

Fondements du Machine Learning

Chapitre 3:

Python: Programmation et bibliothèques pour le machine Learning
(NumPy, pandas, scikit-learn...)

Mme BEDDAD Fatima
fatima.bedad@univ-temouchent.edu.dz

2025-2026

Ce cours introduit certaines **bibliothèques** de Python utilisées dans le domaine de l'apprentissage automatique, telles que: **Numpy**, **Matplotlib**, **Pandas** et **Sklearn**.

Numpy

NumPy est une bibliothèque conçue pour manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux.

Pour aller plus loin, voir : <https://numpy.org/doc/stable/user/quickstart.html>

L'importation de cette bibliothèque se fait comme suit:

```
# importer numpy avec l'alias np
import numpy as np
```

Numpy apporte la structure de données **ndarray** (N-Dimensional array - en français, Tableaux à N-dimensions) à python.

Contrairement à la structure de liste de python (utilisée comme tableau - en anglais, *array*), ndarray *n'est pas dynamique*, et le type de ses éléments doit être *homogène*.

Elle a également l'avantage d'être bien *plus rapide* à traiter que les listes traditionnelles.

Création de tableaux

Les tableaux peuvent être créés avec **np.array()**. On utilise des crochets pour délimiter les listes d'éléments dans les tableaux.

La création de tableaux peut se faire comme suit:

```
a = np.array([1, 2, 3]) # crée un tableau array([1, 2, 3])

b = np.array([[1],[2],[3]]) # crée un tableau array([[1],
                           [2],
                           [3]])

c = np.array([[1, 2, 3], [4, 5, 6]]) # crée un tableau array([[1, 2, 3],
                           [4, 5, 6]])
```

- «array([[1, 2, 3]])» Représente le **vecteur horizontal** $\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$
- «array([[1],[2],[3]])» Représente le **vecteur vertical** $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$
- «array([[1, 2, 3], [4, 5, 6]])» Représente la **matrice** $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

Il existe d'autres fonctions qui permettent de créer des tableaux: (liste non exhaustive)

- **np.arange(start, stop, step)**: permet de créer un tableau à 1 dimension, aux valeurs réparties de façon équitable, allant de la valeur **start** jusqu'à **stop** (exclu) avec un pas **step** (par défaut step=1).

Exemple:

```
d = np.arange(1,10,2) # crée un tableau de valeurs allant de 1 à 10 avec un pas de 2
print(d) # affiche [1 3 5 7 9]
```

- **np.linspace(start, stop, nbr)**: permet de créer un tableau à 1 dimension avec **nbr** éléments, aux valeurs réparties de façon équitable, allant de la valeur **start** jusqu'à **stop** (inclus).

Exemple:

```
e = np.linspace(1,10,2) # crée un tableau de 2 valeurs allant de 1 à 10
print(e) # affiche [ 1. 10.]
f = np.linspace(1,10,5) # crée un tableau de 5 valeurs allant de 1 à 10
print(f) # affiche [ 1. 3.25 5.5 7.75 10. ]
```

- **np.zeros((n,m))** ou **ones((n,m))**: permet de créer un tableau 2D, avec n lignes et m colonnes, rempli de **zéro** ou de **un**. Existe aussi en 1D zeros(n) et ones(n).

Exemple:

```
g = np.zeros(5) # crée un tableau de 5 éléments, rempli de 0
print(g) # affiche [0. 0. 0. 0. 0.]
h = np.ones((2,5)) # crée un tableau 2D, avec 10 éléments, rempli de 1
print(h) # affiche [[1. 1. 1. 1. 1.]
                  [1. 1. 1. 1. 1.]]
```

Les attributs de *ndarray*

L'objet ndarray dispose de certains attributs qui nous permettent d'avoir plus de détails sur nos tableaux.

- 1) **dtype** : retourne le type des données du tableau (int31, int64, float64, etc.).
- 2) **ndim**: retourne le nombre de dimensions (axes) qui définit le tableau (1-dim, 2-dim, 3-dim, etc.).
- 3) **shape**: retourne la forme du tableau, autrement dit, la taille de chaque dimension, présentée sous forme de tuple (n,m), n étant le nombre de lignes et m le nombre de colonnes.
- 4) **size**: retourne la taille du tableau n x m, autrement dit, le nombre total d'éléments dans le tableau, n étant le nombre de lignes et m le nombre de colonnes.

Exemple:

```
c.dtype # retourne int64  
c.ndim # retourne 2  
c.shape # retourne (2, 3)  
c.size # retourne 6
```

Accès aux éléments d'un tableau

Numpy offre la possibilité d'accéder aux éléments d'un tableau facilement, grâce aux techniques d'indexing (indexation) et de Slicing (tranchage).

- T[i] renvoie la ligne i (avec toutes les colonnes)
- T[i,:] renvoie la ligne i (pareil que T[i])
- T[:,j] renvoie la colonne j (de toutes les lignes)
- T[i:] renvoie toutes les lignes à partir de la ligne i (inclus)
- T[:j] renvoie toutes les lignes jusqu'à la ligne j (exclue)
- T[i:j] renvoie toutes les lignes entre les ligne i (inclus) et j (exclue)
- T[i:j,k:l] renvoie toutes les lignes entre les ligne i (inclus) et j (exclue), et toutes les colonnes entre les colonnes k (inclus) et l (exclue)
- T[i:j:k] renvoie toutes les lignes entre les ligne i (inclus) et j (exclue) avec un pas (step) k, k=1 par défaut

Exemple:

```
print(c[:,1]) # affiche [2 5]  
print(c[:,:]) # affiche [[1 2 3]  
#                         [4 5 6]]  
print(c[:,::2]) # affiche [[1 3]  
#                         [4 6]]
```

Matplotlib

Matplotlib est une bibliothèque python utilisée pour représenter des graphiques. Elle permet de produire une grande variété de graphiques de grande qualité.

Le module **pyplot** est l'un des principaux modules de matplotlib. Il regroupe un grand nombre de fonctions qui servent à créer des graphiques et à les personnaliser (travailler sur les axes, changer le type de graphique, ajouter du texte, etc.).

L'importation de ce module se fait comme suit:

```
# importer pyplot à partir de matplotlib avec l'alias plt  
from matplotlib import pyplot as plt
```

Pour aller plus loin: <https://matplotlib.org/stable/index.html>

Avant d'arriver à créer le graphique souhaité, il y a un certain nombre d'étapes à réaliser:

Générer ou importer les données à utiliser

Il est nécessaire d'avoir à disposition les données que nous souhaitons utiliser dans nos graphiques.

Celles-là peuvent être de simples listes, ou des tableaux numpy. En réalité, toutes les séquences seront converties localement en tableaux numpy.

Exemple de génération de données avec numpy:

```
x = np.linspace(0, 2*np.pi, 30) # créer un tableau de 30 valeurs allant de 0 à  
# 2*3.14  
y = np.cos(x) # créer un tableau de 30 valeurs correspondants au cosinus des valeurs  
# du tableau x
```

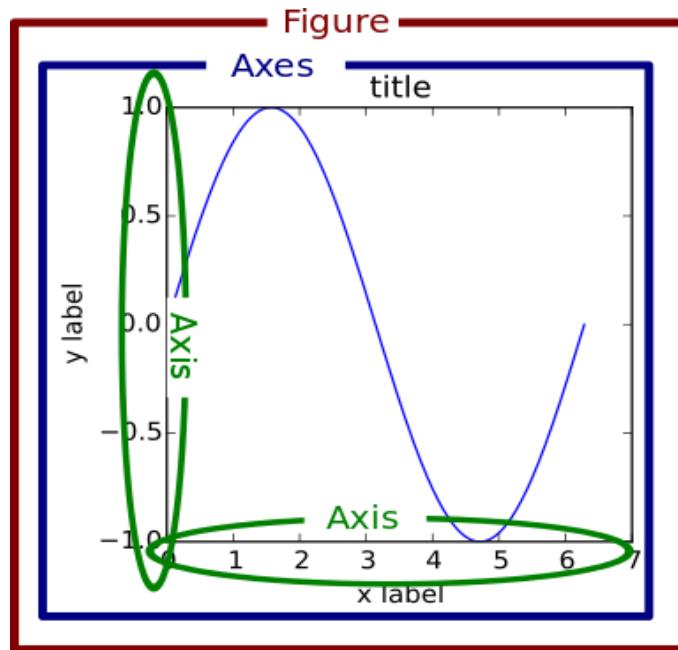
Créer la hiérarchie du graphique

Il est important de comprendre que matplotlib fonctionne avec le concept de hiérarchie d'objets:

- L'objet *Figure* est le conteneur extérieur du graphique, une sorte de fenêtre qui va accueillir les différents graphiques.
- La figure peut contenir de multiples objets *Axes* (à ne pas confondre avec les axes d'un espace vectoriel). Ce sont ces axes qui vont accueillir des graphes. On peut alors

dessiner plusieurs graphes sur une même figure, répartis sur des axes différents. On peut assimiler cela à la création d'une grille de plusieurs graphiques.

- À l'intérieur des axes, nous pouvons avoir une hiérarchie de plus petits objets: plusieurs *graphes*, des *titres*, des *légendes*, du *texte*, etc. Généralement, chacun doit être géré individuellement.



Pyplot nous offre différentes méthodes pour créer différents types de graphiques : plot, scatter, hist, bar, etc., selon le graphe qu'on souhaite tracer: courbe, nuage de points, histogramme, diagrammes à barre, etc.

Pour les besoins de ce TP, nous verrons la méthode plot.

Pour aller plus loin:

<https://matplotlib.org/stable/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py>

Pyplot nous permet de créer des figures de différentes manières, avec un seul ou plusieurs axes :

Plot unique:

Exemple 1: (méthode simple)

```
plt.figure(figsize=(8,6)) # créer la fenêtre figure, avec comme paramètre figsize pour attribuer une taille à la fenêtre
plt.title('titre de la figure') # mettre un titre à la figure
plt.subplot() # créer un axe
plt.plot(x,y) # dessiner la courbe définie par les données x et y
plt.xlabel('x') # mettre un nom (libellé) à l'axe des abscisses
plt.ylabel('cos(x)') # mettre un nom (libellé) à l'axe des ordonnées
plt.show() # afficher la figure à l'écran
```

Exemple 2: (méthode POO)

```
fig, ax = plt.subplots() # subplots avec S initialise fig avec l'objet figure crée un axe ax
fig.suptitle('titre de la figure') # mettre un titre à la figure
ax.plot(x,y) # dessiner la courbe définie par les données x et y
ax.set_xlabel('x') # mettre un nom (libellé) à l'axe des abscisses
ax.set_ylabel('cos(x)') # mettre un nom (libellé) à l'axe des ordonnées
plt.show() # afficher la figure à l'écran
```

Plot multiple:

Exemple 1: (méthode simple)

```
plt.figure(figsize=(12,16)) # créer la fenêtre figure, avec comme paramètre figsize pour attribuer une taille à la fenêtre
plt.title('titre de la figure') # mettre un titre à la figure

plt.subplot(311) # le paramètre 311 indique que la figure est divisée en 3 lignes et 1 colonne = 3 axes , et le dernier 1 indique la position (index) de l'axe sur la figure, ici à l'index 1
plt.plot(x,y) # dessiner la courbe définie par les données x et y
plt.xlabel('x') # mettre un nom (libellé) à l'axe des abscisses
plt.ylabel('cos(x)') # mettre un nom (libellé) à l'axe des ordonnées

plt.subplot(312) # axe positionné à l'index 2
plt.plot(x,np.exp(x)) # dessiner la courbe définie par les données x et y
```

```

plt.xlabel('x') # mettre un nom (libellé) à l'axe des abscisses
plt.ylabel('exp(x)') # mettre un nom (libellé) à l'axe des ordonnées

plt.subplot(313) # axe positionné à l'index 3
plt.plot(x,-np.exp(-x)) # dessiner la courbe définie par les données x et y
plt.xlabel('x') # mettre un nom (libellé) à l'axe des abscisses
plt.ylabel('-exp(-x)') # mettre un nom (libellé) à l'axe des ordonnées
plt.show() # afficher la figure à l'écran

```

Exemple 2: (méthode POO)

```

Fig, (ax1, ax2) = plt.subplots(1,2, figsize=(12,6)) # subplots initialise fig avec
l'objet figure et son argument (1,2) permet de diviser la fenêtre en 1 ligne et 2
colonnes = 2 axes retournés respectivement dans ax1 et ax2

fig.suptitle('titre de la figure') # mettre un titre à la figure, qui peut être
écrasé par les titres des axes

ax1.plot(x,y) # dessiner la courbe définie par les données x et y
ax1.set_xlabel('x') # mettre un nom (libellé) à l'axe des abscisses
ax1.set_ylabel('cos(x)') # mettre un nom (libellé) à l'axe des ordonnées
ax1.set_title('cosinus') # mettre un titre à l'axe

ax2.plot(x,np.exp(x)) # dessiner la courbe définie par les données x et y
ax2.set_xlabel('x') # mettre un nom (libellé) à l'axe des abscisses
ax2.set_ylabel('exp(x)') # mettre un nom (libellé) à l'axe des ordonnées
ax2.set_title('exponentiel') # mettre un titre à l'axe

plt.show() # afficher la figure à l'écran

```

Il est possible de changer la forme du tracé et sa couleur, en ajoutant un paramètre à plot:

- `plot(x, y, 'bo')` # plot x et y en utilisant des marqueurs en forme de cercles de couleur bleue
- `plot(x, y, 'r+')` # plot x et y en utilisant des marqueurs en forme de plus de couleur rouge
- `plot(x, y, 'y-')` # plot x et y en utilisant des marqueurs en forme de tirets de couleur jaune
- etc.

Il est possible de tracer plusieurs graphes sur un même axe, et ceci en :

- Empilant les commandes plot:

```
plt.plot(x, x) # dessiner la 1ère courbe  
plt.plot(x, x**2) # dessiner la 2ème courbe  
plt.plot(x, x**3) # dessiner la 3ème courbe
```

- Ajoutant les données à tracer dans la même commande plot:

```
plt.plot(x, x, 'b-', x, x**2, 'y+', x, x**3, 'r^') # dessiner plusieurs  
courbes sur un même axe. On peut personnaliser les couleurs et les marqueurs pour une  
meilleure lisibilité du graphique final
```

Pandas

Pandas est une bibliothèque Python conçue pour la manipulation et l'analyse de données. Elle est l'une des bibliothèques, si ce n'est la bibliothèque la plus importante à connaître dans le domaine de la data science et de l'intelligence artificielle.

Pour aller plus loin, voir : https://pandas.pydata.org/docs/user_guide/index.html

L'importation de cette bibliothèque se fait comme suit:

```
# importer pandas avec l'alias pd  
import pandas as pd
```

Les structures de données et l'Index

Pandas peut gérer différentes structures de données. Ces dernières sont construites au-dessus de Numpy. On peut donc imaginer que ces données seront sous forme de *ndarray*.

La notion d'Index

Les objets créés avec pandas, sous forme de tableaux, sont accompagnés d'une colonne (placée à gauche) appelée **Index**.

Un index sert à nommer les lignes du tableau. Il peut être, par défaut, une suite continue d'entier commençant par 0 (0, 1, 2, ...), ou des index qu'on aurait défini (suite d'entiers, chaîne de caractères, des dates, etc.). Ces index seront ensuite utilisés pour accéder facilement aux différentes lignes du tableau.

On peut également employer le terme anglais **Label** pour désigner l'index.

Il existe différentes structures de données avec pandas, les plus utilisées sont:

Séries

Une série pandas est un tableau indexé à 1 dimension, qui peut contenir des données homogènes. Cela pourrait également être vu comme une seule colonne dans un tableau, accompagnée des index de ses lignes, à gauche de cette colonne.

DataFrames

L'objet DataFrame de pandas correspond à une *matrice* (tableau à 2 dimensions), également indexée, où les lignes correspondent à des *observations* et les colonnes à des *attributs* (également appelés *variables*).

Comme mentionné plus haut, les lignes sont indexées. Mais dans un dataframe, les colonnes également sont nommées. Rendant, ainsi, la manipulation de données bien plus simple avec pandas.

À noter que, la première colonne affichée par un dataframe n'est pas inclue dans les données, elle représente l'index. Et la première ligne (également appelée *en-tête*) qui correspond aux noms des champs (attributs ou variables), n'est également pas inclue. Les données ne commencent qu'à partir de la seconde ligne.

Voici un exemple d'un dataframe avec deux colonnes: prénom et âge, et 4 lignes indexées de 0 à 3 :

| | prénom | âge |
|---|----------|-----|
| 0 | Fatima | 40 |
| 1 | Ouerdia | 35 |
| 2 | Mohammed | 26 |
| 3 | Ahmed | 67 |

On pourrait facilement assimiler un dataframe à un tableau excel.

Pour les besoins du TP, nous ne verrons en détail que la structure de données DataFrame.

Pour aller plus loin, voir: https://pandas.pydata.org/docs/user_guide/dsintro.html

DataFrame

Il est possible de créer un dataframe à partir de différents type de données, voici une liste non-exhaustive de méthodes pour créer un dataframe :

À partir d'un tableau de tuples

```
data = [(1, 2.0, "Hello"), (2, 3.0, "World")] # chaque élément du tableau représente une ligne dans le dataframe
df = pd.DataFrame(data) # méthode 1 : avec index et nom de colonnes par défaut
print(df)
# affiche
    0      1      2
1 1  2.0  Hello
2 2  3.0  World

df1 = pd.DataFrame(data, index=["premier", "second"], columns=["A", "B", "C"]) # méthode 2 : avec une liste d'index et une liste de colonnes
print(df1)
# affiche
          A      B      C
premier  1  2.0  Hello
second   2  3.0  World
```

À partir d'un dictionnaire de ndarray

```
d = {"one": [1.0, 2.0, 3.0], "two": [4.0, 3.0, 2.0]}
df = pd.DataFrame(d) # méthode 1 : avec index par défaut. Les clés du dictionnaire représentent les noms des colonnes
print(df)
# affiche
    one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0

df1 = pd.DataFrame(d, index=["a", "b", "c"]) # méthode 2 : avec une liste d'index
print(df1)
# affiche
    one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
```

À partir d'une liste de dictionnaires

```
data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]  
  
df = pd.DataFrame(data2) # méthode 1 : avec index par défaut. Les clés du dictionnaire  
représentent les noms des colonnes  
print(df)  
# affiche  
    a    b    c  
1  1    2  NaN  
2  5   10  20.0  
#NaN = Not a Number, valeur par défaut pour les valeurs manquantes  
  
df1 = pd.DataFrame(data2, index=["first", "second"], columns=["a", "b"])  
print(df1)  
# affiche  
      a    b  
first  1    2  
second  5   10
```

Importation de données externes

Il est possible de lire (importer) des données à partir de sources externes (csv, excel, html, json, sql, etc.).

Voici deux méthodes courantes pour la lecture de données :

- 1) df = pd.read_csv('chemin/vers/fichier.csv') : lit un fichier csv (comma-separated values - valeurs séparées par des virgules) dans un dataframe.
- 2) df = pd.read_excel('chemin/vers/fichier.xlsx') : lit un fichier excel (xls, xlsx, xlsm, xlsb, odf, ods et odt) dans un dataframe.

Pour aller plus loin, voir: <https://pandas.pydata.org/pandas-docs/stable/reference/io.html>

Fonctions et attributs d'un dataframe

Un dataframe dispose d'un certain nombre de fonctions et attributs, les plus usuels sont:

- 1) **df.head()**: retourne les 5 premières lignes du jeu de données. Utile pour avoir un coup d'œil rapide sur les données.
- 2) **df.tail()**: retourne les 5 dernières lignes du jeu de données.

- 3) **df.describe()**: retourne une description des données, avec des indicateurs statistiques (moyenne, min, max, fréquence, etc.)
- 4) **df.shape**: retourne les dimensions sous forme d'un tuple (nb_lignes, nb_colonnes). La ligne d'en-tête n'est pas comptabilisée dans le nombre de lignes.
- 5) **df.size**: retourne la taille des données = nb_lignes x nb_colonnes.
- 6) **df.columns**: retourne une liste des noms des différentes colonnes.
- 7) **df.index**: retourne une liste des index.
- 8) **df.dtypes**: retourne le type de chaque colonne.

Manipulation de données

Un Dataframe peut être traité comme un dictionnaire pour tout ce qui est sélection, suppression et ajout de colonnes (attributs).

Manipulation des attributs

Un dataframe peut aussi être considéré comme une concaténation de plusieurs séries (colonnes). En sélectionnant une seule colonne du dataframe nous obtenons une structure en forme de série.

Sélection de colonne

```
df = pd.DataFrame([(1, 2.0, "Hello"), (2, 3.0, "World")], index=["premier", "second"],
columns=["A", "B", "C"] )
print(df["A"]) # fonctionne aussi avec print(df.A)

# les deux affichent la colonne A seulement (similaire à la structure série)
premier    1
second     2
Name: A, dtype: int64

print(df[["A","B"]]) # sélectionne deux colonnes à la
fois # affiche
      A    B
premier 1  2.0
second   2  3.0
```

Le type série étant considéré comme un tableau à 1 dimension, nous pouvons utiliser les indices pour accéder aux différents éléments de la colonne sélectionnée.

```
print(df["A"][0]) # sélectionner le 1er élément de la colonne A. Affiche 1  
print(df["A"][:2]) # sélectionner les éléments de la colonne A (de 0 à 1)
```

Ajout de colonne

```
df["D"] = df["A"] + df["B"] # ajouter une colonne D contenant la somme des valeurs de A et B  
print(df)  
# affiche  
      A      B      C      D  
premier 1  2.0  Hello  3.0  
second   2  3.0  World  5.0  
df["F"] = "OK" # ajouter une colonne F dont les valeurs = "OK"  
print(df)  
# affiche  
      A      B      C      D      F  
premier 1  2.0  Hello  3.0  OK  
second   2  3.0  World  5.0  OK
```

Suppression de colonne

```
del df["C"] #supprimer définitivement la colonne C  
print(df)  
# affiche  
      A      B      D      F  
premier 1  2.0  3.0  OK  
second   2  3.0  5.0  OK  
valeurs_F = df.pop("F") # extraire le contenu de la colonne F dans valeurs_F et la supprimer de df  
print(df)  
# affiche  
      A      B      D  
premier 1  2.0  3.0  
second   2  3.0  5.0  
print(valeurs_F)  
# affiche une série  
premier    OK  
second     OK  
Name: valeurs_F, dtype: object
```

Suppression de lignes et/ou de colonnes

Il est possible de procéder à la suppression de lignes et/ou de colonnes avec une même méthode.

Mais avant cela, il est important de comprendre la notion d'axes (axis en anglais):

- Axis = 0 représente l'axe des lignes
- Axis = 1 représente l'axe des colonnes

La méthode **drop()** permet de supprimer une ou plusieurs lignes et/ou une ou plusieurs colonnes, en spécifiant l'axe sur lequel on souhaite supprimer, ou en spécifiant la liste des labels des lignes et/ou des colonnes à supprimer.

Exemple :

```
df1 = pd.DataFrame(np.random.randn(5, 4), index=list('12345'), columns=list('ABCD'))  
# créer un dataframe de nombres aléatoires, avec 5 lignes et 4 colonnes.  
      A          B          C          D  
1 -0.507802  0.196315  0.927320  0.250089  
2 -1.240852  1.371631 -0.670057  0.713926  
3 -0.692301 -0.169095  0.728527 -0.958735  
4  0.129903  0.955447  0.514052  1.238018  
5  1.173516 -1.555069 -0.941821  0.238360  
  
df1.drop ("A", axis=1, inplace= True) # supprimer la colonne A et appliquer le changement  
directement sur place (inplace= True)  
df1.drop ("1", axis=0, inplace= True) # supprimer la ligne 1  
print(df1)  
# affiche  
      B          C          D  
2 -0.562127 -1.102037 -0.865823  
3 -0.618187  0.799903 -1.158138  
4  0.821559 -0.327911 -1.195187  
5 -0.071999 -1.029290  0.455029  
  
new_df = df1.drop (columns = ["B","D"]) # supprimer les colonnes B et D et retourner les  
données restantes dans new_df. df1 reste inchangé (inplace= False par défaut)  
print(new_df)  
# affiche  
      C
```

```

2 -1.102037
3 0.799903
4 -0.327911
5 -1.029290

new_df1 = df1.drop (index= ['2','3'], columns = ["D"]) # supprimer les lignes 2 et 3, et
la colonne D, puis retourner les données restantes dans new_df1. df1 reste inchangé
print(new_df1)
# affiche
      B          C
4  0.821559 -0.327911
5 -0.071999 -1.029290

```

Indexing et Slicing

On peut accéder aux valeurs du DataFrame via des indices ou plage d'indices. La structure se comporte alors comme une matrice. La cellule en haut et à gauche est de coordonnées (0,0).

Il y a deux manières d'accéder aux valeurs d'un dataframe, les voici :

- `.loc` : sélectionne les lignes et les colonnes par leur *label* (nom)
- `.iloc` : sélectionne les lignes et les colonnes par leur *numéro* de position

Exemple :

```

df2 = pd.DataFrame(np.random.randn(5, 4), index=list('abcde'), columns=list('ABCD'))
# créer un dataframe de nombres aléatoires, avec 5 lignes et 4 colonnes.

      A          B          C          D
a -1.515756 -0.850033 -0.939369 -1.197477
b -0.581837 -0.232735  0.959948 -0.722127
c  1.335379 -0.106731 -0.045989 -1.375990
d  0.897325  1.156948  0.410629 -0.818115
e -1.329477 -0.337404 -0.193044  0.472857

print(df2.loc["b"]) # sélectionne la ligne b
print(df2.iloc[1]) # sélectionne la ligne 1 qui est la ligne b
# les deux affichent le résultat sous forme de série
A   -0.581837
B   -0.232735
C    0.959948

```

```

D    -0.722127
Name: b, dtype: float64

print(df2.loc[:, "B"]) # sélectionne toutes les lignes de la colonne B
print(df2.iloc[:, 1]) # sélectionne toutes les lignes de la colonne 1 qui est la B
# les deux affichent le résultat sous forme de série
a    -0.850033
b    -0.232735
c    -0.106731
d    1.156948
e    -0.337404
Name: B, dtype: float64

print(df2.loc[['a','d'],['B','D']]) # sélectionne les lignes a et d, et les colonnes B et D
print(df2.iloc[[0,3],[1,3]]) # sélectionne les lignes 0 et 3, et les colonnes 1 et 3
# les deux affichent le résultat sous forme de dataframe
      B          D
a -0.850033 -1.197477
d  1.156948 -0.818115

print(df2.loc['a':'c','B':'D']) # sélectionne les lignes comprises entre a et c (inclu),
# et les colonnes comprise entre B et D (inclu)
# affiche le résultat sous forme de dataframe
      B          C          D
a -0.850033 -0.939369 -1.197477
b -0.232735  0.959948 -0.722127
c -0.106731 -0.045989 -1.375990

print(df2.iloc[0:2,1:3])# sélectionne les lignes comprises entre 0 et 2 (exclu), et les
# colonnes comprise entre 1 et 3 (exclu)
# affiche le résultat sous forme de dataframe
      B          C
a -0.850033 -0.939369
b -0.232735  0.959948

```

Données manquantes (Missing Values)

L'une des options de l'analyse et la manipulation de données les plus intéressantes qu'offre pandas est la *gestion des données manquantes* dans les données.

Le marqueur de valeur manquante par défaut est la valeur **NaN** (Not a Number), qui peut être appliqué à tous les types de données (pas seulement aux chiffres). Il en existe d'autres plus spécifiques au dtype que nous n'aborderons pas ici.

Exemple :

```
dff = pd.DataFrame(np.random.randn(7, 3), columns=list("ABC")) # créer un dataframe de
nombres aléatoires, avec 7 lignes et 3 colonnes.
dff.iloc[1:3, 0] = np.nan # attribuer la valeur NaN aux lignes 1 et 2 de la colonne 0
dff.iloc[2:4, 1] = np.nan # attribuer la valeur NaN aux lignes 2 et 3 de la colonne 1
dff.iloc[3:5, 2] = np.nan # attribuer la valeur NaN aux lignes 3 et 4 de la colonne 2

print(dff)
# affiche

      A         B         C
0 -1.303138  0.035097  0.031456
1      NaN  1.435901  0.991497
2      NaN        NaN -0.067726
3  0.609350        NaN        NaN
4 -0.720113 -0.465258        NaN
5  0.597783  1.314623  0.171207
6 -0.390311 -1.219280 -0.603088
```

isna() et notna()

Pandas offre les méthodes **isna()** et **notna()** pour détecter facilement les valeurs manquantes dans les données.

Exemple :

```
print(dff['A'].isna()) # isna() retourne True si valeur = NaN, False sinon
# affiche
1  False
2   True
3   True
```

```

3    False
4    False
5    False
6    False
Name: A, dtype: bool

print(dff.notna()) # notna() retourne True si valeur différente de NaN, False sinon
# affiche
      A      B      C
0  True   True   True
1 False   True   True
2 False  False   True
3  True  False  False
4  True   True  False
5  True   True   True
6  True   True   True

```

fillna()

Il est possible de remplacer les valeurs manquantes par de nouvelles valeurs (au choix) grâce à la méthode **fillna()**.

Exemple:

```

dff['A'].fillna(0.0, inplace= True) # remplacer les NaN de la colonne A par des 0.0
print(dff)
# affiche
      A      B      C
0 -1.303138  0.035097  0.031456
1  0.000000  1.435901  0.991497
2  0.000000      NaN -0.067726
3  0.609350      NaN      NaN
4 -0.720113 -0.465258      NaN
5  0.597783  1.314623  0.171207
6 -0.390311 -1.219280 -0.603088

dff['B'].fillna(dff.mean()['B'], inplace= True) # remplacer les NaN de la colonne B par la
moyenne de ses valeurs
print(dff)
# affiche

```

| | A | B | C |
|---|-----------|-----------|-----------|
| 0 | -1.303138 | 0.035097 | 0.031456 |
| 1 | 0.000000 | 1.435901 | 0.991497 |
| 2 | 0.000000 | 0.220217 | -0.067726 |
| 3 | 0.609350 | 0.220217 | NaN |
| 4 | -0.720113 | -0.465258 | NaN |
| 5 | 0.597783 | 1.314623 | 0.171207 |
| 6 | -0.390311 | -1.219280 | -0.603088 |

`dropna()`

Il est possible de supprimer les lignes ou colonnes qui ont des valeurs manquantes, grâce à la méthode **dropna()**.

Exemple :

```
dff.dropna(axis=0, inplace= True) # supprimer les lignes où apparaissent des valeurs
manquantes NaN
print(dff)
# affiche
      A          B          C
0 -1.303138  0.035097  0.031456
1  0.000000  1.435901  0.991497
2  0.000000  0.220217 -0.067726
5  0.597783  1.314623  0.171207
6 -0.390311 -1.219280 -0.603088
```

Sklearn

Scikit-learn (sklearn) est une bibliothèque libre de Python destinée à l'apprentissage automatique.

Sklearn propose entre autres :

- une syntaxe simple et intuitive.
- des outils simples et efficaces pour le data mining et l'analyse de données.
- une implémentation d'un grand nombre de modèles de machine learning.
- des fonctions pour traiter des problèmes connexes tels que: la préparation de données, la sélection de features, sélection de modèles, etc.
- les fonctions sont construites sur NumPy, SciPy (fonctions de calcul numérique sur l'optimisation, l'intégration, l'algèbre linéaire, les probabilités ...) et matplotlib (affichage de graphes et d'images),

Ce qui en fait un outil assez complet pour débuter dans l'apprentissage automatique.

Dans ce qui suit, nous aborderons certaines notions fondamentales de Sklearn, nécessaires à la réalisation de notre TP n°3.

Mais pour aller plus loin, voir: <https://scikit-learn.org/stable/index.html>

Voir aussi: https://scikit-learn.org/stable/user_guide.html

Chargement du jeu de données

Le package **sklearn.datasets** propose des fonctions de lecture, de génération de données, ainsi que plusieurs jeux de données qu'il n'est plus nécessaire de télécharger.

Pour consulter les différents jeux de données disponibles, voir:

<https://scikit-learn.org/stable/datasets.html>

L'importation de ce package se fait comme suit:

```
# importer datasets à partir de sklearn
from sklearn import datasets
```

Pour pouvoir charger un dataset déjà disponible dans sklearn, il est nécessaire d'utiliser la fonction qui y correspond, à savoir:

- **load_boston()**: charge et retourne le dataset des prix de maison de boston (régression).
- **load_iris()** : charge et retourne le dataset de fleurs d'iris (classification).

- **load_diabetes()** : charge et retourne le dataset pour le diabète (régression).
- **load_digits([n_class])** : charge et retourne le dataset pour la reconnaissance de digits (classification).
- Etc.

Ces datasets sont utiles pour implémenter rapidement différents algorithmes d'AA. Cela dit, ils sont souvent trop petits pour donner une représentation de problèmes du monde réel.

Ces fonctions de chargement (loader) retournent deux éléments: **data** et **target**, qui sont respectivement les variables d'entrée et les résultats (désirés) associés.

Ces éléments sont représentés en général par un tuple (X, y), où **X** est un tableau bidimensionnel NumPy de $n_samples \times n_features$ contenant les valeurs des variables d'entrée (data) et **y** un tableau unidimensionnel NumPy de longueur $n_samples$ contenant les résultats désirés (target).

- $N_samples$ étant le nombre d'observations (nombre de lignes)
- $N_features$ étant le nombre de caractéristiques (nombre de colonnes)

Une description complète de ces datasets est disponible via leur attribut DESCR et certains disposent d'un attribut *features_names* qui donne une liste des caractéristiques des données, et d'un attribut target_names qui liste l'ensemble des résultats désirés.

Exemple :

```
#importer le dataset Iris
from sklearn.datasets import load_iris
iris = load_iris() # chargement du jeu de données de fleurs d'iris
X = iris.data # récupérer les données d'entrée dans X
y = iris.target # récupérer les données de sortie dans y
feature_names = iris.feature_names # récupérer la liste des caractéristiques des fleurs
target_names = iris.target_names # récupérer la liste des résultats désirés
print("Feature names:", feature_names)
# affiche
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
print("Target names:", target_names)
# affiche
Target names: ['setosa' 'versicolor' 'virginica']
```

NB: la bibliothèque pandas n'a pas été utilisée dans ce cas. Les données ici sont sous forme de tableau numpy et non de dataframe.

La version avec pandas sera le sujet du TP n°3.

Diviser le jeu de données

Dans un protocole expérimental d'apprentissage automatique, la partie évaluation et validation du modèle est une étape clé du processus. Et quand on évalue les performances prédictives d'un modèle, il est essentiel que cette étape ne soit pas biaisée.

Pour cela, il est important de faire l'entraînement du modèle sur un certain nombre de données appelées données d'entraînement. Puis, de tester les performances de ce modèle sur des données appelées données de test, qui n'ont jamais été vues auparavant par ce dernier. Autrement, les résultats de l'évaluation seront biaisés. Il est donc nécessaire de disposer de données d'entraînement et de données de test distinctes pour accomplir une expérimentation correcte.

A cet effet, il est nécessaire de diviser (en anglais *split*) notre jeu de données en une partie pour l'entraînement (*training set*) et une autre partie pour le test (*test set*).

Pour cela, sklearn met à notre disposition le package ***model_selection*** qui offre la fonction `train_test_split()`. Cette fonction reçoit en entrée notre jeu de données (X et y), et nous donne en sortie notre jeu de données divisé en un jeu d'entraînement (X_train, y_train) et un jeu de test (X_test, y_test).

- `sklearn.model_selection.train_test_split(arrays, test_size=None, train_size=None, random_state=None):`
 - **arrays**: représente les X et y du jeu de données de base.
 - **test_size**: est un float, allant de 0.0 à 1.0, indiquant la proportion de données allant dans la partie test. Optionnel si `train_size` est déjà spécifié.
 - **train_size**: est un float, allant de 0.0 à 1.0, indiquant la proportion de données allant dans la partie train. Optionnel si `test_size` est déjà spécifié.
 - **random_state**: passer un int pour avoir des données reproductibles à chaque appel à la fonction (garder la même division de données à chaque fois).
 - La fonction retourne une liste des nouveaux set de données: X_train, X_test, y_train, y_test.

Exemple :

```
#importer la fonction train_test_split
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state =
1) # diviser le jeu de données en 70% train et 30% test

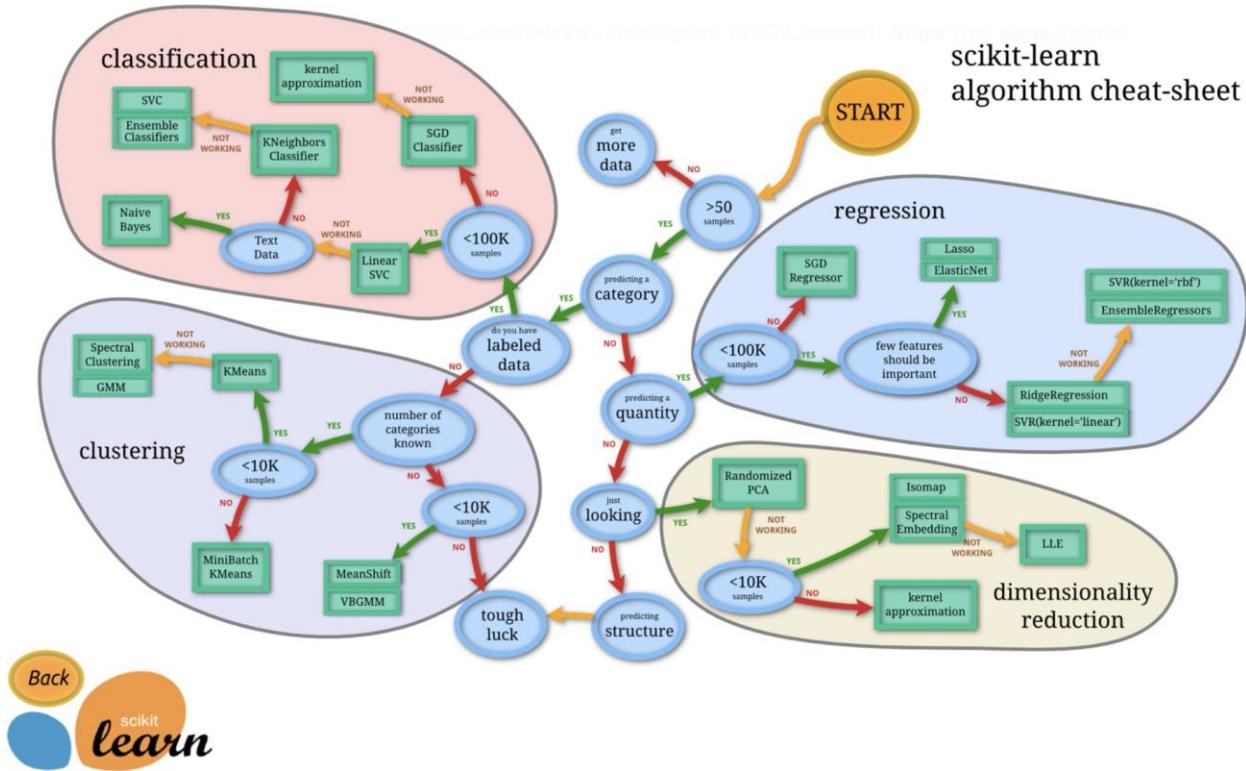
print("X_train.shape: ", X_train.shape)
# affiche X_train.shape: (105, 4)
print("X_test.shape : ", X_test.shape)
# affiche X_test.shape : (45, 4)
print("y_train.shape:", y_train.shape)
# affiche y_train.shape: (105,)
print("y_test.shape:", y_test.shape)
# affiche y_test.shape: (45,)
```

Entraînement des modèles

Sklearn implémente les algorithmes classiques de l'apprentissage automatique et fournit des solutions simples et efficaces pour les problèmes d'apprentissage, rendant l'utilisation de ces algorithmes accessible aux débutants.

Le choix de l'algorithme dépend du type de problème à résoudre et du type de données dont on dispose.

Les algorithmes d'AA que propose Sklearn sont résumés dans la figure ci-dessous:



Pour aller plus loin dans le choix de l'algorithme, voir:

https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

Concrètement, avec sklearn, un algorithme d'AA est implémenté par un estimateur, qui lui est une classe python qui implémente deux méthodes **fit** et **predict** qui vont nous permettre, respectivement, d'entraîner un modèle et de faire des prédictions avec ce dernier.

Une fois le choix de l'algorithme à utiliser fait, et les données à exploiter prêtes, il faut:

- **Importer l'estimateur** de l'algorithme à utiliser (consulter le lien précédent pour prendre connaissance des estimateurs existants).
- **Initialiser les arguments** du constructeur de l'estimateur avec les paramètres du modèle (si nécessaire).
- **Entraîner le modèle** sur les données d'entraînement, en utilisant la méthode *fit*.

Exemple :

```
from sklearn.svm import SVC # importer l'estimateur SVC (Support Vector Classifier) de SVM
model = SVC() # faire appel au constructeur. Ici nous utilisons les paramètres par défaut
model.fit(X_train,y_train) # entraîner le modèle sur les données d'entraînement
```

À la fin de cette étape, nous disposerons d'un modèle que nous pourrons utiliser pour faire des prédictions sur des données jamais vues par ce dernier.

Phase de test et Mesures d'évaluation

Afin d'obtenir le meilleur modèle d'AA possible pour la résolution notre problème, il est nécessaire de prendre connaissance des performances du modèle que l'on a obtenu après la phase d'entraînement, en ayant utilisé les données d'entraînement et les paramètres donnés au constructeur. Et, éventuellement, relancer l'entraînement avec de nouveaux réglages des hyperparamètres pour améliorer ses performances. De ce fait, la phase de "**test**" est primordiale.

(en réalité, dans la pratique, il existe la phase de validation puis vient la phase de test, mais cette phase ne sera pas abordée dans ce cours).

Pour réaliser la phase de test, l'estimateur implémente la méthode predict, qui va prendre en entrée les données de test X_test et qui va retourner les valeurs prédictes par le modèle.

Ces valeurs, ainsi retournées, n'ont pas une réelle signification si nous ne les comparons pas aux données réelles y_test. Cette comparaison va nous permettre de savoir sur combien de données notre modèle s'est trompé, et sur combien il a donné la bonne réponse. Il est clair que nous cherchons à maximiser le taux de bonnes réponses et minimiser le taux de mauvaises réponses.

Pour effectuer cette comparaison, il existe plusieurs indicateurs pour mesurer la performance du modèle. Chacun a ses spécificités, et souvent il est nécessaire d'utiliser plusieurs pour avoir une vision complète de la performance du modèle.

Le choix des indicateurs à utiliser dépend du type de problème à résoudre. Dans ce qui suit, nous verrons certaines mesures d'évaluation appliquées aux problèmes de classification. (les détails théoriques des différentes mesures seront vus en cours)

Sklearn implémente les différentes métriques dans son package *metrics*.

Exemple :

```
# importer toutes les métriques
from sklearn import metrics

y_predict = model.predict(X_test) # tester le modèle en faisant une prédition sur les
                               # données de test
```

```
exactitude = metrics.accuracy_score(y_test, y_predict) # estimer les performances du
modèle en termes d'accuracy, en comparant les prédictions du modèle (prediction) avec les
valeurs réelles (y_test)
print("Accuracy = ", exactitude) # affiche Accuracy = 0.9777777777777777

f_score = metrics.f1_score(y_test, y_predict, average="macro") # estimer les performances
du modèle en termes de f1-score. average="macro" signifie qu'on retourne la moyenne des
mesures des 3 différentes classes
print("F1-score", f) # affiche F1-score 0.9558404558404558

precision = metrics.precision_score(y_test, y_predict, average="macro") # estimer les
performances du modèle en termes de précision
print("Précision = ", precision) # affiche Précision = 0.9761904761904763

rappel = metrics.recall_score(y_test, y_predict, average="macro") # estimer les
performances du modèle en termes de rappel
print("Rappel = ", rappel) #affiche Rappel = 0.9814814814814815
```

Il est possible d'exploiter les résultats de ces métriques pour comparer les performances de différents modèles, et ainsi choisir celui qui fait les meilleures prédictions.