()**

- Allows create many graphs on one drawing
- Syntax functions:

```
fig, ax = plt.subplots(nrows, ncols)
```

- `nrows`: number lines graphs to be created
- `ncols`: number columns graphs to be create
- `figure`: variable to contain generated drawing.
- `axes`: array with axial objects.

- For example:

```
fig, ax = plt.subplots(nrows=2, ncols=2)
```

there are variable `fig` which contains drawing, and variable `ax` which contains array with 4 axes graphs

- For everyone graphics we can change parameters, such as headlines, signatures axes, legends, color etc.
- Seaborn has methods, which allow creating multi graphics, such as `pairplot`, `jointplot`, `FacetGrid`.
- When creating many graphs in one figure, it is important to maintain a balance between the number of graphs and their clarity. Not load the figure with too many graphs and information, as this can cause difficulty in understanding the data.



# Visualization

## Matplotlib Basics and Examples

### Resize charts with

- `plt.figure(figsize=(width, height))`
- `plt.subplots(figsize=(width, height))`

### Save figure

- `fig.savefig('figure.png', dpi=300, format='png', bbox_inches='tight', pad_inches=0.1)`

### # Histogram of 'Age'

```
plt.hist(df['Age'], bins=20)
plt.xlabel('Вік')
plt.ylabel('Частота')
plt.title('Розподіл віку пасажирів Titanic')
plt.show()
```

### # Density plot of 'Age'

```
df['Age'].plot(kind='density')
plt.title('Density Plot of Passenger Ages')
plt.xlabel('Age')
plt.ylabel('Density')
plt.show()
```

```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
```

```
male_data = [df[(df['Pclass']==1) & (df['Sex']=='male')]['Fare'],
             df[(df['Pclass']==2) & (df['Sex']=='male')]['Fare'],
             df[(df['Pclass']==3) & (df['Sex']=='male')]['Fare']]
```

```
female_data = [df[(df['Pclass']==1) & (df['Sex']=='female')]['Fare'],
               df[(df['Pclass']==2) & (df['Sex']=='female')]['Fare'],
               df[(df['Pclass']==3) & (df['Sex']=='female')]['Fare']]
```

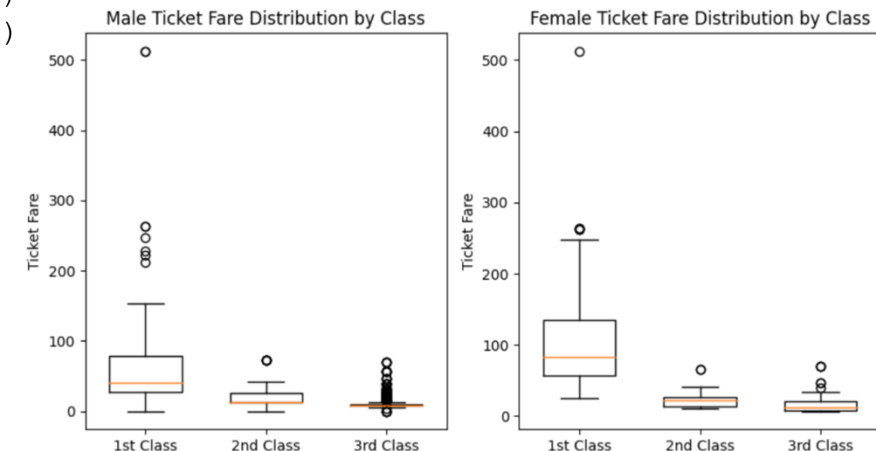
```
axs[0].boxplot(male_data, widths=0.6)
axs[1].boxplot(female_data, widths=0.6)
```

```
axs[0].set_title('Male Ticket Fare Distribution by Class')
axs[1].set_title('Female Ticket Fare Distribution by Class')
```

```
axs[0].set_xticklabels(['1st Class', '2nd Class', '3rd Class'])
axs[1].set_xticklabels(['1st Class', '2nd Class', '3rd Class'])
```

```
axs[0].set_ylabel('Ticket Fare')
axs[1].set_ylabel('Ticket Fare')
```

```
plt.show()
```





# Visualization

## Examples: Pair and Joint Plots, FacetGrid, Combined Graphs, Box-and-Whisker Plots, Heat Map, etc.

### # Loading Data

```
tips = sns.load_dataset('tips')
iris = sns.load_dataset('iris')
titanic = sns.load_dataset('titanic')
```

### # Pair plot

```
sns.pairplot(iris, hue='species', palette='husl')
sns.pairplot(tips, hue='sex', palette='coolwarm')
```

### # Joint plot (1)

```
sns.jointplot(x='total_bill', y='tip', data=tips,
kind='reg', color='blue')
```

### # Joint plot (2)

```
sns.jointplot(data=mpg, x="horsepower", y="mpg",
kind="hex", space=0)
```

### # Count Plot

```
sns.countplot(x='day', data=tips)
```

### # Linear Regression Line

```
sns.lmplot(x='total_bill', y='tip', data=tips)
```

### # Box-and-Whisker Plot

```
sns.boxplot(x='day', y='total_bill', data=tips)
sns.stripplot(x='day', y='total_bill', data=tips)
sns.swarmplot(x='day', y='total_bill', data=tips)
```

### # Correlation Matrix

```
corr = tips.corr()
```

### # Heat Map

```
sns.heatmap(corr, annot=True)
```

### # Combined Graphs

```
sns.set_style("whitegrid")
fig, ax = plt.subplots(figsize=(10, 6))
sns.barplot(x="class", y="survived", hue="sex", data=titanic,
ax=ax)
sns.pointplot(x="class", y="survived", hue="sex",
data=titanic, dodge=True, markers=["o", "x"], linestyle=["-",
"--"], ax=ax)
```

### # FacetGrid for dataset "Tips" (1)

```
g = sns.FacetGrid(tips, col="time", row="smoker",
margin_titles=True)
g.map(sns.scatterplot, "total_bill", "tip", color="purple")
plt.show()
```

### # FacetGrid for dataset "Tips" (2)

```
grid = sns.FacetGrid(tips, col="sex", row="smoker")
grid.map(sns.scatterplot, "total_bill", "tip")
```

### # FacetGrid for dataset "Tips" (3)

```
grid=sns.FacetGrid(tips, col="sex", row="smoker", hue="day")
grid.map(sns.scatterplot, "total_bill", "tip")
grid.add_legend()
```

### # FacetGrid for dataset "Titanic"

```
grid=sns.FacetGrid(titanic, row="class", col="sex")
grid.map(sns.histplot, "age", bins=20)
```

### # FacetGrid for dataset "MPG"

```
mpg = sns.load_dataset('mpg')
grid=sns.FacetGrid(mpg, col="origin", row="cylinders",
hue="model_year", height=3, aspect=1.2)
grid.map(sns.scatterplot, "displacement", "mpg")
grid.add_legend()
```

# Visualization

## Histogram, Distribution, Violin, Boxen, KDE Plots

- **histplot** is a tool for visualizing a numerical variable's distribution using a histogram. The histogram shows the number of values within each interval on the numerical axis.
- **distplot** is a tool for visualizing a numerical variable's distribution using both a histogram and a kernel density estimate (KDE) plot. The KDE plot shows the proportion of values at each point along the numerical axis, providing a smoothed view of the data distribution.
- **kdeplot** is a tool for visualizing a numerical variable's distribution using only a kernel density estimate (KDE) plot. The KDE plot shows the proportion of values at each point along the numerical axis, highlighting the data's distribution density without using bins.
- **violinplot** allows you to create a "violin plot," which displays the distribution of data using kernel density estimation. The violin plot consists of two halves that show the data distribution on the left and right of a central line, which represents the median.
- **boxenplot** allows you to create a "boxen plot," which also displays data distribution but provides a more detailed view of outliers and extreme values. The boxen plot shows quartiles and the median, and it represents outliers using "boxes."
- **catplot** allows for plotting categorical variables. It can be used to create bar plots, strip plots, box plots, and other types of categorical plots.
- **stripplot** allows for plotting the position of individual points on a numerical axis for each categorical variable. Each point represents a single observation.
- **swarmplot** is similar to stripplot but adds functionality to prevent overlapping points, displaying them in a "swarm" arrangement.

```
sns.histplot(data=mpg, x=mpg, bins=10, kde=True)
sns.histplot(data=mpg, x=mpg, bins=10, kde=False,
color="blue")

sns.distplot(mpg[mpg], kde=True) sns.distplot(mpg[mpg],
kde=False)

sns.kdeplot(data=mpg, x=mpg, shade=True, color="red")

sns.violinplot(data=tips, x="day", y="total_bill",
hue="sex", split=True, inner="quartile", palette="Set3")

sns.boxenplot(data=tips, x="day", y="total_bill",
hue="sex", palette="Set2")

sns.catplot(data=tips, x="day", y="total_bill",
kind="bar", hue="sex", palette="Set2")

sns.stripplot(data=tips, x="day", y="total_bill",
hue="sex", jitter=True, dodge=True, palette="Set2")

sns.swarmplot(data=tips, x="day", y="total_bill",
hue="sex", palette="Set2")
```

### Additional parameters:

- <https://seaborn.pydata.org/generated/seaborn.histplot.html>
- <https://seaborn.pydata.org/generated/seaborn.displot.html>
- <https://seaborn.pydata.org/generated/seaborn.kdeplot.html>
- <https://seaborn.pydata.org/generated/seaborn.violinplot.html>
- <https://seaborn.pydata.org/generated/seaborn.boxenplot.html>
- <https://seaborn.pydata.org/generated/seaborn.catplot.html>

# Visualization

## Bar and Point Plots. Heat Map and Cluster Map

- **pointplot** shows the average value of a numerical variable for each category of a categorical variable, along with a confidence interval around this value.
- **barplot** displays the mean or median of a numerical variable for each category of a categorical variable.
- **heatmap** visualize data matrices, where the value of each cell is represented as a color on the plot. Color palettes can be adjusted using the **cmap** and **center** parameters. Additionally, you can display the value of each cell with the **annot** parameter.
- **clustermap** is used to group similar rows and columns in a data matrix. Grouping is performed using a hierarchical clustering algorithm, which helps identify relationships between rows and columns. The resulting plot shows similar rows and columns clustered together and displayed as blocks.

### # Point Plot

```
sns.pointplot(x='day', y='total_bill', hue='sex',
data=tips, ci=None)
plt.title('Total Bill by Day and Gender')
plt.xlabel("Day of theWeek"); plt.ylabel('TotalBill')
```

### # Bar Plot

```
sns.barplot(x='day', y='total_bill', hue='sex',
plt.title('Total Bill by Day and Gender'); plt.xlabel('Day
of theWeek'); plt.ylabel('TotalBill')
```

### # Correlation matrix

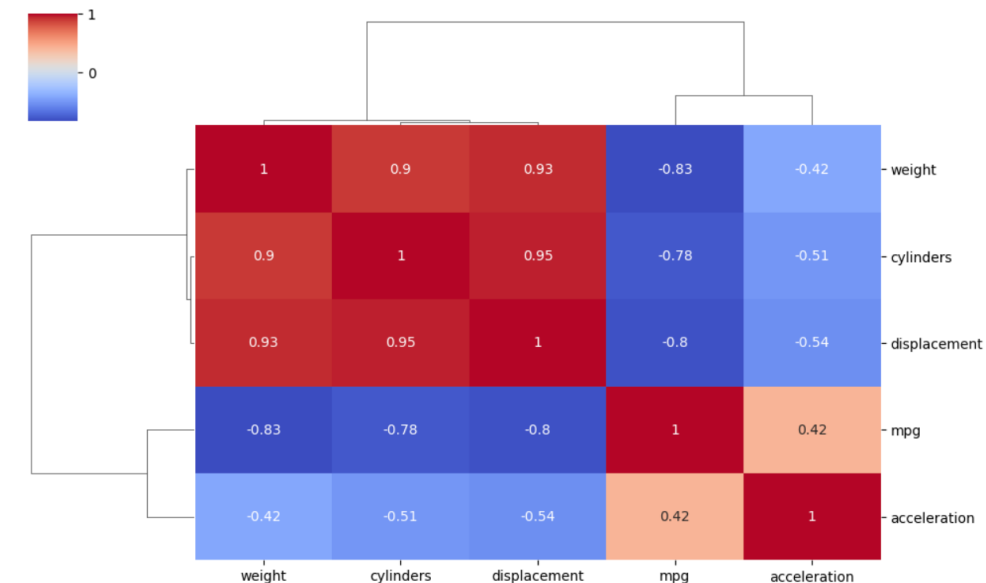
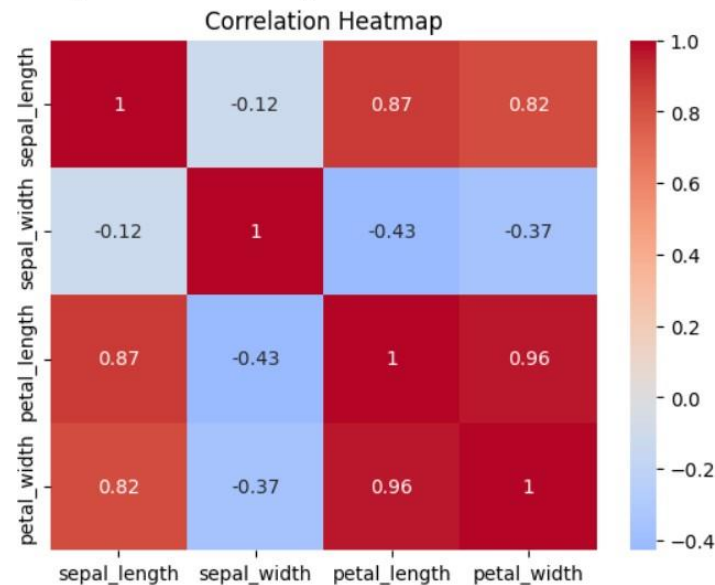
```
corr_matrix= iris.corr()
```

### # Building heatmap

```
sns.heatmap(corr_matrix, annot=True, cmap='cool warm',
center=0)
plt.title('Correlation Heatmap')
```

corr\_matrix

|              | sepal_length | sepal_width | petal_length | petal_width |
|--------------|--------------|-------------|--------------|-------------|
| sepal_length | 1.000000     | -0.117570   | 0.871754     | 0.817941    |
| sepal_width  | -0.117570    | 1.000000    | -0.428440    | -0.366126   |
| petal_length | 0.871754     | -0.428440   | 1.000000     | 0.962865    |
| petal_width  | 0.817941     | -0.366126   | 0.962865     | 1.000000    |



# Visualization

## Line Plot. Relations Plot

```
import yfinance as yf
data = yf.download('AAPL', start='2015-01-01', end='2023-04-19')
```

### # Line Plot

```
sns.set(rc={'figure.figsize': (10, 5)})
sns.lineplot(data=data['Close'])
plt.xlabel('Date')
plt.ylabel('Price')
plt.title('Close and Open Prices for AAPL')
```

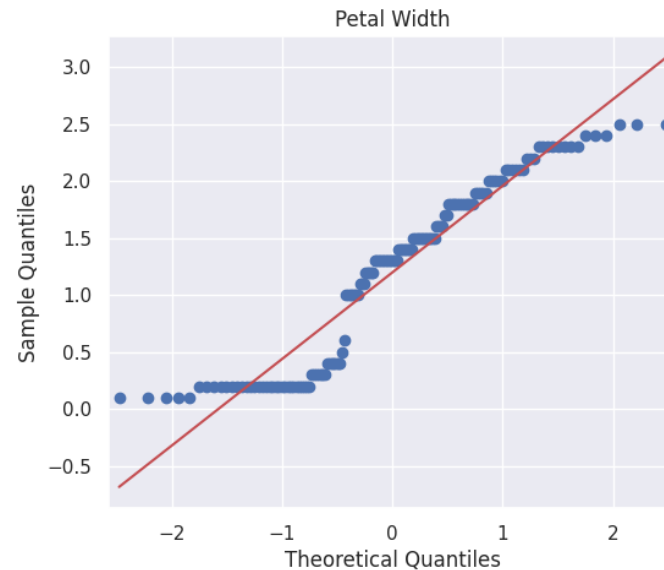
### # Relations Plot

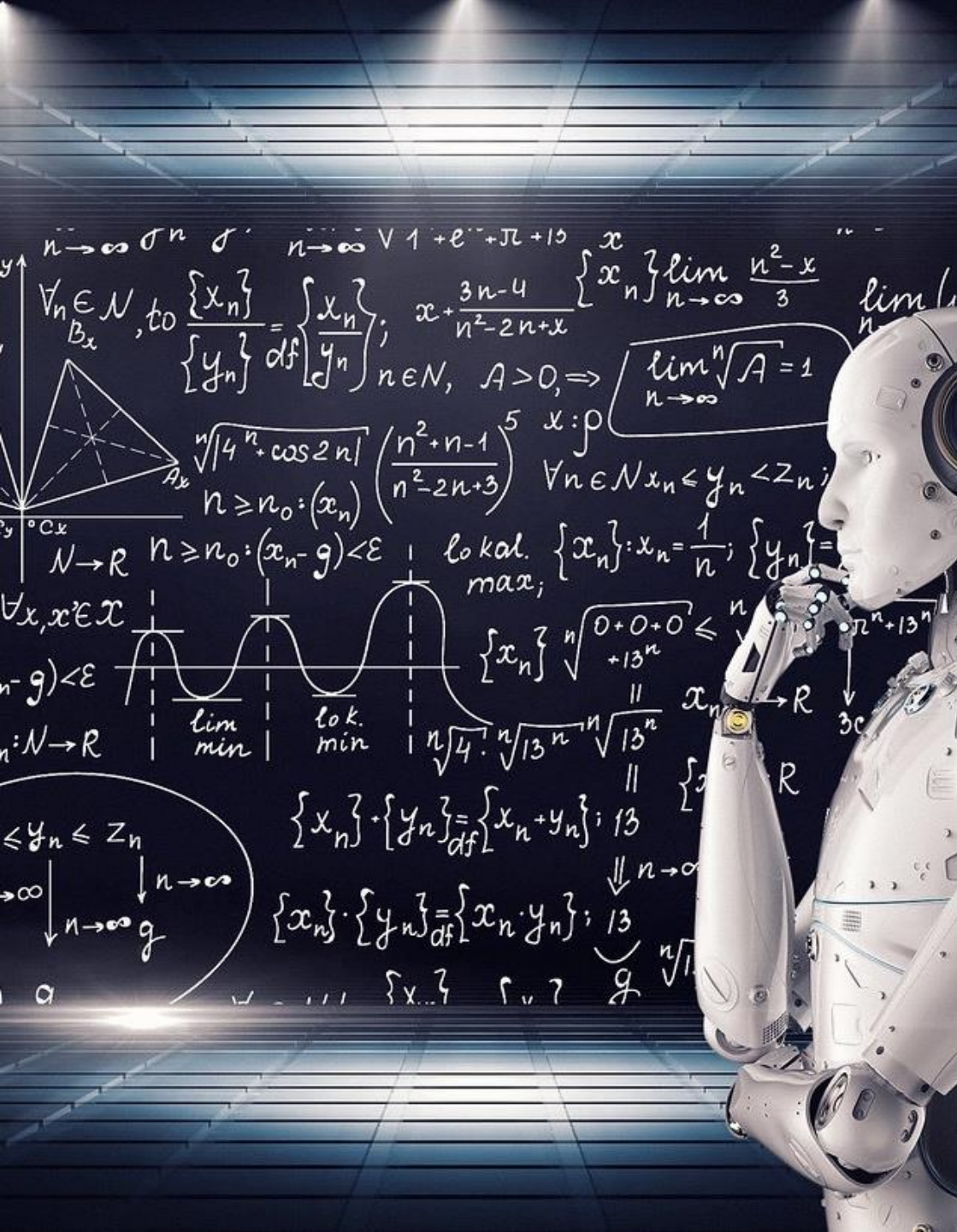
```
sns.relplot(data=data['Close'], label='Close AAPL', height=4.5, aspect=2)
plt.xlabel('Date')
plt.ylabel('Price')
plt.title('Close and Open Prices for AAPL')
plt.show()
```

### # Q-Q Plot

```
iris = sns.load_dataset("iris")
sm.qqplot(iris.iloc[:, 3], line='s')
plt.title('Petal Width')
```

"s" - standardized line, the expected order statistics are scaled by the standard deviation of the given sample and have the mean added to them





## Topic 2

### Correlation and Regression

### Analysis with Python:

### Statistical and Machine

### Learning Methods

## Section 2.1

# Bivariate Analysis and Linear Regression

In bivariate analysis, we consider two attributes at the same time. We are specifically interested in understanding the association or dependence between them, if any. We thus restrict our attention to the two attributes of interest, say  $X_1$  and  $X_2$ , with the data  $D$  represented as an  $n \times 2$  matrix:

$$\mathbf{D} = \begin{pmatrix} X_1 & X_2 \\ x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{n1} & x_{n2} \end{pmatrix}$$

# Correlation

## Pearson Correlation Coefficient

A **correlation coefficient** measures the extent to which two variables tend to change together.

The coefficient describes both the strength (degree) and the direction of the relationship.



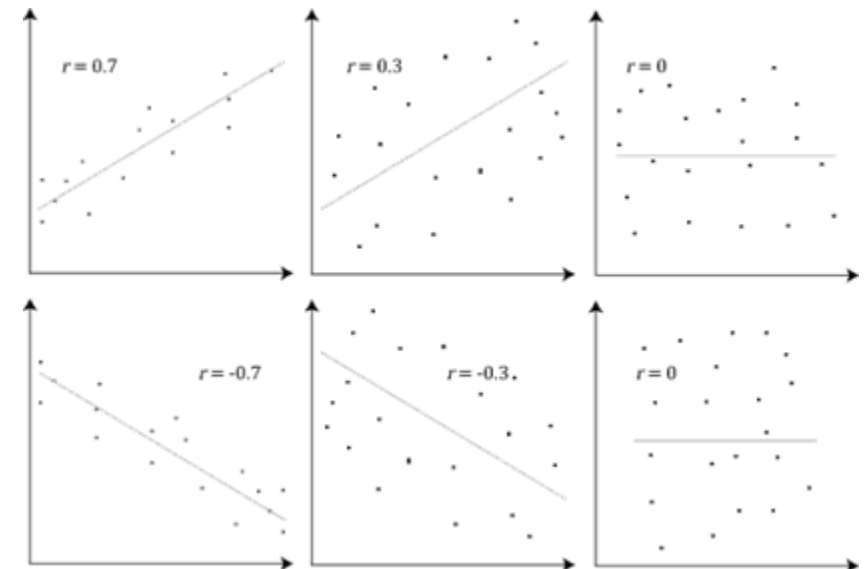
The **Pearson correlation** evaluates the linear relationship between two continuous variables. **A relationship is linear** when a change in one variable is associated with a proportional change in the other variable.

**The values range** between -1.0 and 1.0.

A correlation of -1.0 shows a perfect negative correlation, while a correlation of 1.0 shows a perfect positive correlation.

A correlation of 0.0 shows no linear relationship between the movement of the two variables.

**Example use case:** We can use the Pearson correlation to evaluate whether an increase in age leads to an increase in blood pressure.



# Correlation

## Spearman Correlation Coefficient

**Spearman's rank correlation coefficient** is a nonparametric measure of rank correlation (statistical dependence between the rankings of two variables). It assesses how well the relationship between two variables can be described using a **monotonic function**.

*The Spearman correlation can evaluate a monotonic relationship between two variables – Continuous or Ordinal and it is based on the ranked values for each variable rather than the raw data.*

A **monotonic relationship** is a relationship that does one of the following:

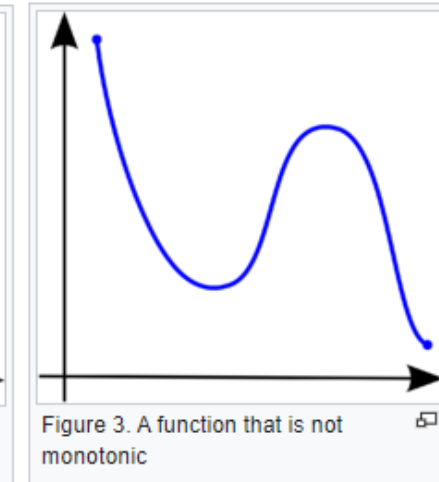
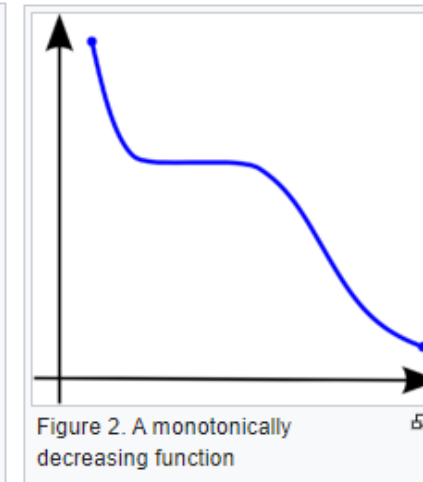
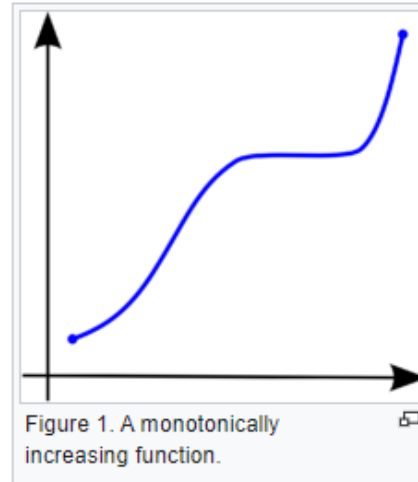
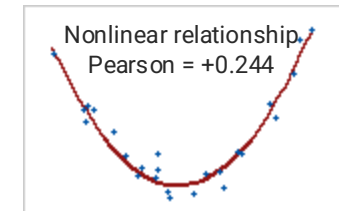
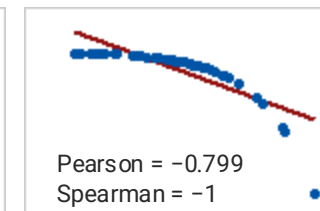
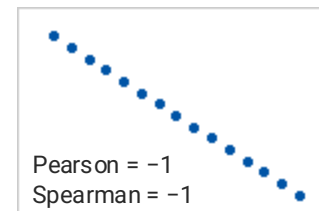
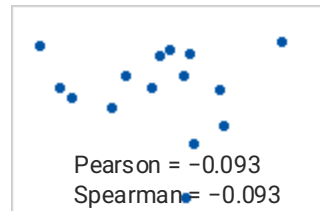
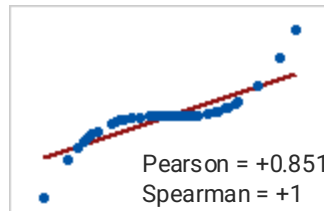
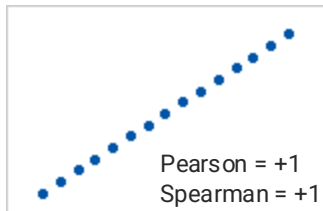
- 1) as the value of one variable increases, so does the value of the other variable, OR,
- 2) as the value of one variable increases, the other variable value decreases.

BUT, not exactly at a constant rate whereas in a linear relationship the rate of increase/decrease is constant.

**Example use case:** Whether the order in which employees complete a test exercise is related to the number of months they have been employed or correlation between the IQ of a person with the number of hours spent in front of TV per week.

### Comparison of Pearson and Spearman coefficients

1. The fundamental difference is that the Pearson coefficient works with a linear relationship between the two variables whereas the Spearman Coefficient works with monotonic relationships as well.
2. One more difference is that Pearson works with raw data values of the variables whereas Spearman works with rank-ordered variables.

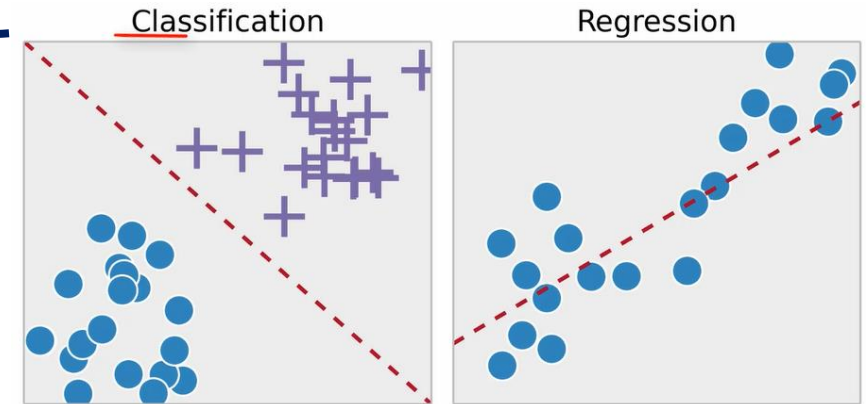


# Supervised and Unsupervised Learning

## Supervised Learning

| ID      | Clump | UnifSize | UnifShape | MargAdh | SingEpiSize | BareNuc | BlandChrom | NormNucl | Mit | Class     |
|---------|-------|----------|-----------|---------|-------------|---------|------------|----------|-----|-----------|
| 1000025 | 5     | 1        | 1         | 1       | 2           | 1       | 3          | 1        | 1   | benign    |
| 1002945 | 5     | 4        | 4         | 5       | 7           | 10      | 3          | 2        | 1   | benign    |
| 1015425 | 3     | 1        | 1         | 1       | 2           | 2       | 3          | 1        | 1   | malignant |
| 1016277 | 6     | 8        | 8         | 1       | 3           | 4       | 3          | 7        | 1   | benign    |
| 1017023 | 4     | 1        | 1         | 3       | 2           | 1       | 3          | 1        | 1   | benign    |
| 1017122 | 8     | 10       | 10        | 8       | 7           | 10      |            | 7        | 1   | malignant |
| 1018099 | 1     | 1        | 1         | 1       | 2           | 10      | 3          | 1        | 1   | benign    |
| 1018561 | 2     | 1        | 2         | H       | 2           | 1       | 3          | 1        | 1   | benign    |
| 1033078 | 2     | 1        | 1         | 1       | 2           | 1       | 1          | 1        | 5   | benign    |
| 1033078 | 4     | 2        | 1         | 1       | 2           | 1       | 2          | 1        | 1   | benign    |

## Types of Supervised Learning



**Classification** is the process of predicting a discrete class label or category.

**Regression** is the process of predicting a continuous value.

## Unsupervised Learning

| Customer Id | Age | Edu | Years Employed | Income | Card Debt | Other Debt | Address | DebtIncomeRatio |
|-------------|-----|-----|----------------|--------|-----------|------------|---------|-----------------|
| 1           | 41  | 2   | 6              | 19     | 0.124     | 1.073      | NBA001  | 6.3             |
| 2           | 47  | 1   | 26             | 100    | 4.582     | 8.218      | NBA021  | 12.8            |
| 3           | 33  | 2   | 10             | 57     | 6.111     | 5.802      | NBA013  | 20.9            |
| 4           | 29  | 2   | 4              | 19     | 0.681     | 0.516      | NBA009  | 6.3             |
| 5           | 47  | 1   | 31             | 253    | 9.308     | 8.908      | NBA008  | 7.2             |
| 6           | 40  | 1   | 23             | 81     | 0.998     | 7.831      | NBA016  | 10.9            |
| 7           | 38  | 2   | 4              | 56     | 0.442     | 0.454      | NBA013  | 1.6             |
| 8           | 42  | 3   | 0              | 64     | 0.279     | 3.945      | NBA009  | 6.6             |
| 9           | 26  | 1   | 5              | 18     | 0.575     | 2.215      | NBA006  | 15.5            |
| 10          | 47  | 3   | 23             | 115    | 0.653     | 3.947      | NBA011  | 4               |
| 11          | 44  | 3   | 8              | 88     | 0.285     | 5.083      | NBA010  | 6.1             |
| 12          | 34  | 2   | 9              | 40     | 0.374     | 0.266      | NBA003  | 1.6             |

## Unsupervised learning techniques:

- Dimension reduction
- Density estimation
- Market basket analysis
- Clustering

ALL OF THIS DATA IS UNLABELED

The model works on its own to discover information.

## Supervised Learning

- **Classification:**  
Classifies labeled data
- **Regression:**  
Predicts trends using previous labeled data
- Has more evaluation methods than unsupervised learning
- Controlled environment

## Unsupervised Learning

- **Clustering:**  
Finds patterns and groupings from unlabeled data
- Has fewer evaluation methods than supervised learning
- Less controlled environment

# Regression

## Linear Regression (LR)

**LR** is the approximation of a linear model used to describe the relationship between two or more variables.

There are two types of linear regression models:

- simple regression
- multiple regression

In **linear regression**, there are two types of variables:

- a dependent variable
- independent variable(s).

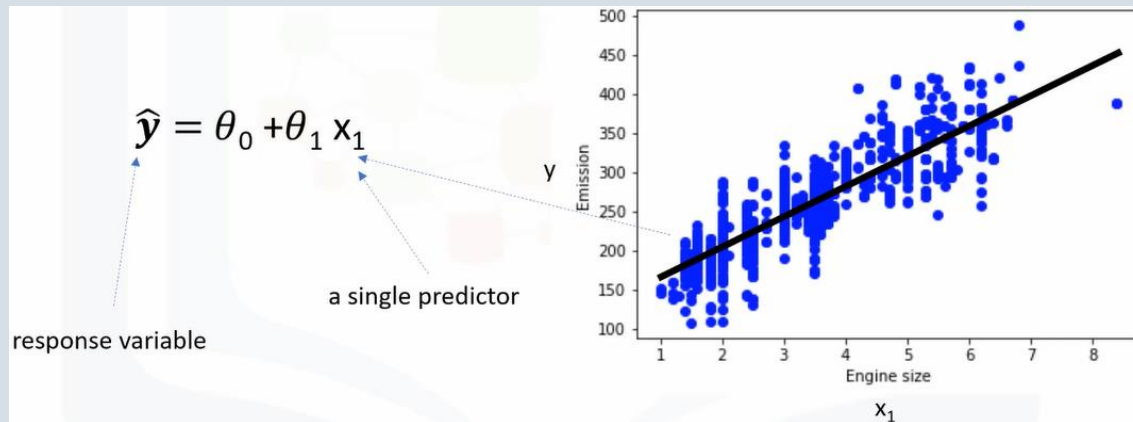
The key point in the linear regression is that our dependent value should be *continuous* and cannot be a *discrete* value. However, the independent variable(s) can be measured on either a *categorical* or *continuous* measurement scale.

X: Independent variable

Y: Dependent variable

|   | ENGINE SIZE | CYLINDERS | FUEL CONSUMPTION_COMB | CO2 EMISSIONS |
|---|-------------|-----------|-----------------------|---------------|
| 0 | 2.0         | 4         | 8.5                   | 196           |
| 1 | 2.4         | 4         | 9.6                   | 221           |
| 2 | 1.5         | 4         | 5.9                   | 136           |
| 3 | 3.5         | 6         | 11.1                  | 255           |
| 4 | 3.5         | 6         | 10.6                  | 244           |
| 5 | 3.5         | 6         | 10.0                  | 230           |
| 6 | 3.5         | 6         | 10.1                  | 232           |
| 7 | 3.7         | 6         | 11.1                  | 255           |
| 8 | 3.7         | 6         | 11.6                  | 267           |
| 9 | 2.4         | 4         | 9.2                   | ?             |

Continuous Values



### Simple Linear Regression

- Predict **CO2 emission** vs **EngineSize** of all cars
  - Independent variable (x): **EngineSize**
  - Dependent variable (y): **CO2 emission**

### Multiple Linear Regression

- Predict **CO2 emission** vs **EngineSize** and **Cylinders** of all cars
  - Independent variable (x): **EngineSize**
  - Dependent variable (y): **CO2 emission**

# Simple Linear Regression

## How to Find the Best Fit?

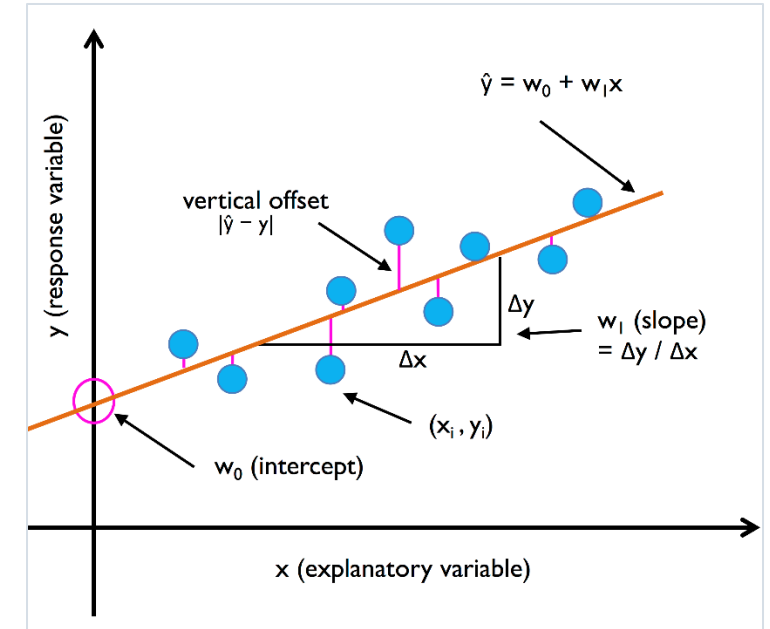
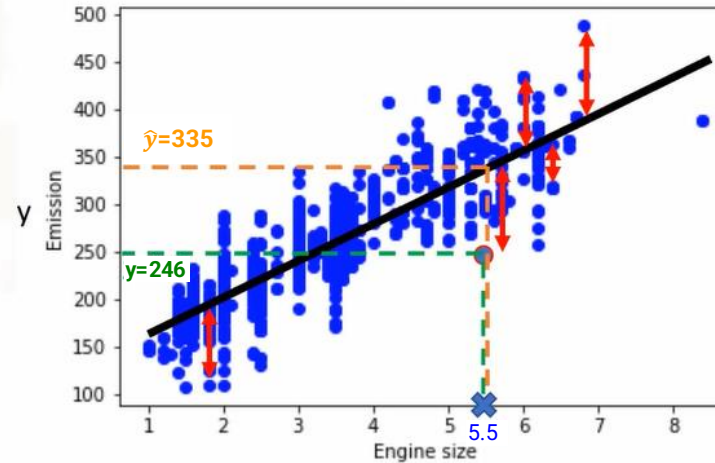
$x_1 = 5.5$  independent variable  
 $y = 246$  actual Co2 emission of  $x_1$

$$\hat{y} = \theta_0 + \theta_1 x_1$$

$\hat{y} = 335$  the predicted emission of  $x_1$

$$\begin{aligned} \text{Error} &= y - \hat{y} \\ &= 246 - 335 \\ &= -89 \end{aligned}$$

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \rightarrow \min$$



### Estimating the parameters

|   | ENGINE SIZE | CYLINDERS | FUEL CONSUMPTION_COMB | CO2 EMISSIONS |
|---|-------------|-----------|-----------------------|---------------|
| 0 | 2.0         | 4         | 8.5                   | 196           |
| 1 | 2.4         | 4         | 9.6                   | 221           |
| 2 | 1.5         | 4         | 5.9                   | 136           |
| 3 | 3.5         | 6         | 11.1                  | 255           |
| 4 | 3.5         | 6         | 10.6                  | 244           |
| 5 | 3.5         | 6         | 10.0                  | 230           |
| 6 | 3.5         | 6         | 10.1                  | 232           |
| 7 | 3.7         | 6         | 11.1                  | 255           |
| 8 | 3.7         | 6         | 11.6                  | 267           |

$$\hat{y} = \theta_0 + \theta_1 x_1$$

$$\theta_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

$$\bar{x} = \frac{2 + 2.4 + 1.5 + \dots}{9} = 3.03$$

$$\bar{y} = \frac{196 + 221 + 136 + \dots}{9} = 226$$

$$\theta_1 = \frac{(2 - 3.03)(196 - 226) + (2.4 - 3.03)(221 - 226) + \dots}{(2 - 3.03)^2 + (2.4 - 3.03)^2 + \dots} = 43.984$$

$$\theta_0 = \bar{y} - \theta_1 \bar{x}$$

$$\theta_0 = 226.22 - 43.984 \cdot 3.03 = 92.8$$

- $\theta_0$  is also called the **bias coefficient**
- $\theta_1$  is the **coefficient** for the **CO2Emission** column.

So

$$\begin{aligned} \text{CO2Emission} &= 92.8 + 43.984 \times 2.4 \\ &= 198 \end{aligned}$$

$$\begin{aligned} \text{CO2Emission} &= 92.8 + 43.984 \times 5.4 \\ &= 335 \end{aligned}$$

# Regression Line

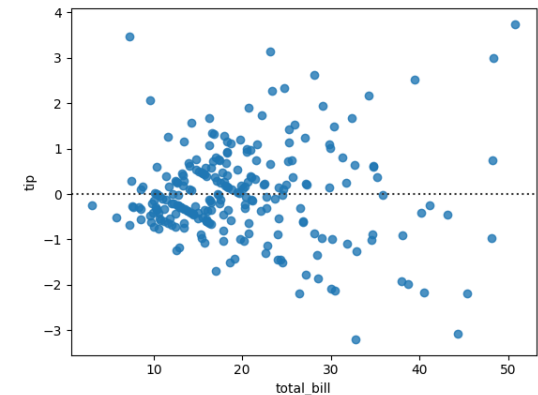
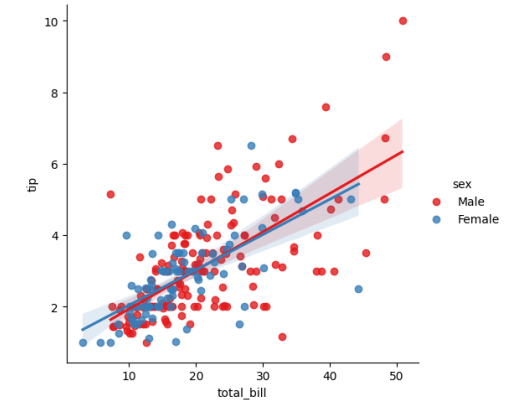
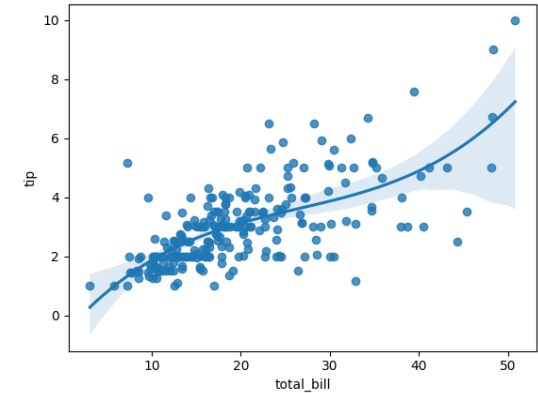
## Visualization with Matplotlib and Seaborn

- **regplot** is used to plot the line regressions and displaying points on the graph
- **lmlplot** allows you to plot dependency graphs between variables for different subsets of the data.
- **residplot** allows you to display a graph of residuals regression model.
- **regplot** and **lmlplot** are designed for building graphs that show dependencies between variables and regression lines. However, **lmlplot** allows you to build graphs for different subgroups of data using the **hue** parameter. Additionally, you can split graphs into columns and rows based on variable values using the **col** and **row** parameters.
- **residplot** is used for displaying the residuals of a regression line. This plot helps to assess how well the linear regression fits the dependency between variables. The residuals are displayed on the Y-axis, while the predicted variable values are on the X-axis. An optional zero line can be added to the plot to indicate the range of values where residuals are either missing or minimal.

```
sns.regplot(data=tips, x="total_bill", y="tip", order = 3)
```

```
sns.lmlplot(data=tips, x="total_bill", y=tip, hue="sex", palette="Set1")
```

```
sns.residplot(data=tips, x="total_bill", y=tip)
```



### Additional parameters:

- <https://seaborn.pydata.org/generated/seaborn.lmlplot.html>

# Multiple Linear Regression

## Multiple Regression Model

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n + \varepsilon$$

where:  $\theta_0$  (intercept),  $\theta_1, \theta_2, \dots, \theta_n$  are the parameters and  $\varepsilon$  is a random variable called the *error term*

### (Estimated) Multiple Regression Equation

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

For a good multiple linear regression model, we should exclude the multicollinearity.

**Multicollinearity** means that the independent variables are highly correlated. It can be dealt with regularization.

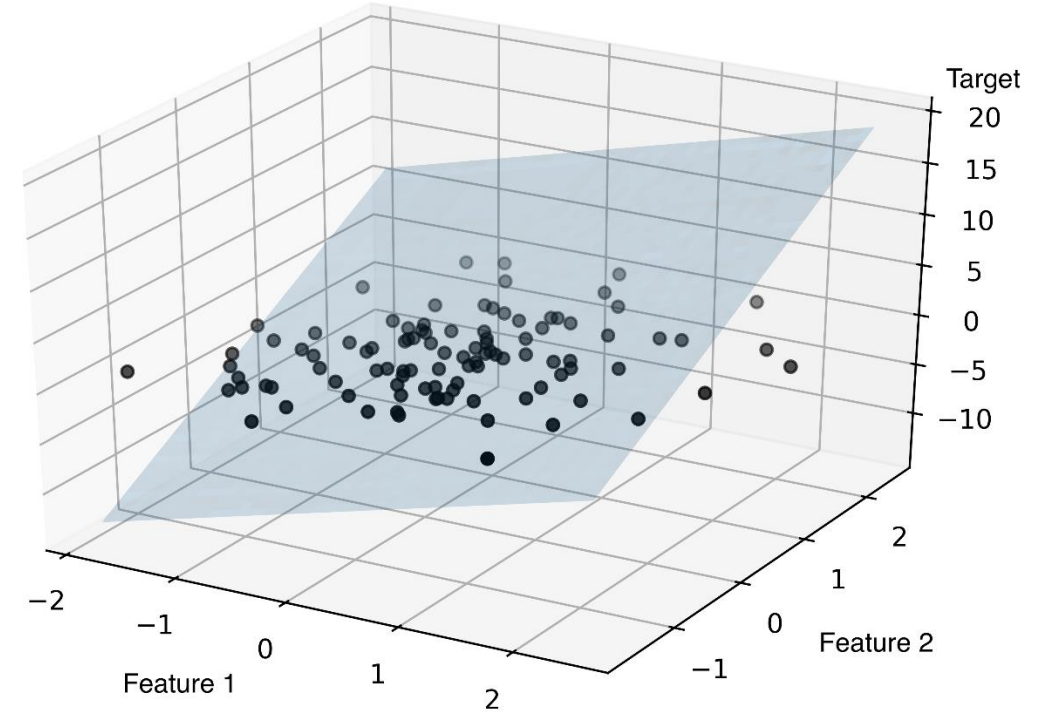
### Ordinary Least Squares (OLS)

OLS is a method for estimating the unknown parameters in a linear regression model. OLS chooses the parameters of a linear function of a set of explanatory variables by minimizing the sum of the squares of the differences between the target dependent variable and those predicted by the linear function. In other words, it tries to minimize the sum of squared errors (SSE) or mean squared error (MSE) between the target variable ( $y$ ) and our predicted output ( $\hat{y}$ ) over all samples in the dataset.

OLS can find the best parameters using of the following methods:

- Solving the model parameters analytically using closed-form equations
- Using an optimization algorithm (Gradient Descent, Stochastic Gradient Descent, Newton's Method, etc.)

## Two features (predictors, factors)



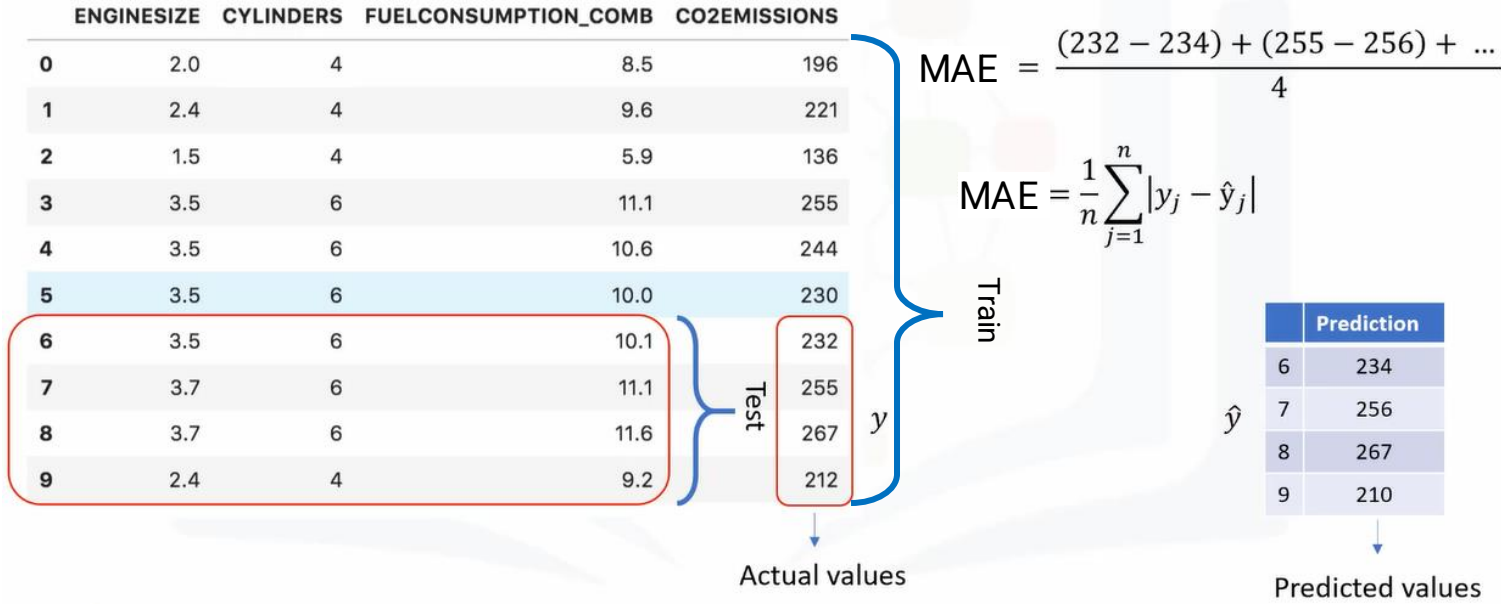
|            | Coeffic. | Std. Err. | t Stat | P-value |
|------------|----------|-----------|--------|---------|
| Intercept  | 3.17394  | 6.15607   | 0.5156 | 0.61279 |
| Experience | 1.4039   | 0.19857   | 7.0702 | 1.9E-06 |
| Test Score | 0.25089  | 0.07735   | 3.2433 | 0.00478 |

$$\text{SALARY} = 3.174 + 1.404(\text{EXPER}) + 0.251(\text{SCORE})$$

# Regression

## Model evaluation

### Train and test on the same dataset



### Metrics:

- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)
- Mean Absolute Error (MAE)
- R Squared ( $R^2$ )
- Adjusted R Squared ( $R^2_{adj}$ )
- Mean Squared Percentage Error (MSPE)
- Mean Absolute Percentage Error (MAPE)
- Root Mean Squared Logarithmic Error (RMSLE)

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} = \sqrt{MSE}$$

$$R^2 = 1 - \frac{MSE(model)}{MSE(baseline)} \quad R^2_{adj} = 1 - \left[ \frac{(1 - R^2)(n-1)}{n-k-1} \right]$$

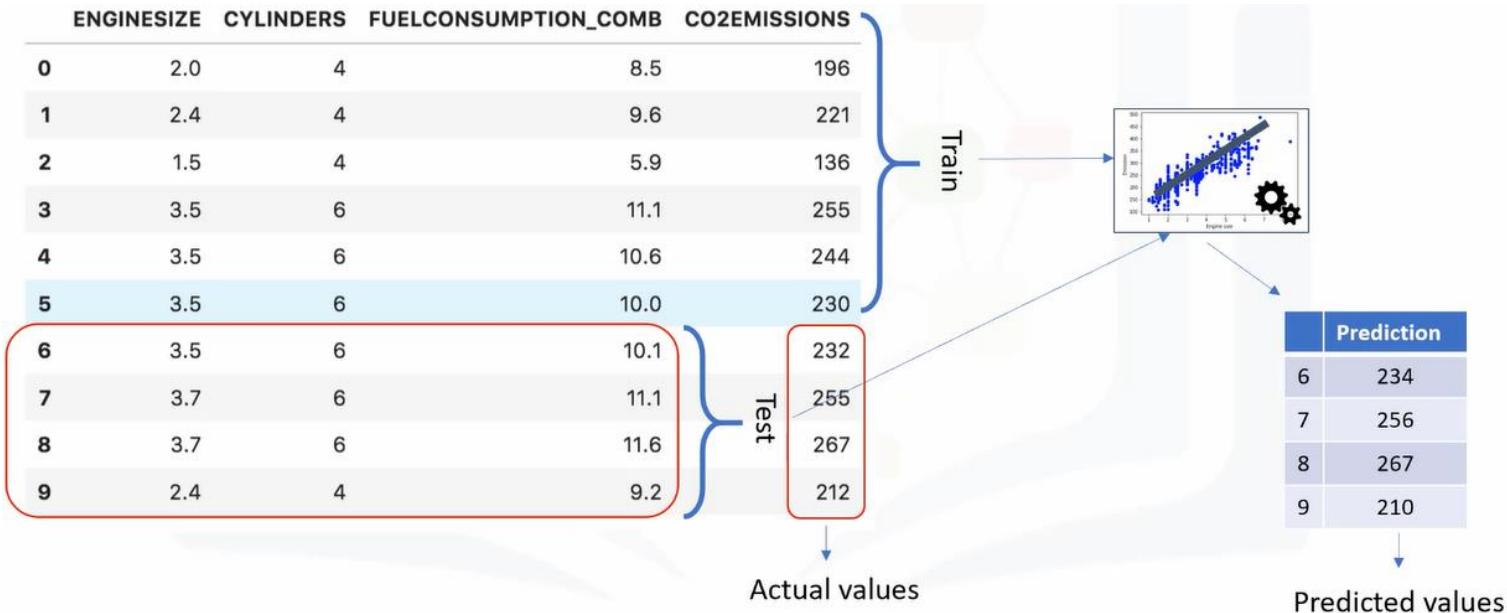
$$MSE(baseline) = \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2$$

$$MSPE = \frac{100\%}{n} \sum_{i=1}^n \left( \frac{y_i - \hat{y}_i}{y_i} \right)^2$$

$$MAPE = \frac{100\%}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

$$RMSLE = \sqrt{\frac{1}{n} \sum_{i=1}^n (\log(y_i + 1) - \log(\hat{y}_i + 1))^2}$$

### Train and test split



# Regression

## StatsModels Library

**Statsmodels** is a statistical modeling library that provides a wide range of statistical methods, including regression, classification, clustering, time series, and methods for working with panel data.

<https://www.statsmodels.org/stable/index.html>

### Key Features and Methods:

- **Exploratory Data Analysis (EDA):**
  - data visualization
  - summary statistics
  - hypothesis testing:
    - t-tests, chi-squared tests, ANOVA, etc.
    - Non-parametric tests: Mann-Whitney U test, Kruskal-Wallis test, etc.
- **Regression Analysis:**
  - Ordinary least squares (OLS)
  - Weighted Least Squares (WLS) regression
  - Generalized Linear Models (GLM)
  - Poisson regression, and more.
- **Time Series Analysis:**
  - Autoregressive Integrated Moving Average (ARIMA) models
  - Vector Autoregression (VAR) models
  - VECM models, and others.
- **Multivariate Analysis:**
  - Principal component analysis (PCA)
  - Factor analysis
  - Structural equation modeling.

```
import statsmodels.api models as sm
```

```
# We separate the dependent variable and the independent variables
```

```
y = tips["tip"]
```

```
X = tips[["total_bill", "size"]]
```

```
# We add a constant to the independent variables
```

```
X = sm.add_constant(X)
```

```
# Creating a regression model and its training
```

```
model = sm.OLS(y, X).fit()
```

```
# Display the results of the model
```

```
print(model.summary())
```

```
# Graph of the dependence of the variable "tip" on "total_bill"
```

```
sns.scatterplot(x="total_bill", y="tip", data=tips)
```

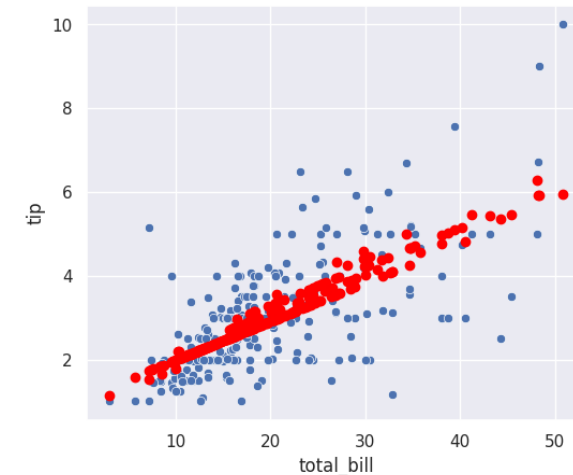
```
# Predicted values of the model
```

```
tips["predicted_tip"] = model.predict(X)
```

```
# Linear regression line
```

```
sns.lineplot(x="total_bill", y="predicted_tip", data=tips, color="red")
```

| OLS Regression Results |                  |                     |          |       |        |        |
|------------------------|------------------|---------------------|----------|-------|--------|--------|
| Dep. Variable:         | tip              | R-squared:          | 0.468    |       |        |        |
| Model:                 | OLS              | Adj. R-squared:     | 0.463    |       |        |        |
| Method:                | Least Squares    | F-statistic:        | 105.9    |       |        |        |
| Date:                  | Tue, 29 Oct 2024 | Prob (F-statistic): | 9.67e-34 |       |        |        |
| Time:                  | 10:50:51         | Log-Likelihood:     | -347.99  |       |        |        |
| No. Observations:      | 244              | AIC:                | 702.0    |       |        |        |
| Df Residuals:          | 241              | BIC:                | 712.5    |       |        |        |
| Df Model:              | 2                |                     |          |       |        |        |
| Covariance Type:       | nonrobust        |                     |          |       |        |        |
|                        | coef             | std err             | t        | P> t  | [0.025 | 0.975] |
| const                  | 0.6689           | 0.194               | 3.455    | 0.001 | 0.288  | 1.050  |
| total_bill             | 0.0927           | 0.009               | 10.172   | 0.000 | 0.075  | 0.111  |
| size                   | 0.1926           | 0.085               | 2.258    | 0.025 | 0.025  | 0.361  |
| Omnibus:               | 24.753           | Durbin-Watson:      | 2.100    |       |        |        |
| Prob(Omnibus):         | 0.000            | Jarque-Bera (JB):   | 46.169   |       |        |        |
| Skew:                  | 0.545            | Prob(JB):           | 9.43e-11 |       |        |        |
| Kurtosis:              | 4.831            | Cond. No.           | 67.6     |       |        |        |





# Regression

## Scikit-Learn Library

```
from sklearn.linear_model import LinearRegression

# Load the 'tips' dataset
tips = sns.load_dataset('tips')

# Assuming 'tips' is a pandas DataFrame loaded with your
data
y = tips["tip"]
X = tips[["total_bill", "size"]]

# Building and training the model
model = LinearRegression().fit(X, y)

# The coefficients and intercept can be accessed like
this
print(f'Coefficients: {model.coef_}')
print(f'Intercept: {model.intercept_}')

# Plotting the relationship between "tip" and
"total_bill"
sns.scatterplot(x="total_bill", y="tip", data=tips)

# Predicting tips
tips["predicted_tip"] = model.predict(X)

# Adding the linear regression line to the scatterplot
sns.lineplot(x="total_bill", y="predicted_tip",
data=tips, color="red")

Coefficients: [0.09271334 0.19259779] Intercept:
0.6689447408125022
<Axes: xlabel='total_bill', ylabel='tip'>
```

## Regression process

1. Understand data relationships
2. Select features
3. Select metrics
4. Create model
5. Evaluate model
6. Improve model
  1. Start by understanding the residuals (errors)
  2. Filter or transform the data
  3. Better feature engineering
  4. Improve feature selection
  5. Use a different type of model
  6. Choice of model parameters
7. Validate model

## Required Conditions for the Error Variable $\varepsilon$

1. The error  $\varepsilon$  is normally distributed.
2. The mean is equal to zero and the standard deviation is constant (se) for all possible values of the  $X_i$ s.
3. All errors are independent.

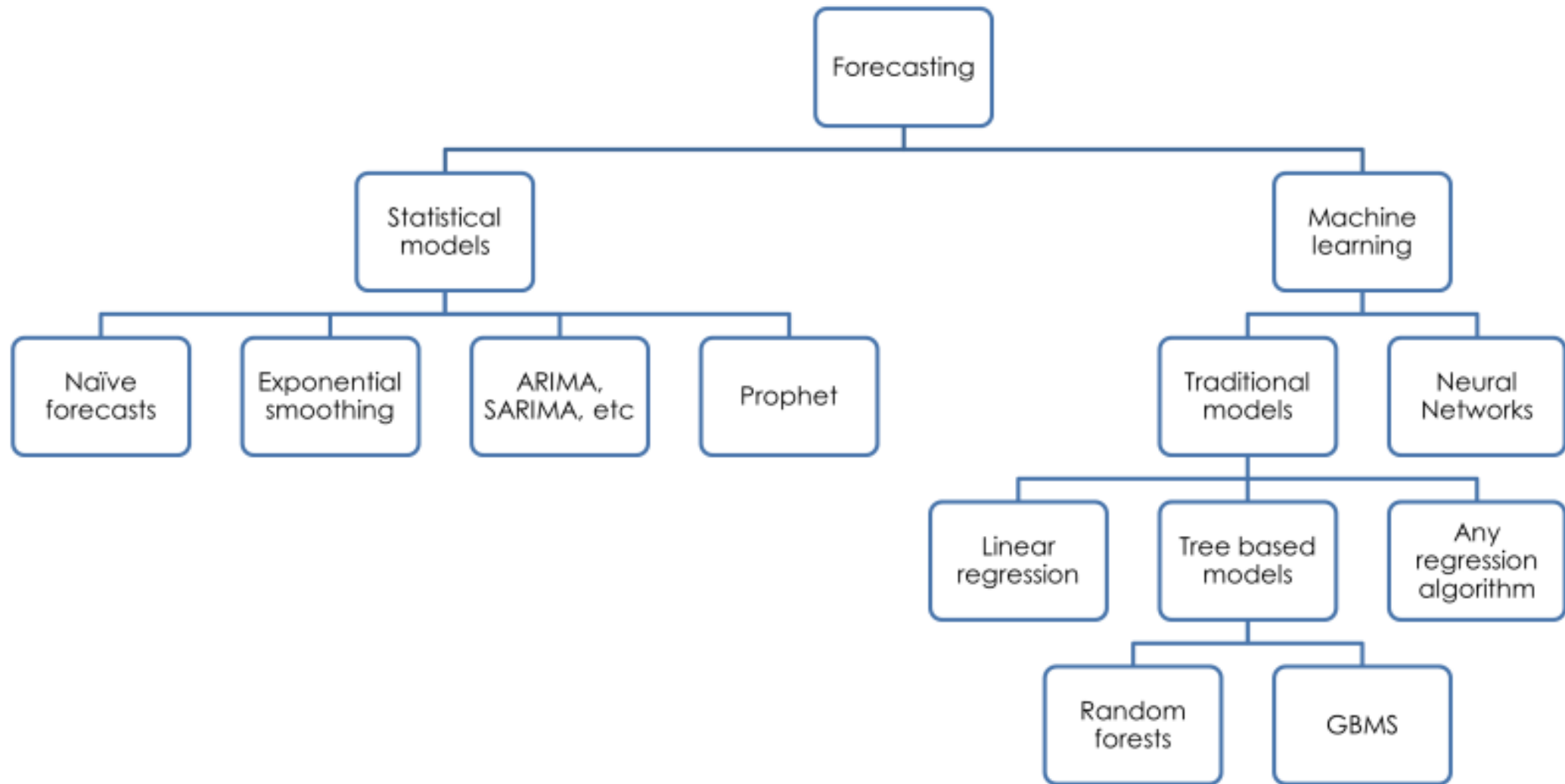


## Section 2.2

# **Time Series Analysis with Python**



# Time Series Analysis with Python





# Time Series Analysis

## Augmented Dickey-Fuller Test. Stationary. Decomposition into Trend, Seasonal and Residual

```
import statsmodels.api as sm
# Get Data
url =
"https://raw.githubusercontent.com/facebook/prophet/main/example
es/example_air_passengers.csv"
df = pd.read_csv(url, header=0, parse_dates=[0], index_col=0,)
```

Before creating a Time Series Model, we need to ensure that the data is stationary. A stationary dataset has constant mean and variance over time. We will use the Augmented Dickey-Fuller (ADF) test to check for stationarity:

```
from statsmodels.tsa.stattools import adfuller
def adf_test(data):
    result = adfuller(data)
    print('ADF Statistic: {}'.format(result[0]))
    print('p-value: {}'.format(result[1]))
    print('Critical Values:')
    for key, value in result[4].items():
        print('\t{}: {}'.format(key, value))
    adf_test(df['y'])
ADF Statistic: 0.8153688792060498
p-value: 0.991880243437641
Critical Values:
1%: -3.4816817173418295
5%: -2.8840418343195267
10%: -2.578770059171598
```

If the p-value is less than 0.05, the data is stationary. If not, we need to transform the data to make it stationary.

**Make the data stationary:** One common technique to make the data stationary is to apply differencing. Let's apply second-order differencing and check for stationarity again:

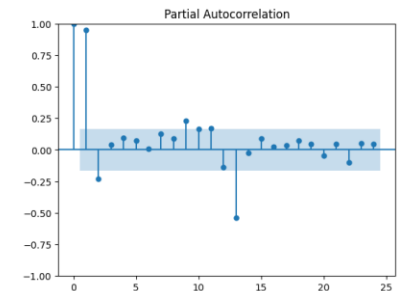
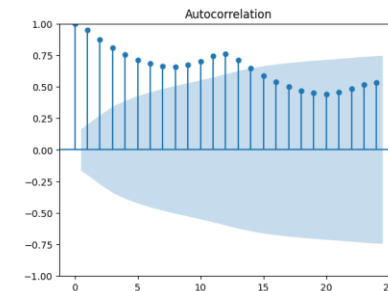
```
df_diff = df.diff(2).dropna()
adf_test(df_diff['y'])
```

**Decompose the dataset:** Now that our data is stationary, let's decompose it into its trend, seasonal, and residual components:

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(df_diff)
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
plt.figure(figsize=(12, 8))
plt.subplot(411)
plt.plot(df, label='Original')
plt.legend(loc='upper left')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='upper left')
plt.subplot(413)
plt.plot(seasonal, label='Seasonal')
plt.legend(loc='upper left')
plt.subplot(414)
plt.plot(residual, label='Residual')
plt.legend(loc='upper left')
```

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

```
plot_acf(df_diff['y'], lags=24)
plot_pacf(df_diff['y'], lags=24)
```



# Time Series Analysis

## ARIMA and SARIMA. Prophet

```
# Fit an ARIMA model
model = sm.tsa.ARIMA(df, order=(14, 2, 1))
results = model.fit()

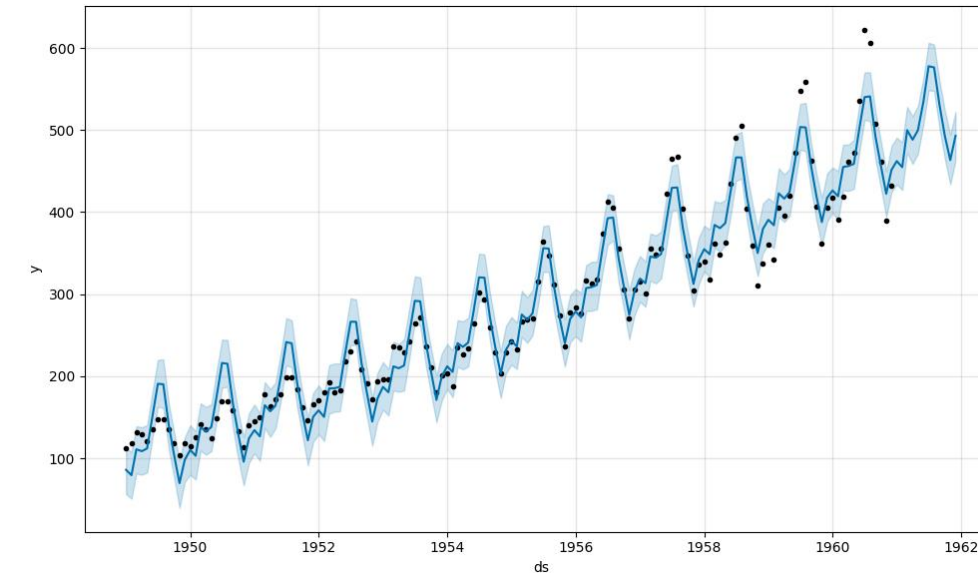
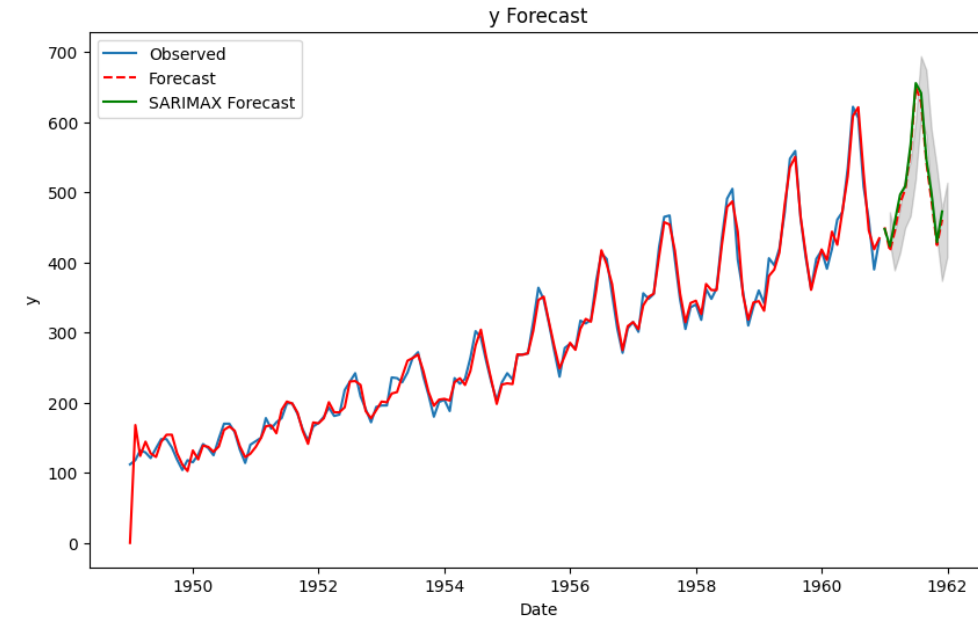
# Forecast
forecast = results.get_forecast(steps=12)
forecast_index = pd.date_range(start=df.index[-1]+ pd.DateOffset(months=1),
                                periods=12, freq='M')
forecast_df = pd.DataFrame(forecast.predicted_mean, index=forecast_index,
                            columns=['Forecast'])

# Fit an SARIMA model
sarimax_model = sm.tsa.SARIMAX(df, order=(1,2,1), seasonal_order=(1,1,1,12))
sarimax_result = sarimax_model.fit()
sarimax_forecast = sarimax_result.forecast(steps=12)

# Plot
plt.figure(figsize=(10, 6))
plt.plot(df['y'], label='Observed')
plt.plot(forecast.predicted_mean, label='Forecast', color='red',
         linestyle='--')
plt.plot(results.fittedvalues, color='red')
plt.plot(sarimax_forecast, label='SARIMAX Forecast', color='green')
plt.xlabel('Date'); plt.ylabel('y'); plt.title('y Forecast'); plt.legend()
plt.fill_between(forecast_index, forecast.conf_int().iloc[:, 0],
                 forecast.conf_int().iloc[:, 1], color='k', alpha=.15)

print(results.summary())

# Prophet
from prophet import Prophet
model = Prophet()
model.fit(df.reset_index())
future = model.make_future_dataframe(periods=12, freq='M')
forecast = model.predict(future)
model.plot(forecast)
```



# Time Series Analysis

## Forecasting with machine learning models in Python: library “skforecast”.

```
from skforecast.ForecasterAutoreg import ForecasterAutoreg
from sklearn.linear_model import Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
```

### # Model 1: Linear Regression

```
forecaster_lr =
ForecasterAutoreg(regressor=Lasso(random_state=10),
lags=12)
forecaster_lr.fit(y=pd.Series(train['y']))
predictions_lr = forecaster_lr.predict(steps=12)
```

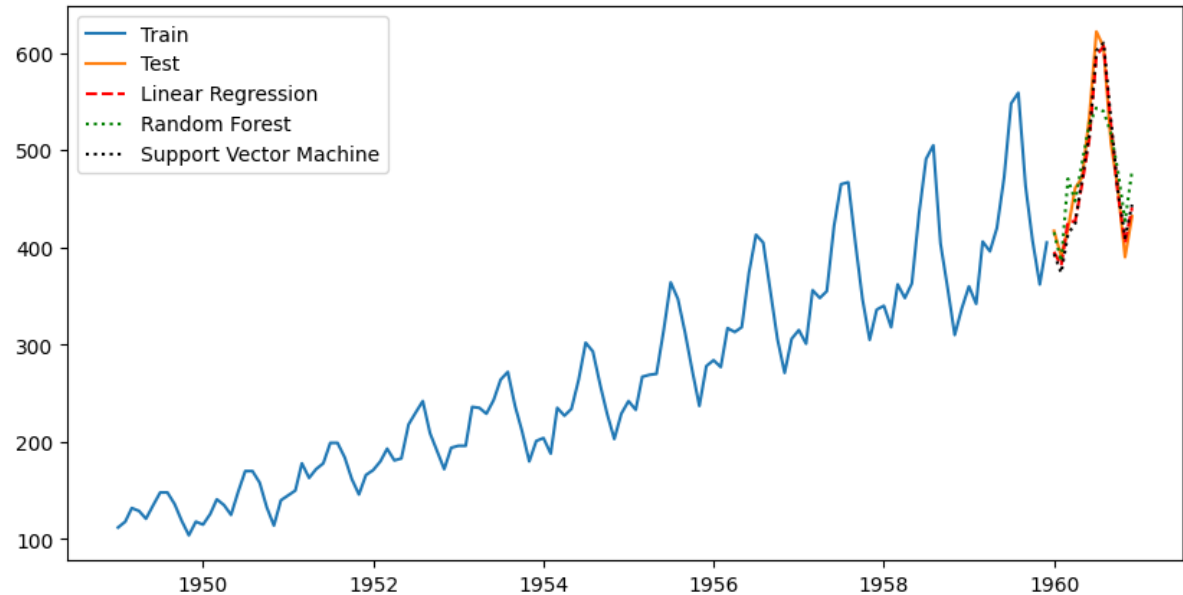
### # Model 2: Random Forest

```
forecaster_rf = ForecasterAutoreg(regressor=
RandomForestRegressor(n_estimators=100, random_state=42),
lags=12)
forecaster_rf.fit(y=pd.Series(train['y']))
predictions_rf = forecaster_rf.predict(steps=12)
```

### # Model 3: Support Vector Machine (SVR)

```
forecaster_svr =
ForecasterAutoreg(regressor=SVR(kernel='linear'), lags=12)
forecaster_svr.fit(y=pd.Series(train['y']))
predictions_svr = forecaster_svr.predict(steps=12)
```

```
plt.figure(figsize=(10, 5))
plt.plot(train, label='Train')
plt.plot(test, label='Test')
plt.plot(test.index, predictions_lr, label='Linear
Regression', linestyle='--', color='red')
plt.plot(test.index, predictions_rf, label='Random
Forest', linestyle=':', color='green')
plt.plot(test.index, predictions_svr, label='Support Vector
Machine', linestyle=':', color='black')
plt.legend()
```





# Time Series Analysis

## Forecasting with Deep Learning

```
from math import sqrt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import r2_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
```

```
from skforecast.datasets import fetch_dataset
```

```
data = fetch_dataset('bike_sharing', raw=True)
dataset = data['users'].values.reshape(-1, 1)
```

```
# scale down the values between 0 and 1
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

```
# train-test split
```

```
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:],
dataset[train_size:len(dataset),:]
```

bike\_sharing ----- Hourly usage of the bike share system in the city of Washington D.C. during the years 2011 and 2012. In addition to the number of users per hour, information about weather conditions and holidays is available. Fanaee-T,Hadi. (2013). Bike Sharing Dataset. UCI Machine Learning Repository. <https://doi.org/10.24432/C5W894>. Shape of the dataset: (17544, 12)

```
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
```

```
look_back = 1
```

```
# separate the ylabel
```

```
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
trainX = np.reshape(trainX, (trainX.shape[0], 1,
trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1,
testX.shape[1]))
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=6, batch_size=1,
verbose=2)
```

```
# get the final rmse after prediction
```

```
testPredict = model.predict(testX)
testScore = sqrt(mean_squared_error(testY,
testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
print('R2_score: %.2f' % r2_score(testY, testPredict[:,0]))
```

```
RMSE: 0.12
```

```
R2_score: 0.69
```



# Topic 3

## **Classification with Python: Decision Trees, KNN, SVM, and Machine Learning Methods**

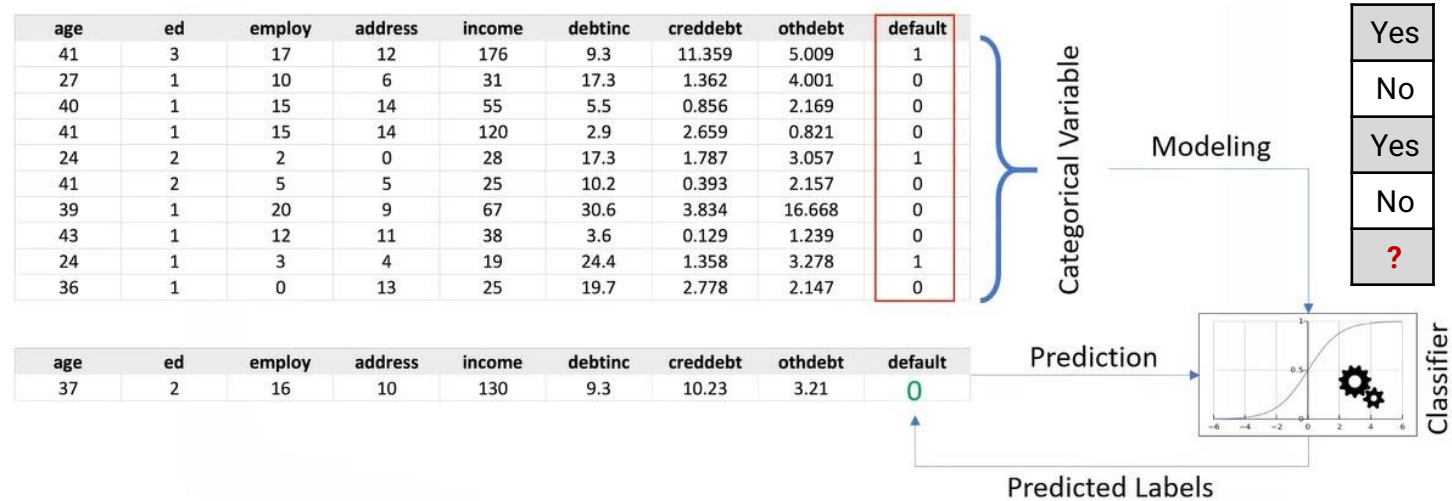
# Classification

## Introduction

**Classification** is a supervised learning approach, which can be thought of as a means of categorizing or "classifying" some unknown items into a discrete set of "classes."

**Classification** attempts to learn the relationship between a set of feature variables and a target variable of interest.

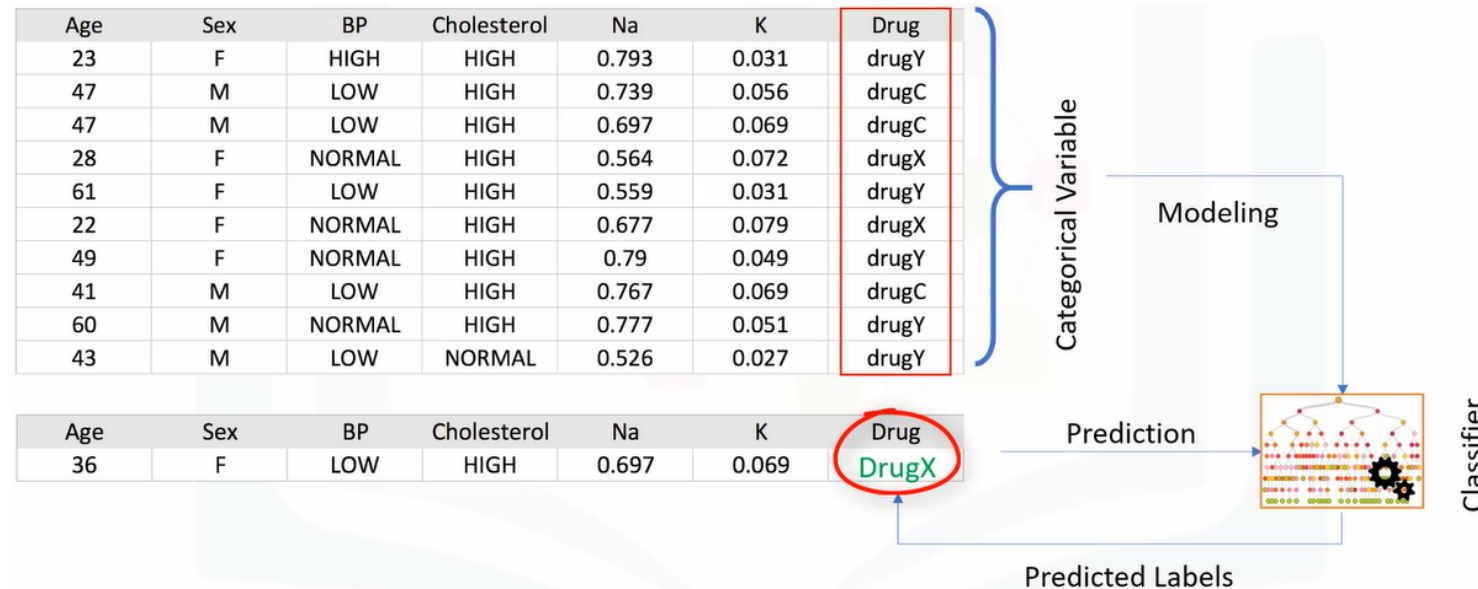
The target attribute in classification is a categorical variable with discrete values.



### Multi-Class Classification

Classification has different business use cases, e.g.:

- To predict the category to which a customer belongs;
- For churn detection, where we predict whether a customer switches to another provider or brand;
- To predict whether or not a customer responds to a particular advertising campaign.



# Classification

## KNN algorithm: concepts

The types of classification algorithms in machine learning:

- Decision Trees,
- Naïve Bayes,
- Linear Discriminant Analysis,
- K-nearest neighbors (KNN),
- Logistic regression,
- Neural Networks,
- Support Vector Machines.

**K-Nearest Neighbors** is an algorithm for supervised learning. Where the data is 'trained' with data points corresponding to their classification. Once a point is to be predicted, it takes into account the 'K' nearest points to it to determine its classification.

X: Independent variable

Y: Dependent variable

|   | region | age | marital | address | income | ed | employ | retire | gender | reside | custcat |
|---|--------|-----|---------|---------|--------|----|--------|--------|--------|--------|---------|
| 0 | 2      | 44  | 1       | 9       | 64     | 4  | 5      | 0      | 0      | 2      | 1       |
| 1 | 3      | 33  | 1       | 7       | 136    | 5  | 5      | 0      | 0      | 6      | 4       |
| 2 | 3      | 52  | 1       | 24      | 116    | 1  | 29     | 0      | 1      | 2      | 3       |
| 3 | 2      | 33  | 0       | 12      | 33     | 2  | 0      | 0      | 1      | 1      | 1       |
| 4 | 2      | 30  | 1       | 9       | 30     | 1  | 2      | 0      | 0      | 4      | 3       |
| 5 | 2      | 39  | 0       | 17      | 78     | 2  | 16     | 0      | 1      | 1      | 3       |
| 6 | 3      | 22  | 1       | 2       | 19     | 2  | 4      | 0      | 1      | 5      | 2       |
| 7 | 2      | 35  | 0       | 5       | 76     | 2  | 10     | 0      | 0      | 3      | 4       |
| 8 | 3      | 50  | 1       | 7       | 166    | 4  | 31     | 0      | 0      | 5      | ?       |

| Value | Label         |
|-------|---------------|
| 1     | Basic Service |
| 2     | E-Service     |
| 3     | Plus Service  |
| 4     | Total Service |

## Euclidian distance



| Customer 1 |
|------------|
| Age        |
| 54         |



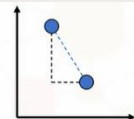
| Customer 2 |
|------------|
| Age        |
| 50         |

## 1-dimensional space

$$\text{Dis}(x_1, x_2) = \sqrt{\sum_{i=0}^n (x_{1i} - x_{2i})^2}$$

$$\text{Dis}(x_1, x_2) = \sqrt{(34 - 30)^2} = 4$$

| Customer 1 |
|------------|
| Age        |
| 54         |
| Income     |
| 190        |

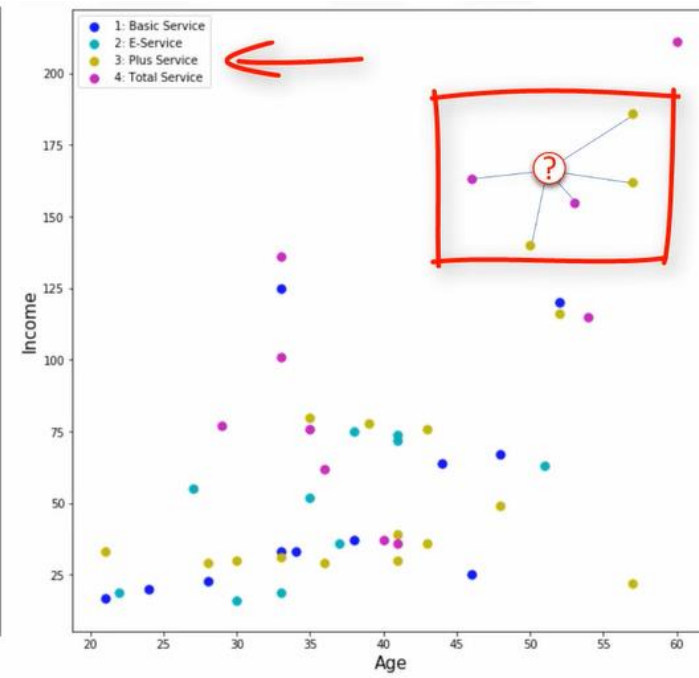
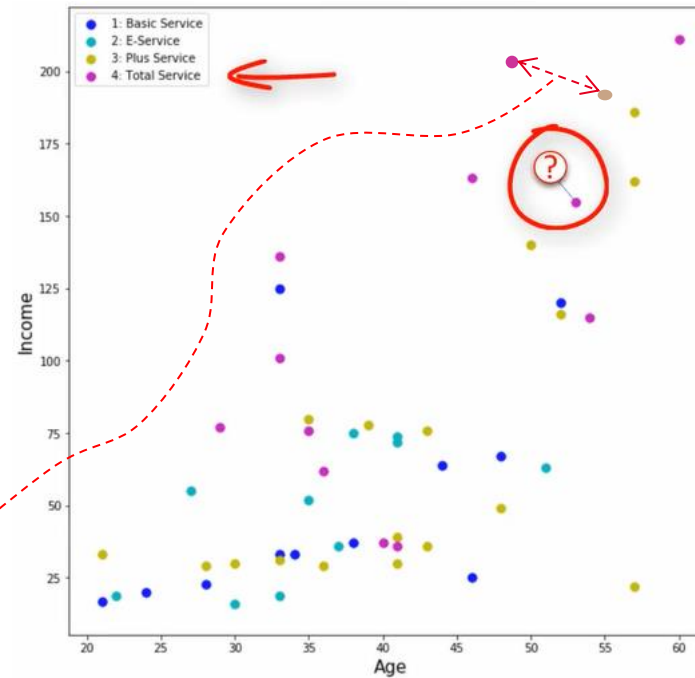


| Customer 2 |
|------------|
| Age        |
| 50         |
| Income     |
| 200        |

## 2-dimensional space

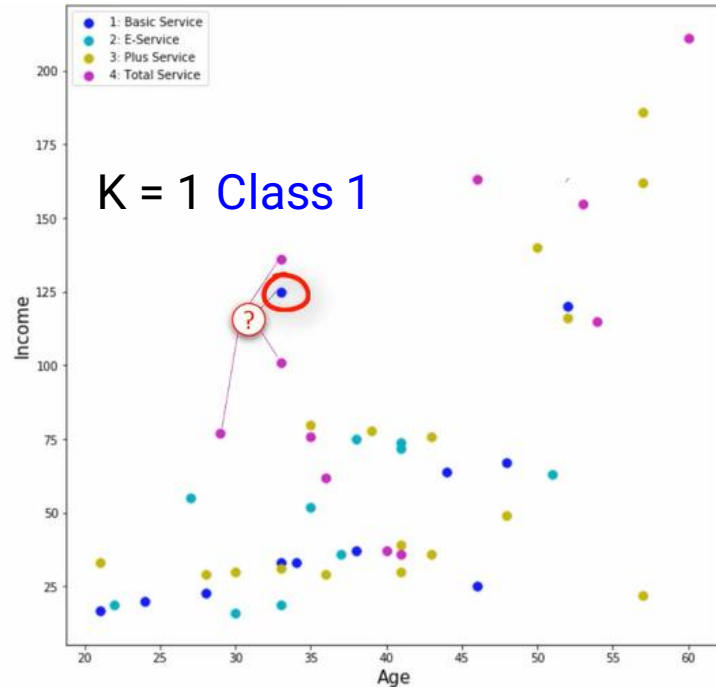
$$\text{Dis}(x_1, x_2) = \sqrt{\sum_{i=0}^n (x_{1i} - x_{2i})^2}$$

$$= \sqrt{(54 - 50)^2 + (190 - 200)^2} = 10.77$$



# Classification

## KNN algorithm: best value of K

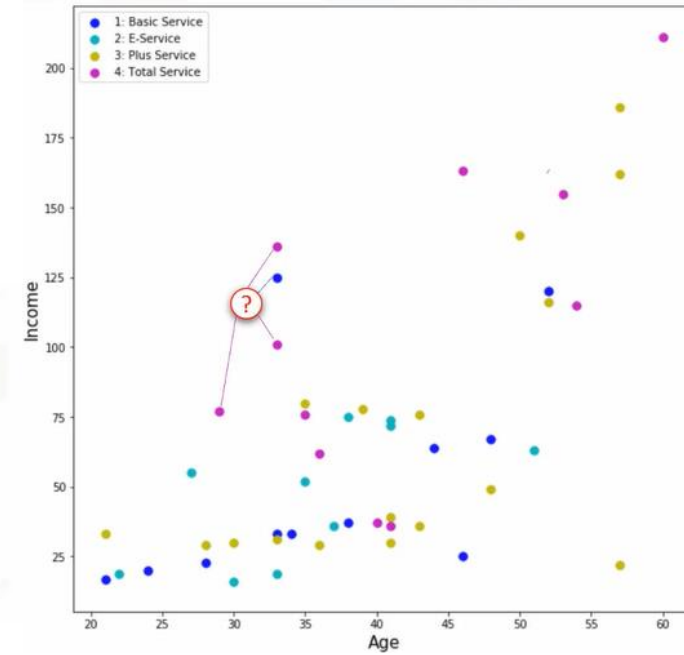


K = 1 Class 1

...



K = 4 Class 4



The **best choice of k** depends upon the data; generally, larger values of k reduces effect of the noise on the classification, but make boundaries between classes less distinct.

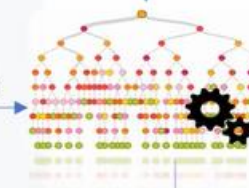
# Classification

## SVM algorithm: concepts

| ID      | Clump | UnifSize | UnifShape | MargAdh | SingEpiSize | BareNuc | BlandChrom | NormNucl | Mit | Class     |
|---------|-------|----------|-----------|---------|-------------|---------|------------|----------|-----|-----------|
| 1000025 | 5     | 1        | 1         | 1       | 2           | 1       | 3          | 1        | 1   | benign    |
| 1002945 | 5     | 4        | 4         | 5       | 7           | 10      | 3          | 2        | 1   | benign    |
| 1015425 | 3     | 1        | 1         | 1       | 2           | 2       | 3          | 1        | 1   | malignant |
| 1016277 | 6     | 8        | 8         | 1       | 3           | 4       | 3          | 7        | 1   | benign    |
| 1017023 | 4     | 1        | 1         | 3       | 2           | 1       | 3          | 1        | 1   | benign    |
| 1017122 | 8     | 10       | 10        | 8       | 7           | 10      |            | 7        | 1   | malignant |
| 1018099 | 1     | 1        | 1         | 1       | 2           | 10      | 3          | 1        | 1   | benign    |
| 1018561 | 2     | 1        | 2         | H       | 2           | 1       | 3          | 1        | 1   | benign    |
| 1033078 | 2     | 1        | 1         | 1       | 2           | 1       | 1          | 1        | 5   | benign    |
| 1033078 | 4     | 2        | 1         | 1       | 2           | 1       | 2          | 1        | 1   | benign    |

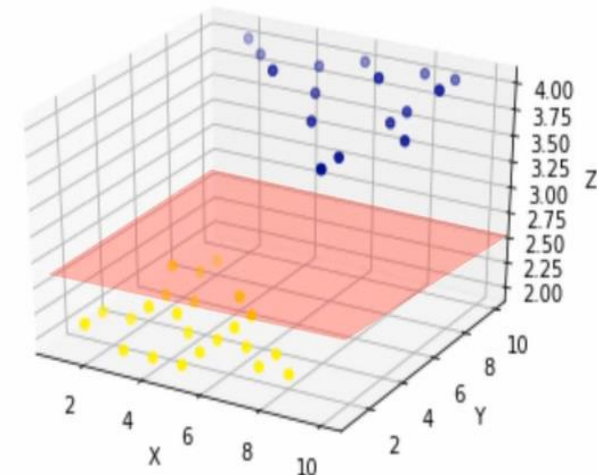
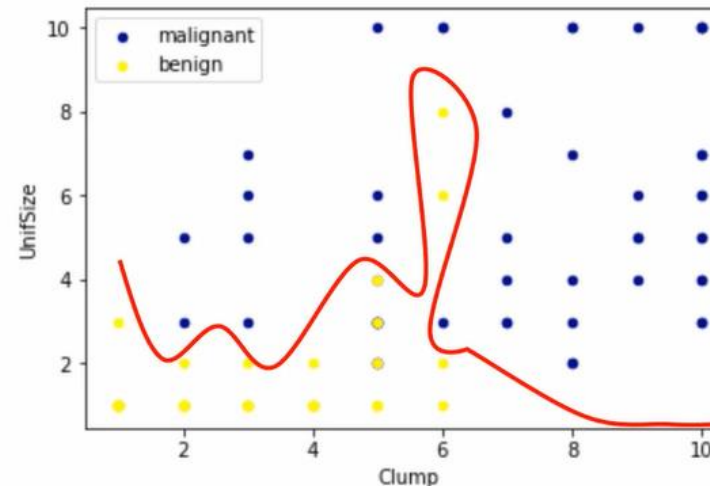
Modeling

Prediction



Accuracy = 89%

| Clump | UnifSize | UnifShape | MargAdh | SingEpiSize | BareNuc | BlandChrom | NormNucl | Mit | Class     |
|-------|----------|-----------|---------|-------------|---------|------------|----------|-----|-----------|
| 5     | 1        | 1         | 1       | 2           | 1       | 3          | 1        | 1   | benign    |
| 5     | 4        | 4         | 5       | 7           | 10      | 3          | 2        | 1   | benign    |
| 3     | 1        | 1         | 1       | 2           | 2       | 3          | 1        | 1   | malignant |
| 6     | 8        | 8         | 1       | 3           | 4       | 3          | 7        | 1   | benign    |
| 4     | 1        | 1         | 3       | 2           | 1       | 3          | 1        | 1   | benign    |
| 8     | 10       | 10        | 8       | 7           | 10      |            | 7        | 1   | malignant |
| 1     | 1        | 1         | 1       | 2           | 10      | 3          | 1        | 1   | benign    |
| 2     | 1        | 2         | H       | 2           | 1       | 3          | 1        | 1   | benign    |
| 2     | 1        | 1         | 1       | 2           | 1       | 1          | 1        | 5   | benign    |
| 4     | 2        | 1         | 1       | 2           | 1       | 2          | 1        | 1   | benign    |



A **Support Vector Machine** is a supervised algorithm that can classify cases by finding a separator.

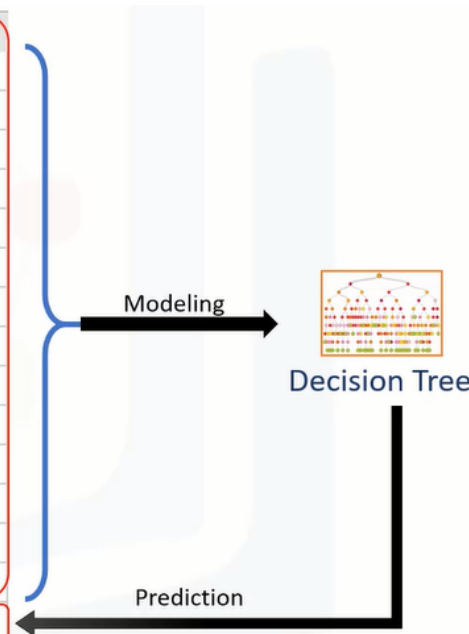
SVM works by first, mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. Then, a separator is estimated for the data. The data should be transformed in such a way that a separator could be drawn as a hyperplane.

# Classification

## Decision Tree: concepts

- **Root node:** This is the topmost node from which a decision tree is constructed. It represents the entire population/sample, and it further gets divided into two or more nodes (The purple root node in the figure above).
- **Parent/Child node:** Root node is the parent node, and all other nodes branched from a parent/root node is called a child node.
- **Branch/Sub Tree:** Formed by splitting the tree/node.
- **Splitting:** Dividing the root/sub node into different parts on the basis of some condition
- **Leaf node:** Node that cannot be further divided / segregated
- **Pruning:** Removing unwanted branches from a tree.

| Patient ID | Age        | Sex | BP     | Cholesterol | Drug   |
|------------|------------|-----|--------|-------------|--------|
| p1         | Young      | F   | High   | Normal      | Drug A |
| p2         | Young      | F   | High   | High        | Drug A |
| p3         | Middle-age | F   | Hiigh  | Normal      | Drug B |
| p4         | Senior     | F   | Normal | Normal      | Drug B |
| p5         | Senior     | M   | Low    | Normal      | Drug B |
| p6         | Senior     | M   | Low    | High        | Drug A |
| p7         | Middle-age | M   | Low    | High        | Drug B |
| p8         | Young      | F   | Normal | Normal      | Drug A |
| p9         | Young      | M   | Low    | Normal      | Drug B |
| p10        | Senior     | M   | Normal | Normal      | Drug B |
| p11        | Young      | M   | Normal | High        | Drug B |
| p12        | Middle-age | F   | Normal | High        | Drug B |
| p13        | Middle-age | M   | High   | Normal      | Drug B |
| p14        | Senior     | F   | Normal | High        | Drug A |
| p15        | Middle-age | F   | Low    | Normal      | ?      |

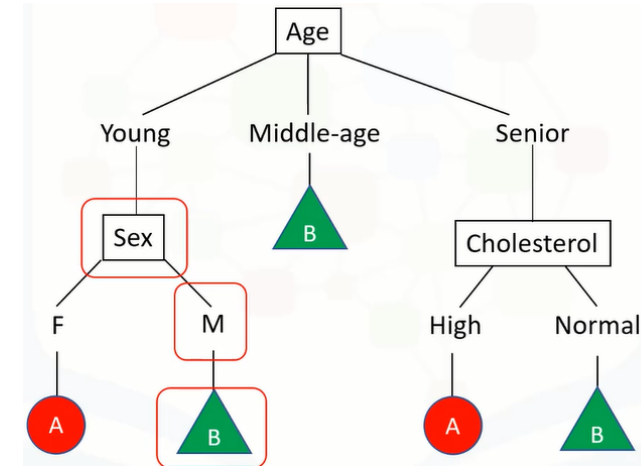
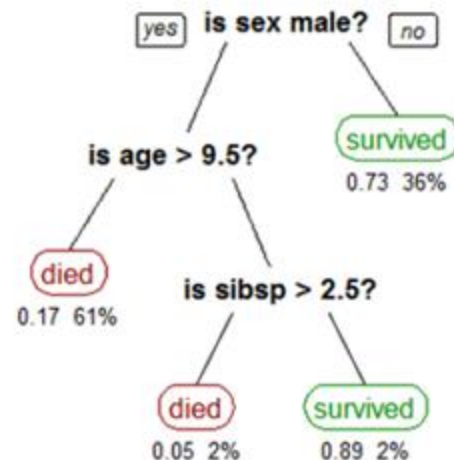


### Advantages:

- Easy to understand
- Can handle both categorical and numerical data
- Robust to outliers

### Disadvantages:

- Prone to Overfitting
- Need to be careful with parameter tuning
- Can create biased learned trees if some classes dominate



- Each **internal node** corresponds to a test
- Each **branch** corresponds to a result of the test
- Each **leaf node** assigns a classification

# Decision Trees

## Creating a Decision Tree Based on Information Gain Calculated Using Entropy or Gini Index

Information gain for multiple classification:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^m \frac{N_j}{N} I(D_j)$$

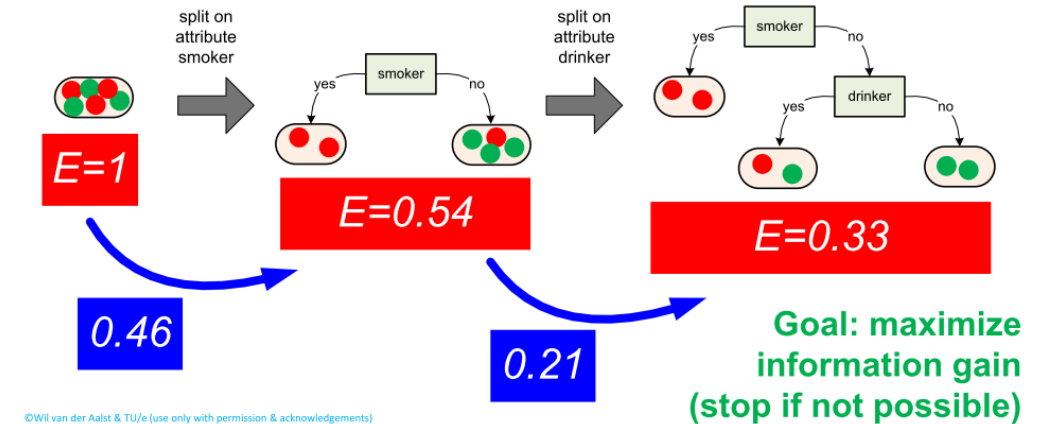
f: feature split on  
 $D_p$ : dataset of the parent node  
 $D_j$ : dataset of the jth child node  
 $I$ : impurity criterion  
 $N$ : total number of samples  
 $N_j$ : number of samples at jth child node

Information gain for binary classification:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N} I(D_{left}) - \frac{N_{right}}{N} I(D_{right})$$

f: feature split on  
 $D_p$ : dataset of the parent node  
 $D_{left}$ : dataset of the left child node  
 $D_{right}$ : dataset of the right child node  
 $I$ : impurity criterion (Gini Index or Entropy)  
 $N$ : total number of samples  
 $N_{left}$ : number of samples at left child node  
 $N_{right}$ : number of samples at right child node

The idea of Information Gain based on Entropy



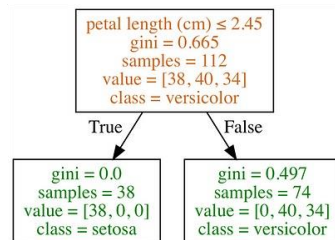
©Wil van der Aalst & TU/e (use only with permission & acknowledgements)

### Gini Formula

$$I_G = 1 - \sum_{j=1}^c p_j^2$$

$p_j$ : proportion of the samples that belongs to class  $c$  for a particular node

### Decision Tree



### Information Gain Calculation

Gini for Parent

$$1 - ((38.0/112)**2 + (40.0/112)**2 + (34.0/112)**2)$$

0.665

Gini for Child Node (left)

$$1 - ((38.0/38)**2 + (0.0/38)**2 + (0.0/38)**2)$$

0.000

Gini for Child Node (right)

$$1 - ((0.0/74)**2 + (40.0/74)**2 + (34.0/74)**2)$$

0.497

Information Gain

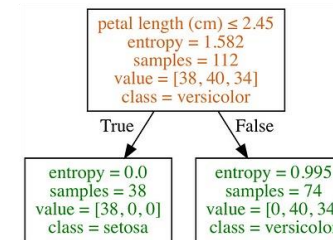
$$.665 - 0 - (74/112) * .497$$

0.337

### Entropy Formula

$$I_H = - \sum_{j=1}^c p_j \log_2(p_j)$$

### Decision Tree



### Information Gain Calculation

Entropy for Parent

$$-1 * ((38.0/112) * np.log2(38.0/112) + (40.0/112) * np.log2(40.0/112) + (34.0/112) * np.log2(34.0/112))$$

1.581711119299905

Entropy for Child Node (left)

$$-1 * ((38.0/38) * np.log2(38.0/38))$$

-0.0

Entropy for Child Node (right)

$$-1 * ((40.0/74) * np.log2(40.0/74) + (34.0/74) * np.log2(34.0/74))$$

0.9952525494396791

Information Gain

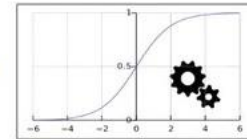
$$1.5817 - 0 - (74/112) * .995$$

0.924

# Classification

## Evaluation metrics

|   | tenure | age  | address | income | ed  | employ | equip | callcard | wireless | churn |
|---|--------|------|---------|--------|-----|--------|-------|----------|----------|-------|
| 3 | 11.0   | 33.0 | 7.0     | 136.0  | 5.0 | 5.0    | 0.0   | 1.0      | 1.0      | 1     |
| 1 | 33.0   | 33.0 | 12.0    | 33.0   | 2.0 | 0.0    | 0.0   | 0.0      | 0.0      | 1     |
| 2 | 23.0   | 30.0 | 9.0     | 30.0   | 1.0 | 2.0    | 0.0   | 0.0      | 0.0      | 0     |
| 3 | 38.0   | 35.0 | 5.0     | 76.0   | 2.0 | 10.0   | 1.0   | 1.0      | 1.0      | 0     |
| 4 | 7.0    | 35.0 | 14.0    | 80.0   | 2.0 | 15.0   | 0.0   | 1.0      | 0.0      | 0     |
| 5 | 68.0   | 52.0 | 17.0    | 125.0  | 1.0 | 24.0   | 0.0   | 0.0      | 0.0      | 0     |
| 6 | 42.0   | 40.0 | 7.0     | 37.0   | 2.0 | 8.0    | 1.0   | 0.0      | 0.0      | 0     |
| 7 | 9.0    | 21.0 | 1.0     | 17.0   | 2.0 | 2.0    | 0.0   | 0.0      | 0.0      | 0     |
| 8 | 36.0   | 30.0 | 26.0    | 140.0  | 2.0 | 21.0   | 0.0   | 0.0      | 0.0      | 0     |
| 9 | 49.0   | 51.0 | 27.0    | 63.0   | 4.0 | 19.0   | 0.0   | 0.0      | 0.0      | 0     |



### Test set

|   | tenure | age  | address | income | ed  | employ | equip | callcard | wireless | churn |
|---|--------|------|---------|--------|-----|--------|-------|----------|----------|-------|
| 0 | 11.0   | 33.0 | 7.0     | 136.0  | 5.0 | 5.0    | 0.0   | 1.0      | 1.0      | 1     |
| 1 | 33.0   | 33.0 | 12.0    | 33.0   | 2.0 | 0.0    | 0.0   | 0.0      | 0.0      | 1     |
| 2 | 23.0   | 30.0 | 9.0     | 30.0   | 1.0 | 2.0    | 0.0   | 0.0      | 0.0      | 0     |
| 3 | 38.0   | 35.0 | 5.0     | 76.0   | 2.0 | 10.0   | 1.0   | 1.0      | 1.0      | 0     |
| 4 | 7.0    | 35.0 | 14.0    | 80.0   | 2.0 | 15.0   | 0.0   | 1.0      | 0.0      | 0     |

Test

Actual Labels  $y$

| Predicted churn |
|-----------------|
| 1               |
| 1               |
| 0               |
| 1               |
| 0               |

$\hat{y}$  Predicted Labels

We compare the actual values in the test set with the values predicted by the model, to calculate the accuracy of the model.

### Evaluation metrics:

- Jaccard index
- F1-score
- Log Loss

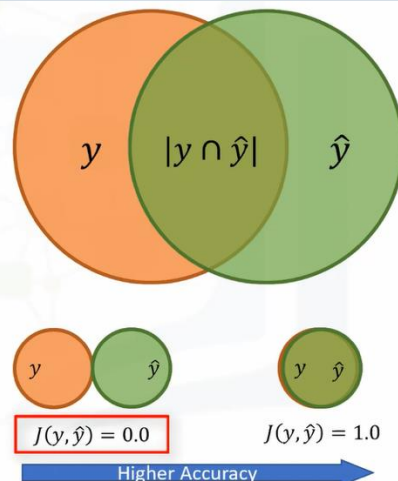
$y$ : Actual labels  
 $\hat{y}$ : Predicted labels

$$J(y, \hat{y}) = \frac{|y \cap \hat{y}|}{|y \cup \hat{y}|} = \frac{|y \cap \hat{y}|}{|y| + |\hat{y}| - |y \cap \hat{y}|}$$

$y$ : [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]  
 $\hat{y}$ : [1, 1, 0, 0, 0, 1, 1, 1, 1, 1]

$$J(y, \hat{y}) = \frac{8}{10+10-8} = 0.66$$

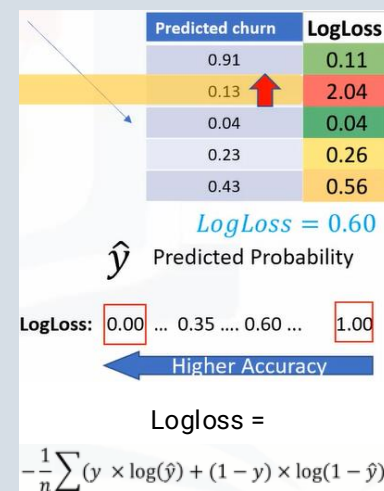
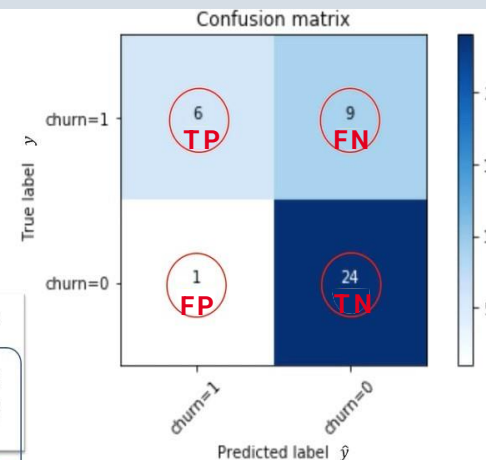
Jaccard index



- Precision =  $TP / (TP + FP)$
- Recall =  $TP / (TP + FN)$
- F1-score =  $2x (prc \times rec) / (prc + rec)$

F1-score: 0.00 ... 0.20 ... 0.55 ... 0.83 ... 1.00  
 Higher Accuracy

|                | precision | recall | f1-score |
|----------------|-----------|--------|----------|
| Churn = 0      | 0.73      | 0.96   | 0.83     |
| Churn = 1      | 0.86      | 0.40   | 0.55     |
| Avg Accuracy = |           |        | 0.72     |





# Classification

## Evaluation metrics

| Churn |
|-------|
| 1     |
| 1     |
| 0     |
| 0     |
| 0     |
| 1     |
| 0     |
| 0     |

If the target attribute can belong to one of two classes (“**churn**” or “**retention**”), there are four possible outcomes:

1. Predicted churn and actual churn
2. Predicted retention, but actual churn
3. Predicted churn and actual retention
4. Predicted retention and actual retention

Let's move to the terminology:

- “**Positive**” = customer **churn** (service cancellation)
- “**Negative**” = customer **retention** (continuing to purchase services).

In the **confusion matrix**, cases where the actual outcome did not match the model's prediction are highlighted in **red**, and correct predictions are highlighted in **green**.

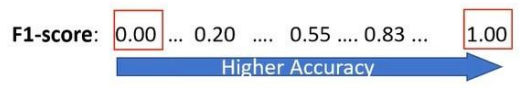
$$\textbf{Precision} = \text{True Positives} / (\text{True Positives} + \text{False Positives}) = \text{Correct Positives} / \text{Predicted Positives}$$

This metric shows the percentage of records predicted as positive (churn) by the model that actually represent churn.

$$\textbf{Recall} = \text{True Positives} / (\text{True Positives} + \text{False Negatives}) = \text{Correct Positives} / \text{Actual Positives}$$

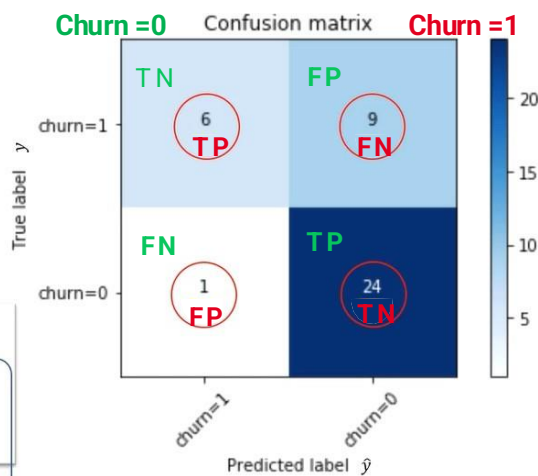
This metric indicates the percentage of actual churn cases correctly identified by the model out of all churn cases in the dataset.

- Precision = TP / (TP + FP)
- Recall = TP / (TP + FN)
- F1-score = 2x (prc x rec) / (prc+rec)



|           | precision | recall | f1-score |
|-----------|-----------|--------|----------|
| Churn = 0 | 0.73      | 0.96   | 0.83     |
| Churn = 1 | 0.86      | 0.40   | 0.55     |

Avg Accuracy = 0.72



| Outcomes           | Actual Positive                                                | Actual Negative                                            |
|--------------------|----------------------------------------------------------------|------------------------------------------------------------|
| Predicted Positive | True Positives (TP) = 6                                        | False Positives (FP) = 9<br>Type I errors<br>"False Alarm" |
| Predicted Negative | False Negatives (FN) = 1<br>Type II errors<br>"Missed targets" | False Negatives (FN) = 24                                  |



# Decision Trees

## Python code (1)

### #### Load libraries

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn.tree import DecisionTreeClassifier # import Decision Tree Classifier
from sklearn.model_selection import train_test_split # import train_test_split function
from sklearn import metrics # import scikit-learn metrics module for accuracy calculation
from sklearn.metrics import classification_report
from sklearn.preprocessing import OneHotEncoder
```

### #### Loading Data from Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
churn = pd.read_csv("/content/drive/My Drive/Churn_Modelling.csv")
churn.rename(columns={'NumOfProducts': 'NumOfProds'}, inplace = True)
```

### #### Feature Selection

```
# Split dataset into dependent (target variable) and independent variables (feature variables)
feature_cols = ['CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProds', 'HasCrCard',
                'IsActiveMember', 'EstimatedSalary']

X = churn[feature_cols] # Features
y = churn.Exited # Target variable
```

### #### One-Hot Encoding

```
categorical_columns = X.select_dtypes(include=['object']).columns.tolist()
# Create a one-hot encoder object
encoder = OneHotEncoder(sparse_output=False)
# Apply one-hot encoding to the categorical columns
one_hot_encoded = encoder.fit_transform(X[categorical_columns])
# Create a DataFrame with the one-hot encoded columns
# We use get_feature_names_out() to get the column names for the encoded data
one_hot_df = pd.DataFrame(one_hot_encoded, columns=encoder.get_feature_names_out(categorical_columns))
# Concatenate the one-hot encoded dataframe with the original dataframe
df_encoded = pd.concat([X, one_hot_df], axis=1)
```



# Decision Trees

## Python code (2)

```
# Drop the original categorical columns
```

```
X_encoded = df_encoded.drop(categorical_columns, axis=1)
```

```
#### Split dataset into training set and test set
```

```
X_train,X_test, y_train,y_test = train_test_split(X_encoded, y, test_size=0.4, random_state=1) # 60% train and 40% test
```

```
#### Building Decision Tree Model
```

```
# Create Decision Tree classifier object
```

```
clf = DecisionTreeClassifier(criterion="entropy", max_depth=4)
```

```
# Train Decision Tree Classifier
```

```
clf = clf.fit(X_train,y_train)
```

```
# Predict the response for test dataset
```

```
y_pred = clf.predict(X_test)
```

```
#### Evaluating the Model
```

```
# Model Accuracy, how often is the classifier correct?
```

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
print("Classification Report: \n", classification_report(y_test, y_pred))
```

```
# Feature Importances
```

```
feature_importance = clf.feature_importances_
```

```
feature_names = X_encoded.columns
```

```
feature_importances = pd.DataFrame({'feature':feature_names,  
                                   'importance':feature_importance})
```

```
feature_importances = feature_importances.sort_values('importance',  
                                                       ascending=False)
```

```
print(feature_importances)
```

```
#### Plot Decision Tree
```

```
plt.figure(figsize=(16,16), dpi=300)
```

```
plot_tree(clf, feature_names=feature_cols, filled=True,  
          class_names=["Exited", "Stayed"])
```

```
plt.title("Decision Tree"); plt.show()
```

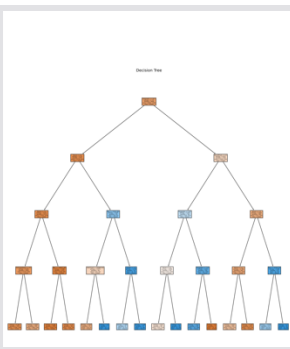
Accuracy: 0.84425

Classification Report:

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.84      | 0.99   | 0.91     | 3159    |
| 1            | 0.90      | 0.29   | 0.44     | 841     |
| accuracy     |           |        | 0.84     | 4000    |
| macro avg    | 0.87      | 0.64   | 0.68     | 4000    |
| weighted avg | 0.85      | 0.84   | 0.81     | 4000    |

Feature Importance:

|   | feature           | importance |
|---|-------------------|------------|
| 1 | Age               | 0.449575   |
| 4 | NumOfProds        | 0.333398   |
| 6 | IsActiveMember    | 0.120444   |
| 3 | Balance           | 0.049963   |
| 9 | Geography_Germany | 0.031907   |
| 7 | EstimatedSalary   | 0.009467   |
| 2 | Tenure            | 0.005247   |
| 0 | CreditScore       | 0.000000   |



Volodymyr Kravchenko

National University of Life and Environmental  
Sciences

+380 99 246 99 19

[v.kravchenko@nubip.edu.ua](mailto:v.kravchenko@nubip.edu.ua)

[nubip.edu.ua](http://nubip.edu.ua)

