FACULTÉ DES SCIENCES DHAR EL MAHRAZ
UNIVERSITÉ SIDI MOHAMED BEN ABDELLAH

Université Sorbonne Paris Nord

RAPPORT DES TRAVAUX PRATIQUES DE MODULE : SDN

Master: WISD & EID2

Module : Science des Données Numériques

20/12/2021

Réalisé par :

Mohamed LAMGARRAJ

Responsable de module :

Pr. Younès BENNANI

Responsable des TPs:

Pr. Guénaël Cabanes

Data Science

ANNEE UNIVERSITAIRE 2021/2022

Table des matières

1. Introduction

La science des données (data science) est le processus d'extraction de connaissance d'ensembles de données.

C'est une spécialité récente, qui s'est développée avec l'essor des données dans le monde. Elle provient du croisement des domaines de l'extraction de données, aussi appelé forage de données ou *data mining*, et de l'analyse statistique.

La mission principale du data scientist est d'élaborer des stratégies d'analyse de données, mais également de préparer ses données pour leur analyse, puis d'explorer et analyser ces informations. Le data scientist doit ensuite créer des modèles avec ces données, en s'appuyant sur des langages de programmation (python den notre cas) afin de déployer ces modèles dans des applications.

Dans ce travail nous allons appliquer Le processus d'analyse et d'exploitation de données sur lequel se fonde la data science, en utilisant le module Sickit-Learn.

2. TP1 : Introduction au module scikit-learn

Scikit-Learn est une bibliothèque Python spécialisée dans les travaux de Data Science. C'est une bibliothèque facilement accessible, et puissante, qui s'intègre naturellement dans l'écosystème plus large des outils de science des données basés sur Python.
Scikit-learn a été initialement développé par *David Cournapeau* dans le cadre du projet Google summer of code en 2007. Plus tard, Matthieu Brucher a rejoint le projet et a commencé à l'utiliser dans le cadre de son travail de thèse. En 2010, l'INRIA s'est impliqué et la première version publique (v0.1 beta) a été publiée fin janvier 2010.Le projet compte maintenant plus de 30 contributeurs actifs et a bénéficié du soutien financier de l'INRIA, de Google, de Tinyclues et de la Python Software Foundation.

2.1 Importation des libraires

1- Importez **scikit-learn :**

Dans Python, la plupart des fonctions sont incluses dans des librairies, qu'il faut importer pour pourvoir les utiliser. Par exemple pour importer **scikit-learn** il faut écrire :

*from sklearn import **

2- Importez les librairies **numpy** (calcul scientifique) et **matplotlib.pyplot** (figures).

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn import *
```

2.2 Manipulation d'un jeu de données

1- Chargez les données Iris avec la commande :

iris = datasets.load_iris()

La variable *iris* est un objet qui contient la matrice des données (*iris.data*), un vecteur de numéro de classe (*target*), ainsi que les noms des variables (*feature_names*) et le nom des classes (*target_names*).

```
[ ]  iris = datasets.load_iris()
```

```
[ ]  print(iris.keys())

     dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

2- Affichez les données, les noms des variables et le nom des classes (utilisez *print*).

Les données d'Iris contiennent 150 lignes et **4** colons ça veut dire qu'il contient les valeurs de ***4 caractéristiques*** pour ***150 fleurs*** :

```
iris.data.shape

(150, 4)
```

```
[ ]  print(iris.data)

[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
```

142 lignes de plus

Les données d'iris sont divisées sur ***3 classes*** :

```
[]  print(iris.target)

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Les noms des classes sont :

```
print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

Les ***4 variables*** (caractéristiques des fleurs) que iris contient sont :

```
print(iris.feature_names)
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

3- Affichez le nom des classes pour chaque donnée

Les 50 premiers lignes appartiennent à la classe **0** (setosa).
De 51 à 100 appartiennent à la classe **1** (versicolor).
De 1001 à 150 appartiennent à la classe **2** (virginica) :

```
for i in range(iris.target.size):
    print("{} {}".format(iris.data[i], iris.target_names[iris.target[i]]))
```

```
[5.1 3.5 1.4 0.2] setosa
[4.9 3.  1.4 0.2] setosa
[4.7 3.2 1.3 0.2] setosa
[4.6 3.1 1.5 0.2] setosa
[6.4 3.2 4.5 1.5] versicolor
[6.9 3.1 4.9 1.5] versicolor
[5.5 2.3 4.  1.3] versicolor
[6.5 2.8 4.6 1.5] versicolor
[5.7 2.8 4.5 1.3] versicolor
[6.3 3.3 6.  2.5] virginica
[5.8 2.7 5.1 1.9] virginica
[7.1 3.  5.9 2.1] virginica
[6.3 2.9 5.6 1.8] virginica
[6.5 3.  5.8 2.2] virginica
```

iris.data est une matrice, mais c'est aussi un objet, comme toute variable en Python, qui inclut des méthodes. Par exemple, iris.data.mean(0) donne la moyenne de chaque variable, et iris.data.mean(1) donne la moyenne de chaque donnée. Il existe aussi des attributs associés aux objets.

4- Affichez la moyenne (*mean*), l'ecart-type (*std*), le *min* et le *max* pour chaque variable.

```
print("mean")
print(iris.data.mean(0))

print("l'ecart-type")
print(iris.data.std(0))

print("min")
print(iris.data.min(0))

print("max")
print(iris.data.max(0))
```

```
mean
[5.84333333 3.05733333 3.758      1.19933333]
l'ecart-type
[0.82530129 0.43441097 1.75940407 0.75969263]
min
[4.3 2.  1.  0.1]
max
[7.9 4.4 6.9 2.5]
```

5- En utilisant les attributs *size* et *shape*, affichez le nombre de données, le nombre de variables et le nombre de classes.

Data.shape() nous donnes le nombre des lignes et le nombre des colonnes
Data.size() nous donnes le nombre total des éléments dans la matrice iris.data
Target.size() nous donnes le nombre des entrées dans le vecteur iris.target
Target_names.size() nous donnes le nombre des classes

```
[ ] print(iris.data.shape)
    print(iris.data.size)
    print(iris.target.size)
    print(iris.target_names.size)
```

```
(150, 4)
600
150
3
```

2.3 Téléchargement et importation de données

Il est possible de charger un jeu de donnée directement depuis le site mldata.org qui contient de nombreux jeux de données gratuits, grâce à la fonction datasets.fetch_mldata.

1- Importez les données 'MNIST original'.

Quelques informations sur MNIST la base de donnes des Digits(0-9) manuscrites :

```
mnist = datasets.fetch_openml('mnist_784')
mnist.details
```

```
{'creator': ['Yann LeCun', 'Corinna Cortes', 'Christopher J.C. Burges'],
 'default_target_attribute': 'class',
 'description_version': '1',
 'file_id': '52667',
 'format': 'ARFF',
 'id': '554',
 'language': 'English',
 'licence': 'Public',
 'md5_checksum': '0298d579eb1b86163de7723944c7e495',
 'name': 'mnist_784',
 'processing_date': '2020-11-20 20:12:09',
 'status': 'active',
 'tag': ['AzurePilot',
  'OpenML-CC18',
```

2- Affichez la matrice des données, le nombre de données et de variables, les numéros de classes pour chaque donnée, ainsi que la moyenne, l'écart type, les valeurs min et max pour chaque variable ; enfin donnez le nombre de classe avec la fonction *unique*.

La matrice des données : contienne les valeurs de **784** pixels pour **7000** images :

```
print(mnist.data)
```

```
       pixel1  pixel2  pixel3  pixel4  ...  pixel781  pixel782  pixel783  pixel784
0         0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
1         0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
2         0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
3         0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
4         0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
...       ...     ...     ...     ...  ...       ...       ...       ...       ...
69995     0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
69996     0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
69997     0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
69998     0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0
69999     0.0     0.0     0.0     0.0  ...       0.0       0.0       0.0       0.0

[70000 rows x 784 columns]
```

```
[ ]  print(mnist.data.shape)
```

```
(70000, 784)
```

Le numéro de classe pour chaque donnée : Les classe sont {0,1,2,3,4,5,6,7,8,9} qui représente les 10 Digits :

```
[ ]  print(mnist.target)
```

```
0        5
1        0
2        4
3        1
4        9
        ..
69995    2
69996    3
69997    4
69998    5
69999    6
Name: class, Length: 70000, dtype: category
Categories (10, object): ['0', '1', '2', '3', ..., '6', '7', '8', '9']
```

Le moyenne pour chaque Variable : les valeurs des pixels sont entre 0 et 255 donc les moyennes est aussi :

```
[ ]  print(mnist.data.mean(0))
```

```
pixel1      0.000000
pixel2      0.000000
pixel3      0.000000
pixel4      0.000000
pixel5      0.000000
              ...
pixel780    0.001714
pixel781    0.000000
pixel782    0.000000
pixel783    0.000000
pixel784    0.000000
Length: 784, dtype: float64
```

Les écart-types pour chaque variable :

```
[ ]  print(mnist.data.std(0))
```

```
pixel1      0.000000
pixel2      0.000000
pixel3      0.000000
pixel4      0.000000
pixel5      0.000000
              ...
pixel780    0.320889
pixel781    0.000000
pixel782    0.000000
pixel783    0.000000
pixel784    0.000000
Length: 784, dtype: float64
```

Le minimum et le maximum pour chaque variable : les min et les max sont entre 0 et 255 (niveaux de gris) :

```
print(mnist.data.min(0))
```

```
pixel1      0.0
pixel2      0.0
pixel3      0.0
pixel4      0.0
pixel5      0.0
           ...
```

```
print(mnist.data.max(0))
```

```
pixel1       0.0
pixel2       0.0
pixel3       0.0
pixel4       0.0
pixel5       0.0
            ...
pixel780    62.0
pixel781     0.0
pixel782     0.0
pixel783     0.0
pixel784     0.0
Length: 784, dtype: float64
```

Le nombre des classes avec la méthode **unique()** de **numpy** qui donne le nombre des valeurs existent dans le matrice sans redondance, dans la base MNIST c'est 10 car on parle des chiffre de 0 à 9 :

```
[ ]  print(np.unique(mnist.target).size)

     10
```

2.4 Génération de données et affichage

1. Utiliser l'aide (*help*) pour voir comment utiliser la fonction *datasets.make_blobs*.

La fonction ***make_blobs()*** peut être utilisée pour générer des gouttes de points avec une distribution gaussienne. Nous pouvons contrôler le nombre de blobs à générer et le nombre d'échantillons à générer, ainsi qu'une multitude d'autres propriétés.

```
help(datasets.make_blobs)
```

```
Help on function make_blobs in module sklearn.datasets._samples_generator:

make_blobs(n_samples=100, n_features=2, centers=None, cluster_std=1.0, center_box=(-10.0, 10.0), shuffle=True, random_state=None)
    Generate isotropic Gaussian blobs for clustering.

    Read more in the :ref:`User Guide <sample_generators>`.

    Parameters
    ----------
    n_samples : int or array-like, optional (default=100)
        If int, it is the total number of points equally divided among
        clusters.
        If array-like, each element of the sequence indicates
        the number of samples per cluster.

    n_features : int, optional (default=2)
        The number of features for each sample.

    centers : int or array of shape [n_centers, n_features], optional
```

Les hyperparamètres de fonction make_blobs()

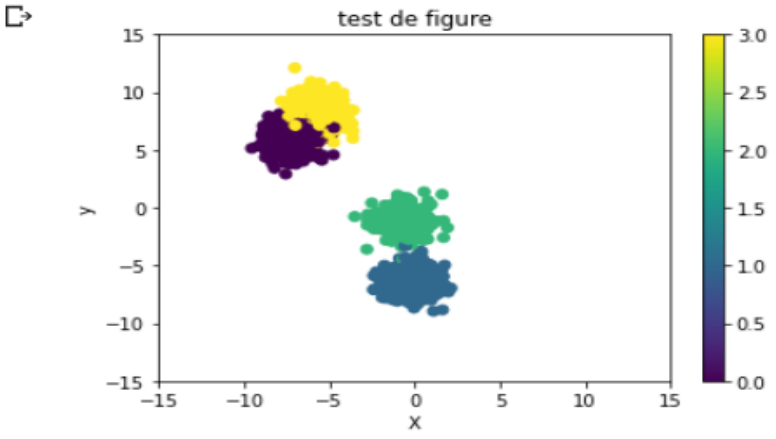
2. Générez 1000 données de deux variables réparties en 4 groupes.

n_samples est le nombre des données , *centers* est le nombre des classes et *n_features* est le nombre des variables (caractéristiques) :

```
X, y = datasets.make_blobs(n_samples=1000, centers=4, n_features=2)
```

3. Utilisez les fonctions *figure, scatter, title, xlim, ylim, xlabel, ylabel* et *show* pour afficher les données avec des couleurs correspondant aux classes. Les axes x et y seront dans l'intervalle [-15, 15] et devrons avoir un titre. La figure doit aussi avoir un titre.

```
[ ]  f = plt.figure()
     plt.scatter(X[:,0], X[:,1],c=y)
     plt.colorbar()
     plt.title('test de figure')
     plt.xlabel('X')
     plt.ylabel('y')
     plt.xlim(-15,15)
     plt.ylim(-15,15)
     plt.show()
     plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

4. Générez 100 données de deux variables réparties en 2 groupes, puis 500 données de deux variables réparties en 3 groupes. Concaténez (*vstack* et *hstack*) les deux jeux de données et les numéros de classe pour fusionner les deux jeux de données. Affichez les trois ensembles avec *scatter* comme précédemment.

Génération des deux datasets :

```
[ ]  X_100, y_100 = datasets.make_blobs(n_samples=100, centers=2, n_features=2)
     X_500, y_500 = datasets.make_blobs(n_samples=500, centers=3, n_features=2)
```

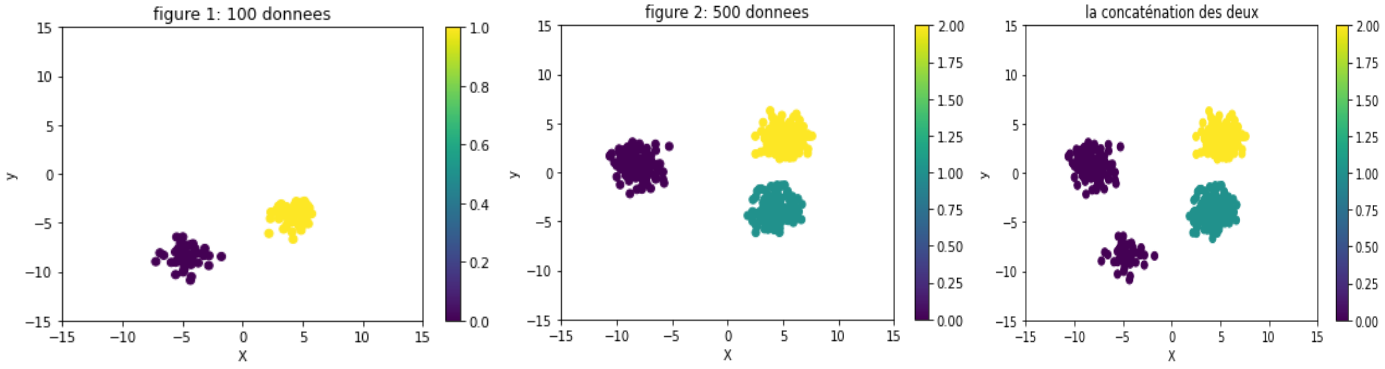
Concaténation des deux datasets :

```
X_600 = np.vstack((X_100, X_500))
y_600 = np.hstack((y_100, y_500))
```

Création d'une fonction d'affichage des figures :

```
def sc(X, y, title):
    plt.scatter(X[:,0], X[:,1], c=y)
    plt.colorbar()
    plt.title(title)
    plt.xlabel('X')
    plt.ylabel('y')
    plt.xlim(-15,15)
    plt.ylim(-15,15)
    plt.show()
```

Visualisation des trois datasets :



Le premier TP est fini ici, le but de cette partie était de familialiser avec le module sikit-learn. Vous trouverez le code complet de **TP N° 1** dans mon notebook Google-colab suivant : https://colab.research.google.com/drive/1jrCg7Cpppdvq92_8O4H3Ah7JVBbTpJxW?usp=sharing

3. TP2 : Prétraitement et visualisation des données

Le package **sklearn.preprocessing** offre plusieurs fonctions pour la transformations de données, i.e. changer les caractéristiques initiales de données en une représentation qui est plus approprié pour le traitement de ces données. Ce traitement est souvent nécessaire pour les données en grande dimension.
Dans cette partie, nous allons démarrer avec le package **sklearn.preprocessing** de bibliothèque Scikit-learn pour voir comment faire quelque traitement sur les données avant d'entrer l'exploitation des données.

3.1 Normalisation des données

La normalisation est le processus de mise à l'échelle des échantillons individuels pour avoir une norme unitaire. Ce processus peut être utile si vous prévoyez d'utiliser une forme quadratique telle que le produit scalaire ou tout autre noyau pour quantifier la similarité de n'importe quelle paire d'échantillons.
Cette hypothèse est la base du modèle d'espace vectoriel souvent utilisé dans les contextes de classification et de regroupement de textes.

Dans cet atelier nous allons ignorer la forme de distribution des données et on simplement transforme les données en centrent en retirant la valeur moyenne de chaque variable, puis divisant les valeurs des variables par l'écart-type.

Pour ces besoins nous allons utiliser *NumPy*, qu'est une bibliothèque pour langage de programmation Python, destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux.

Importez les librairies **numpy** (calcul scientifique) et preprocessing (prétraitement de données)

1- Créez la matrice **X** suivante :

```
1, -1,  2,
2,  0,  0,
0,  1, -1
```

Numpy.array() est la méthode qui permet de créer des matrices (des vecteurs) de n'importe qu'elle taille en séparant les ligne par des virgules :

```
[1]  import numpy as np
     from sklearn import *
     import matplotlib.pyplot as plt
```

```
X = np.array([[1, -1, 2],[2, 0, 0],[0, 1, -1]])
print(X)
```

```
[[ 1 -1  2]
 [ 2  0  0]
 [ 0  1 -1]]
```

2- Visualisez X et calculez la moyenne et la variance de X.

Les méthodes mean() est std() retourne la moyenne et la variance de tout objet de type ***numpy.array***

```
[ ]  m = X.mean() #moyenne
     print("nmoyenne =",m)
```

```
nmoyenne = 0.4444444444444444
```

```
[ ]  v = X.std() #variance
     print("variance =", v)
```

```
variance = 1.0657403385139377
```

3- Utilisez la fonction **scale** pour normaliser la matrice X. Que constatez vous ?

On remarque que les valeurs de nouveau matrice sont la soustraction de moyenne de matrice, puis divise par l'écart-type, les valeurs toujours entre la valeur maximale de X : **2** et la valeur minimale **-2**:

```
NormalisedX = preprocessing.scale(X)
print(NormalisedX)
```

```
[[ 0.         -1.22474487  1.33630621]
 [ 1.22474487  0.         -0.26726124]
 [-1.22474487  1.22474487 -1.06904497]]
```

4- Calculer la moyenne et la variance de la matrice X normalisé. Expliquez le résultat obtenu.

On remarque que la Moyenne est presque **nul** et la variance est **1** car le scaling est la mise en seule échelle toutes les valeurs de notre matrice :

```
Normalised_mean = NormalisedX.mean()
Normalised_var = NormalisedX.std()
print("normalized moyenne = ",Normalised_mean)
print("normalized variance = ",Normalised_var)
```

```
normalized moyenne =  4.9343245538895844e-17
normalized variance =  1.0
```

3.2 Normalisation MinMax

Le procédé de normalisation MIN-MAX n'a besoin que du min et du max. L'idée est la suivante, on ramène toutes les valeurs de la variable entre 0 et 1, tout en conservant les distances entre les valeurs.

1- Créez la matrice de données **X2** suivante :

```
1, -1,  2,
2,  0,  0,
0,  1, -1
```

```
X2 = np.array([[1, -1, 2], [2, 0, 0], [0, 1, -1]])
print(X2)
```

```
[[ 1 -1  2]
 [ 2  0  0]
 [ 0  1 -1]]
```

2- Visualisez la matrice et calculez la moyenne sur les variables.

```
m2 = X2.mean(0)
print("nmoyenne sur les variables =",m2)
```

```
nmoyenne sur les variables = [1.         0.         0.33333333]
```

3- Normalisez les données dans l'intervalle [0 1]. Visualisez les données normalisées et calculez la moyenne sur les variables. Que constatez-vous ?

La moyenne pour les variables après la normalisation est de **½** :

```
normalizedX2 = preprocessing.minmax_scale(X2, feature_range=(0,1))
print(normalizedX2)
```

```
[[0.5        0.         1.        ]
 [1.         0.5        0.33333333]
 [0.         1.         0.        ]]
```

```
[ ] print("nmoyenne sur les variables", normalizedX2.mean(0))
```

```
nmoyenne sur les variables [0.5        0.5        0.44444444]
```

3.3 Visualisation des données

La visualisation de données est une étape clé pour comprendre l'ensemble des données et en tirer des conclusions, pour visualiser les données **Iris**, nous allons utiliser la bibliothèque **Matplotlib** de Python pour ces raisons, et utilisons son module **pyplot** pour le tracé de nos données et **cm** pour la palette de couleurs.

1- Chargez les données Iris

Iris contient les données de **4** caractéristiques pour **150** fleurs divisé sur **3** classes :

```
[ ] iris = datasets.load_iris()
    X = iris.data
    print(X)
    y = iris.target
    print(y)
```

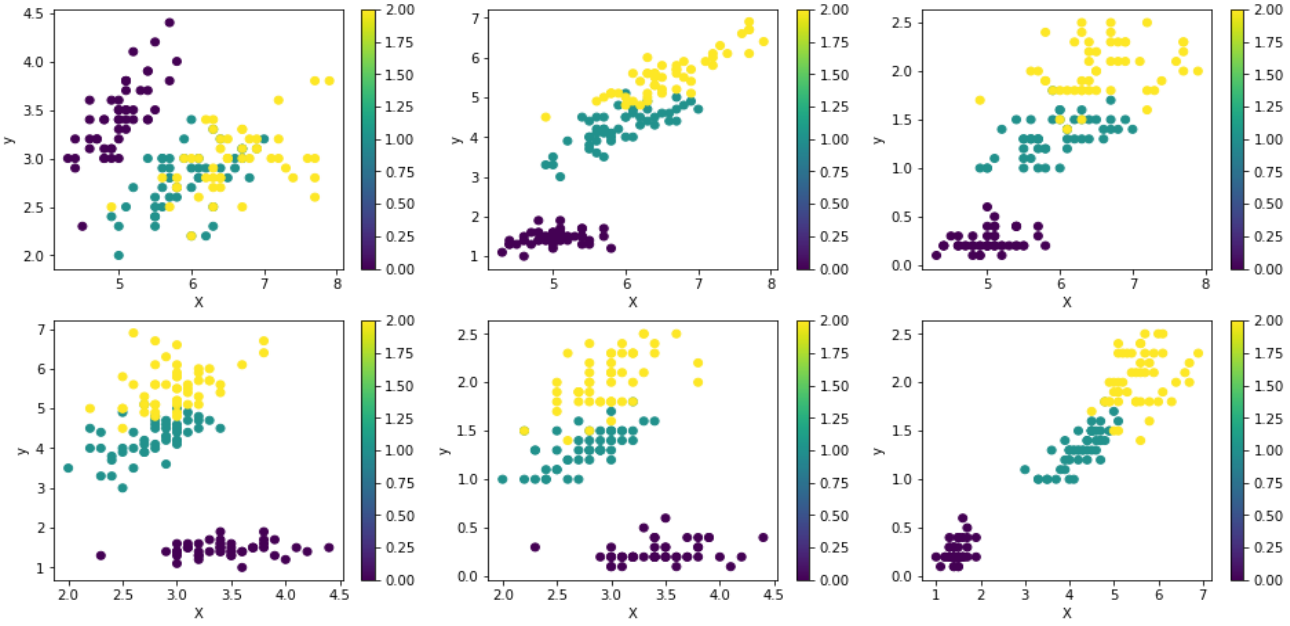
```
 [5.4 3.4 1.5 0.4]
 [5.2 4.1 1.5 0.1]
 [5.5 4.2 1.4 0.2]
 [4.9 3.1 1.5 0.2]
 [5.  3.2 1.2 0.2]
 [5.5 3.5 1.3 0.2]
 [4.9 3.6 1.4 0.1]

 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

2- Visualisez le nuage de points en 2D avec des couleurs correspondant aux classes en utilisant toutes les combinaisons de variables. Quelle est la meilleure visualisation ? Justifiez votre réponse.

Pour **4** variables on a **12** combinaisons possibles, mai la combinaison (X,Y) est le même (Y,X) avec les axes inversés, donc il nous reste **6** combinaisons sans redondance sont les suites :

```
fig = plt.figure(figsize=(16,8))
plt.subplot(231); plt.scatter(X[:,0], X[:,1], c=y); plt.colorbar(); plt.xlabel('X'); plt.ylabel('y')
plt.subplot(232); plt.scatter(X[:,0], X[:,2], c=y); plt.colorbar(); plt.xlabel('X'); plt.ylabel('y')
plt.subplot(233); plt.scatter(X[:,0], X[:,3], c=y); plt.colorbar(); plt.xlabel('X'); plt.ylabel('y')
plt.subplot(234); plt.scatter(X[:,1], X[:,2], c=y); plt.colorbar(); plt.xlabel('X'); plt.ylabel('y')
plt.subplot(235); plt.scatter(X[:,1], X[:,3], c=y); plt.colorbar(); plt.xlabel('X'); plt.ylabel('y')
plt.subplot(236); plt.scatter(X[:,2], X[:,3], c=y); plt.colorbar(); plt.xlabel('X'); plt.ylabel('y')
plt.show()
```

On peut remarquer que la meilleure combinaison est **(2,3)** et aussi dans un certaine niveau la combinaison **(0,3)** car les données sont séparées plus que dans les autres visualisations :

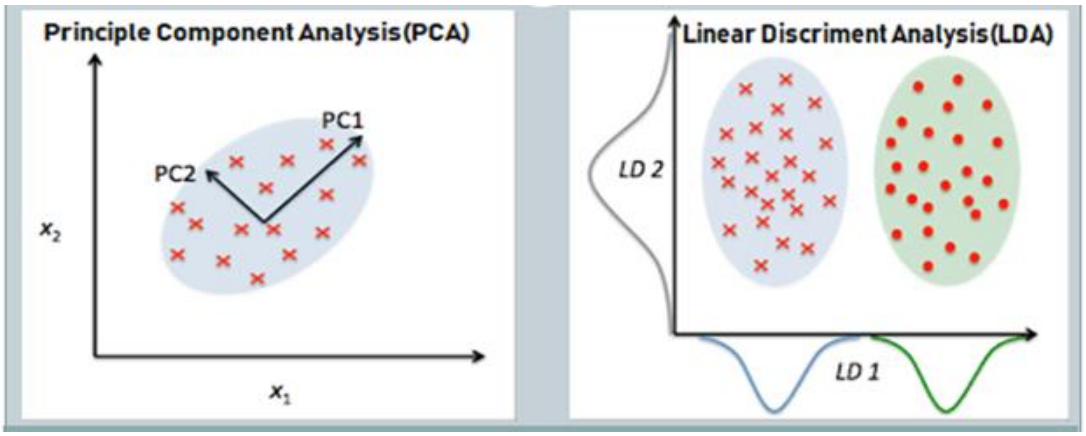


3.4 Réduction de dimensions et visualisation de données

Dans Machine Learning et Statistiques, la réduction de la dimensionnalité est le processus de réduction du nombre de variables aléatoires considérées via l'obtention d'un ensemble de variables principales. Il peut être divisé en sélection de caractéristiques et extraction de caractéristiques.

Dans ce travail Nous traiterons de deux algorithmes principaux en réduction de dimensionnalité :

- **Analyse en Composantes Principales (PCA) :** L'ACP est une technique d'extraction de caractéristiques. Ainsi, il combine nos variables d'entrée d'une manière spécifique, puis nous pouvons supprimer les variables "les moins importantes" tout en conservant les parties les plus précieuses de toutes les variables.
- **Analyse discriminante linéaire (LDA) :** est un type de combinaison linéaire, un processus mathématique utilisant divers éléments de données et appliquant une fonction à ce site pour analyser séparément plusieurs classes d'objets ou d'éléments.



Dans cette partie nous allons appliquer les deux méthodes sur la base de données Iris :

1- Les méthodes **PCA** et **LDA** peuvent etre importé à partir des package suivants :

```
import from sklearn.decomposition
import PCA from sklearn.lda import LDA
```

```
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

help(LDA)
help(PCA)
```

```
Help on class LinearDiscriminantAnalysis in module sklearn.discriminant_analysis:

class LinearDiscriminantAnalysis(sklearn.base.BaseEstimator, sklearn.linear_model._base.LinearClassifierMixin, sklearn.base.TransformerMixin)
 |  LinearDiscriminantAnalysis(solver='svd', shrinkage=None, priors=None, n_components=None, store_covariance=False, tol=0.0001)
 |
 |  Linear Discriminant Analysis
 |
 |  A classifier with a linear decision boundary, generated by fitting class
 |  conditional densities to the data and using Bayes' rule.
```

```
Help on class PCA in module sklearn.decomposition._pca:

class PCA(sklearn.decomposition._base._BasePCA)
 |  PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None)
 |
 |  Principal component analysis (PCA).
 |
 |  Linear dimensionality reduction using Singular Value Decomposition of the
 |  data to project it to a lower dimensional space. The input data is centered
 |  but not scaled for each feature before applying the SVD.
```

2- Analysez le manuel d'aide pour ces deux fonctions (**pca** et **lda**) et appliquez les sur la base Iris. Il faudra utiliser **pca.fit(Iris).transform(Iris)** et sauvegardez les résultats dans IrisPCA pour la PCA et IrisLDA pour la LDA.

J'ai donné le nombre des caractéristiques à conserver **nbr_component=2**, donc on va supprimer les deux variables les moins significatives selon chaque méthode, et bien le résultat est la même data avec juste **2** colonnes :

```
[2]  pca = PCA(n_components=2)
     lda = LDA(n_components=2)
```

```
irisPCA = pca.fit(X).transform(X)
irisLDA = lda.fit(X,y).transform(X)
# ou bien :
#irisLDA= lda.fit_transform(X,y)
#irisPCA= pca.fit_transform(X)
```

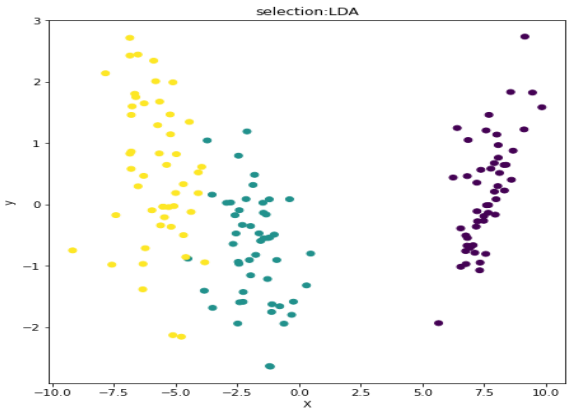
```
[13] irisPCA.shape

     (150, 2)
```

```
[14] irisLDA.shape

     (150, 2)
```

3- Visualisez les nuages de points avec les nouvelles axes obtenus : une image pour l'ACP et une autre pour l'ADL et utiliser la classe de Iris comme couleurs de points. Quelle différence constatez-vous entre les deux visualisations? Expliquer votre raisonnement.

```
fig = plt.figure(figsize=(16,8))
plt.subplot(121); plt.scatter(irisPCA[:,0], irisPCA[:,1], c=y); plt.title('selection:PCA'); plt.xlabel('X'); plt.ylabel('y')
plt.subplot(122); plt.scatter(irisLDA[:,0], irisLDA[:,1], c=y); plt.title('selection:LDA'); plt.xlabel('X'); plt.ylabel('y')
plt.show()
```

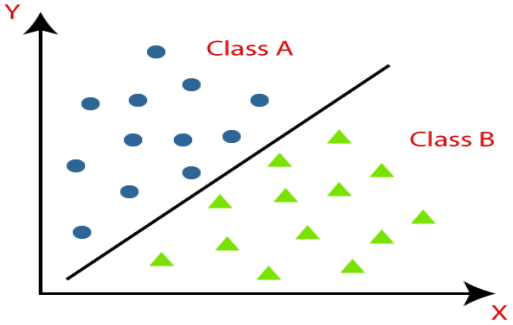
On peut remarquer que les visualisations sont les mêmes, ce qui montre que les deux méthodes ont sélectionné les mêmes variables, sauf que le degré d'importance sont déférents, ce qui couse que les axes sont inversés.



Le deuxième TP est fini ici, le but de cette partie était de faire quelques taches de prétraitement et visualisation des donnés. Vous trouverez le code complet de TP N° **2** dans mon notebook Google-colab suivant : https://colab.research.google.com/drive/1eNWbhvnvftYvT3GLucsWjCCvVwiRxhD ?usp=sharing

4. TP3 : Introduction à la classification

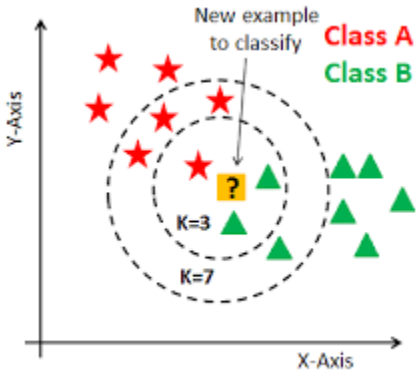
La classification est une technique d'apprentissage supervisé qui est utilisée pour identifier la catégorie de nouvelles observations sur la base des données d'apprentissage. Dans la classification, un programme apprend à partir de l'ensemble de données ou des observations donnés, puis classe la nouvelle observation en un certain nombre de classes ou de groupes. Par exemple, oui ou non, 0 ou 1, spam ou non spam, chat ou chien, etc. Les classes peuvent être appelées cibles/étiquettes ou catégories.
Contrairement à la régression, la variable de sortie de la classification est une catégorie, pas une valeur, telle que "vert ou bleu", "fruit ou animal", etc. Étant donné que l'algorithme de classification est une technique d'apprentissage supervisé, il prend donc des données d'entrée étiquetées, qui signifie qu'il contient une entrée avec la sortie correspondante.



L'objectif de ce **TP** est de programmer et de tester deux algorithmes de classification, très simples mais très efficaces : l'algorithme du Plus Proche Voisin **(NN et KNN)** et le Classifieur Bayesien Naïf **(Naive-Bays)**. Nous n'étudions ici que les versions les plus simple de ces algorithmes.
Pour ce **TP** nous aurons besoin d'importer ***sklearn*** et **numpy**. Les tests pourront se faire sur les données (data) prédéfinies de *sklearn* qui sont fournies avec leurs étiquettes de classe (target), comme **iris** qui contiens **iris.data** et **iris.target.**

4.1 Plus Proche Voisin (NN et KNN)

Les K plus proches voisins KNN ou la recherche du voisin le plus proche pour k=1(NN), en tant que forme de recherche de proximité, est le problème d'optimisation consistant à trouver le point dans un ensemble donné qui est le plus proche (ou le plus similaire) d'un point donné. La proximité est généralement exprimée en termes de fonction de dissemblance : moins les objets sont similaires, plus les valeurs de la fonction sont grandes.



L'intuition derrière l'algorithme des K plus proches voisins est l'une des plus simples de tous les algorithmes de Machine Learning supervisé :

- Étape 1 : Sélectionnez le nombre K de voisins
- Étape 2 : Calculez la distance

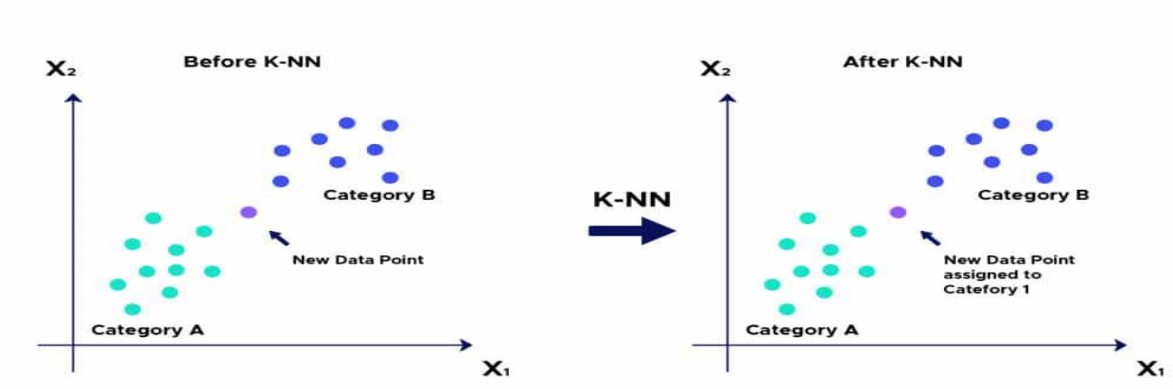
$$\sum_{i=1}^{n} |x_i - y_i|$$ **Euclidienne**

$$\sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$ **Manhattan**

Du point non classifié aux autres points.

- Étape 3 : Prenez les K voisins les plus proches selon la distance calculée.
- Étape 4 : Parmi ces K voisins, comptez le nombre de points appartenant à chaque catégorie.

- Étape 5 : Attribuez le nouveau point à la catégorie la plus présente parmi ces K voisins.
- Étape 6 : Notre modèle est prêt :



1. Créez une fonction PPV(X,Y) qui prend en entrée des données X et des étiquettes Y et qui renvoie une étiquette, pour chaque donnée, prédite à partir du plus proche voisin de cette donnée. Ici on prend chaque donnée, une par une, comme donnée de test et on considère toutes les autres comme données d'apprentissage. Cela nous permet de tester la puissance de notre algorithme selon une méthode de validation par validation croisée (cross validation) de type "leave one out".
2. La fonction PPV calcule une étiquette prédite pour chaque donnée. Modifiez la fonction pour calculer et renvoyer l'erreur de prédiction : c'est à dire le pourcentage d'étiquettes mal prédites.
3. Testez sur les données Iris.
4. Testez la fonction des K Plus Proches Voisins de sklearn (avec ici $K = 1$). Les résultats sont-ils différents ? Testez avec d'autres valeurs de K.
5. **BONUS** : Modifiez la fonction PPV pour qu'elle prenne en entrée un nombre K de voisins (au lieu de 1). La classe prédite sera alors la classe *majoritaire* parmi les K voisins.

Implémentation d'algorithme KNN avec K par défaut est **1** donc l'algorithme NN mais en peut passer une autre valeur par paramètre pour avoir le KNN, utilisant comme mesure de similarité la distance euclidienne :

```
def PPV(X, y, k=1):
    targets = np.zeros(len(X))
    for i in range(len(X)):
        X_test = X[i]
        X_train = np.delete(X, i, 0)
        l = np.array(metrics.pairwise.euclidean_distances(X_train, [X_test])).flatten()
        indexs = np.argsort(l)[:k]
        cs = []
        for j in range(k): cs.append(y[indexs[j]])
        c = np.bincount(cs).argmax()
        targets[i] = c
    error = 1 - metrics.accuracy_score(targets, y)
    return targets, error
```

Avec k=1 l'erreur de classification est de 4%

```
pred, error = PPV(X,y, k=1)
print(f"erreur de classification = {round(error, 2)*100} %")
print(pred)
```

```
erreur de classification = 4.0 %
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 1.
 2. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2.]
```

Avec K=10 l'erreur est de 2% :

```
pred, error = PPV(X,y, k=10)# changer le k pour changer la taille de voisinage
print(f"erreur de classification = {round(error, 2)*100} %")
print(pred)
```

```
erreur de classification = 2.0 %
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2.]
```

4.2 Classifieur Bayesien Naïf (Naive-bayes)

L'algorithme du Classifieur Bayesien Naïf est un algorithme de classification basé sur le calcul de probabilité d'appartenance à chaque classe. C'est à dire que la donnée de test (à classer) sera affectée à la classe la plus probable.
Les probabilités d'appartenances à chaque classe sont calculées à partir des données d'apprentissage de la façon suivante :

$$classe(x) = argmax_{\omega_k}\{\prod_i P(x_i/\omega_k)P(\omega_k)\}$$

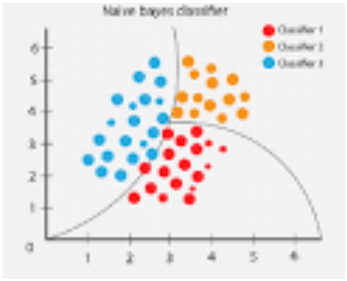
Ici P(w) est la probabilité à priori d'appartenir à la classe k. autrement dit c'est la probabilité d'obtenir une donnée de la classe k si on tire une donnée au hasard. P(xi=w) est la probabilité qu'une donnée x ait la valeur xi pour la variable i, si on connaît sa classe w. Pour chaque classe k, cette probabilité peut se calculer comme $1 - d_{xk}/\sum d_{xb}$, , avec **dxk** la distance entre la donnée x et le

barycentre de k (c'est à dire la moyenne de la classe), et $\sum d_{xb}$ la somme des distances entre cette donnée et chaque barycentre de chaque classe.

$$P(A|B) = \frac{P(B|A)\ P(A)}{P(B)}$$

using Bayesian probability terminology, the above equation can be written as

$$\text{Posterior} = \frac{\text{prior x likelihood}}{\text{evidence}}$$



Dans cette partie je vais essayer d'implémenter l'algorithme de naive bayes et comparer les résultats obtenus avec les résultats données par la méthode prédéfini dans Sklearn :

1. Créez une fonction CBN(X,Y) qui prend en entrée des données X et des étiquettes Y et qui renvoie une étiquette, pour chaque donnée, prédite à partir de la classe la plus probable selon l'équation (1). Ici encore, on prend chaque donnée, une par une, comme donnée de test et on considère toutes les données comme données d'apprentissage. Il est conseillé de calculer d'abord les barycentres et les probabilités à priori $P(\omega_k)$ pour chaque classe, puis de calculer les probabilités conditionnelles $P(x_i/\omega_k)$ pour chaque classe et chaque variable.

Méthode pour regrouper les éléments attribués à la même classe :

```
[ ]  def separate_by_class(X,y):
         d = {}
         for i in range(len(X)):
             if y[i] not in d:
                 d[y[i]] = list()
             d[y[i]].append(list(X[i]))
         return d
```

Méthode pour calculer la probabilité de chaque classe, j'ai testé la méthode sur les **target iris** , et elle a donné une équiprobabilité puisque dans iris on a **50** éléments dans chaque une des 3 classes, donc les résultats sont exactes :

```
[ ]  def probabilite_class(y):
         classes = np.unique(y)
         probs = {}
         for c in classes:
             probs[c] = list(y).count(c)/len(y)
         return probs
     probabilite_class(y)
```

```
{0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0.3333333333333333}
```

Méthode pour calculer le barycentre de chaque classe en fonction des variables, j'ai testé la méthode sur les données **iris**, et elle m'a donnée le cordonnées des barycentres des 3 classes (4 cordonnées = 4 caractéristiques) :

```
[ ]  def barycenter_class(X,y):
         classes = np.unique(y)
         sep = separate_by_class(X,y)
         barycenters = {}
         for c in classes:
             barycenters[c] = np.array(sep[c]).mean(0)
         return barycenters
     barycenter_class(X,y)
```

```
{0: array([5.006, 3.428, 1.462, 0.246]),
 1: array([5.936, 2.77 , 4.26 , 1.326]),
 2: array([6.588, 2.974, 5.552, 2.026])}
```

La méthode principale ***CBN()*** qui fait le travail d'un classifieur naive bayes :

```
def CBN(X, y):
    targets = np.zeros(len(X))
    for i in range(len(y)):
        X_test = X[i]; X_train = np.delete(X, i, 0)
        y_test = y[i]; y_train = np.delete(y, i)
        classes = np.unique(y_train)
        #calcule des P(Wk) probabilités de chaque classe k
        probabilites = probabilite_class(y_train)
        #calcule des barycentres de chaque classe k
        barycentres = barycenter_class(X_train, y_train)
        #calculer la probabilité conditionelle des valeurs des données sachant que l
        distance = np.array([abs(X_test-b) for b in barycentres.values()])
        dist_total = np.sum(distance, axis=0)
        proba = abs(1-(distance/dist_total))
        produit = []
        j = 0
        for p in probabilites.values():
            produit.append(p*np.prod(proba[j]))
            j += 1
        targets[i] = np.argmax(np.array(produit))
    #l'erreur de classification
    error = 1 - metrics.accuracy_score(targets, y)
    return targets, error
```

2. La fonction CBN calcule une étiquette prédite pour chaque donnée. Modifiez la fonction pour calculer et renvoyer l'erreur de prédiction : c'est à dire le pourcentage d'étiquettes mal prédites. Testez sur les données Iris.

Notre méthode a une erreur de classification de 13% :

```
[ ]  pred, error = CBN(X,y)
     print(f"error = {round(error, 2)*100} %")
     print(pred)
```

```
error = 13.0 %
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 2. 2. 2. 1. 1. 1. 2. 1. 1. 1. 1. 1. 1. 1. 2. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 2. 1. 2. 1. 1. 1. 1. 1. 1. 1. 2. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 2. 1. 2. 2. 2. 2. 1. 2. 2. 2. 2. 2. 2. 1. 1. 2. 2. 2. 2. 1.
 2. 1. 2. 1. 2. 2. 1. 2. 2. 2. 2. 2. 2. 1. 1. 2. 2. 2. 2. 2. 2. 2. 1. 2.
 2. 2. 2. 2. 2. 2.]
```

3. Testez la fonction du Classifieur Bayesien Naïf inclut dans sklearn. Cette fonction utilise une distribution Gaussienne au lieu des distances aux barycentres. Les résultats sont-ils différents ?

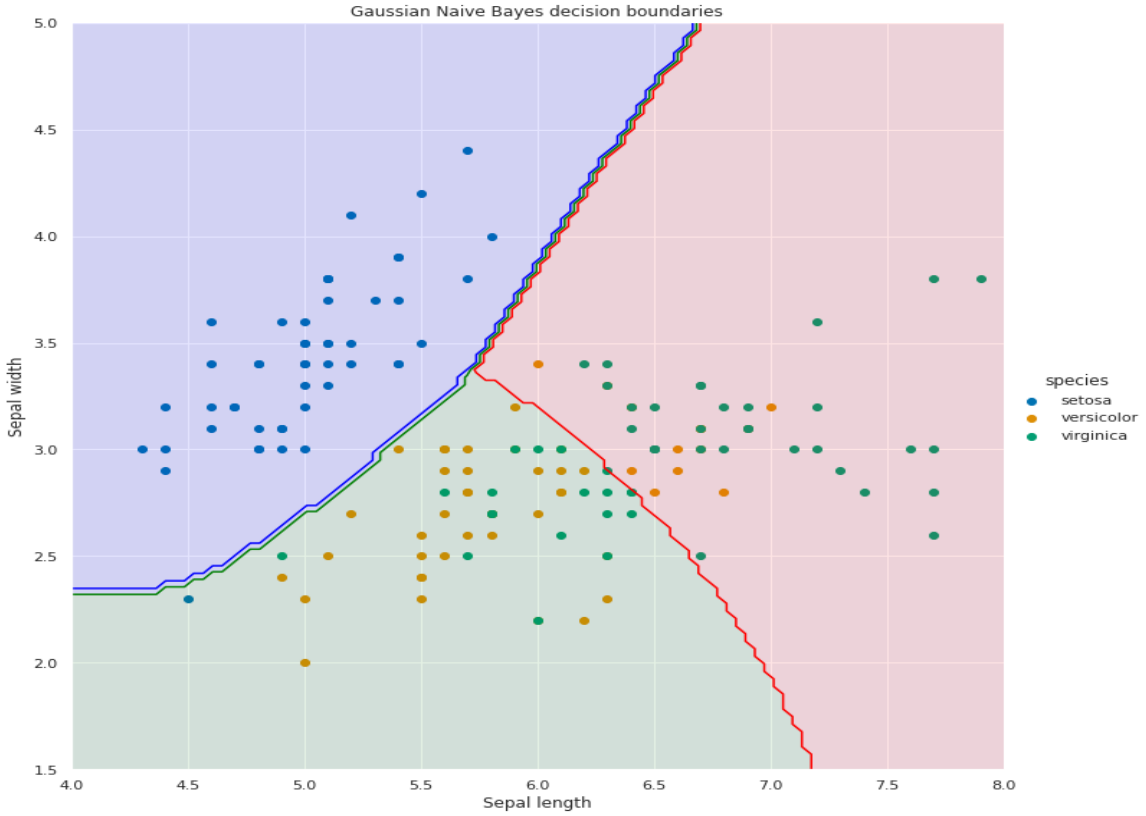
Le modèle naive bayes prédéfini dans la bibliothèque **sklearn** a donné une erreur de classification de **4**%, notre méthode a donné **13**% d'erreur pour les mêmes données ! c''est possible que dans sklearn les mésures sont bien calculé ou bien elle utilise un type de distance plus efficace :

```
model = naive_bayes.GaussianNB()
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.3,random_state=109)
model.fit(X_train, y_train)
pred = model.predict(X_test)
error = 1 - metrics.accuracy_score(pred, y_test)
print("****************** erreur*****************")
print(f"error = {round(error, 2)*100} %")
print("****************** prediction*****************")
print(pred)
```

```
****************** erreur*****************
error = 4.0 %
****************** prediction*****************
[2 1 2 0 2 1 0 2 1 2 2 0 1 0 0 0 1 1 0 1 1 0 2 0 0 2 2 1 1 2 1 2 1 2 2 1 0
 2 2 1 1 1 1 2 0]
```

J'ai fait une tache supplémentaire pour la visualisation des résultats de classification par algorithme naive bays, la visualisions est selon les variables petal-lenght et petal-width :

```
color_list = ['Blues','Greens','Reds']
my_norm = colors.Normalize(vmin=-1.,vmax=1.)
g = sns.FacetGrid(iris, hue="species", size=10, palette = 'colorblind') .map(plt.scatter, "1_sepal_length", "2_sepal_width",)  .add_legend()
my_ax = g.ax # i will try sepal ou lieu de petal
#Computing the predicted class function for each value on the grid
zz = np.array(  [predict_NB_gaussian_class( np.array([xx,yy]).reshape(-1,1), mu_list, std_list, pi_list)
                 for xx, yy in zip(np.ravel(X), np.ravel(Y)) ] )
print(zz.shape)
print(X.shape)
#Reshaping the predicted class into the meshgrid shape
Z = zz.reshape(X.shape)
#Plot the filled and boundary contours
my_ax.contourf( X, Y, Z, 2, alpha = .1, colors = ('blue','green','red'))
my_ax.contour( X, Y, Z, 2, alpha = 1, colors = ('blue','green','red'))
# Addd axis and title
my_ax.set_xlabel('Sepal length')
my_ax.set_ylabel('Sepal width')
my_ax.set_title('Gaussian Naive Bayes decision boundaries')

plt.show()
```



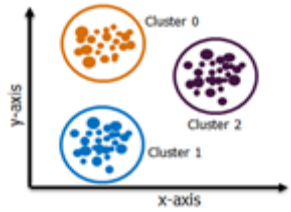
Le troisième TP est fini ici, le but de cette partie était de faire quelques taches d'apprentissage supervisé en implémentant des algorithme classification : ***KNN*** et ***naive-bayes***. Vous trouverez le code complet de **TP N° 3** dans mon notebook Google-colab suivant : **https://colab.research.google.com/drive/17jatxGPnaYHuwQOQbBIMfsmtUyIe2vg7?usp=sharing**

5. TP4 : Clustering des données

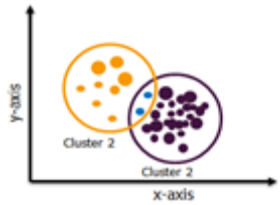
Le clustering est une méthode d'apprentissage automatique qui consiste à regrouper des points de données par similarité ou par distance. C'est une méthode d'apprentissage non supervisée et une technique populaire d'analyse statistique des données. Pour un ensemble donné de points, vous pouvez utiliser des algorithmes de classification pour classer ces points de données individuels dans des groupes spécifiques. En conséquence, les points de données d'un groupe particulier présentent des propriétés similaires. Dans le même temps, les points de données de différents groupes ont des caractéristiques différentes.

Il y a déférents types de clustering et le choix de type approprié dépend de problématique traité et nature des données :

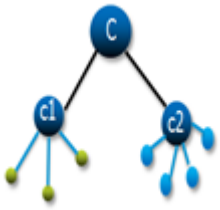
Clustering exclusif : dans le clustering exclusif, un élément appartient exclusivement à un cluster, pas à plusieurs. Dans l'image, vous pouvez voir que les données appartenant au cluster 0 n'appartiennent pas au cluster 1 ou au cluster 2. Le clustering k-means est un type de clustering exclusif.



Clustering en chevauchement : ici, un élément peut appartenir à plusieurs clusters avec un degré d'association différent entre chaque cluster. L'algorithme Fuzzy C-means est basé sur une classification par chevauchement.



Clustering hiérarchique : Dans le clustering hiérarchique (ascendant et descendant), les clusters ne sont pas formés en une seule étape mais suivent une série de partitions pour créer des clusters finaux. Cela ressemble à un arbre comme visible sur l'image.



Le but de ce TP est de créer une fonction Python capable de détecter des clusters de données homogènes dans un ensemble de données, puis d'analyser un jeu de données réelles, donc nous allons coder et appliquer une méthode de clustering exclusif très connue : Le K-means (k-moyennes), et aussi nous allons utiliser d'autre méthodes de clustering.

5.1 K-Moyennes (K-means)

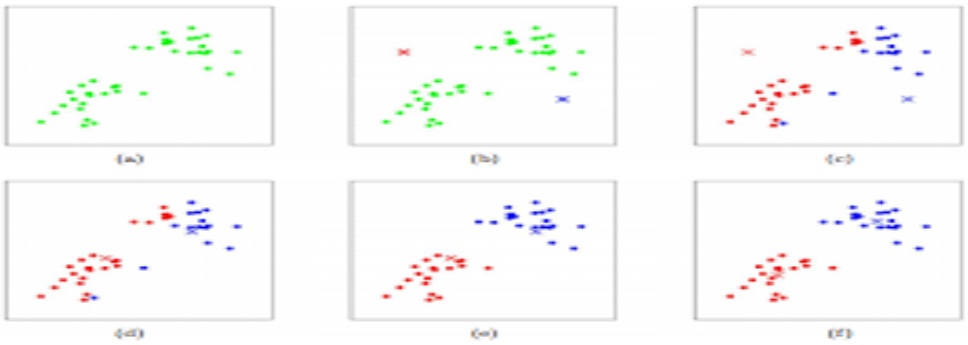
L'élément central de cet algorithme est le centroïde. Un **centroïde** est un point du jeu de données que l'on choisira comme le « centre » d'un cluster. C'est en fonction du centroïde que nous définirons l'appartenance à un cluster.
Un autre élément important dans cet algorithme est la distance. Une distance est une application qui associe un couple de vecteurs à un nombre réel positif. Il existe plusieurs types de distances dont la plus connue est la distance euclidienne. Soit, en dimension 2 : D (X, Y) = (X2+Y2)
Plus D (X, Y) est petit, plus X et Y sont proches.

L'Algorithme qu'on vas implémenter est comme suite :

Initialisation : On commence par choisir, au hasard, k centroïdes. Qui seront les centres des clusters de départ.

Boucle :

- **On construit k clusters** : Chaque point est dans le cluster du centroïde qui lui est le plus proche.
- **On calcule les nouveaux centroïdes** : Pour chacun des clusters qu'on vient de former, on calcule la moyenne. Celle-ci devient le nouveau centroïde (n'est pas nécessairement un point du jeu de donnée).
- **On recommence jusqu'à ce qu'à ce qu'il y ait convergence** : La convergence correspond au fait que les centroïdes ne changent pas après une mise à jour.



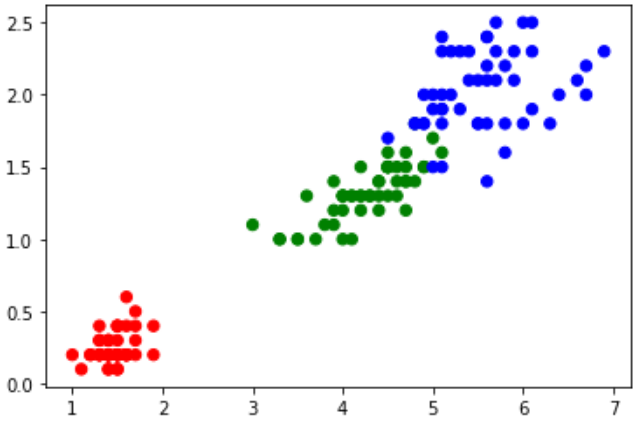
Le module ***Sikit-learn*** contient une méthode prête à utiliser K-mean nous allons tester cette méthode sur les données iris :

```
import pandas as pd
import numpy as np
import sklearn.metrics as sm
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn import datasets
iris = datasets.load_iris()
print(iris.feature_names)
print(iris.target)
print(iris.target_names)
#Stocker les données en tant que DataFrame Pandas
x=pd.DataFrame(iris.data)
# définir les noms de colonnes
x.columns=['Sepal_Length','Sepal_width','Petal_Length','Petal_width']
y=pd.DataFrame(iris.target)
a=y.columns=['Targets']
#Cluster K-means
model=KMeans(n_clusters=3)
#adapter le modèle de données
model.fit(x)
print(model.labels_)
plt.scatter(x.Petal_Length, x.Petal_width)
colormap=np.array(['Red','green','blue'])
```

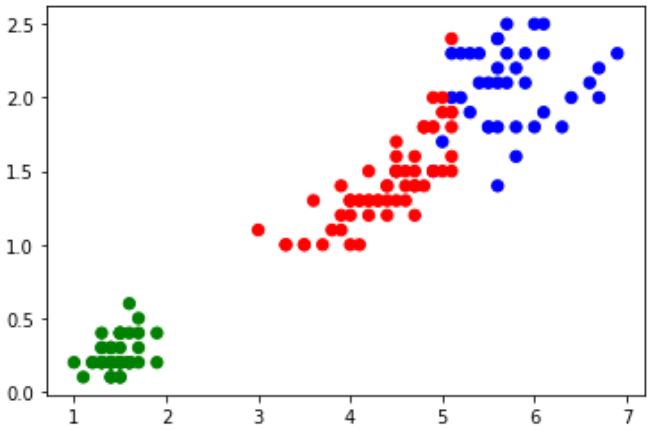
```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']#variables
['setosa' 'versicolor' 'virginica']#classes

#Les classes de chaque entrée dans iris :
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
#Les classes obtenus par K-means de sk-learn rouge = erreur :

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 2 2 2 0 2 2 2 2
```

```
 2 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 0 2 2 2 2 2 0 2 2 2 2 0 2 2 2 0 2 2 2 0 2
 2 0]
```

La distribution des clusters défini dans iris :



Le résultat obtenus par K-means de sk-learn:



Les résultats sont presque les mêmes avec un petit taux d'erreur.

1. Écrivez en python l'algorithme des K-Moyennes sous la forme d'une fonction. Vous trouverez une description de l'algorithme dans le cours ou sur internet. La fonction prend en entrée deux paramètres : la matrice des données et le nombre de clusters que l'on souhaite. Testez sur les données *Iris* ou sur des données que vous générez. Comparez avec la fonction *Kmeans* de *sklearn*.

Importer les outils nécessaires :

```
import pandas as pd
import numpy as np
import random as rd
from sklearn import *
import matplotlib.pyplot as plt
```

Importer le jeu des données iris :

```
[ ]  from sklearn import *
     from scipy.spatial.distance import euclidean
     iris = datasets.load_iris()
     x=iris.data
     y=iris.target
```

Méthode pour calculer la distance euclidienne entre deux points de données :

```
[ ]  def d_euclidean(x,y):
         return metrics.pairwise.euclidean_distances([x],[y])
```

La méthode principale pour regrouper les éléments dans des clusters :

```
def k_moyennes(data, k, max_iter = 100):
    n_iter = 0
    centroids = {}
    for i in range(k): centroids[i] = data[i]
    for i in range(max_iter):
        classes = {}
        targets = []
        for j in range(k):
            classes[j] = []
        for j in range(len(data)):
            distances = [d_euclidean(data[j], centroids[cen]) for cen in centroids]
            classe = np.argmin(distances)
            classes[classe].append(data[j])
            targets.append(classe)
        prev_centroids = dict(centroids)
        for classe in classes:
            centroids[classe] = np.mean(classes[classe], axis=0)
```

```
        for c in centroids:
            original_centroid = prev_centroids[c]
            current_centroid = centroids[c]

        ccens = np.array([v for v in centroids.values()])
        pcens = np.array([v for v in prev_centroids.values()])
        n_iter += 1

        if (ccens == pcens).all():
            break

    for k,v in classes.items(): classes[k] = np.array(v)
    return targets, n_iter
```

Tester notre méthode sur les données iris :

```
targets, n_iter = k_moyennes(x,3)
print(targets)
```

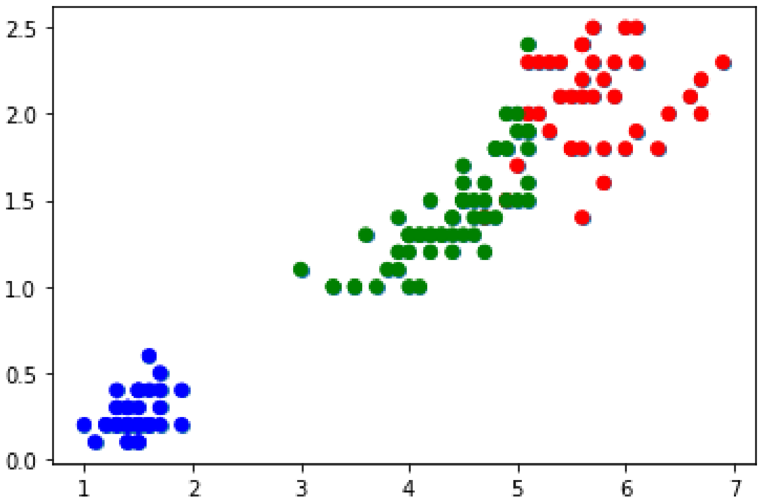
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1]

Visualiser les résultats : ***targets*** est le vecteur des ***labels*** obtenue par notre méthode :

```
X=pd.DataFrame(iris.data)
X.columns=['Sepal_Length','Sepal_width','Petal_Length','Petal_width']
```

```
[ ] plt.scatter(X.Petal_Length, X.Petal_width)
    colormap=np.array(['Red','green','blue'])
    plt.scatter(X.Petal_Length, X.Petal_width,c=colormap[targets],s=40)
```

Les résultats obtenus par notre méthode sont favorables, un taux d'erreur raisonnable, mais il reste moins efficace que k-mean de sk-learn :



3. Utiliser l'indice de Silhouette (qui est dans le package *sklearn*) pour stabiliser les résultats et sélectionner automatiquement le nombre de groupes. Pour ce faire, créez un script qui applique K-moyenne sur les données pour différents nombres de clusters allant de 2 à 10, 10 fois pour chaque nombre de clusters (soit 90 fois en tout) et qui renvoie la solution ayant le meilleur score de Silhouette.

Un indicateur de la qualité de Clustering est donné par le coefficient de silhouette, qui implique une combinaison de cohésion et de séparation entre les données**.**

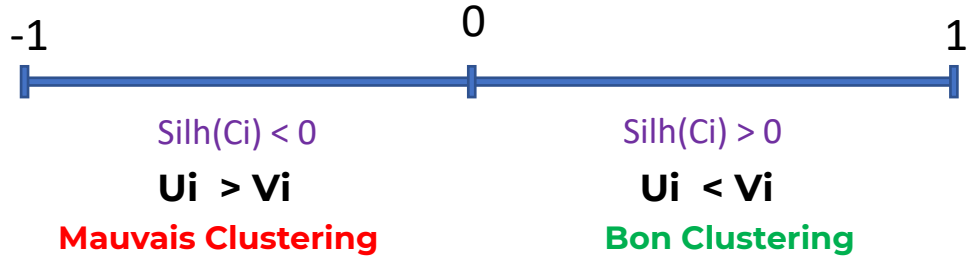
Soit : **ui :** La distance moyenne de xi de toutes les autres observations appartenant au même cluster.

Vi: Le minimum des distances wif (Pour chaque cluster Cf autre que le cluster auquel xi appartient, le la distance moyenne wif entre xi et toutes les observations dans Cf).

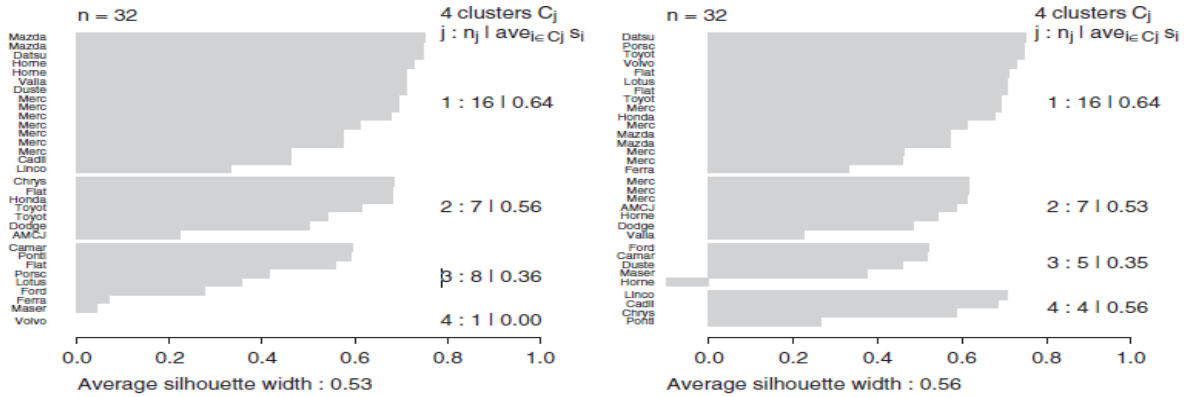
Le coefficient de silhouette de xi est défini par :

$$\text{silh}(\mathbf{x}_i) = \frac{v_i - u_i}{\max(u_i, v_i)}.$$

Le coefficient de silhouette global d'un clustering est la moyenne des coefficients de silhouette de toutes les observations dans l'ensemble de données.



Le diagramme de silhouette : les observations sont placées sur l'axe vertical, et les valeurs du coefficient de silhouette sont indiquées sur l'axe horizontal.



Méthode pour obtenir le meilleur score par l'indice de silhouette :

```python
def get_meilleur_score(X):
    scores = []
    for i in range(2, 11):
        for j in range(10):
            kmeans = cluster.KMeans(n_clusters=i).fit(X)
            pred = kmeans.fit_predict(x)
            score = metrics.silhouette_score(X, pred)
            scores.append((j+1,i,score))
    scores.sort(key = lambda x:x[2], reverse=True)
    return scores[0]
```

```python
[ ] (iterations, n_clusters, score) = get_meilleur_score(x)
```

```python
[ ] print(f"le meilleur score est : {score} \n le nombre des clusters est: {n_clusters}
```

```
le meilleur score est : 0.681046169211746
 le nombre des clusters est: 2
  numero d'iteration  1
```

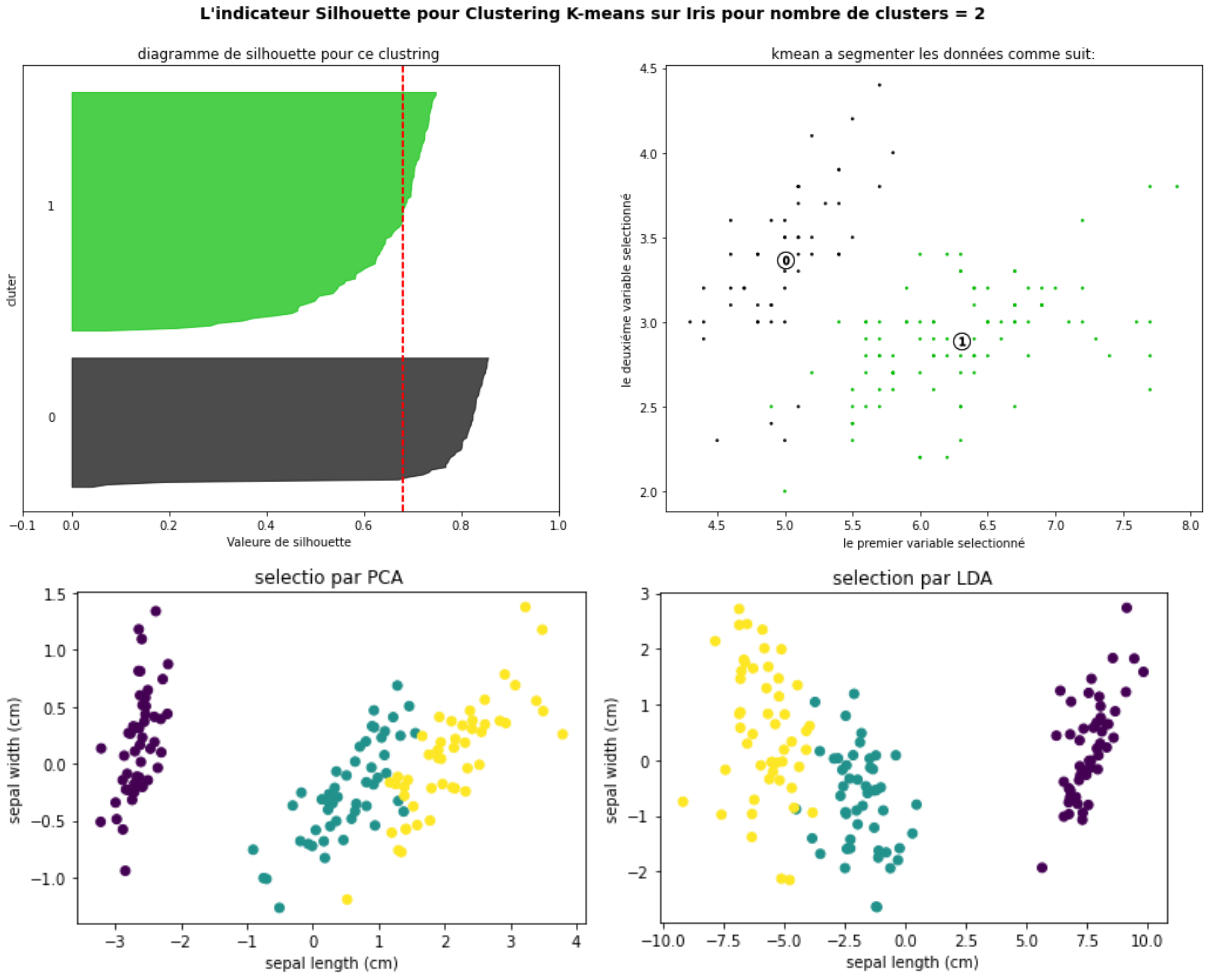
4. Utiliser une ACP (fonction *PCA* de *sklearn*) pour vérifier visuellement la cohérence des groupes obtenus. Vérifier aussi visuellement la séparabilité et la compacité de ces groupes à l'aide d'une ADL (fonction *LinearDiscriminantAnalysis* de *sklearn)*. Quelle est la différence entre les deux méthodes ?

```python
range_n_clusters = [2, 3, 4]
for n_clusters in range_n_clusters:
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)
    ax1.set_xlim([-0.1, 1])
    ax1.set_ylim([0, len(X) + (n_clusters + 1) * 10])


    clusterer = KMeans(n_clusters=n_clusters, random_state=10)
    cluster_labels = clusterer.fit_predict(X)
    silhouette_avg = silhouette_score(X, cluster_labels)
    print("pour lr nombre de clusters =", n_clusters,"l'indice silhouette est :", silhouette_avg)

    # calculer silhouette pour chaque lement
    sample_silhouette_values = silhouette_samples(X, cluster_labels)
```

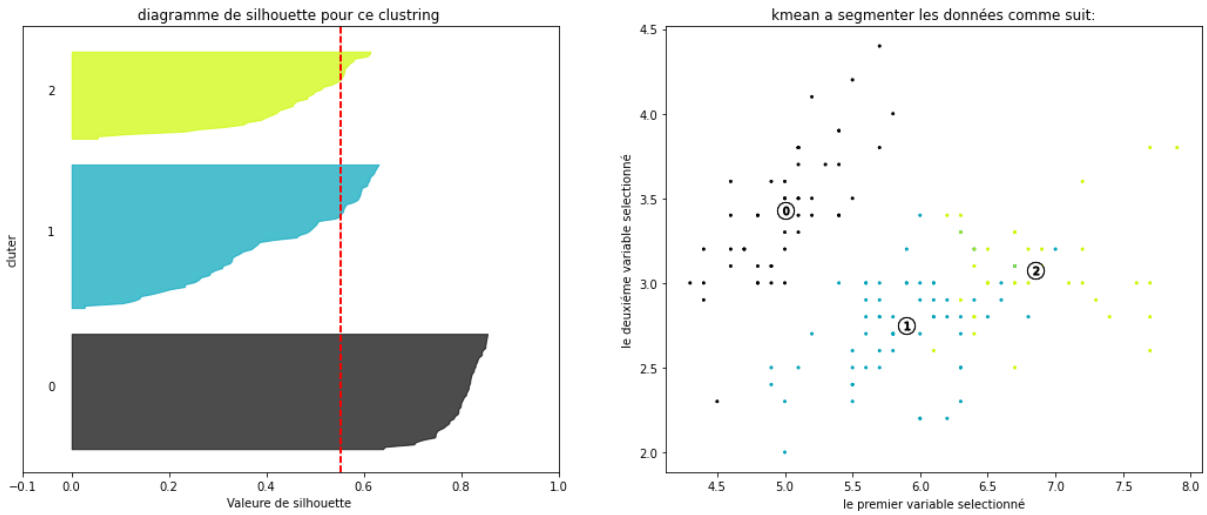
```python
#ploting tout les visualisations pca,lda et silhouette diagramme
    plt.show()
    pca = PCA(n_components=2)
    IrisPCA = pca.fit(X).transform(X)
    plt.title('selectio par PCA ')
    plt.scatter(IrisPCA[:, 0], IrisPCA[:, 1], c=iris.target, cmap='viridis')
    plt.xlabel(iris.feature_names[0])
```

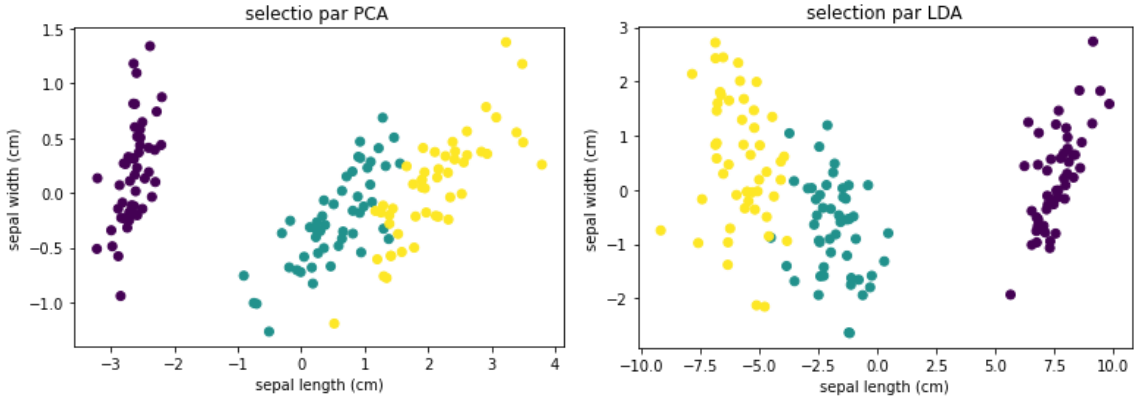
Les résultats sont comme suit :

pour le nombre de clusters = 2 l'indice silhouette est : 0.681046169211746

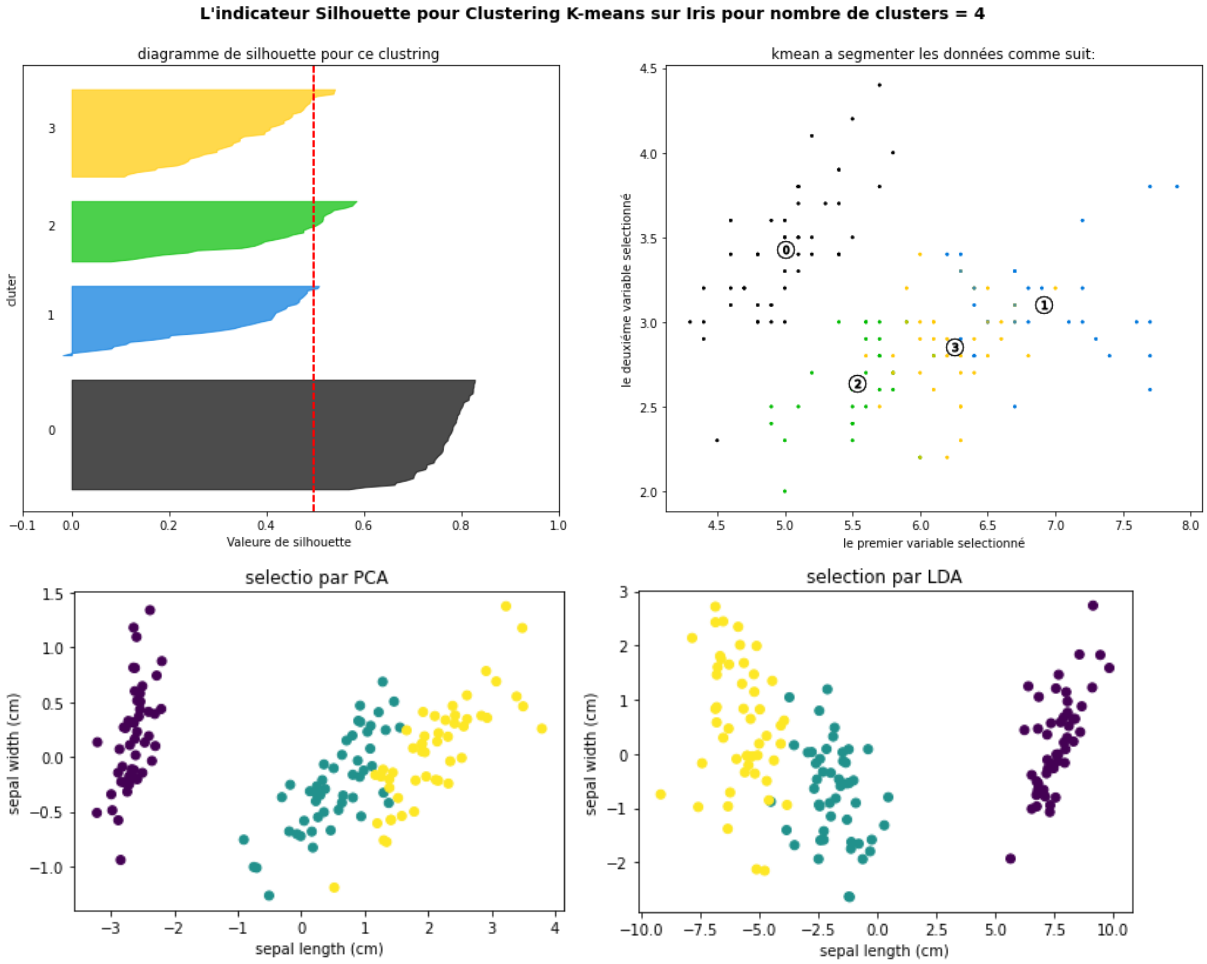


pour le nombre de clusters = 3 l'indice silhouette est : 0.5528190123564091





pour le nombre de clusters = 4
l'indice silhouette est : 0.4980505049972867



Le meilleur coefficient de silhouette est obtenu est **0.68** pour une clustering à **deux groupes**, en peut expliquer ces résultats par la distribution de nuage des points dans l'espace des caractéristiques, car dans la visualisation des données ***iris***, nous voir deux groupes bien **homogènes** bien **séparées** même si en réalité le deuxième groupe regroupe deux types des fleurs.

5.2 Analyse des données « choix projet »

Les données ***choixprojetstab.csv*** contiennent des données anonymisées de vœux faits par les étudiants du master 2 pour des projets à réaliser en binômes. Chaque étudiant a fait des vœux sur les différents projets en leur donnant une mention très bien, bien ou jamais. Les projets non mentionnés peuvent être considérés comme « moyens » et on peut dire que très bien = 3, bien = 2, moyens = 1, jamais = 0. L'objectif est de former des clusters d'étudiants ayant faits des choix similaires.

1. Utilisez le package *csv* (ou l'importation de variable de Spyder) pour lire le fichier et remplir deux variables : la liste des codes « C » représentant les étudiant (première colonne) et la matrice « M » des données (tout sauf la première ligne et la première colonne). La matrice M doit être de type *array* du package *numpy*. Faites attentions à ce que les valeurs dans M soient bien numériques (1, 2, 3) et non textuelle ('1', '2', '3'). Vous pouvez utiliser la méthode *astype* de *numpy* en cas de besoin.

Lire le fichier *choixprojetstab.csv* dans le daframe data :

```
from google.colab import drive
import pandas as pd
drive.mount('/content/drive')
data = pd.read_csv('/content/drive/My Drive/DataMining/choixprojetstab.csv', sep=';')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount(
```

Le fichier contient le vote de **71** étudiants sur **81** sujets de projet :

```
data.head
```

```
<bound method NDFrame.head of        étudiant·e  ga.vTZVmBFaC.  ...
0   bl/.vSDYCGrSs              1  ...              1              3
1   bl/1NiMubceBs              1  ...              1              1
2   bl/dvgMTLVSvk              0  ...              0              0
3   bl1NWhKcNADF2              1  ...              1              1
4   bl1ao5B7htJfQ              1  ...              0              2
..            ...            ...  ...            ...            ...
66  blvqKhCDhQab.              1  ...              1              1
67  blwyn40XjgvEs              1  ...              1              0
68  bly3Oqsvf6WlU              1  ...              0              2
69  bly69tf4z1FiM              1  ...              1              1
70  blzon470EHguA              1  ...              1              1

[71 rows x 81 columns]>
```

Isoler la première colonne pour qu'il soit notre vecteur des labels :

```
[ ]  targety=data['étudiant·e']
```

```
[ ]  targety.head
```

```
<bound method NDFrame.head of 0     bl/.vSDYCGrSs
1      bl/1NiMubceBs
2      bl/dvgMTLVSvk
3      bl1NWhKcNADF2
4      bl1ao5B7htJfQ
           ...
66     blvqKhCDhQab.
67     blwyn40XjgvEs
68     bly3Oqsvf6WlU
69     bly69tf4z1FiM
70     blzon470EHguA
Name: étudiant·e, Length: 71, dtype: object>
```

On garde dans le dataframe **data** que les votes :

```
data.drop(columns=data.columns[0],
          axis=1,
          inplace=True)
```

```
[ ]  datax=data
     datax.head
```

```
<bound method NDFrame.head of      ga.vTZVmBFaC.  ga/mLSm4ai/6g  ...  g
0                1              1  ...              1              3
1                1              1  ...              1              1
2                0              0  ...              0              0
3                1              1  ...              1              1
4                1              1  ...              0              2
..             ...            ...  ...            ...            ...
66               1              1  ...              1              1
67               1              1  ...              1              0
68               1              1  ...              0              2
69               1              1  ...              1              1
70               1              1  ...              1              1

[71 rows x 80 columns]>
```

Transformation des données **data** en objet array de numpy :

```
[ ]  datax=np.array(datax)
```

```
     print(datax)
```

```
[[1 1 1 ... 1 1 3]
 [1 1 1 ... 1 1 1]
 [0 0 0 ... 1 0 0]
 ...
 [1 1 1 ... 1 0 2]
 [1 1 0 ... 1 1 1]
 [1 1 1 ... 0 1 1]]
```

```
[ ]  print(type(datax))
     print(type(targety))
     print(len(datax))
     print(len(targety))
```

```
<class 'numpy.ndarray'>
<class 'pandas.core.series.Series'>
71
71
```

2. Dans *sklearn.cluster* il existe différents algorithmes de clustering. Testez les différents algorithmes du package et proposez le meilleur clustering possible des données selon l'indice Silhouette.

Le clustering K-means:

```
[ ]  from sklearn.cluster import KMeans
     kmeans = KMeans(n_clusters=4).fit(datax)
     kmeans_pred = kmeans.labels_
     print(len(kmeans_pred))
     print(kmeans_pred)
```

```
71
[1 0 3 0 3 0 0 0 1 0 0 2 3 0 0 0 3 0 3 0 0 0 3 1 0 0 0 0 0 0 0 3 0 2 0 3 0
 3 0 0 2 0 0 0 1 3 0 0 0 0 0 0 3 0 0 3 0 0 0 0 3 0 1 0 2 3 0 0 0 0 0]
```

```
[ ]  from sklearn.metrics import silhouette_score
     silhouette_avg_kmeans = silhouette_score(datax, kmeans_pred)
```

```
[23] print("l'indice sihouette pour ce clustering est : ",silhouette_avg_kmeans)
```

```
l'indice sihouette pour ce clustering est :  0.29043701764448937
```

Le clustering hiérarchique ascendant:

```
from sklearn.cluster import AgglomerativeClustering
ward = AgglomerativeClustering(n_clusters=4, linkage="ward").fit(datax)
label = ward.labels_
```

```
print(label)
```

```
[3 2 1 2 1 0 0 2 3 2 2 0 1 2 2 2 1 2 1 2 2 0 1 3 2 2 2 2 2 2 2 1 2 0 2 1 2
 1 0 2 0 2 2 2 3 1 2 2 2 2 2 2 1 2 2 1 2 2 2 2 1 2 0 2 0 1 2 2 0 2 2]
```

```
[26] from sklearn.metrics import silhouette_score
     silhouette_avg_herarchi = silhouette_score(datax, labels)
```

```
[27] print("l'indice sihouette pour ce clustering est : ",silhouette_avg_herarchi)
```

```
l'indice sihouette pour ce clustering est :  0.2884593801353125
```

Le clustering par affinity propagation:

```
from sklearn.cluster import AffinityPropagation
# Fit Affinity Propagation with Scikit learn
afprop = AffinityPropagation(max_iter=800)
afprop.fit(datax)
cluster_centers_indices = afprop.cluster_centers_indices_
n_clusters_ = len(cluster_centers_indices)
# Predict the cluster for all the samples
P = afprop.predict(datax)
print(P)
```

```
[10  3  5  3  0  1  2  3 10  3  3  7  4  3  3  3  5 11  0  3  3  2  6 10
  3  3  3  3  3  3  3  8  3  7  3  8  3  5  3  3  9  3 11  3 10  4  3 11
  3  3  3  3  5  3  3  5  3  3  3  3  5  3 12  3  9  6  3  3  3  3  3]
```

```
[30] from sklearn.metrics import silhouette_score
     silhouette_avg_aff_p = silhouette_score(datax, P)
```

```
[31] print("l'indice sihouette pour ce clustering est : ",silhouette_avg_herarchi)
```

```
l'indice sihouette pour ce clustering est :  0.2884593801353125
```

Optic clustering:

```
from sklearn.cluster import OPTICS
from numpy import *
from matplotlib import pyplot
# define the model
model = OPTICS(eps=0.9, min_samples=4)
# fit model and predict clusters
model.fit(datax)
lanels = model.fit_predict(datax)
print(labels)
```

```
[3 2 1 2 1 0 0 2 3 2 2 0 1 2 2 2 1 2 1 2 2 0 1 3 2 2 2 2 2 2 2 1 2 0 2 1 2
 1 0 2 0 2 2 2 3 1 2 2 2 2 2 2 1 2 2 1 2 2 2 2 2 1 2 0 2 0 1 2 2 0 2 2]
```

```
[35] from sklearn.metrics import silhouette_score
     silhouette_avg_optic = silhouette_score(datax, labels)
```

```
[37] print("l'indice sihouette pour ce clustering est : ",silhouette_avg_optic)
```

```
l'indice sihouette pour ce clustering est :  0.2884593801353125
```

On constate que les résultats de clustering sont pas de très bon niveau d'après l'indice silhouette, et nous pouvons expliquer ces résultats par la déférence entre

les méthodes de clustering, chaque méthode est utile pour un type de données, et pour des besoins précis (données cycliques, données de distribution normale…).

Le Quatrième TP est fini ici, le but de cette partie était de faire quelques taches d'apprentissage non supervisé en implémentant des algorithmes de clustering : ***K-means***.et utiliser d'autre algorithme de ce genre, comme on a vu quelques mesures pour l'évaluation de clustering.
Vous trouverez le code complet de **TP N° 4** dans mon notebook Google-colab suivant :
https://colab.research.google.com/drive/10npy3rfDz4TzS5a67KEUx-YkSH3fgzSo?usp=sharing

6. TP5 : Descente de gradient

Le gradient descente est un algorithme permettant de trouver le minimum local d'une fonction différentiable. La descente de gradient est simplement utilisée pour trouver des valeurs aux paramètres d'une fonction permettant d'atteindre ce minimum local.
Pour être plus précis, la descente de gradient est un moyen de minimiser une fonction objectif **J(θ)** en mettant à jour les paramètres dans le sens inverse du gradient de la fonction objectif.
On commence avec des valeurs initiales (aléatoires), puis on calcule la fonction de mise à jour avec ces valeurs.
On répète l'opération jusqu'à trouver les valeurs qui minimisent la fonction de coût.

$$\theta := \theta - \alpha \frac{\partial}{\partial \theta} J(\theta)$$

Où **θ** est le paramètre de la fonction à minimiser, **α** le taux d'apprentissage et **J(θ)** est la fonction à minimiser.

Ce TP présente ce qu'est une Descente de Gradient (DG). La DG s'applique lorsque l'on cherche le minimum d'une fonction dont on connaît l'expression analytique, qui est dérivable, mais dont le calcul direct du minimum est difficile. C'est un algorithme fondamental à connaître car utilisé partout sous des formes dérivées. Nous n'étudions ici que la version de base.

6.1 Descente de gradient

Cette partie présente la DG sur la minimisation de la fonction E(x) quelconque. Le problème est de trouver la valeur de x qui minimise E(x). Pour trouver analytiquement le minimum de la fonction E, il faut trouver les racines de l'équation E0(x) = 0, donc trouver ici les racines d'un polynôme de degré 3, ce qui est des fois "difficile". Donc on va utiliser la DG. La DG consiste à construire une suite de valeurs xi (avec x0 fixé au hasard) de manière itérative :

$$x_{i+1} = x_i - \eta E'(x_i).$$

On peut donner un critère de fin à la DG par exemple si $x_{i+1} - x_i < \epsilon$ ou si $i > nombre_{max}$. Pour ce problème, nous utilisons $\epsilon = 0.01$ et $nombre_{max} = 1000$.

1. Calculez l'expression analytique de la fonction $E(x) = (x-1)(x-2)(x-3)(x-5)$ et sa dérivée.

```
[2]  # E(x) fonction à minimiser
     def E(x):
       return x**4 - 11*(x**3) + 41*x*x - 61*x + 30 #défacteurisation de (x-1)*(x-2)*(x-3)*(x-5)
     #derivé de E(x)
     def dE(x):
       return 4*(x**3) - 33*x*x + 82*x - 61
```

test de fonction pou x=0 set x=1

```
print(f"E(0) = {E(0)},     E(1) = {E(1)}")
print(f"dE(0) = {dE(0)},   dE(1) = {dE(1)}")
```

```
E(0) = 30,     E(1) = 0
dE(0) = -61,   dE(1) = -8
```

2. Implémentez l'algorithme DG sous Python pour la fonction $E(x)$.

```
def DG(x0, pas, e = 0.01, max_iter = 1000):
    x = x0
    for i in range(max_iter):
        temp = x
        x = temp - (pas*dE(temp))
        if abs(x - temp) < e:
            break
    return {"    min" : round(x, 3), "   nbr_iteration" : i+1, "    E(min)" : E(x)}
```

3. Pour comprendre ce que fait effectivement la DG, testez l'algorithme implémenté en utilisant des exemples d'exécutions avec des valeurs initiales de x_0 et η suivantes :
   (a) $x_0 = 5$ et $\eta = 0.001$
   (b) $x_0 = 5$ et $\eta = 0.01$
   (c) $x_0 = 5$ et $\eta = 0.1$
   (d) $x_0 = 5$ et $\eta = 0.17$
   (e) $x_0 = 5$ et $\eta = 1$
   (f) $x_0 = 0$ et $\eta = 0.001$

```
print("pour : x0=5 et pas=0.001", DG(x0=5, pas=0.001))
print("pour : x0=5 et pas=0.01 ",DG(x0=5, pas=0.01))
print("pour : x0=5 et pas=0.1  ",DG(x0=5, pas=0.1))
print("pour : x0=5 et pas=0.17 ",DG(x0=5, pas=0.17))
print("pour : x0=0 et pas=0.001",DG(x0=0, pas=0.001))
```

```
pour : x0=5 et pas=0.001 {'    min': 4.661, '   nbr_iteration': 22, '    E(min)': -5.480963002094541}
pour : x0=5 et pas=0.01  {'    min': 4.361, '   nbr_iteration': 10, '    E(min)': -6.9011757851308175}
pour : x0=5 et pas=0.1   {'    min': 4.105, '   nbr_iteration': 1000, '    E(min)': -6.463930507818617}
pour : x0=5 et pas=0.17  {'    min': 1.658, '   nbr_iteration': 1000, '    E(min)': -1.0083047800537628}
pour : x0=0 et pas=0.001 {'    min': 0.949, '   nbr_iteration': 39, '    E(min)': 0.44149105986255677}
```

Pour le cas **(e)** la descente de gradient ne converge pas car le **pas = 1** est très grande pour un intervalle de convection trop petit, ce qui couse une oscillation entre des point loin de minimum de fonction

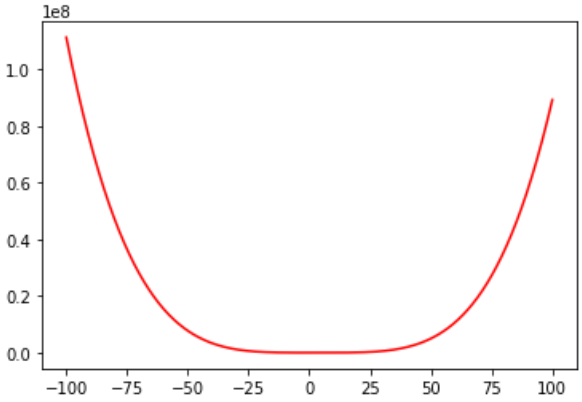
pour le pas=1 on remarque que l'algorithme prend beaucoup de temps pour le calcule et en fin elle diverge

```
[ ]  print("pour : x0=5 et pas=1   ",DG(x0=5, pas=1))
```

5. Visualisez l'évolution des minimums de la fonction $E(x)$ trouvés au cours des itérations.

```
from matplotlib.ticker import EngFormatter
from scipy import misc

import matplotlib.pyplot as plt
import numpy as np
# Plot function
x = np.arange(-100, 100, 0.01)      #la visualisation des point visité pondant la rechrche de
y = E(x)                            # car la fonction est casi constante dans grandes interva
                                   # plus facile a decouvrir
plt.plot(x, y,'r-')
plt.show()
```

Pour ce cas la fonction est casi constante dans un grande intervalle les points visités par le décent de gardien sont plus proche d'une zone où la fonction semble stable donc c'est une mauvaise fonction pour donner une visualisation des itérations de descente de gradient.



Pour bien visualiser les itérations de descente de gradient en vas utiliser la fonction convexe suivante :

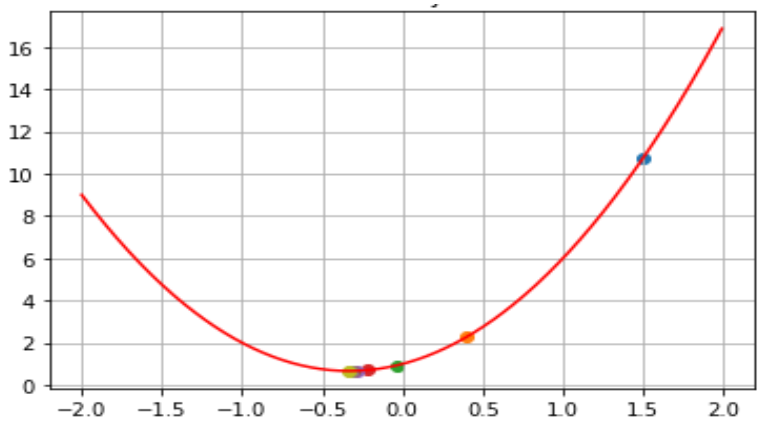
$$E(x)=3\ x^2 +2x +1$$

Avec :

```
alpha = 0.1 #le pas
nb_max_iter = 100 # Nb max d'itération
eps = 0.0001 # stop condition
x0 = 1.5 # point de depart
```

Et voici les coordonnées des point visitées par la descente de gradient et la représentation graphique :

```
            X                   Y           epsilon
0.3999999999999999 2.2799999999999994 8.47
-0.040000000000000036 0.9248 1.3551999999999995
-0.216 0.707968 0.2168319999999999
-0.2864 0.67327488 0.03469312000000002
-0.31456 0.6677239808 0.00555089920000007
-0.32582400000000006 0.666835836928 0.0008881438719999801
-0.33032959999999995 0.66669373390848 0.00014210301952000126
-0.3321318399999999 0.6666709974253568 2.2736483123120266e-05
```



6. Testez votre algorithme avec d'autres valeurs de ϵ et $nombre_{max}$.

Pour epsilon= **0.1** et un maximum d'itérations de **1500** itérations :

```
max_iter=1500
epsilon=0.1
```

```
print("pour : x0=5 et pas=0.001", DG(x0=5, pas=0.001,e=epsilon,max_iter=max_iter))
print("pour : x0=5 et pas=0.01 ",DG(x0=5, pas=0.01,e=epsilon,max_iter=max_iter))
print("pour : x0=5 et pas=0.1  ",DG(x0=5, pas=0.1,e=epsilon,max_iter=max_iter))
print("pour : x0=5 et pas=0.17 ",DG(x0=5, pas=0.17,e=epsilon,max_iter=max_iter))
print("pour : x0=0 et pas=0.001",DG(x0=0, pas=0.001,e=epsilon,max_iter=max_iter))
```

```
pour : x0=5 et pas=0.001 {'    min': 4.976, '   nbr_iteration': 1, '    E(min)': -0.5611480842240439}
pour : x0=5 et pas=0.01  {'    min': 4.547, '   nbr_iteration': 3, '    E(min)': -6.329232464124345}
pour : x0=5 et pas=0.1   {'    min': 2.658, '   nbr_iteration': 2, '    E(min)': 0.8742514433457984}
pour : x0=5 et pas=0.17  {'    min': 1.658, '   nbr_iteration': 1500, '    E(min)': -1.0083047800537628}
pour : x0=0 et pas=0.001 {'    min': 0.061, '   nbr_iteration': 1, '    E(min)': 26.429078054841}
```

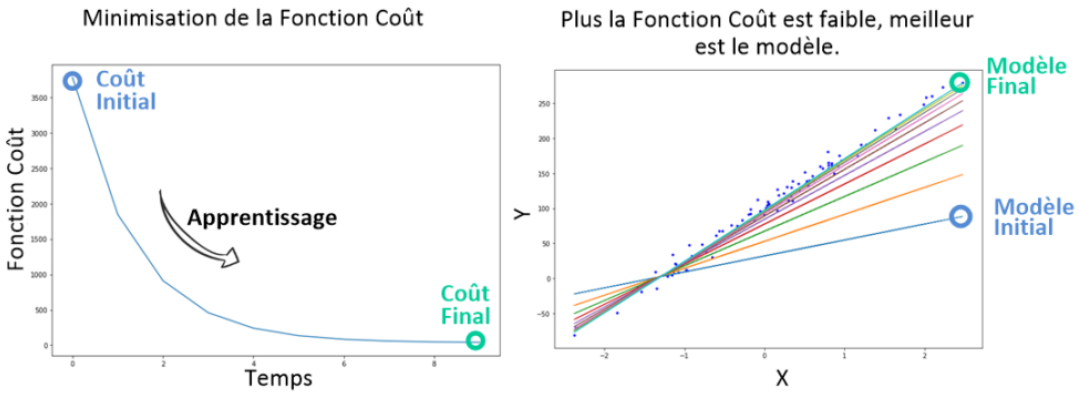
Pour epsilon= **0.1** et un maximum d'itérations de **1500** itérations :

6.2 Descente de gradient pour la régression linéaire

Le principe de la régression linéaire est de modéliser une variable dépendante quantitative Y, au travers d'une combinaison linéaire de p variables explicatives quantitatives, X_1, X_2, ..., X_p. Le modèle déterministe (ne prenant pas en compte d'aléa) s'écrit pour une observation i :

$$y_i = a_1x_{1i} + a_2x_{2i} + \ldots + a_px_{pi} + e_i$$

Pour trouver les a_i optimales qui sépare bien les données, on doit minimiser l'erreur quadratique moyenne : $J(a,b) = \frac{1}{2m}\sum_{i=1}^{m}(ax^{(i)} + b - y^{(i)})^2$ et pour cela on doit appliquer la descente de gradient.



Dans ce cas, la fonction de coût à minimiser est la suivante :

$$E(a,b) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2.$$

```
[ ]  def E(a,b,x,y):
       return sum([(a*x[i]+b)-y[i] for i in range(1, len(x))])
```

1. Calculez les dérivées partielles de la fonction $E(a,b)$ selon a et b.

```
[ ]  nb_sample_data = x.shape[0]
     rg_sample = range(1, nb_sample_data)
     grad_a = sum([2*(a*x[j]**2+b*x[j]-y[j]*x[j]) for j in rg_sample])
     grad_b = sum([2*(a*x[j]+b-y[j]) for j in rg_sample])
```

2. Implémentez l'algorithme DG sous Python pour la fonction $E(a,b)$. NB : Pour le cas avec deux dimensions, les règles de mis à jour de la DG deviennent :

$$a_{i+1} = a_i - \eta \frac{\partial E(a,b)}{\partial a}(a_i),$$

$$b_{i+1} = b_i - \eta \frac{\partial E(a,b)}{\partial b}(b_i).$$

```
import numpy as np
def DG_reg(max_iter, e, n, x, y):
    nb_sample_data = x.shape[0]
    rg_sample = range(1, nb_sample_data)
    a = np.random.random(x.shape[1])[0]
    b = np.random.random(x.shape[1])[0]
    Eab = E(a,b,x,y)
    for i in range(max_iter):
        #Calcule du gradient
        grad_a = sum([2*(a*x[j]**2+b*x[j]-y[j]*x[j]) for j in rg_sample])
        grad_b = sum([2*(a*x[j]+b-y[j]) for j in rg_sample])
        #Descente de gradient
        prev_a = a - n * grad_a[0]
        prev_b = b - n * grad_b[0]
        a = prev_a
        b = prev_b
        #Calcule de l'erreur
        err = sum([a*x[j]+b-y[j] for j in rg_sample])
        if abs(Eab-err) < e: break
        Eab = err
    return {'a': a, 'b': b, 'n_iter': i+1, "E(amin, bmin)": E(a,b,x,y)}
```

3. Importez la fonction *datasets.make_regression* et utilisez la pour générez un problème de régression aléatoire de 100 example avec une seule variable. Appliquez l'algorithme implémenté au jeu de données généré avec les paramètres suivants :
   (a) $\eta = 0.001$, $nombre_{max} = 100$
   (b) $\eta = 0.001$, $nombre_{max} = 500$
   (c) $\eta = 0.001$, $nombre_{max} = 1000$
   (d) $\eta = 0.01$, $nombre_{max} = 1000$
   (e) $\eta = 1$, $nombre_{max} = 1000$

4. Affichez les coefficients trouvés, ainsi que la valeur de $E(a_{min}, b_{min})$ et le nombre d'itérations. Que constatez-vous ?

```
from sklearn.datasets import make_regression
(x,y) = make_regression(n_samples=100, n_features=1, noise=7)
```

```
print(f"n = 0.001, n_max = 100 :\n{DG_reg(max_iter = 100, e = 0.0001, n = 0.001, x=x, y=y)}")
print(f"\nn = 0.001, n_max = 500 :\n{DG_reg(max_iter = 500, e = 0.0001, n = 0.001, x=x, y=y)}")
print(f"\nn = 0.001, n_max = 1000 :\n{DG_reg(max_iter = 1000, e = 0.0001, n = 0.001, x=x, y=y)}")
print(f"\nn = 0.01, n_max = 1000 :\n{DG_reg(max_iter = 1000, e = 0.0001, n = 0.01, x=x, y=y)}")
#print(f"n = 1, n_max = 1000 ===> {DG_reg(max_iter = 1000, e = 0.0001, n = 1, x=x, y=y)}")
```

```
n = 0.001, n_max = 100 :
{'a': 76.4324397291805, 'b': -0.7370699719712466, 'n_iter': 91, 'E(amin,
bmin)': array([-0.00051189])}

n = 0.001, n_max = 500 :
{'a': 76.4324397572481, 'b': -0.7370699540422365, 'n_iter': 91, 'E(amin,
bmin)': array([-0.00051049])}

n = 0.001, n_max = 1000 :
{'a': 76.43243972632494, 'b': -0.7370699736211307, 'n_iter': 91, 'E(amin,
bmin)': array([-0.00051201])}

n = 0.01, n_max = 1000 :
{'a': -7.05194012941271e+61, 'b': 1.1070908042456394e+62, 'n_iter': 1000,
'E(amin, bmin)': array([1.1898295e+64])}
```

Pour le dernier cas ou le ***n = 1*** la fonction prendre beaucoup de temps de calcule et en fin elle diverge, en déduire que le pas de descente de gradient doit être minimale pour que la méthode converge, et ne doit pas être infiniment petit pour que la méthode converge vite.

5. Importez la fonction stats.linregress de scipy. Utilisez cette fonction pour résoudre le même problème. Comparez les résultats obtenus. Que constatez-vous ?

```
from scipy.stats import linregress

slope, intercept, r_value, p_value, std_err = linregress(x.flatten(), y)

print(f"a:{slope}    b: {intercept}")
```

```
a:76.45210351879082    b: -0.781668390718357
```

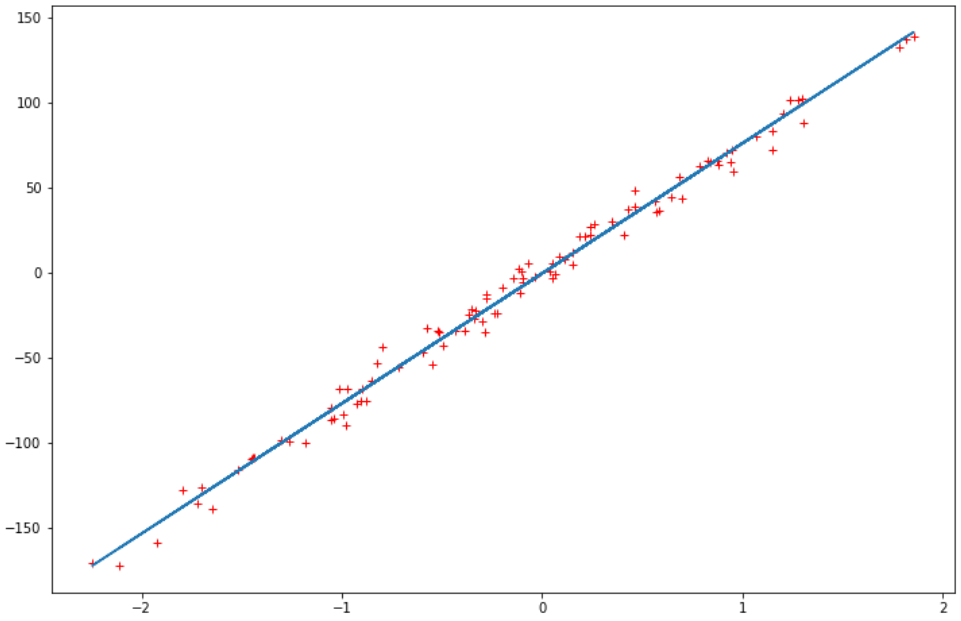
En peut remarquer que c'est le même résultat obtenu par notre méthode pour le **pas=0.001 :**

```
n = 0.001, n_max = 1000 :
{'a': 76.43243972632494, 'b': -0.7370699736211307, 'n_iter': 91,
'E(amin, bmin)': array([-0.00051201])}
```

6. Visualisez le jeu de données généré avec les fonctions approximatives obtenues en utilisant les deux méthodes.

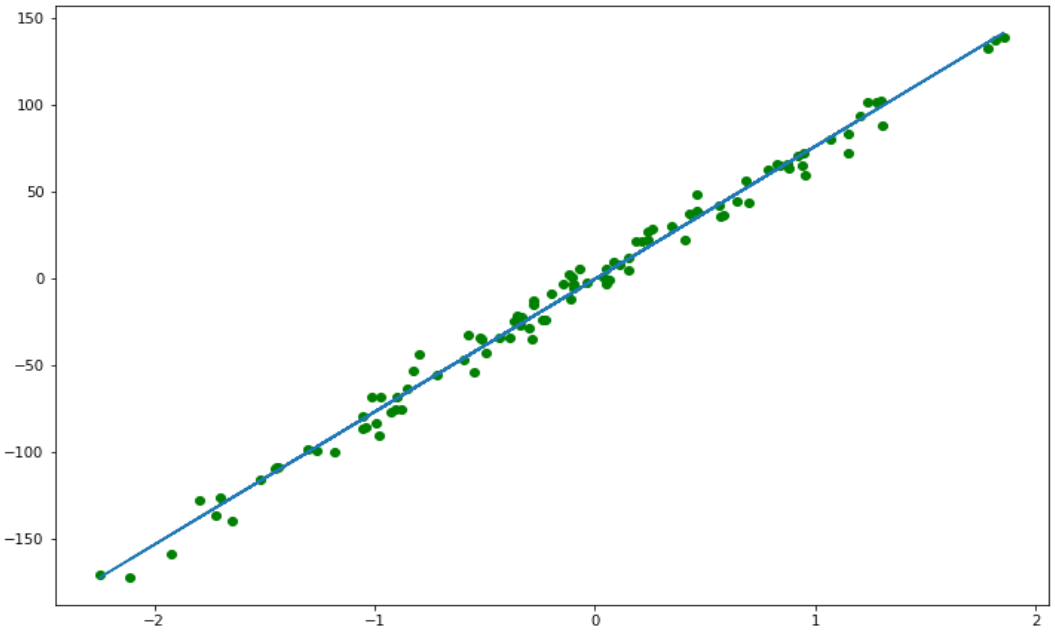
Visualisation des résultats de régression par la méthode prédéfinie dans skipy :

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12,8))
plt.plot(x,y, 'r+')
plt.plot(x, intercept+slope*x)
plt.show()
```



Visualisation des résultats de régression par notre méthode :

```
[ ]  res = DG_reg(max_iter = 100, e = 0.0001, n = 0.001, x=x, y=y)
     plt.figure(figsize=(12,8))
     plt.plot(x,y, 'go')
     plt.plot(x, res['b']+res['a']*x)
     plt.show()
```



Les résultats sont similaires donc notre fonction répondue bien au besoin.

Le Cinquième TP est fini ici, le but de cette partie était d'implémentant l'algorithmes de décente de gradient .et voir quelque utilité de cet algorithme comme dans la régression linaire.

Vous trouverez le code complet de **TP N° 5** dans mon notebook Google-colab suivant :

https://colab.research.google.com/drive/1AyMjVvck3QiugwsVU6sHyde460gW9DuJ?usp=sharing

7. TP6 : Récapitulatif et Bilan

Le but de ce TP est de créer une fonction **analyse(X,Y)** capable d'analyser un jeu de données, de comprendre ses caractéristiques et de prédire les liens entre observations et labels. Cette fonction prend en entrée une matrice de données X et une liste de labels Y. Elle devra fournir une description de chaque variable, visualiser les liens existants entre les variables puis tester et comparer différents algorithmes de classification.

7.1 Implémentation de la fonction

```
def analyse(x,y):
    #*****************  Part A ******************************
    a=x.shape
    b=unique(y).size
    c=np.array(unique(y))
    xhelp=pd.DataFrame(x)
    print("\n \n  ---------------------- start ---------------------
------ ")
    print("\n \n le nembre des données est: ",a[0])
    print("le nembre des variables est: ",a[1])
    print("les classes sont : ",unique(y))
    print("le nom de classe pour chaque donées :")
    pt = PrettyTable()
    pt.field_names = ['données','classe']
    for i in range(y.size): pt.add_row([x[i], f"{y[i]}"])
    print(pt)
    print("la moyenne pour chaque variable est : ")
    print(x.mean(0))
    print("l'ecart-type pour chaque variable est : ")
    print(x.std(0))
    print("minimum pour chaque variable est : ")
    print(x.min(0))
    print("maximum pour chaque variable est : ")
    print(x.max(0))
    print("La correlation entre les variables : ")
    print(xhelp.corr())
    sns.heatmap(xhelp.corr())
    #*****************  Part B ******************************
    print("\n \n visualisation des données à l'aide d'un ACP et ADL
: ")
    pca = PCA(n_components=2)
    lda = LDA(n_components=2)
    dPCA = pca.fit(x).transform(x)
    dLDA = lda.fit(x,y).transform(x)
```

```
    fig = plt.figure(figsize=(16,8))
    plt.subplot(121); plt.scatter(dPCA[:,0], dPCA[:,1], c=y); plt.ti
tle('selection:PCA'); plt.xlabel('X'); plt.ylabel('y')
    plt.subplot(122); plt.scatter(dLDA[:,0], dLDA[:,1], c=y); plt.ti
tle('selection:LDA'); plt.xlabel('X'); plt.ylabel('y')
    plt.show()
    print("\n \n ********* visualisation des données selon chaque co
mbinaison des paramètres")
    fig = plt.figure(figsize=(16,8))
    plt.subplot(231); plt.scatter(x[:,0], x[:,1], c=y); plt.colorbar
(); plt.xlabel('X'); plt.ylabel('y')
    plt.subplot(232); plt.scatter(x[:,0], x[:,2], c=y); plt.colorbar
(); plt.xlabel('X'); plt.ylabel('y')
    plt.subplot(233); plt.scatter(x[:,0], x[:,3], c=y); plt.colorbar
(); plt.xlabel('X'); plt.ylabel('y')
    plt.subplot(234); plt.scatter(x[:,1], x[:,2], c=y); plt.colorbar
(); plt.xlabel('X'); plt.ylabel('y')
    plt.subplot(235); plt.scatter(x[:,1], x[:,3], c=y); plt.colorbar
(); plt.xlabel('X'); plt.ylabel('y')
    plt.subplot(236); plt.scatter(x[:,2], x[:,3], c=y); plt.colorbar
(); plt.xlabel('X'); plt.ylabel('y')
    plt.show()
    print("****************** Regression liniear ********************
***")
    slope, intercept, r_value, p_value, std_err = linregress(x[:,2],
 y)
    plt.figure(figsize=(12,8))
    plt.title("linear regression")
    plt.plot(x,y, 'r+')
    plt.plot(x, intercept+slope*x)
    plt.show()
     #*****************  Part C ******************************
    X_train, X_test, y_train, y_test = model_selection.train_test_sp
lit(x, y, test_size=0.3,random_state=109)
    print("\n \n  ----------------------------------
- classification ---------------------------- ----------------------
----- ")

    print("\n \n********************** KNN *************************"
)
    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    error = 1 - sm.accuracy_score(pred, y_test)
    print(classification_report(y_test,pred ))
    print(f"\n erreur  = {round(error, 2)*100} %")

    print("\n \n********************** naive bayes ****************
*******")
```

```
    model = naive_bayes.GaussianNB()
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    error = 1 - sm.accuracy_score(pred, y_test)
    print(classification_report(y_test,pred))
    print(f"\n erreur  = {round(error, 2)*100} %")

    print("\n \n********************** SVM ************************
***")
    model = SVC()
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    error = 1 - sm.accuracy_score(pred, y_test)
    print(classification_report(y_test,pred))
    print(f"\n erreur  = {round(error, 2)*100} %")

    print("\n \n************************ MLP ***********************
*****")
    model = MLPClassifier(hidden_layer_sizes=(150,100,50), max_iter=
300,activation = 'relu',solver='adam',random_state=1)
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    error = 1 - sm.accuracy_score(pred, y_test)
    print(classification_report(y_test,pred))
    print(f"\n erreur  = {round(error, 2)*100} %")

    #*****************  Part D ******************************
    print("\n \n  ---------------------------------- Clustering --
------------------------- --------------------------- ")
    print("\n \n********************** Kmeans ********************
*****")
    model=KMeans(n_clusters=b)
    model.fit(x)
    l=model.labels_
    silhouettescore = silhouette_score(x,l)
    DB=davies_bouldin_score(x,l)
    print("silhouette score pour Hierarchical clustering est : ",sil
houettescore)
    print("davies_bouldin_score pour optic clustering est : ",DB)

    print("\n \n***************** Hierarchical clustering (Agglomera
tive) **********************")
    model = AgglomerativeClustering(n_clusters=b, linkage="ward").fi
t(x)
    labels = model.labels_
    silhouettescore = silhouette_score(x,labels)
    DB=davies_bouldin_score(x,labels)
    print("silhouette score pour Hierarchical clustering est : ",sil
houettescore)
```

```
    print("davies_bouldin_score pour optic clustering est : ",DB)

    print("\n \n ****** Optic Clustering *****************")
    model = OPTICS(eps=0.9, min_samples=b)
    model.fit(x)
    labels = model.fit_predict(x)
    silhouettescore = silhouette_score(x,labels)
    DB=davies_bouldin_score(x,labels)
    print("silhouette score pour optic clustering est : ",silhouette
score)
    print("davies_bouldin_score pour optic clustering est : ",DB)
    print("\n \n \n \n ---------------------- End ------------------
--------- ")
```

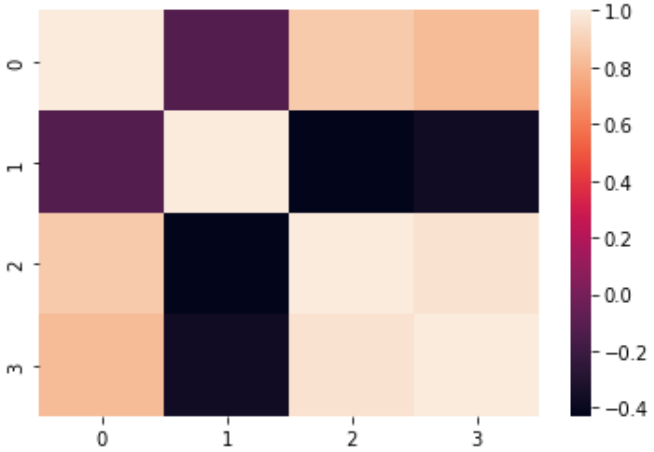
7.2 Test de la fonction sur les données « iris »

```
iris = datasets.load_iris()
x=iris.data
y=iris.target
analyse(x,y)
```

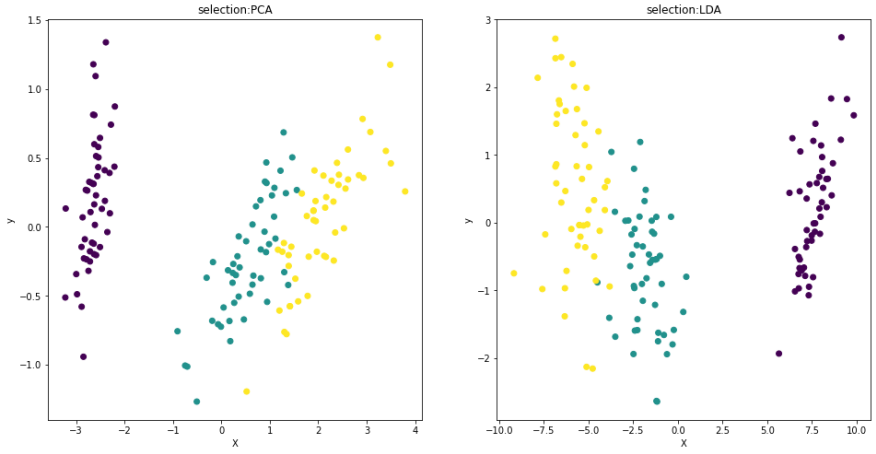
Résultat :

```
---------------------- start --------------------------


le nembre des données est:  150
le nembre des variables est:  4
les classes sont :  [0 1 2]
le nom de classe pour chaque données :
+-------------------+--------+
|      données      | classe |
+-------------------+--------+
| [5.1 3.5 1.4 0.2] |   0    |
| [4.9 3.  1.4 0.2] |   0    |
| [4.7 3.2 1.3 0.2] |   0    |
| [4.6 3.1 1.5 0.2] |   0    | # 50 données pour la classe 0
| [7.  3.2 4.7 1.4] |   1    |
| [6.4 3.2 4.5 1.5] |   1    |
| [6.9 3.1 4.9 1.5] |   1    |
| [5.5 2.3 4.  1.3] |   1    |
| [6.5 2.8 4.6 1.5] |   1    |
| [5.7 2.8 4.5 1.3] |   1    |
| [6.3 3.3 4.7 1.6] |   1    |# 50 données pour la classe 1
| [6.3 3.3 6.  2.5] |   2    |
| [5.8 2.7 5.1 1.9] |   2    |
| [7.1 3.  5.9 2.1] |   2    |
| [6.3 2.9 5.6 1.8] |   2    |# 50 données pour la classe 2
+-------------------+--------+
la moyenne pour chaque variable est :
[5.84333333 3.05733333 3.758      1.19933333]
l'ecart-type pour chaque variable est :
[0.82530129 0.43441097 1.75940407 0.75969263]
minimum pour chaque variable est :
[4.3 2.  1.  0.1]
```

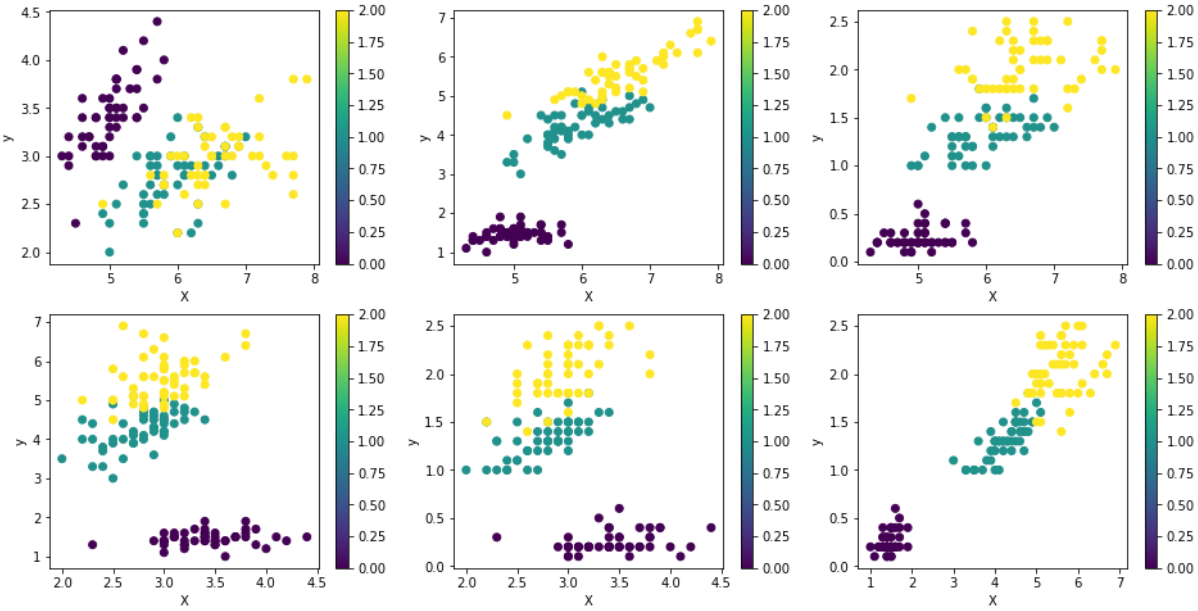
```
maximum pour chaque variable est :
[7.9 4.4 6.9 2.5]
La correlation entre les variables :
          0         1         2         3
0  1.000000 -0.117570  0.871754  0.817941
1 -0.117570  1.000000 -0.428440 -0.366126
2  0.871754 -0.428440  1.000000  0.962865
3  0.817941 -0.366126  0.962865  1.000000
```



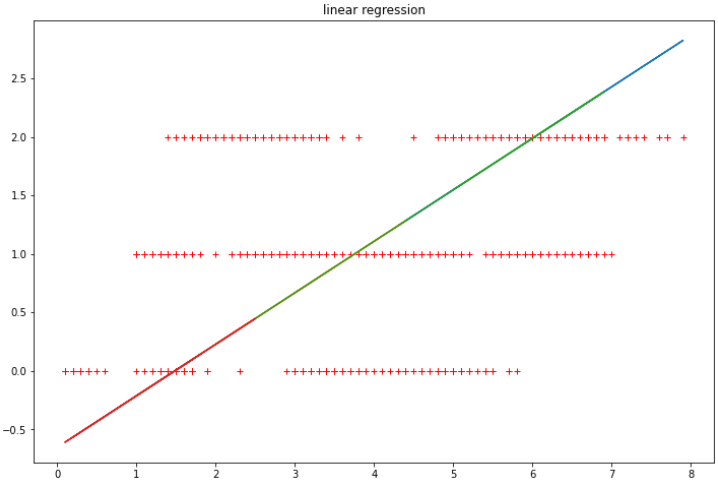
visualisation des données à l’aide d’un ACP et ADL :



********* visualisation des données selon chaque combinaison des paramètres



********** Regression liniear **********



********** KNN **********

	precision	recall	f1-score	support
0	1.00	1.00	1.00	12
1	0.93	0.82	0.87	17
2	0.83	0.94	0.88	16
accuracy			0.91	45
macro avg	0.92	0.92	0.92	45
weighted avg	0.92	0.91	0.91	45

erreur = 9.0 %

********** naive bayes **********

	precision	recall	f1-score	support
0	1.00	1.00	1.00	12
1	0.94	0.94	0.94	17
2	0.94	0.94	0.94	16

```
    accuracy                           0.96        45
   macro avg       0.96      0.96      0.96        45
weighted avg       0.96      0.96      0.96        45


 erreur  = 4.0 %


********************** SVM ***************************
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       0.84      0.94      0.89        17
           2       0.93      0.81      0.87        16

    accuracy                           0.91        45
   macro avg       0.92      0.92      0.92        45
weighted avg       0.91      0.91      0.91        45


 erreur  = 9.0 %


************************ MLP *****************************
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       1.00      0.82      0.90        17
           2       0.84      1.00      0.91        16

    accuracy                           0.93        45
   macro avg       0.95      0.94      0.94        45
weighted avg       0.94      0.93      0.93        45


 erreur  = 7.000000000000001 %


********************** Kmeans ****************************
silhouette score pour Hierarchical clustering est :  0.5528190123564091
davies_bouldin_score pour optic clustering est :  0.6619715465007511


***************** Hierarchical clustering (Agglomerative)
*********************
silhouette score pour Hierarchical clustering est :  0.5543236611296415
davies_bouldin_score pour optic clustering est :  0.6562564540642065


 ************************ Optic Clustering
*******************************
silhouette score pour optic clustering est :  -0.23849716821389966
davies_bouldin_score pour optic clustering est :  2.2901847700847484


 ---------------------- End ---------------------------
```

7.3 Test de la fonction sur les données « wine »

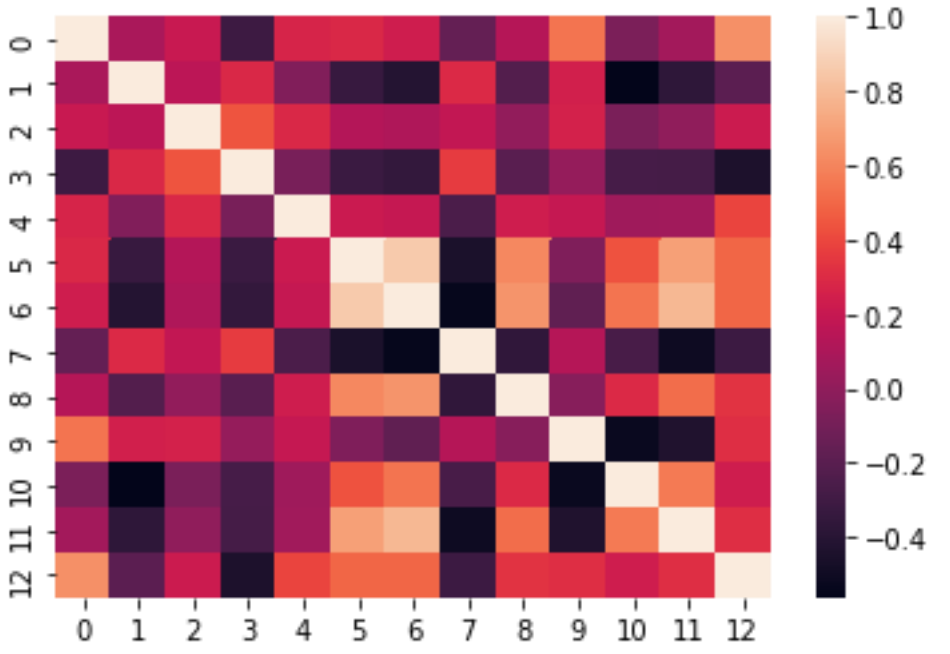
```
from sklearn.datasets import load_wine
win= load_wine()
x=win.data
y=win.target

analyse(x,y)
```

Résultat :

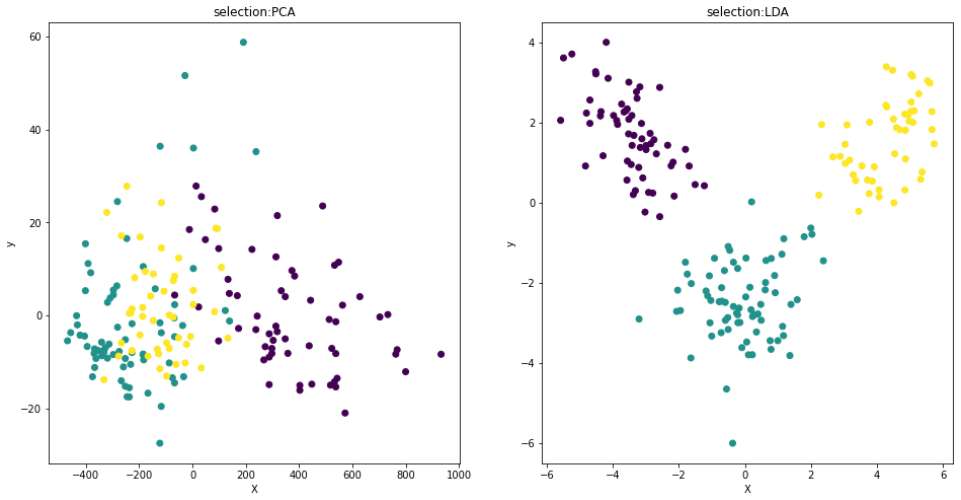
```
---------------------- start --------------------------

le nembre des données est:  178
le nembre des variables est:  13
les classes sont :  [0 1 2]
```

```
     le nom de classe pour chaque donées :
+-----------------------------------------------------------------------------+---------+
|                                 données                                     | classe  |
+-----------------------------------------------------------------------------+---------+
|  [1.423e+01 1.710e+00 2.430e+00 1.560e+01 1.270e+02 2.800e+00 3.060e+00     |    0    |
|       2.800e-01 2.290e+00 5.640e+00 1.040e+00 3.920e+00 1.065e+03]          |         |
| [1.32e+01 1.78e+00 2.14e+00 1.12e+01 1.00e+02 2.65e+00 2.76e+00 2.60e-01    |    0    |
|               1.28e+00 4.38e+00 1.05e+00 3.40e+00 1.05e+03]                 |         |
|  [1.316e+01 2.360e+00 2.670e+00 1.860e+01 1.010e+02 2.800e+00 3.240e+00     |    0 +60
|
|       3.000e-01 2.810e+00 5.680e+00 1.030e+00 3.170e+00 1.185e+03]          |         |
|  [1.237e+01 9.400e-01 1.360e+00 1.060e+01 8.800e+01 1.980e+00 5.700e-01     |    1    |
|       2.800e-01 4.200e-01 1.950e+00 1.050e+00 1.820e+00 5.200e+02]          |         |
|  [1.233e+01 1.100e+00 2.280e+00 1.600e+01 1.010e+02 2.050e+00 1.090e+00     |    1 +60
|
|       6.300e-01 4.100e-01 3.270e+00 1.250e+00 1.670e+00 6.800e+02]          |         |
|  [1.286e+01 1.350e+00 2.320e+00 1.800e+01 1.220e+02 1.510e+00 1.250e+00     |    2    |
|       2.100e-01 9.400e-01 4.100e+00 7.600e-01 1.290e+00 6.300e+02]          |         |
|  [1.288e+01 2.990e+00 2.400e+00 2.000e+01 1.040e+02 1.300e+00 1.220e+00     |    2 +60
|
|       2.400e-01 8.300e-01 5.400e+00 7.400e-01 1.420e+00 5.300e+02]          |         |
+-----------------------------------------------------------------------------+---------+
     la moyenne pour chaque variable est :
     [1.30006180e+01 2.33634831e+00 2.36651685e+00 1.94949438e+01
      9.97415730e+01 2.29511236e+00 2.02926966e+00 3.61853933e-01
      1.59089888e+00 5.05808988e+00 9.57449438e-01 2.61168539e+00
      7.46893258e+02]

     l'ecart-type pour chaque variable est :
     [8.09542915e-01 1.11400363e+00 2.73572294e-01 3.33016976e+00
      1.42423077e+01 6.24090564e-01 9.96048950e-01 1.24103260e-01
      5.70748849e-01 2.31176466e+00 2.27928607e-01 7.07993265e-01
      3.14021657e+02]
```

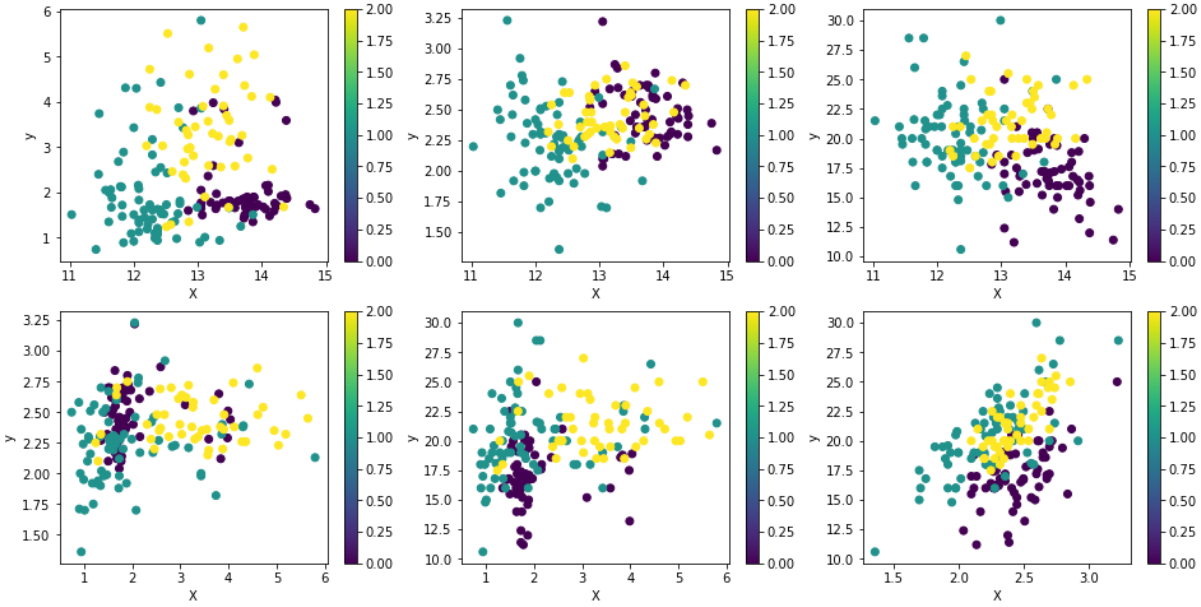
```
minimum pour chaque variable est :
[1.103e+01 7.400e-01 1.360e+00 1.060e+01 7.000e+01 9.800e-01 3.400e-01
 1.300e-01 4.100e-01 1.280e+00 4.800e-01 1.270e+00 2.780e+02]
maximum pour chaque variable est :
[1.483e+01 5.800e+00 3.230e+00 3.000e+01 1.620e+02 3.880e+00 5.080e+00
 6.600e-01 3.580e+00 1.300e+01 1.710e+00 4.000e+00 1.680e+03]


La correlation entre les variables :
           0         1         2   ...        10        11        12
0   1.000000  0.094397  0.211545  ... -0.071747  0.072343  0.643720
1   0.094397  1.000000  0.164045  ... -0.561296 -0.368710 -0.192011
2   0.211545  0.164045  1.000000  ... -0.074667  0.003911  0.223626
3  -0.310235  0.288500  0.443367  ... -0.273955 -0.276769 -0.440597
4   0.270798 -0.054575  0.286587  ...  0.055398  0.066004  0.393351
5   0.289101 -0.335167  0.128980  ...  0.433681  0.699949  0.498115
6   0.236815 -0.411007  0.115077  ...  0.543479  0.787194  0.494193
7  -0.155929  0.292977  0.186230  ... -0.262640 -0.503270 -0.311385
8   0.136698 -0.220746  0.009652  ...  0.295544  0.519067  0.330417
9   0.546364  0.248985  0.258887  ... -0.521813 -0.428815  0.316100
10 -0.071747 -0.561296 -0.074667  ...  1.000000  0.565468  0.236183
11  0.072343 -0.368710  0.003911  ...  0.565468  1.000000  0.312761
12  0.643720 -0.192011  0.223626  ...  0.236183  0.312761  1.000000

[13 rows x 13 columns]
```



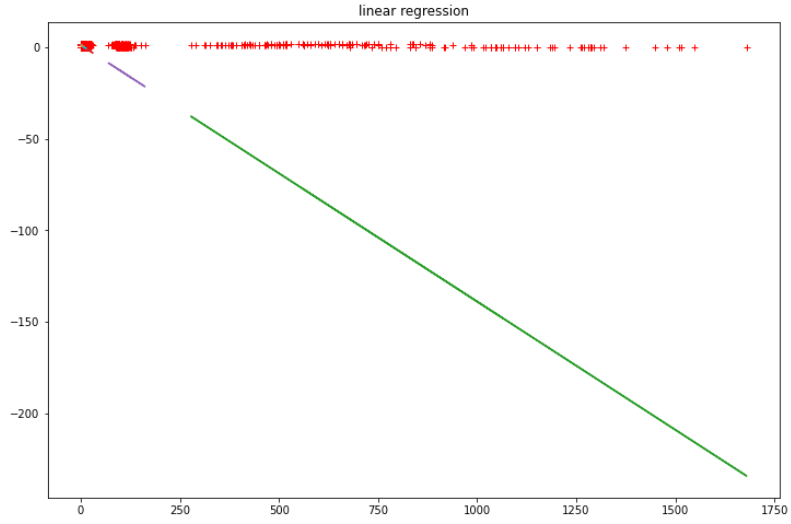
visualisation des données à l’aide d’un ACP et ADL :



********* visualisation des données selon chaque combinaison des paramètres



********** Regression liniear **********



```
  ------------------------------------ classification ----------------------
------- ---------------------------


********************** KNN ************************
              precision    recall  f1-score   support

           0       0.75      0.86      0.80        21
           1       0.55      0.58      0.56        19
           2       0.50      0.36      0.42        14

    accuracy                           0.63        54
   macro avg       0.60      0.60      0.59        54
weighted avg       0.61      0.63      0.62        54
```

erreur = 37.0 %

```
********************** naive bayes ************************
              precision    recall  f1-score   support

           0       0.91      0.95      0.93        21
           1       0.94      0.79      0.86        19
           2       0.88      1.00      0.93        14

    accuracy                           0.91        54
   macro avg       0.91      0.91      0.91        54
weighted avg       0.91      0.91      0.91        54
```

erreur = 9.0 %

```
************************ SVM ***************************
              precision    recall  f1-score   support

           0       0.85      0.81      0.83        21
           1       0.61      0.74      0.67        19
           2       0.45      0.36      0.40        14

    accuracy                           0.67        54
   macro avg       0.64      0.63      0.63        54
weighted avg       0.66      0.67      0.66        54
```

erreur = 33.0 %

```
************************ MLP ***************************
/usr/local/lib/python3.7/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:696:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (300) reached
and the optimization hasn't converged yet.
  ConvergenceWarning,
              precision    recall  f1-score   support

           0       0.87      0.95      0.91        21
           1       0.94      0.79      0.86        19
           2       0.93      1.00      0.97        14
```

```
    accuracy                           0.91        54
   macro avg       0.91      0.91      0.91        54
weighted avg       0.91      0.91      0.91        54
```

erreur = 9.0 %

```
  ---------------------------------- Clustering -------------------------
--- --------------------------
```

```
********************** Kmeans *************************
silhouette score pour Hierarchical clustering est :  0.5711381937868844
davies_bouldin_score pour optic clustering est :  0.5342431775436273
```

```
***************** Hierarchical clustering (Agglomerative)
********************
silhouette score pour Hierarchical clustering est :  0.5644796401732074
davies_bouldin_score pour optic clustering est :  0.5357343073560216
```

```
 ************************ Optic Clustering
*******************************
silhouette score pour optic clustering est :  0.12550445659052656
davies_bouldin_score pour optic clustering est :  5.40890086305377
```

```
 ---------------------- End --------------------------
```

7.4 Test de la fonction sur les données « Digits »

```
from sklearn.datasets import load_digits
digits = load_digits()
print(digits.data.shape)
x=digits.data
y=digits.target
analyse(x,y)
```

Résultat :

```
---------------------- start --------------------------
```

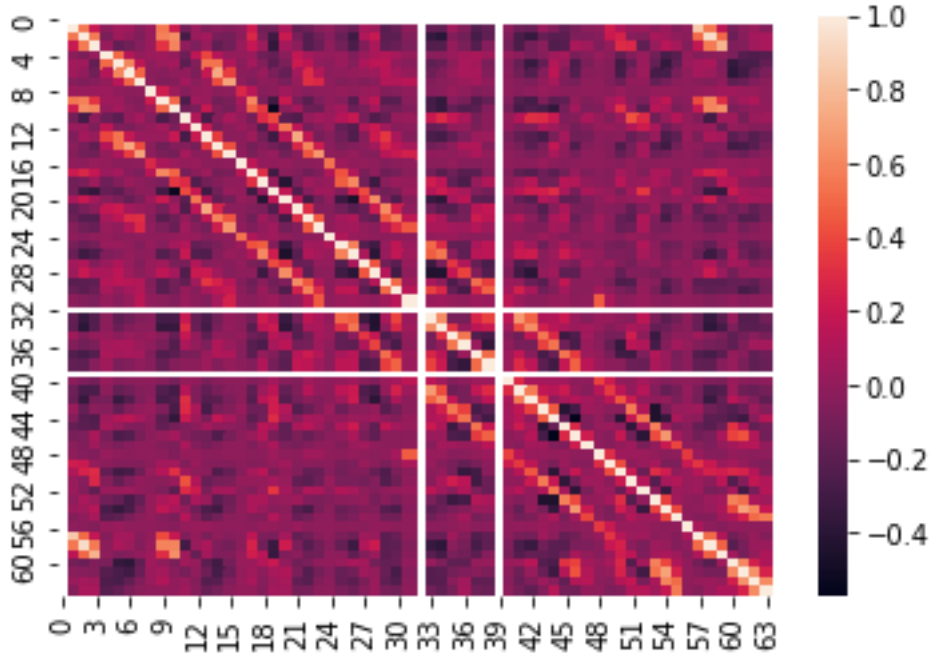
```
 le nembre des données est:  1797
le nembre des variables est:  64
les classes sont :  [0 1 2 3 4 5 6 7 8 9]
le nom de classe pour chaque donées : # 1797 lignes à afficher

la moyenne pour chaque variable est :
[0.00000000e+00 3.03839733e-01 5.20478575e+00 1.18358375e+01
 1.18480801e+01 5.78185865e+00 1.36227045e+00 1.29660545e-01
 5.56483027e-03 1.99387869e+00 1.03823038e+01 1.19794101e+01
```

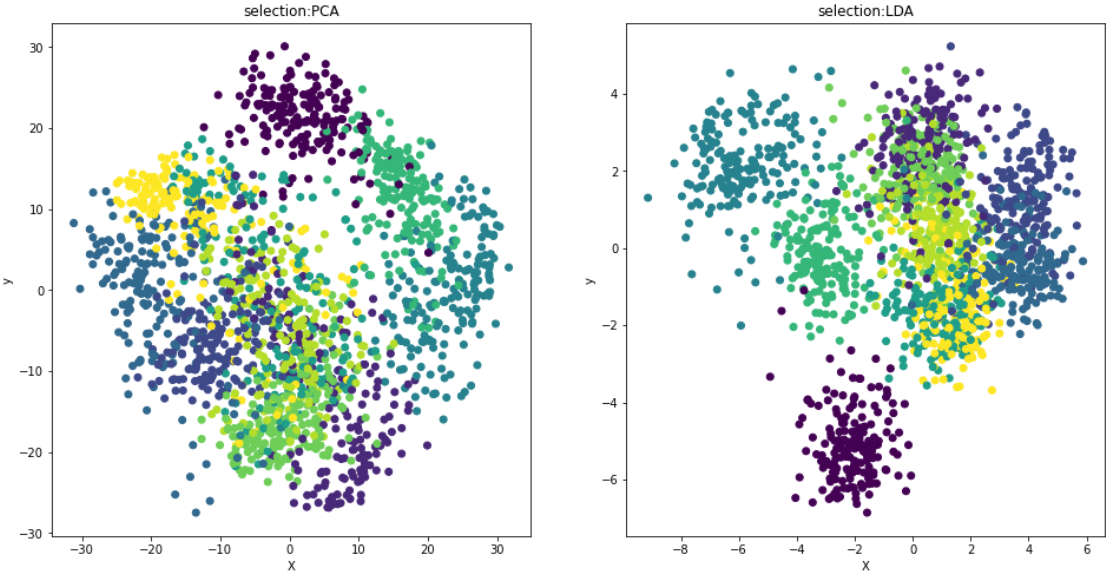
```
 1.02793545e+01 8.17584864e+00 1.84641068e+00 1.07957707e-01
 2.78241514e-03 2.60155815e+00 9.90317195e+00 6.99276572e+00
 7.09794101e+00 7.80634391e+00 1.78853645e+00 5.00834725e-02
 1.11296605e-03 2.46967168e+00 9.09126322e+00 8.82136895e+00
 9.92710072e+00 7.55147468e+00 2.31775181e+00 2.22593211e-03
 0.00000000e+00 2.33945465e+00 7.66722315e+00 9.07178631e+00
 1.03016138e+01 8.74401781e+00 2.90929327e+00 0.00000000e+00
 8.90372844e-03 1.58375070e+00 6.88146912e+00 7.22815804e+00
 7.67223150e+00 8.23650529e+00 3.45631608e+00 2.72676683e-02
 7.23427935e-03 7.04507513e-01 7.50695604e+00 9.53923205e+00
 9.41624930e+00 8.75848637e+00 3.72509738e+00 2.06455203e-01
 5.56483027e-04 2.79354480e-01 5.55759599e+00 1.20890373e+01
 1.18091263e+01 6.76405120e+00 2.06789093e+00 3.64496383e-01]

l'ecart-type pour chaque variable est :
[0.         0.90693964 4.75350317 4.24765948 4.28619491 5.66484088
 3.32484969 1.03709417 0.09419533 3.19527098 5.41994694 3.97643575
 4.78134964 6.05127561 3.58532293 0.82768465 0.06235094 3.57530605
 5.68918332 5.80104695 6.17400993 6.19559718 3.25896254 0.43847543
 0.03334258 3.14565685 6.19031469 5.88129939 6.15038083 5.87092136
 3.68543009 0.04712725 0.         3.4794038  6.32292731 6.26664682
 5.93183902 5.86901393 3.53629836 0.         0.14514503 2.98098645
 6.53613529 6.43958504 6.25776954 5.69394162 4.32974601 0.30727036
 0.20416633 1.74566694 5.64292531 5.22549314 5.30057302 6.02947606
 4.91803706 0.98412698 0.02358333 0.9340418  5.1015993  4.37347662
 4.93257433 5.89898069 4.08940957 1.85960409]

minimum pour chaque variable est :
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

maximum pour chaque variable est :
[ 0.  8. 16. 16. 16. 16. 16. 15.  2. 16. 16. 16. 16. 16. 16. 12.  2. 16.
 16. 16. 16. 16. 16.  8.  1. 15. 16. 16. 16. 16. 15.  1.  0. 14. 16. 16.
 16. 16. 14.  0.  4. 16. 16. 16. 16. 16. 16.  6.  8. 16. 16. 16. 16. 16.
 16. 13.  1.  9. 16. 16. 16. 16. 16. 16.]
La correlation entre les variables :
    0         1         2         3   ...        60        61        62
63
0  NaN       NaN       NaN       NaN  ...       NaN       NaN       NaN
NaN
1  NaN  1.000000  0.556618  0.207814  ... -0.102349 -0.029870  0.026547 -
0.043889
2  NaN  0.556618  1.000000  0.560180  ... -0.134754 -0.041183  0.072599
0.082523
3  NaN  0.207814  0.560180  1.000000  ... -0.065957 -0.054936  0.053437
0.081971
4  NaN -0.018761 -0.084235  0.023938  ... -0.082125 -0.215809 -0.250699 -
0.215349
..  ..       ...       ...       ...  ...       ...       ...       ...
...
59 NaN  0.147646  0.499840  0.767945  ...  0.058390 -0.094956  0.006849
0.099671
60 NaN -0.102349 -0.134754 -0.065957  ...  1.000000  0.609515  0.243305
0.103015
61 NaN -0.029870 -0.041183 -0.054936  ...  0.609515  1.000000  0.648328
0.261991
62 NaN  0.026547  0.072599  0.053437  ...  0.243305  0.648328  1.000000
0.620428
```

63 NaN -0.043889 0.082523 0.081971 ... 0.103015 0.261991 0.620428
1.000000

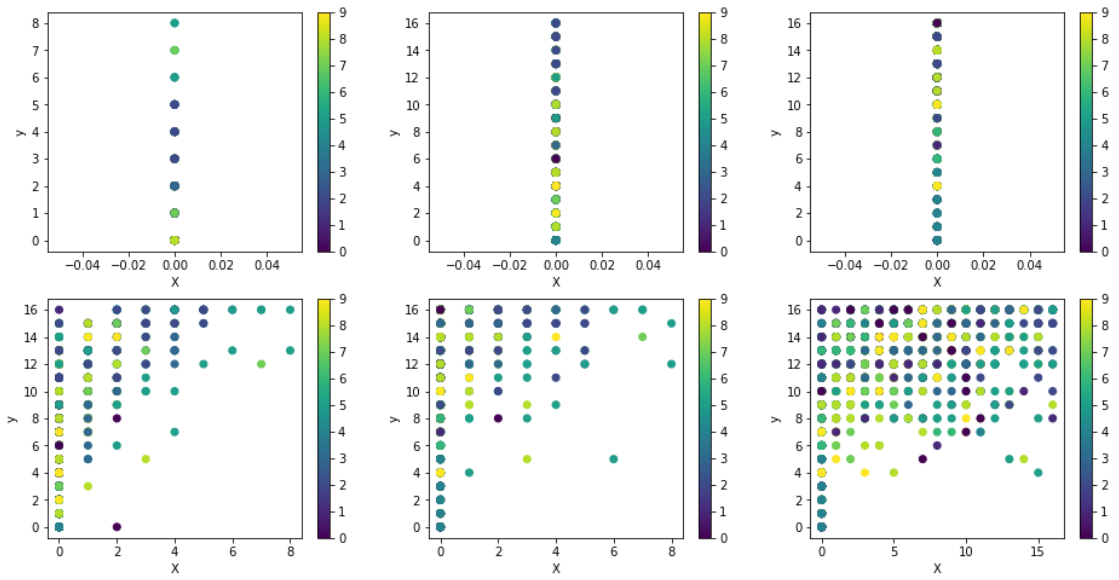
[64 rows x 64 columns]

visualisation des données à l’aide d’un ACP et ADL :

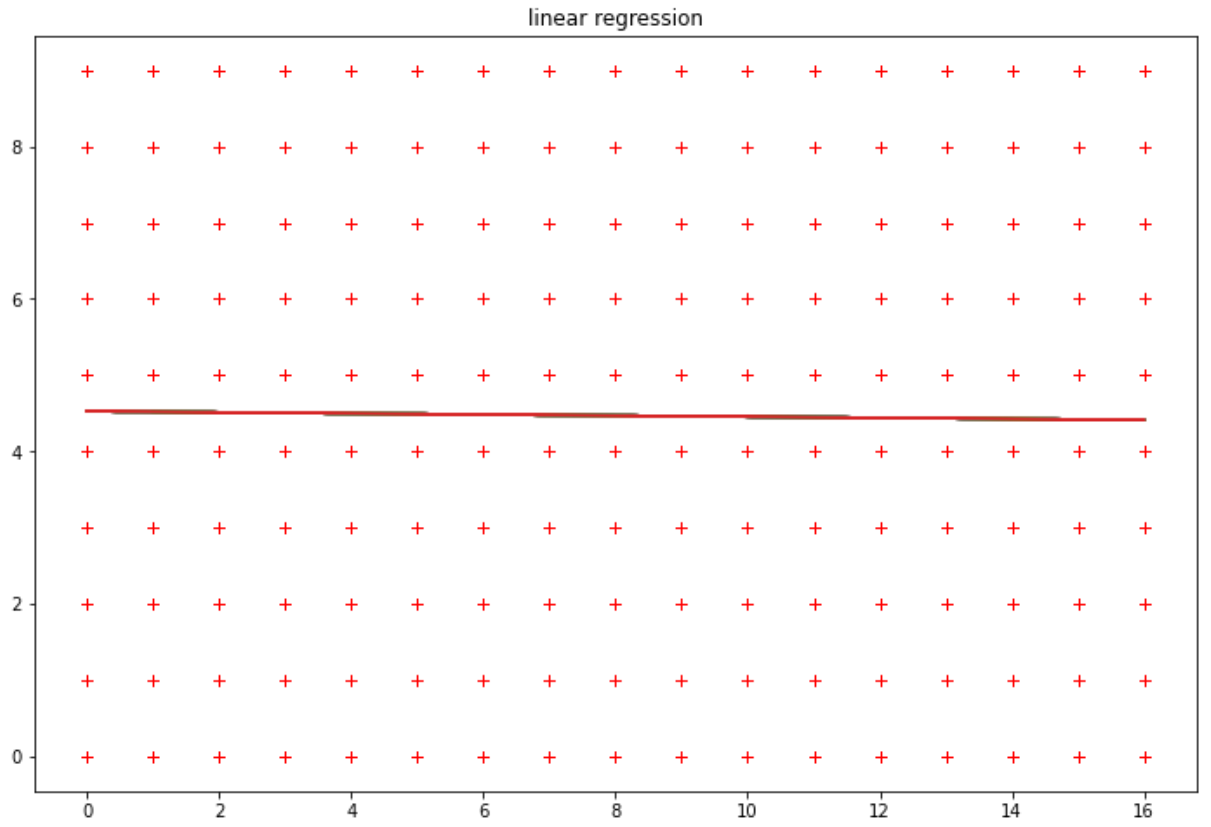




********* visualisation des données selon chaque combinaison des paramètres



********** Regression liniear **********



---------- classification ----------
------- ----------

********** KNN **********

	precision	recall	f1-score	support
0	1.00	1.00	1.00	43
1	0.97	1.00	0.98	56
2	1.00	1.00	1.00	51
3	0.98	1.00	0.99	60

4	1.00	0.98	0.99	48
5	0.98	0.98	0.98	49
6	0.98	1.00	0.99	58
7	0.98	0.98	0.98	64
8	0.98	0.95	0.97	60
9	0.98	0.96	0.97	51
accuracy			0.99	540
macro avg	0.99	0.99	0.99	540
weighted avg	0.99	0.99	0.99	540

erreur = 1.0 %

********** naive bayes **********

	precision	recall	f1-score	support
0	0.98	1.00	0.99	43
1	0.66	0.88	0.75	56
2	0.97	0.61	0.75	51
3	0.88	0.88	0.88	60
4	0.95	0.81	0.88	48
5	0.96	0.92	0.94	49
6	0.95	0.95	0.95	58
7	0.80	1.00	0.89	64
8	0.74	0.80	0.77	60
9	0.85	0.65	0.73	51
accuracy			0.85	540
macro avg	0.87	0.85	0.85	540
weighted avg	0.87	0.85	0.85	540

erreur = 15.0 %

********** SVM **********

	precision	recall	f1-score	support
0	1.00	1.00	1.00	43
1	0.95	1.00	0.97	56
2	1.00	1.00	1.00	51
3	1.00	0.98	0.99	60
4	1.00	1.00	1.00	48
5	0.96	0.98	0.97	49
6	0.98	1.00	0.99	58
7	1.00	0.98	0.99	64
8	0.96	0.92	0.94	60
9	0.94	0.94	0.94	51
accuracy			0.98	540
macro avg	0.98	0.98	0.98	540
weighted avg	0.98	0.98	0.98	540

erreur = 2.0 %

********** MLP **********

	precision	recall	f1-score	support

```
              0       0.96      1.00      0.98        43
              1       0.95      0.98      0.96        56
              2       0.98      0.98      0.98        51
              3       0.97      0.97      0.97        60
              4       1.00      0.98      0.99        48
              5       0.98      0.98      0.98        49
              6       0.98      0.95      0.96        58
              7       0.97      0.97      0.97        64
              8       0.95      0.95      0.95        60
              9       0.98      0.96      0.97        51

    accuracy                            0.97       540
   macro avg       0.97      0.97      0.97       540
weighted avg       0.97      0.97      0.97       540
```

erreur = 3.0 %

```
  ------------------------------------ Clustering -------------------------
--- ---------------------------
```

```
********************** Kmeans *************************
silhouette score pour Hierarchical clustering est :  0.18241832571369485
davies_bouldin_score pour optic clustering est :  1.92622222748616
```

```
***************** Hierarchical clustering (Agglomerative)
*********************
silhouette score pour Hierarchical clustering est :  0.17849659940596496
davies_bouldin_score pour optic clustering est :  1.8889885974945109
```

```
 ************************* Optic Clustering
*******************************
silhouette score pour optic clustering est :  -0.03003367750986449
davies_bouldin_score pour optic clustering est :  1.6126235073233708
```

```
 ---------------------- End --------------------------
```

Cette Partie est fini ici, le but était d'implémentant une fonction analyse(x,y) qui regroupe une partie de tout ce que nous avons fait dans les TPs précédents.
Vous trouverez le code complet de **TP N° 6** dans mon notebook Google-colab suivant :
https://colab.research.google.com/drive/1jvFA03C0jKlN4UNkTCC76s7OObfBhKwY?usp=sharing

8. Conclusion

Dans ce rapport j'ai cité tout ce que j'ai fait pour répondre aux travaux demandés pour le module de science des données numérique, sous la direction de monsieur Guénaël Cabanes, en exploitant ce que nous avons vu dans le cours avec monsieur Younès Bennani.

Vous trouverez le code python de chaque partie dans les fichiers `.pynb` joints avec ce rapport, ou vous pouvez les consulter à partir des liens vers mes Notebooks google-colab suivants :

TP1:
https://colab.research.google.com/drive/1jrCg7Cpppdvq92_8O4H3Ah7JVBbTpJxW?usp=sharing

TP2 :

https://colab.research.google.com/drive/1eNWbhvnvftYvT3GLucsWjCCvVwiRxhD_?usp=sharing

TP3 :

https://colab.research.google.com/drive/17jatxGPnaYHuwQOQbBIMfsmtUyIe2vg7?usp=sharing

TP4 :

https://colab.research.google.com/drive/10npy3rfDz4TzS5a67KEUx-YkSH3fgzSo?usp=sharing

TP5 :

https://colab.research.google.com/drive/1AyMjVvck3QiugwsVU6sHyde460gW9DuJ?usp=sharing

TP6:

https://colab.research.google.com/drive/1jvFA03C0jKlN4UNkTCC76s7OObfBhKwY?usp=sharing