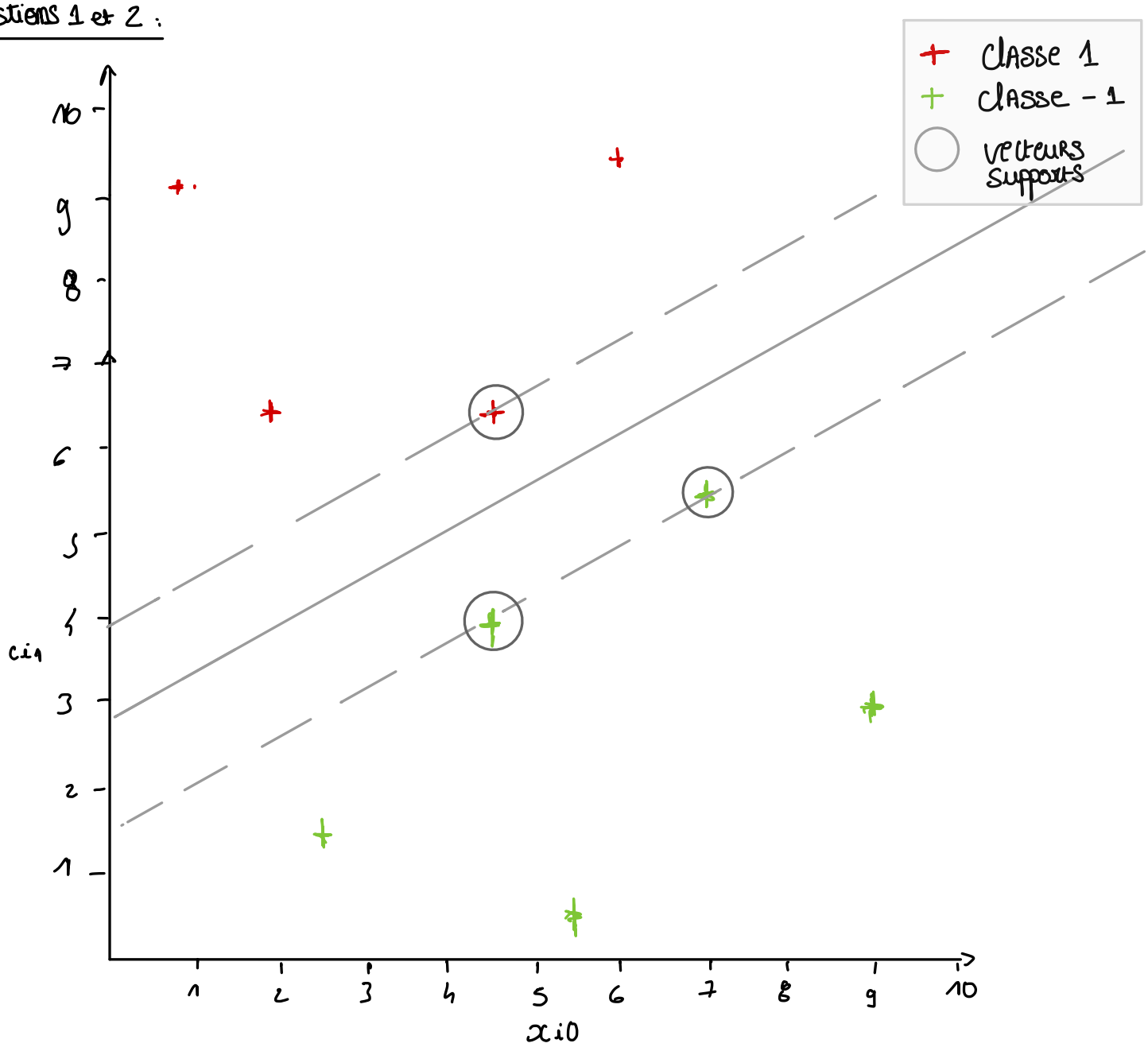
# Exercice

questions 1 et 2 :




Voici les vecteurs supports correspondant aux points entourés sur le graphique. soit les points

| | | |
|---|---|---|
| $x_1 (4,5 ; 4)$ | $x_2 (7,5 ; 5,5)$ | $x_3 (4,5 ; 6,5)$ |
| $y_1 = -1$ | $y_2 = -1$ | $y_3 = +1$ |

## Question 3 :

- paramètres de la solution primale

On a $y_i(w.x_i + b) = 1$ pour chaque vecteur support.

On a donc :

$$\begin{cases} -1\,(w_1 \cdot 4{,}5 + w_2 \cdot 4 + b) = 1 \\ -1\,(w_1 \cdot 7{,}5 + w_2 \cdot 5{,}5 + b) = 1 \\ 1\,(w_1 \cdot 4{,}5 + w_2 \cdot 6{,}5 + b) = 1 \end{cases}$$

$$\begin{cases} -4{,}5\,w_1 - 4\,w_2 - b = 1 & (1) \\ -7{,}5\,w_1 - 5{,}5\,w_2 - b = 1 & (2) \\ 4{,}5\,w_1 + 6{,}5\,w_2 + b = 1 & (3) \end{cases}$$

On veut résoudre ce système :

(1) + (3) $\quad -4{,}5 w_1 + 4{,}5 w_1 - 4 w_2 + 6{,}5 w_2 - b + b = -2$

$$2{,}5\,w_2 = -2$$

$$\boxed{w_2 = -0{,}8}$$

Dans (1) : $\quad -4{,}5\,w_1 - 4 \times 0{,}8 - b = 1$

$$-4{,}5\,w_1 - 3{,}2 - b = 1$$

$$w_1 = -\frac{4{,}2 + b}{4{,}5}$$

Dans (2) : $\quad -7{,}5 \times \left(-\frac{4{,}2 + b}{4{,}5}\right) - 5{,}5 \times 0{,}8 - b = 1$

$$\frac{7{,}5 \times 4{,}2 + 7{,}5 \times b}{4{,}5} - 5{,}5 \times 0{,}8 - b = 1$$

$$7{,}5 \times 4{,}2 + 7{,}5 \times b - 5{,}5 \times 0{,}8 \times 4{,}5 - 4{,}5\,b = 4{,}5$$

$$31{,}5 + 7{,}5\,b - 19{,}8 - 4{,}5\,b = 4{,}5$$

$$11{,}7 + 3b = 4{,}5$$

$$b = \frac{4{,}5 - 11{,}7}{3}$$

$$b = \frac{7{,}2}{3}$$

$$\boxed{b = 2{,}4} \quad \text{d'où} \quad \boxed{w_1 = -\frac{4{,}2 - 2{,}4}{4{,}5} = 0{,}4}$$

Les valeurs de l'hyperplan sont donc : $\boxed{w_1 = 0{,}4\ ,\ w_2 = -0{,}8 \text{ et } b = 2{,}4}$

## • paramètres de la solu° duale

La solution duale concerne la détermination des multiplicateurs de Lagrange $\alpha$.
On obtient les valeurs de $\alpha$ en résolvant le système suivant :

$$\begin{cases} W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{ij=1} y_i y_j (x_i \cdot x_j) \\ \sum_{i=1}^{m} \alpha_i y_i = 0 \quad \text{et} \quad \alpha_i \geq 0 \end{cases}$$

On a $W(\alpha_1, \alpha_2, \alpha_3) = \alpha_1 + \alpha_2 + \alpha_3 - \frac{1}{2}[\alpha_1^2 y_1^2 + \alpha_2^2 y^2 + \alpha_3^3 y_3^2 + 2\alpha_1\alpha_2 y_1 y_2 \langle x_1, x_i \rangle$

$+ 2\alpha_1\alpha_3 y_1 y_3 \langle x_1, x_3 \rangle + 2\alpha_2\alpha_3 y_2 y_3 \langle x_2, x_3 \rangle]$

On calcule les produits scalaires :

$\langle x_1, x_2 \rangle = 4{,}5 \times 7{,}5 + 4 \times 5{,}5 = 55{,}75$

$\langle x_1, x_3 \rangle = 46{,}25$

$\langle x_2, x_3 \rangle = 69{,}5$

On a donc $W(\alpha_1, \alpha_2, \alpha_3) = \alpha_1 + \alpha_2 + \alpha_3 \quad [\alpha_1^2 + \alpha_2^2 + \alpha_3^2 - 2\alpha_1\alpha_2 \times 55{,}75 - 2\alpha_1\alpha_3$

$\times 46{,}25 + 2\alpha_2\alpha_3 \times 69{,}5$

On resoud le système :

$\alpha_1 > 0$

$\alpha_1 y_1 + \alpha_2 y_2 + \alpha_3 y_3 = 0 \Rightarrow -\alpha_1 - \alpha_2 + \alpha_3 = 0$

$$\begin{cases} \frac{\partial W}{\partial \alpha_1} = 1 - \alpha_1 + \alpha_2 \times 55{,}75 + \alpha_3 \times 46{,}25 = 0 & (1) \\ \frac{\partial W}{\partial \alpha_2} = 1 - \alpha_2 + \alpha_1 \times 55{,}75 + \alpha_3 \times 69{,}5 = 0 & (2) \\ \frac{\partial W}{\partial \alpha_3} = 1 - \alpha_3 + \alpha_1 \times 46{,}25 - \alpha_2 \times 69{,}5 = 0 & (3) \\ -\alpha_1 - \alpha_2 + \alpha_3 = 0 & (4) \end{cases}$$

Après la résolution de ce système, nous obtenons

$\alpha_1 = 0{,}43 \; ; \quad \alpha_2 = 0{,}20 \; ; \quad \alpha_3 = 0{,}24$

# Exercice de Compréhension

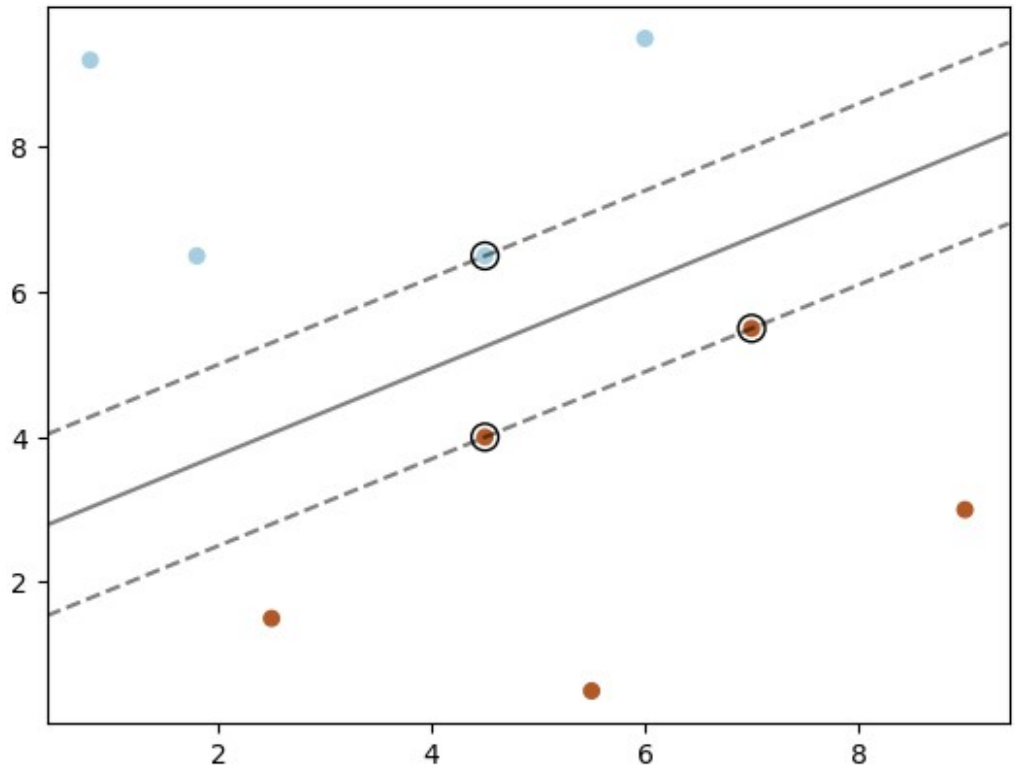
Nous avons fait une vérification de nos valeurs trouvées par les calculs.

```
pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in
/usr/local/lib/python3.10/dist-packages (1.2.2)
Requirement already satisfied: numpy>=1.17.3 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.23.5)
Requirement already satisfied: scipy>=1.3.2 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.11.3)
Requirement already satisfied: joblib>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.2.0)
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC

# Données
X = np.array([[6, 9.5], [0.8, 9.2], [1.8, 6.5], [4.5, 6.5], [7, 5.5],
              [4.5, 4], [9, 3], [2.5, 1.5], [5.5, 0.5]])
y = np.array([-1, -1, -1, -1, 1, 1, 1, 1, 1])

# Entraînement du modèle SVM
model = SVC(kernel='linear')
model.fit(X, y)

# Visualisation
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# Tracer l'hyperplan
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Créer la grille pour évaluer le modèle
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = model.decision_function(xy).reshape(XX.shape)

# Tracer la frontière de décision et les marges
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])
```

```
# Tracer les vecteurs supports
ax.scatter(model.support_vectors_[:, 0], model.support_vectors_[:, 1],
s=100,
           linewidth=1, facecolors='none', edgecolors='k')
plt.show()

# Paramètres du modèle
print("Vecteur normal (w):", model.coef_)
print("Biais (b):", model.intercept_)
print("Multiplicateurs de Lagrange (alpha):",
np.abs(model.dual_coef_))
```




```
Vecteur normal (w): [[ 0.47983062 -0.8       ]]
Biais (b): [2.04090335]
Multiplicateurs de Lagrange (alpha): [[0.43515935 0.19193225 0.2432271
]]
```

## Gérér l'affichage des courbes

On va utiliser pyplot du module matplotlib pour afficher les courbes et les graphiques La commande %matplotlib inline fait en sorte que les courbes apparaissent dans le notebook. Si vous voulez sauvegarder les courbes sans les afficher, il faut ajouter la commande matplotlib.use('Agg') entre les 2 commandes suivantes :

```
# utiliser matplotlib
%matplotlib inline
#matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

# Machines à Vecteurs de Support (SVM)

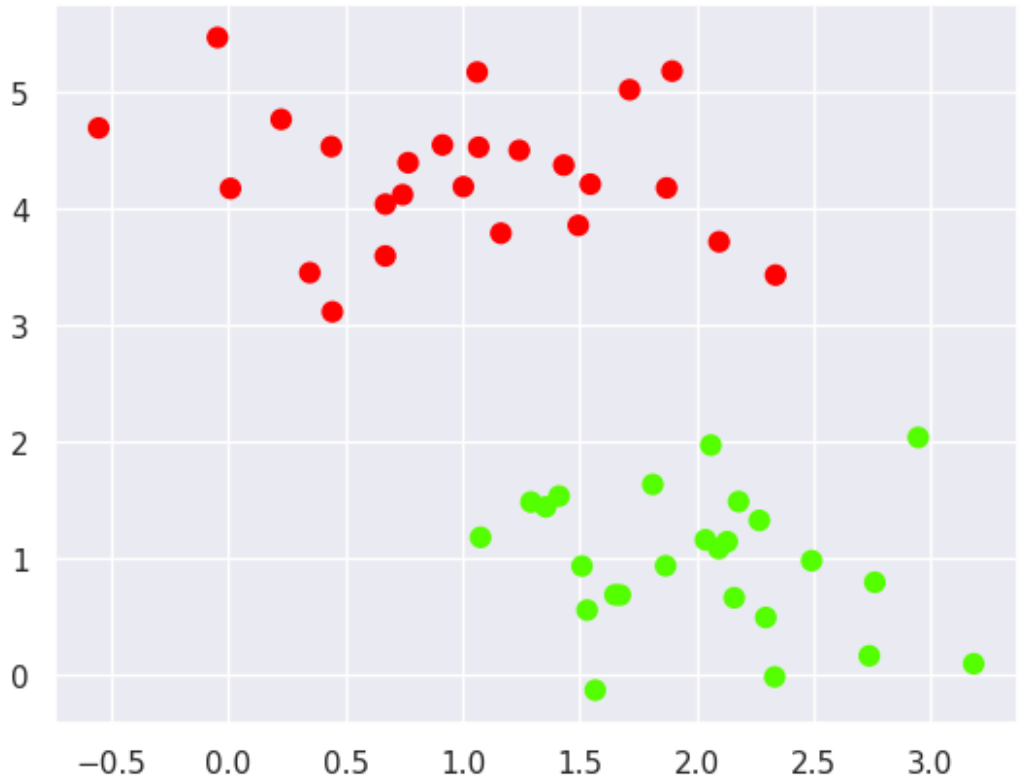
Si anaconda3 n'est pas installé, il nous faut d'abord installer les modules nécessaires.

```
"""
!pip3 -q install sklearn
!pip3 -q install matplotlib
!pip3 -q install seaborn
"""

{"type":"string"}
```

# Première partie : prise en main des SVM

Cette partie est librement inspirée du travail de Jake VenderPlas, auteur du livre Python Data Science Handbook. Son GitHub (en anglais) regorge de fichiers utiles. Dans un premier temps, on va générer des données jouets, linéairement séparables :

```
%matplotlib inline
import matplotlib.pyplot as plt
#Un petit environment qui donne de meilleurs graphes
import seaborn as sns; sns.set()
# fonction sklearn pour générer des données simples
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=50, centers=2,
 random_state=0, cluster_std=0.60)
# Affichage des données
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='prism');
```




## SVM linéaire (Séparateur à vaste marge)

On va commencer par apprendre un SVM linéaire (sans noyau) à l'aide de scikit-learn :

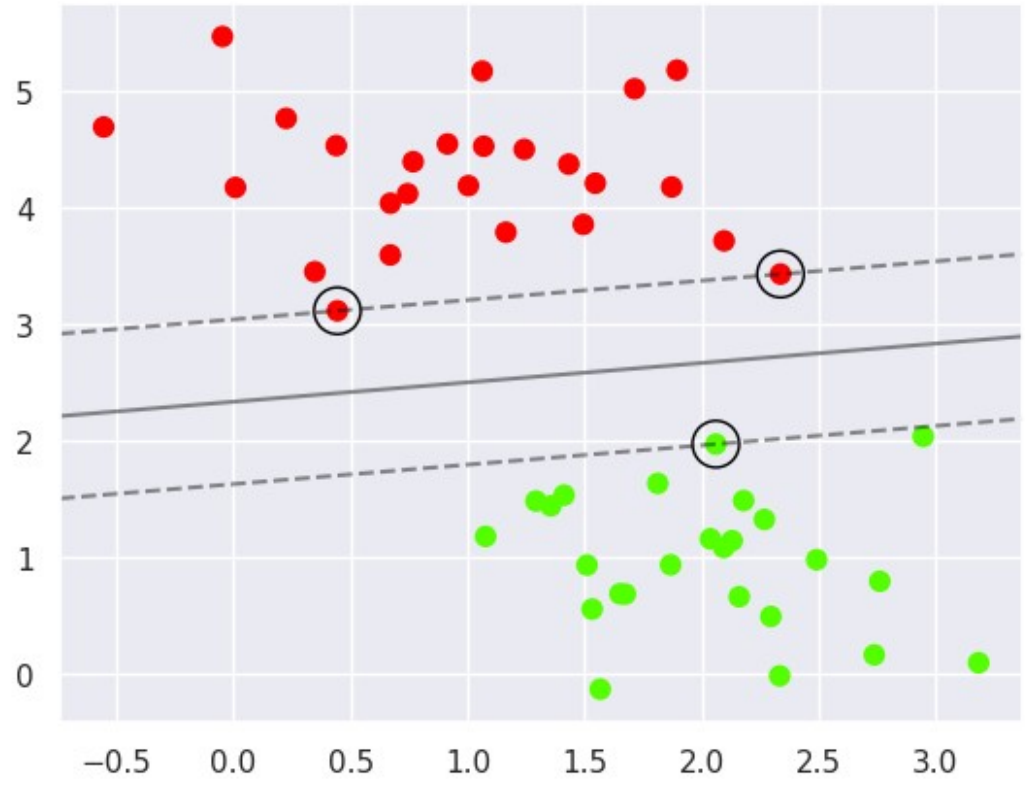
```
#import de la classe - qui s'appelle SVC et pas SVM...
from sklearn.svm import SVC
#Définition du modèle
model = SVC(kernel='linear', C=1E10)
#Apprentissage sur les donnée
model.fit(X, y)
```

```
SVC(C=10000000000.0, kernel='linear')
```

On va utiliser une fonction d'affichage qui va bien, où tout ce qui est nécessaire est affiché.

```
import numpy as np
def affiche_fonction_de_decision(model, ax=None, plot_support=True):
 """Affiche le séparateur, les marges, et les vecteurs de support
d'un SVM en 2D"""
 if ax is None:
  ax = plt.gca()
 xlim = ax.get_xlim()
 ylim = ax.get_ylim()

 # création de la grille pour l'évaluation
```

```
 x = np.linspace(xlim[0], xlim[1], 30)
 y = np.linspace(ylim[0], ylim[1], 30)
 Y, X = np.meshgrid(y, x)
 xy = np.vstack([X.ravel(), Y.ravel()]).T
 P = model.decision_function(xy).reshape(X.shape)

 # affichage de l'hyperplan et des marges
 ax.contour(X, Y, P, colors='k', levels=[-1, 0, 1], alpha=0.5,
linestyles=['--', '-', '--'])

 # Affichage des vecteurs de support
 if plot_support:
  ax.scatter(model.support_vectors_[:, 0], model.support_vectors_[:,
1],s=300, linewidth=1, facecolors='none',
edgecolor='black');
 ax.set_xlim(xlim)
 ax.set_ylim(ylim)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='prism')
affiche_fonction_de_decision(model);
```



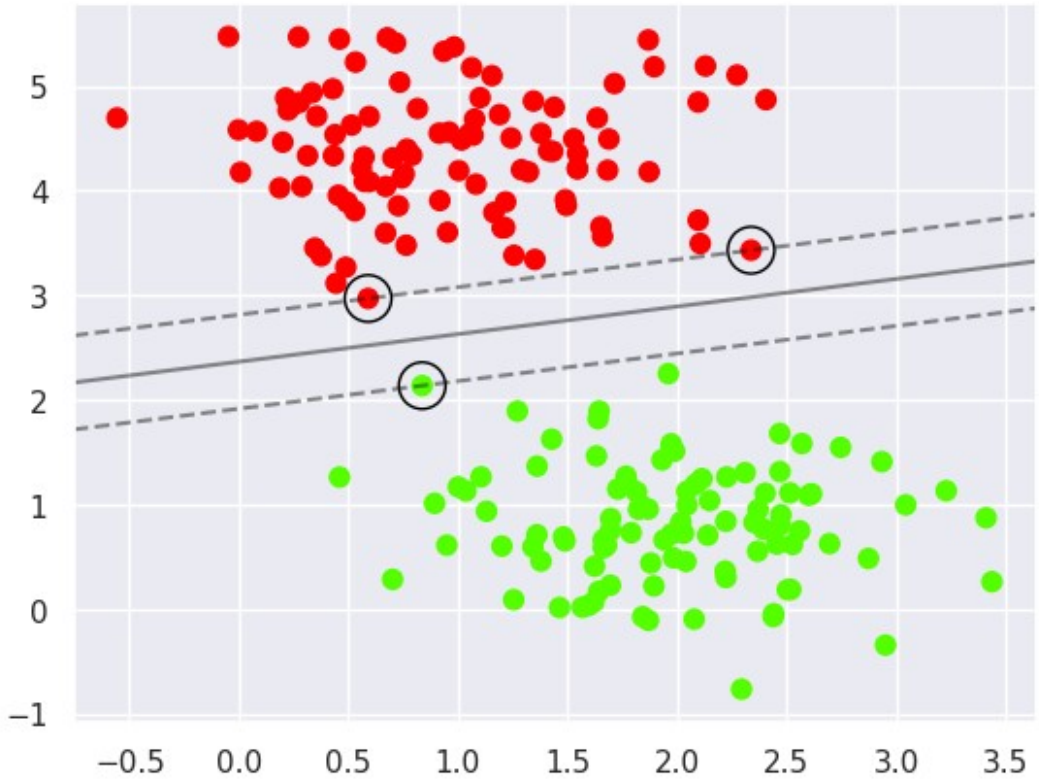

Sur ce graphe, on voit le séparateur (ligne pleine), les vecteurs de support (points entourés) et la marge (matérialisée par des lignes discontinues). On a ici le séparateur qui maximise la marge. Scikit-learn nous permet, après apprentissage, de récupérer les vecteurs de supports:

```
model.support_vectors_
```

```
array([[0.44359863, 3.11530945],
       [2.33812285, 3.43116792],
       [2.06156753, 1.96918596]])
```

Seules trois données sont utiles pour classer de nouvelles données. On peut s'en assurer en rajoutant des données sans changer le modèle :

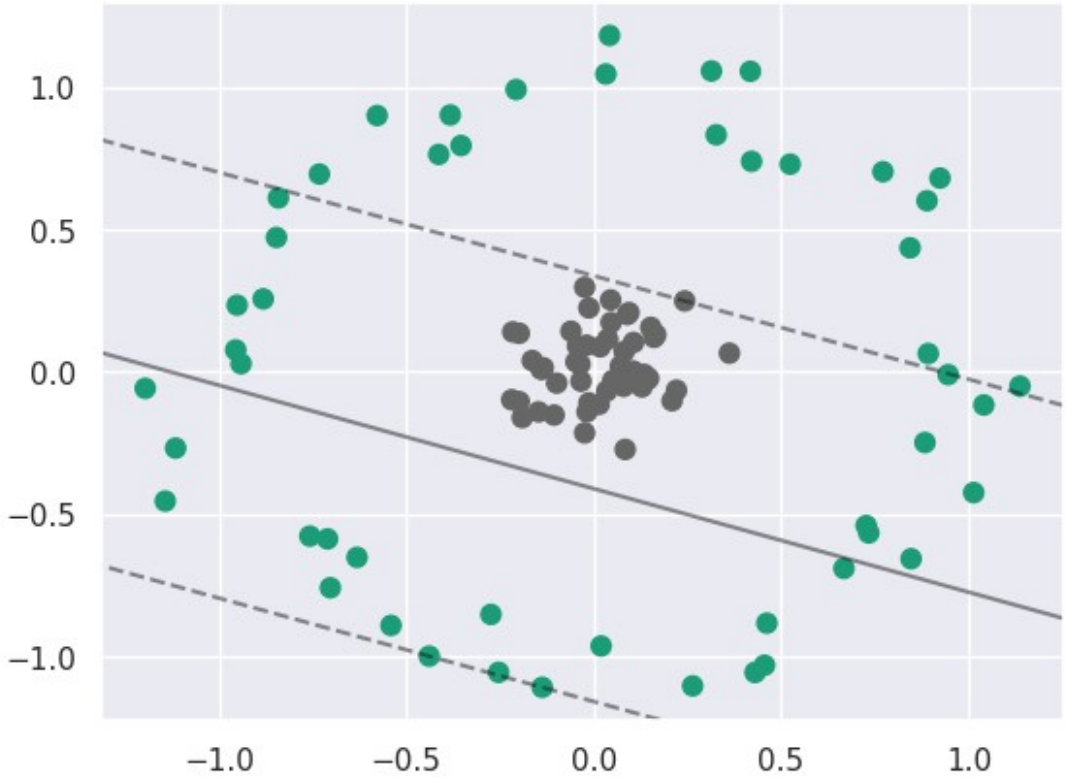
```
X2, y2 = make_blobs(n_samples=200, centers=2,
 random_state=0, cluster_std=0.60)
model2 = SVC(kernel='linear', C=1E10)
model2.fit(X2, y2)
plt.scatter(X2[:, 0], X2[:, 1], c=y2, s=50, cmap='prism')
affiche_fonction_de_decision(model2);
```




## SVM non linéaire

Comme vu en cours, la puissance des séparateurs linéaires est limitée (à des données linéairement séparables). Mais il est possible de contourner cette limitation par l'utilisation de noyaux. On va commencer par générer des données non-linéairement séparables, puis on apprend un classifieur SVM linéaire et on affiche le résultat :

```
from sklearn.datasets import make_circles
X, y = make_circles(100, factor=.1, noise=.1)
clf = SVC(kernel='linear').fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='Dark2')
affiche_fonction_de_decision(clf, plot_support=False)
```



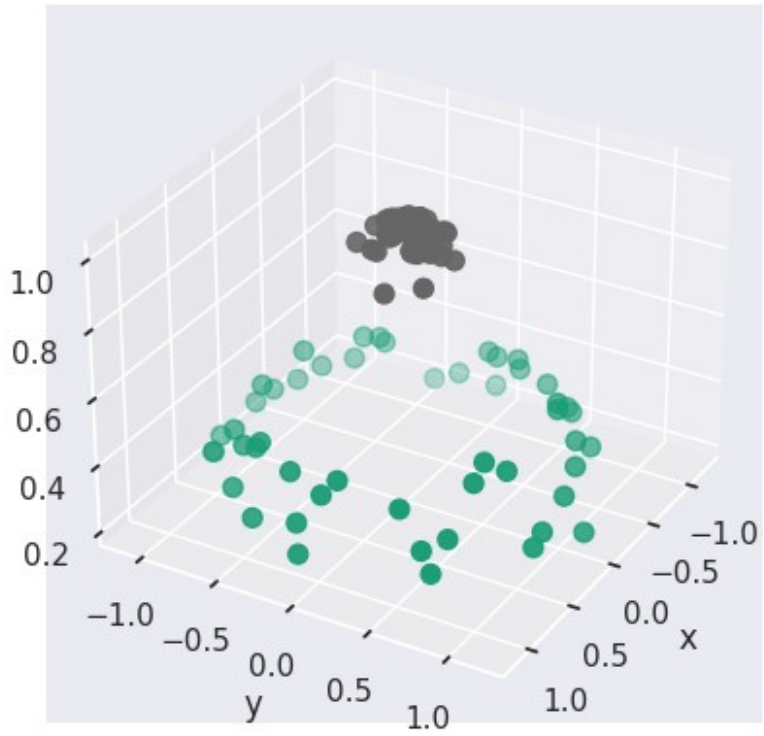

Clairement notre apprentissage de séparateur linéaire a échoué... On va manuellement ajouter une troisième dimension z :

```
z = np.exp(-(X ** 2).sum(1))
```

On peut afficher les données augmentées et se rendre compte qu'elles sont linéairement séparables dans ce nouvel espace de dimension plus grande :

```
from mpl_toolkits.mplot3d import Axes3D
ax = plt.subplot(projection='3d')
ax.scatter3D(X[:, 0], X[:, 1], z, c=y, s=50, cmap='Dark2')
ax.view_init(elev=30, azim=30)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
```

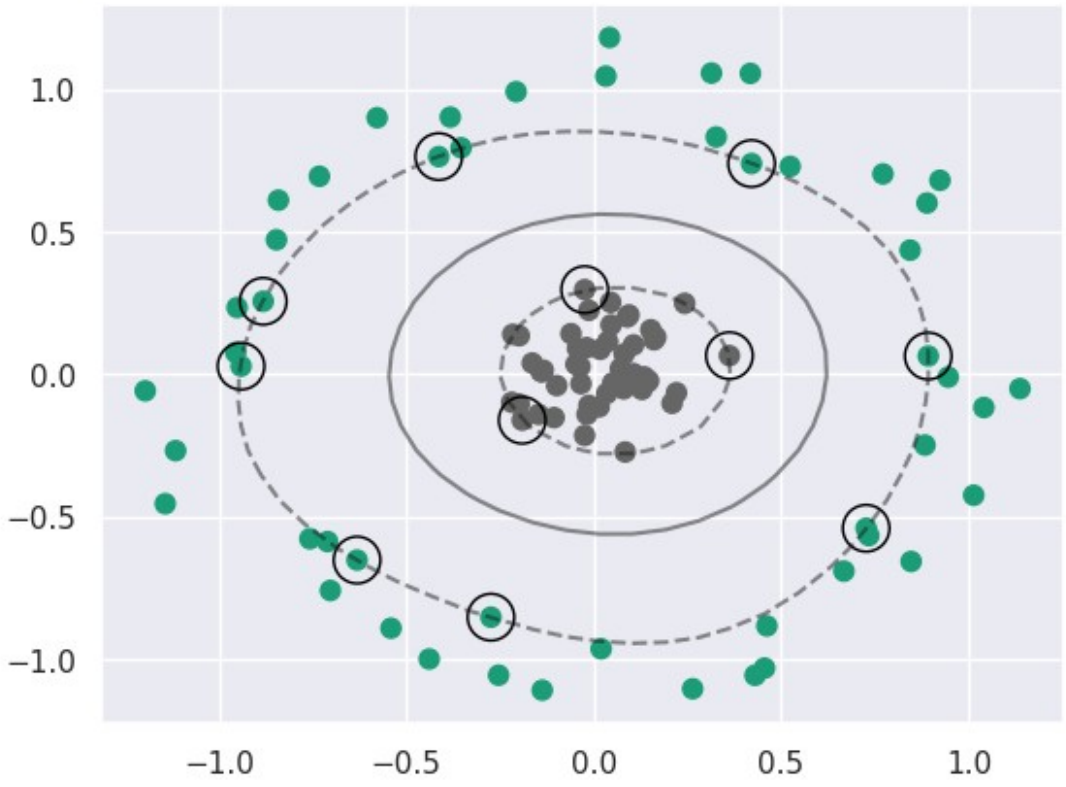
```
Text(0.5, 0, 'z')
```




Le plan définit par z=0.7 (par exemple) sépare les 2 classes parfaitement. Bien entendu, la projection en plus grande dimension est capitale, et en choisissant un autre calcul pour z on aurait probablement obtenu des données non linéairement séparables. Et s'il fallait faire effectivement la projection, cela limiterait drastiquement la dimension de l'espace de plongement ainsi que le nombre de données traitables.

C'est pourquoi l'utilisation de noyaux (kernels en anglais) est d'une grande efficacité. En Scikit-Learn, il suffit de modifier le paramètre kernel : jusqu'à présent, nous avons utilisé 'linear' comme valeur. On peut par exemple utiliser rbf pour 'radial basis function', le noyau gaussien (celui qui transforme notre espace de description initial vers le 3D avec z précédent), et il nous reste à trouver la bonne valeur du paramètre :

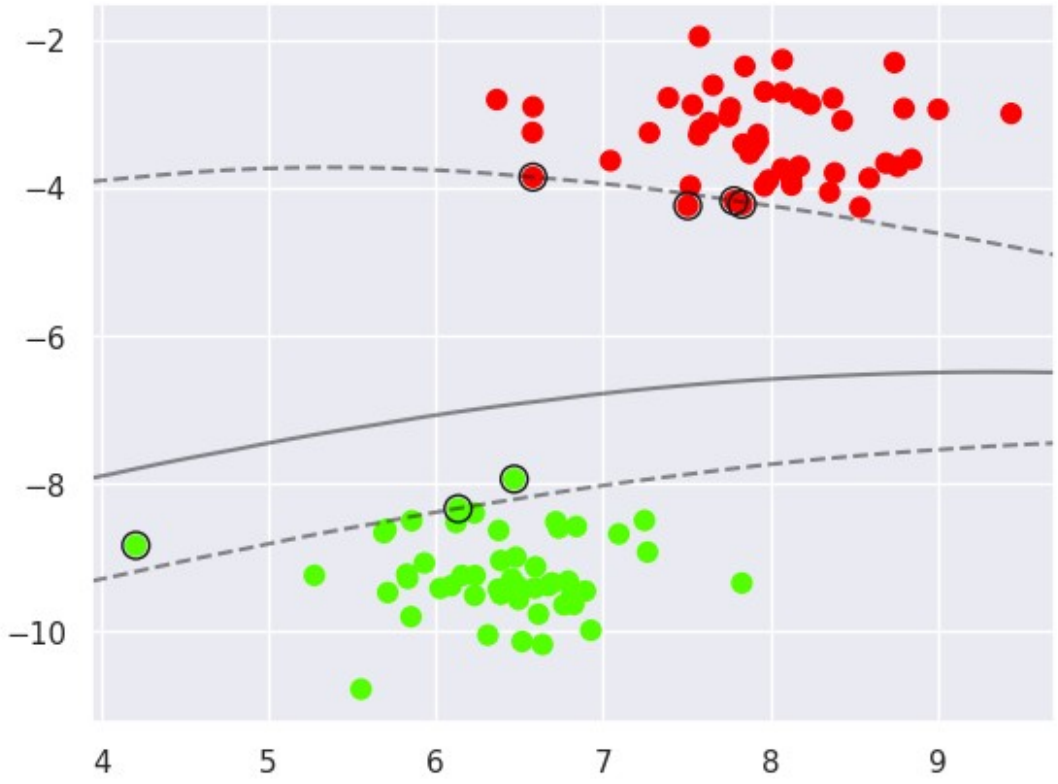
```
clf = SVC(kernel='rbf', C=1E10)
clf.fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='Dark2')
affiche_fonction_de_decision(clf)
```




**Exercice :** Exécuter les instructions permettant un apprentissage avec un autre noyau -- pour un plongement dans un autre espace (par exemple noyau polynomial de degré 5), et la visualisation du séparateur. Vous devriez constater que ce n'est pas un noyaux très adapté !

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC

# Step 1: Define and train the SVM model
clf = SVC(kernel='poly', degree=5)
clf.fit(X, y)

# Step 2: Visualization function
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # Create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
```

```
    P = model.decision_function(xy).reshape(X.shape)

    # Plot decision boundary and margins
    ax.contour(X, Y, P, colors='k', levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # Plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=100, linewidth=1, facecolors='none',
edgecolors='k')
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)

# Plotting the data and the decision function
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='prism')
plot_svc_decision_function(clf)
plt.show()
```



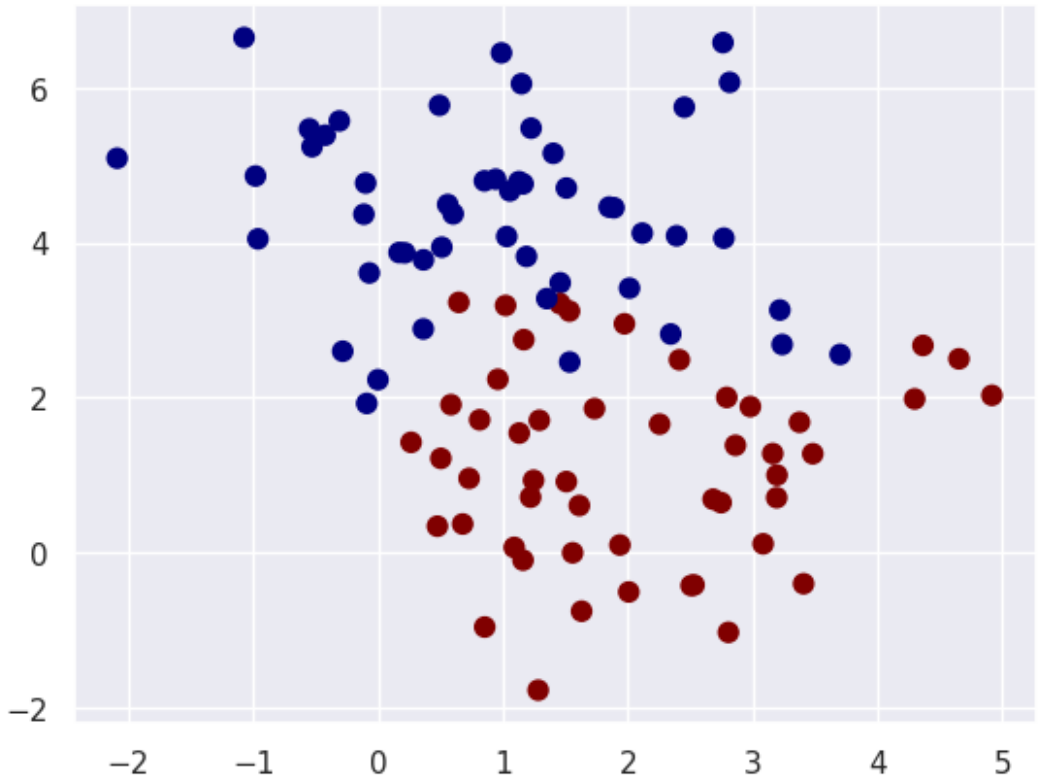

On voit ici que le séparateur (et la marge associée) ne sont pas linéaire dans l'espace des données, mais qu'ils peuvent s'y représenter sans difficulté. Notons aussi que le nombre de vecteurs de support reste très petit.

## SVM à marge douce

Il est aussi possible que le problème soit linéairement séparable (dans la dimension initiale des données ou dans un plongement) mais que le bruit (=la mauvaise qualité des données) empêche l'apprenant de trouver un séparateur.

On utilise alors ce que l'on appelle un classifieur à marge douce : on autorise certains points à être dans la marge, voire du mauvais côté de l'hyperplan. C'est le role du paramètre C : pour des grosses valeurs, on est quasiment en marge dure, mais plus C prend des petites valeurs, plus les marges deviennent permissibles. On va prendre des données qui se chevauchent un peu : (à ce stade, il est important de comprendre la spécificité des données que l'on génére ci-après: en cas de doute appelez votre enseignant)

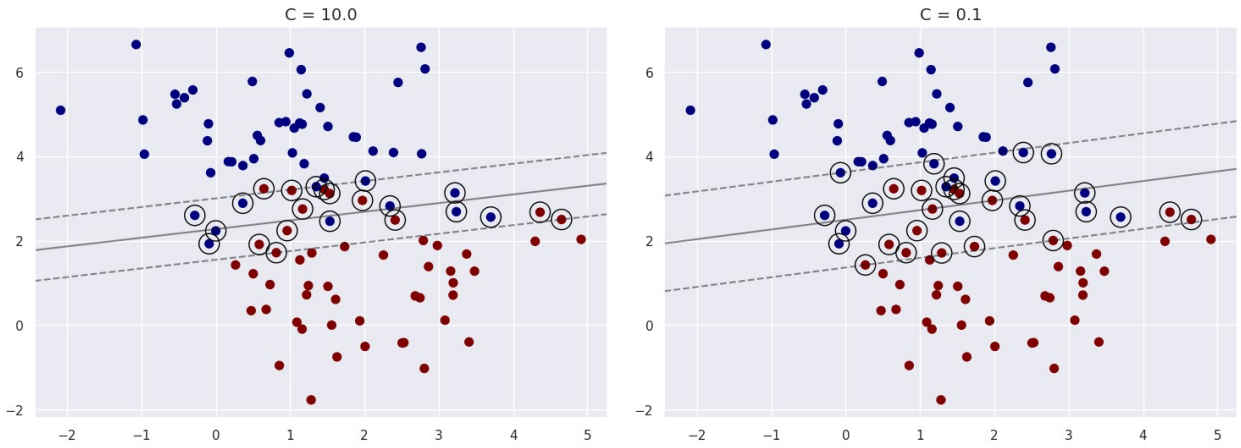
```
X, y = make_blobs(n_samples=100, centers=2,
 random_state=0, cluster_std=1.2)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet');
```




On joue alors avec la valeur de C

```
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, C in zip(ax, [10.0, 0.1]):
 model = SVC(kernel='linear', C=C).fit(X, y)
 axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')
 affiche_fonction_de_decision(model, axi)
```

```
axi.scatter(model.support_vectors_[:, 0],
model.support_vectors_[:, 1],
s=300, lw=1, facecolors='none');
axi.set_title('C = {0:.1f}'.format(C), size=14)
```

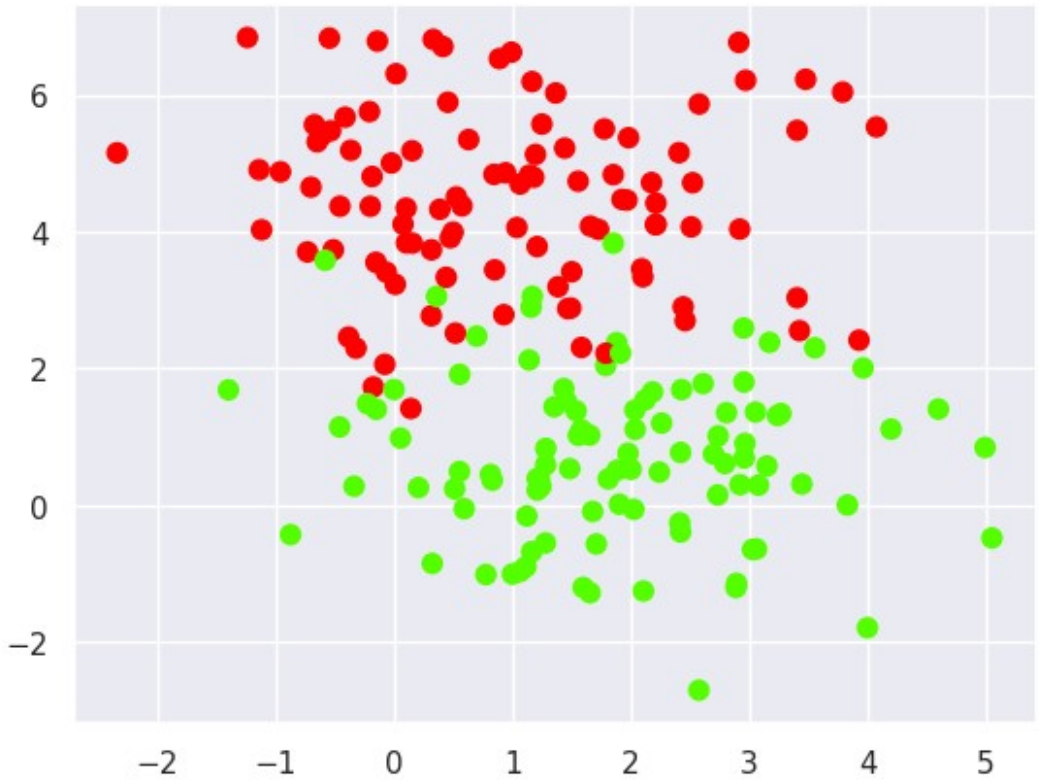



## Paramétrer (tuner) un SVM

Tous les noyaux sont paramétrés : il est question ici d'étudier l'impact d'un (hyper)paramètre sur la qualité de l'apprentissage. Pour cela, on va générer des données qui ne sont pas linéairement séparables :

```
X, y = make_blobs(n_samples=200, centers=2,
 random_state=0, cluster_std=1.3)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='prism')
```

```
<matplotlib.collections.PathCollection at 0x78ef13a317e0>
```




On va étudier 2 noyaux différents

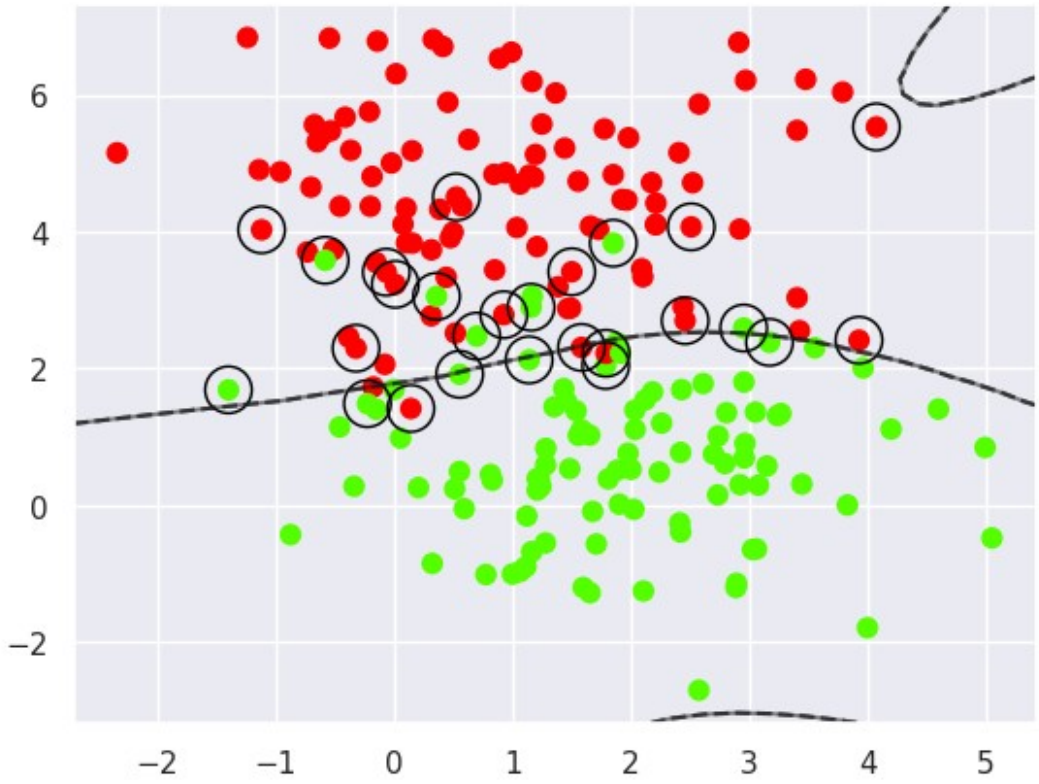
• le noyau polynomial (kernel='poly') qui a 2 paramètres, degree qu'il faut faire varier entre 2 et 6 (au minimum), et C (lié à la 'douceur' de la marge)

• le noyau gaussien (kernel='rbf') qui a aussi 2 paramètres, gamma, qu'il faut faire varier de 1 à 0.01, et C

A chaque fois, en plus de l'affichage des séparateurs, et de l'estimation de l'erreur, il serait intéressant de regarder combien de vecteurs de support le classifieur appris a besoin.

```
#Exemple avec le noyau gaussien et des valeurs pour gamma et C
clf = SVC(kernel='rbf', gamma=0.01, C=1E10)
clf.fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='prism')
affiche_fonction_de_decision(clf)
print("Nombre de vecteurs de support (sur 200 données) :",
len(clf.support_vectors_))
```

```
Nombre de vecteurs de support (sur 200 données) : 26
```



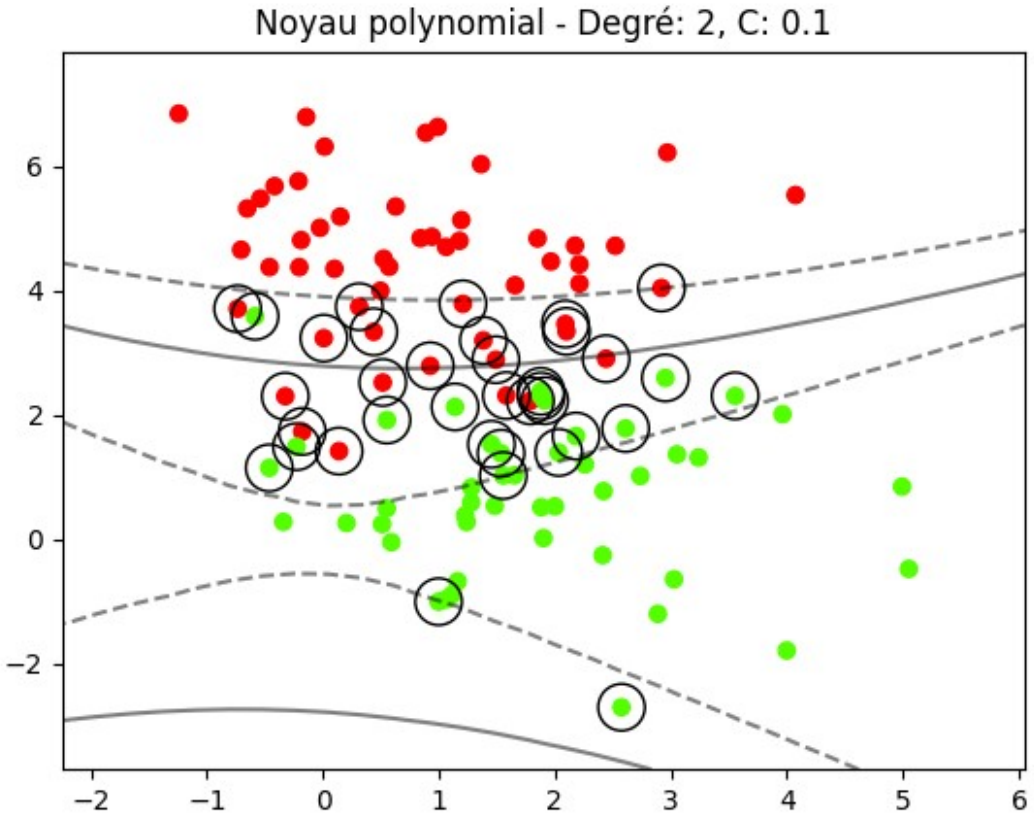

A vous de jouer ! (pour chaque noyau, faire varier les hyper-paramètres dans les intervalles mentionnés, et pour chaque couple d'hyper-paramètres : afficher la frontière de décison, le nombre de vecteurs supports du modèle (le plus petit est le mieux), et le score estimé sur un échantillon test de taille 100 généré de la même façon que l'échantillon d'apprentissage.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split

# Génération des données d'apprentissage et de test
X, y = make_blobs(n_samples=200, centers=2, random_state=0,
cluster_std=1.3)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=100, random_state=42)

# Définition de la fonction d'affichage
def affiche_fonction_de_decision(model, ax, X, y):
    # Ajuster les limites des axes en fonction des données
    ax.set_xlim([X[:, 0].min() - 1, X[:, 0].max() + 1])
    ax.set_ylim([X[:, 1].min() - 1, X[:, 1].max() + 1])

    xlim = ax.get_xlim()
    ylim = ax.get_ylim()
```

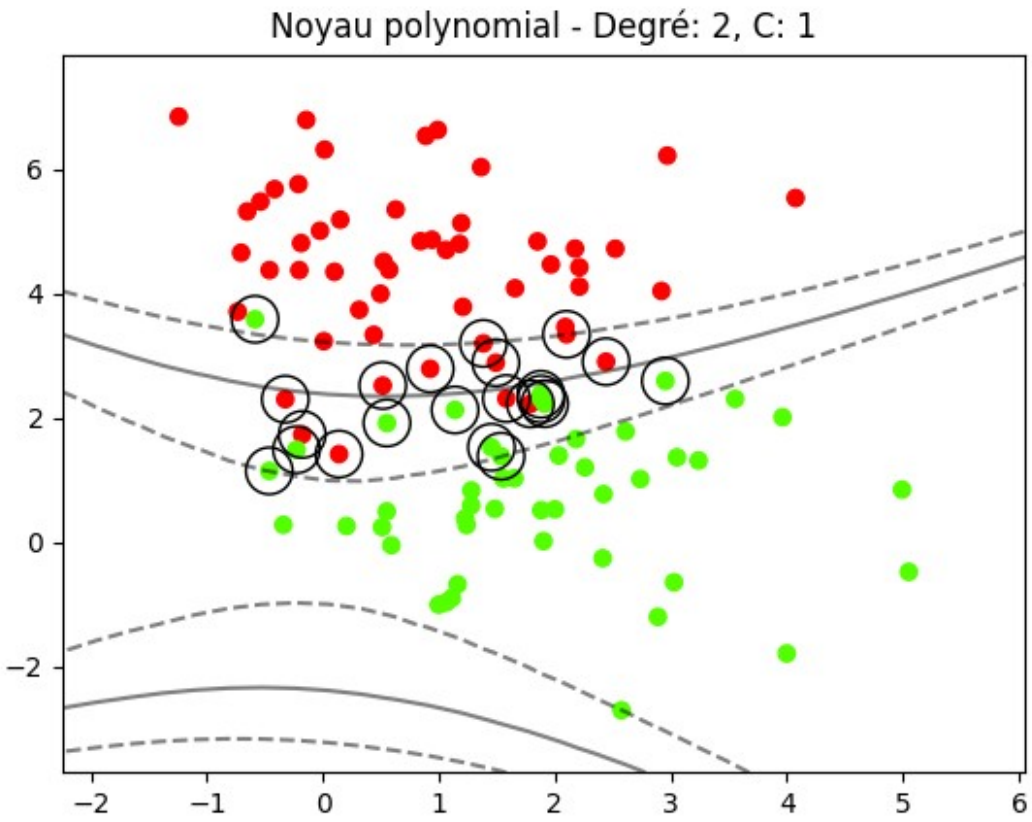
```
    # Création de la grille pour l'évaluation
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # Affichage de l'hyperplan et des marges
    ax.contour(X, Y, P, colors='k', levels=[-1, 0, 1], alpha=0.5,
linestyles=['--', '-', '--'])

    # Affichage des vecteurs de support
    ax.scatter(model.support_vectors_[:, 0], model.support_vectors_[:,
1], s=300, linewidth=1, facecolors='none', edgecolor='black')

# Paramètres pour le noyau polynomial et gaussien
degree_values = range(2, 7)
C_values = [0.1, 1, 10, 100]
gamma_values = [1, 0.1, 0.01, 0.001]

# Exploration du noyau polynomial
for degree in degree_values:
    for C in C_values:
        model = SVC(kernel='poly', degree=degree, C=C)
        model.fit(X_train, y_train)
        score = model.score(X_test, y_test)
        print(f"Noyau polynomial - Degré: {degree}, C: {C}, Score:
{score:.2f}, Vecteurs supports: {len(model.support_vectors_)}")

        # Affichage
        fig, ax = plt.subplots()
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
cmap='prism')
        affiche_fonction_de_decision(model, ax, X_train, y_train)
        ax.set_title(f'Noyau polynomial - Degré: {degree}, C: {C}')
        plt.show()

# Exploration du noyau gaussien
for gamma in gamma_values:
    for C in C_values:
        model = SVC(kernel='rbf', gamma=gamma, C=C)
        model.fit(X_train, y_train)
        score = model.score(X_test, y_test)
        print(f"Noyau gaussien - Gamma: {gamma}, C: {C}, Score:
{score:.2f}, Vecteurs supports: {len(model.support_vectors_)}")

        # Affichage
        fig, ax = plt.subplots()
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
```

```
cmap='prism')
        affiche_fonction_de_decision(model, ax, X_train, y_train)
        ax.set_title(f'Noyau gaussien - Gamma: {gamma}, C: {C}')
        plt.show()
```

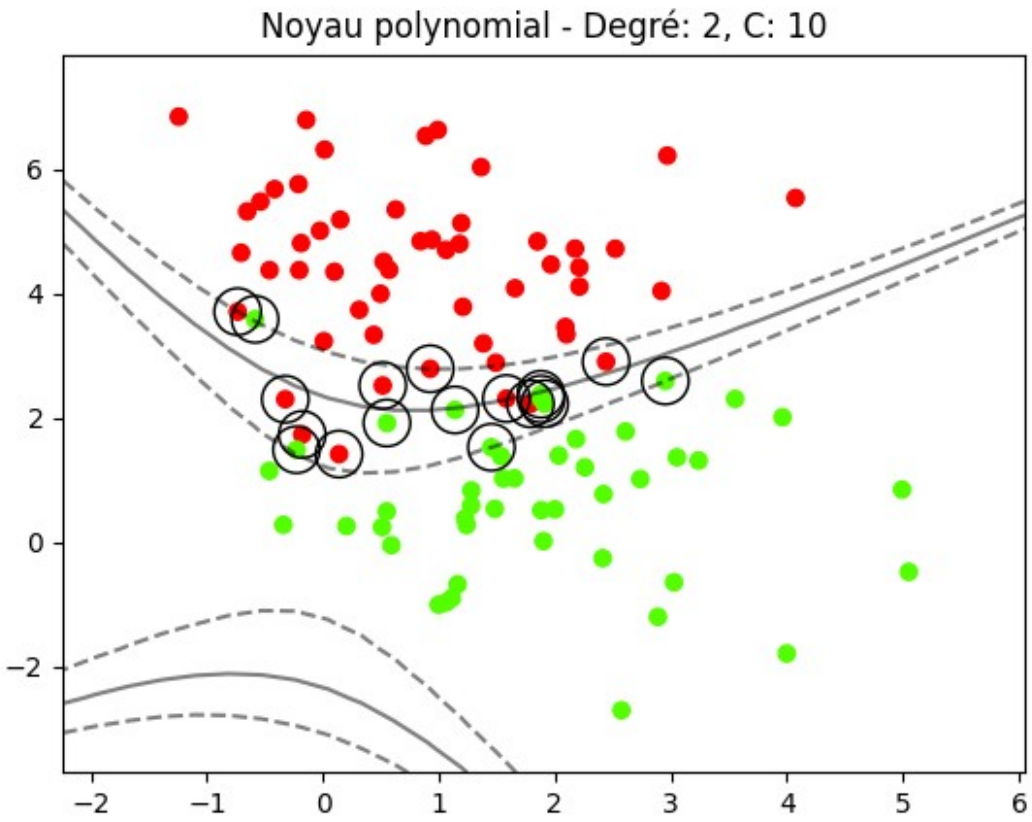
```
Noyau polynomial - Degré: 2, C: 0.1, Score: 0.90, Vecteurs supports:
36
```



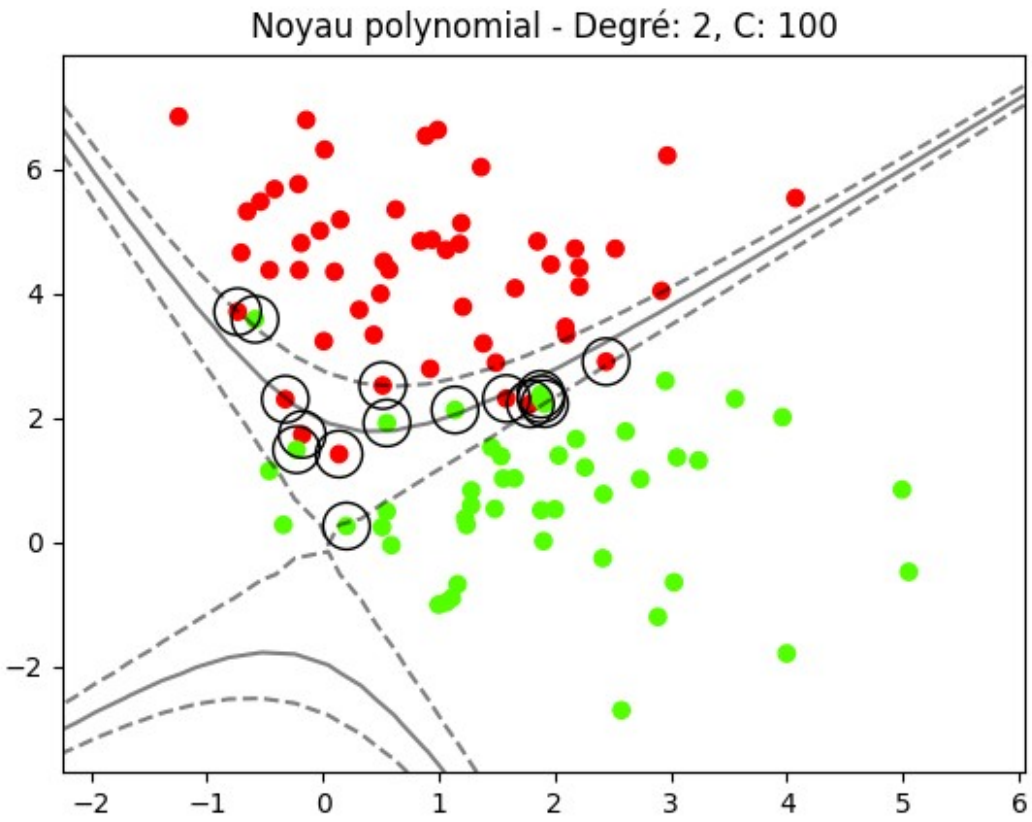

```
Noyau polynomial - Degré: 2, C: 1, Score: 0.89, Vecteurs supports: 22
```



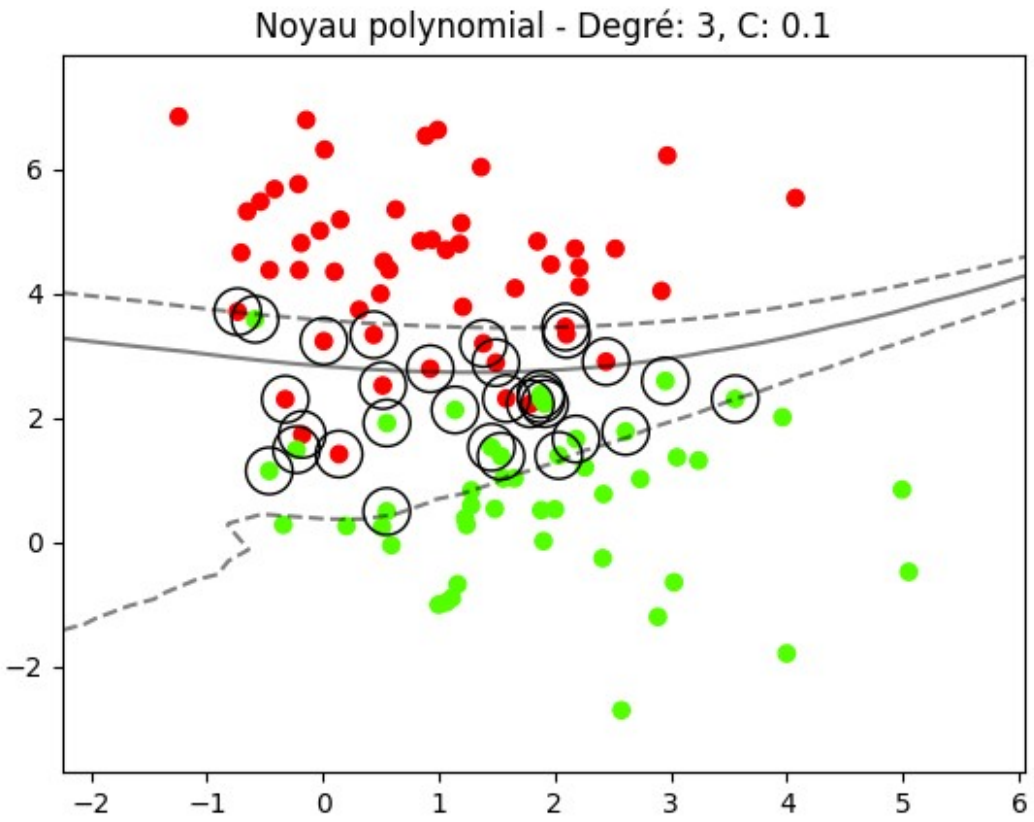

```
Noyau polynomial - Degré: 2, C: 10, Score: 0.88, Vecteurs supports: 18
```



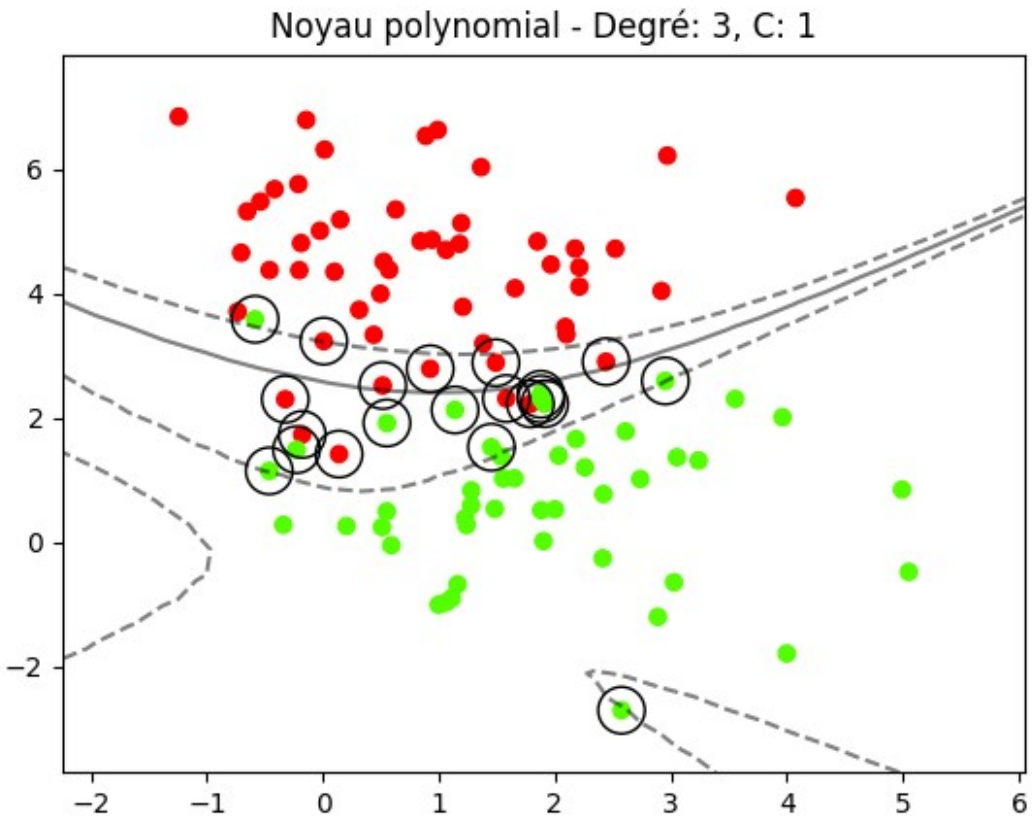

```
Noyau polynomial - Degré: 2, C: 100, Score: 0.90, Vecteurs supports:
16
```



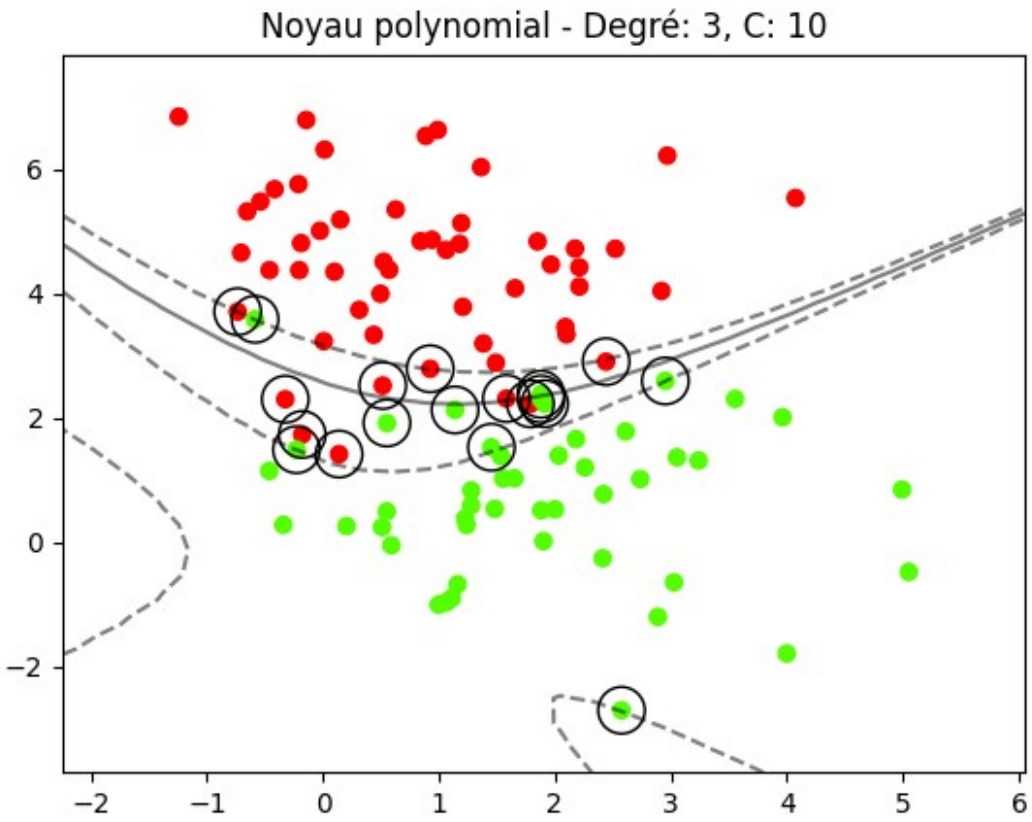

```
Noyau polynomial - Degré: 3, C: 0.1, Score: 0.89, Vecteurs supports:
31
```



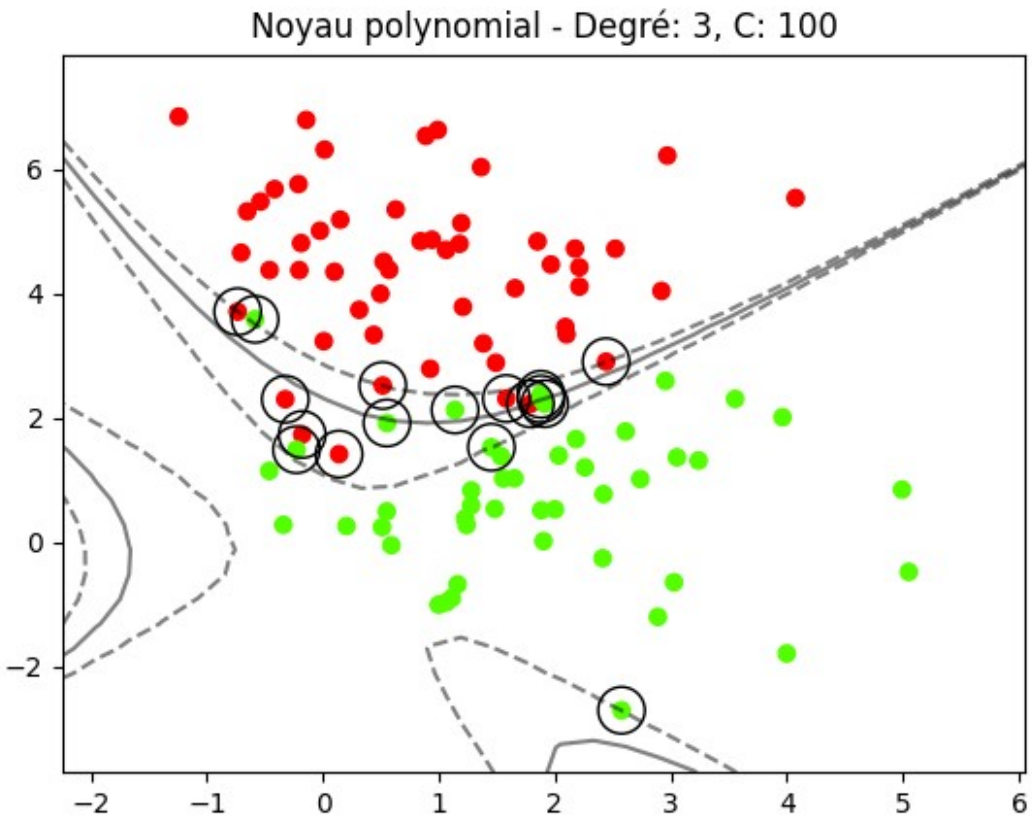

```
Noyau polynomial - Degré: 3, C: 1, Score: 0.89, Vecteurs supports: 21
```



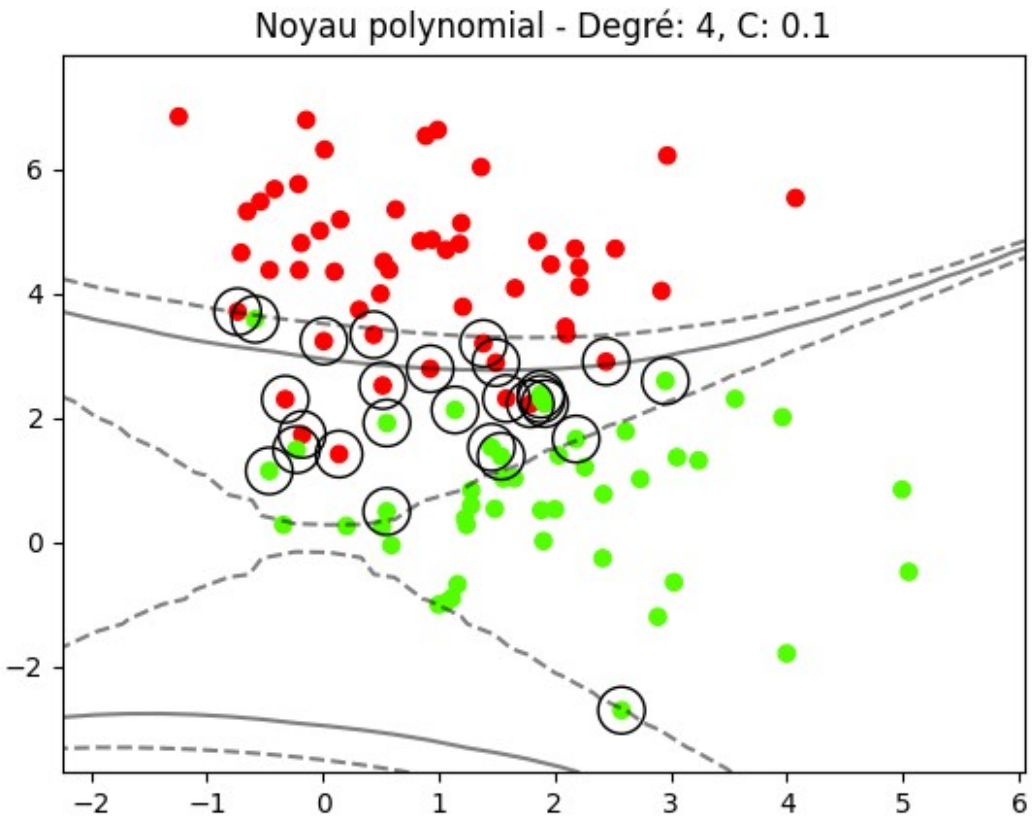

```
Noyau polynomial - Degré: 3, C: 10, Score: 0.90, Vecteurs supports: 19
```



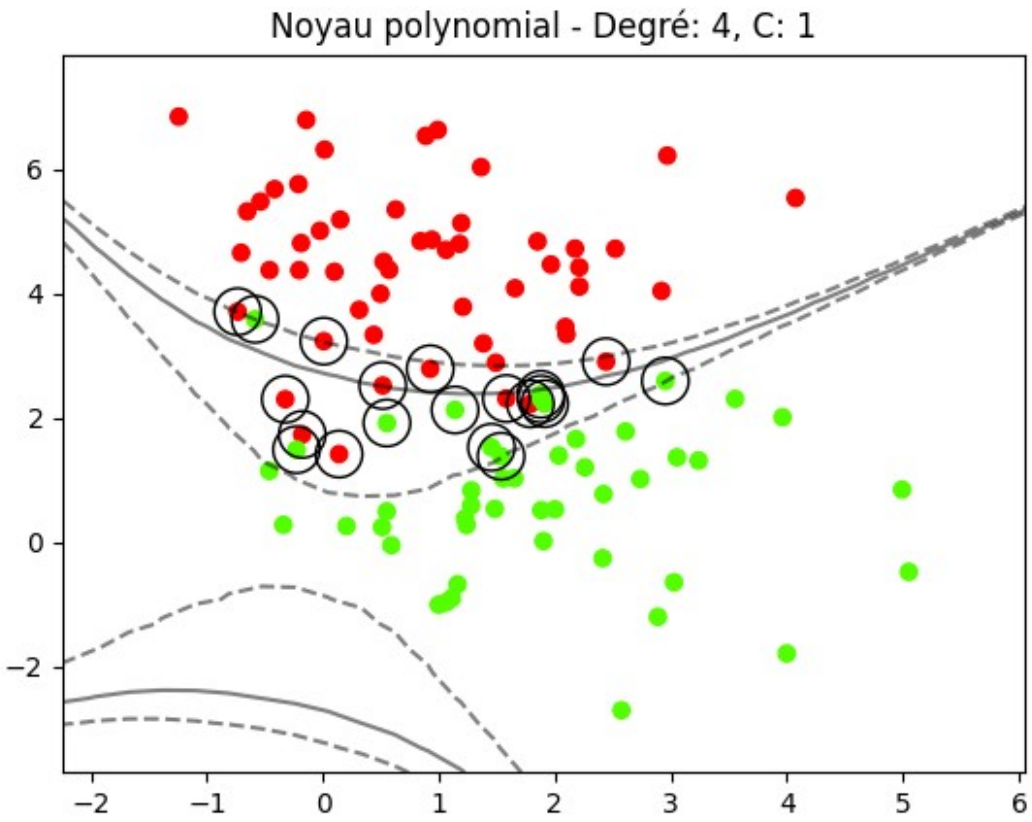

```
Noyau polynomial - Degré: 3, C: 100, Score: 0.88, Vecteurs supports:
17
```



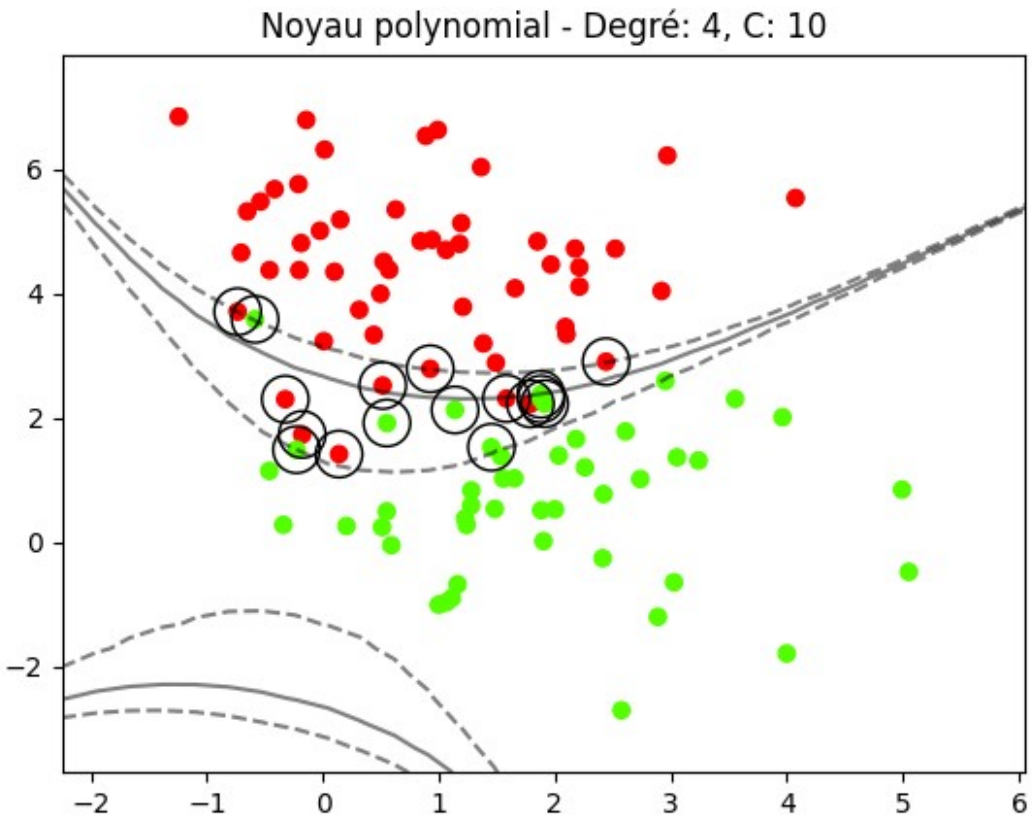

```
Noyau polynomial - Degré: 4, C: 0.1, Score: 0.89, Vecteurs supports:
27
```



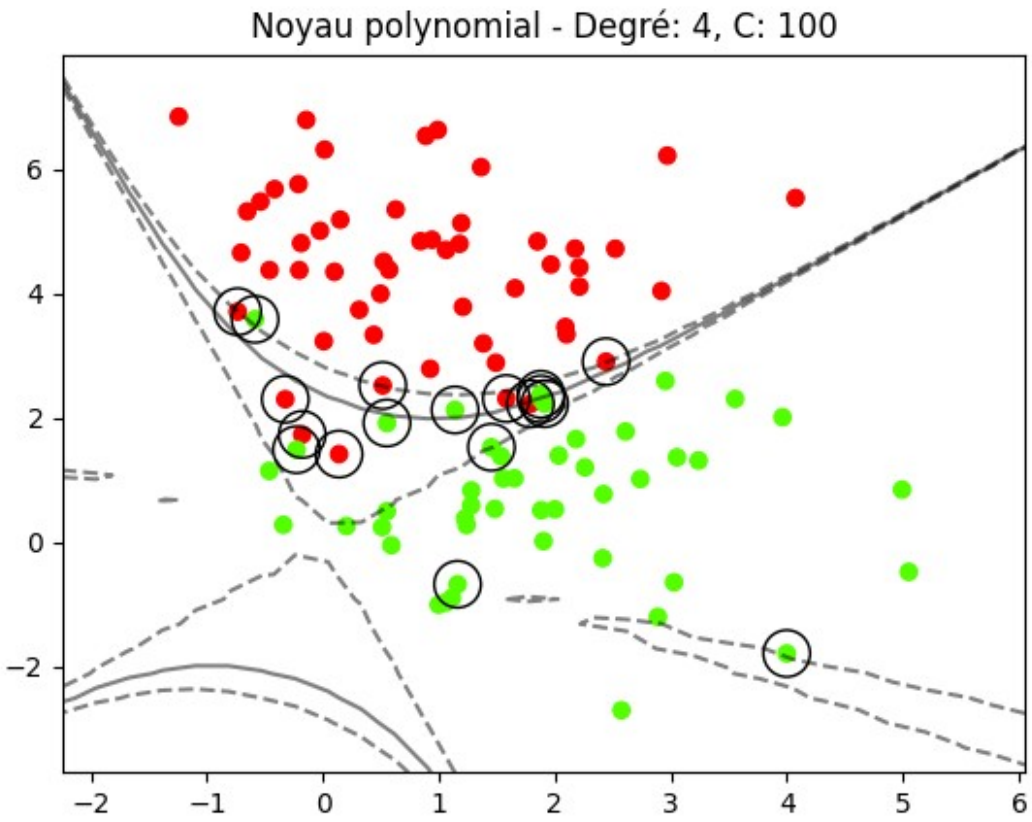

```
Noyau polynomial - Degré: 4, C: 1, Score: 0.89, Vecteurs supports: 20
```



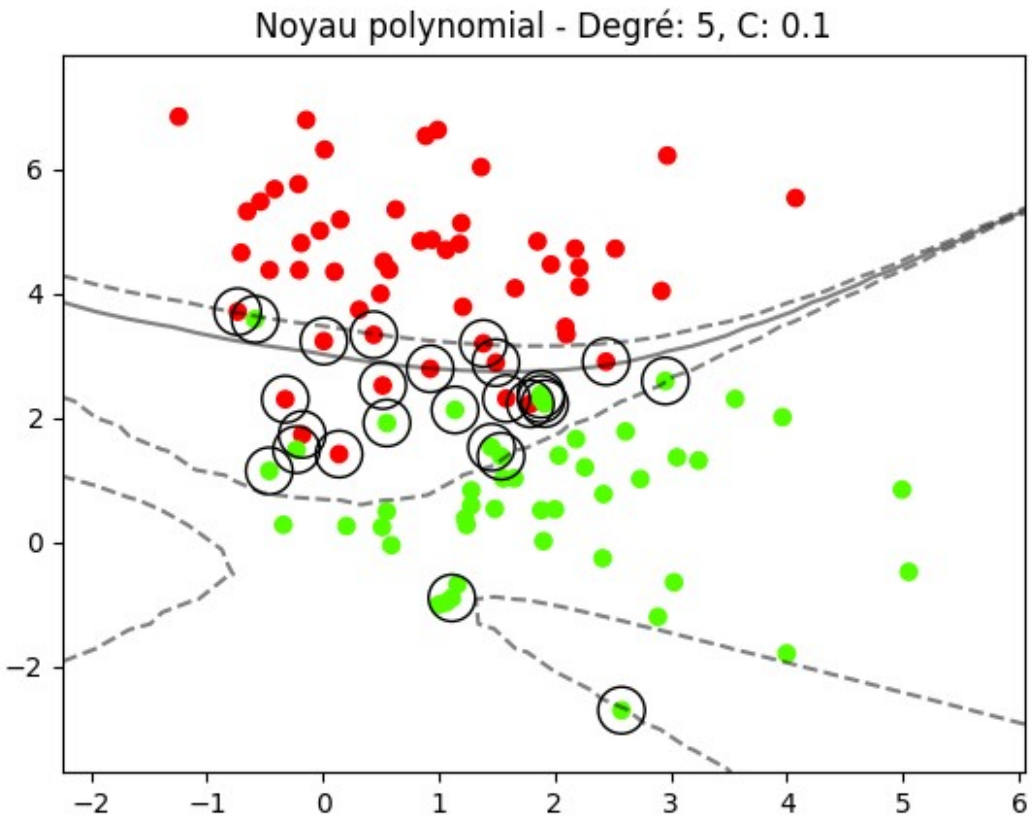

```
Noyau polynomial - Degré: 4, C: 10, Score: 0.89, Vecteurs supports: 17
```



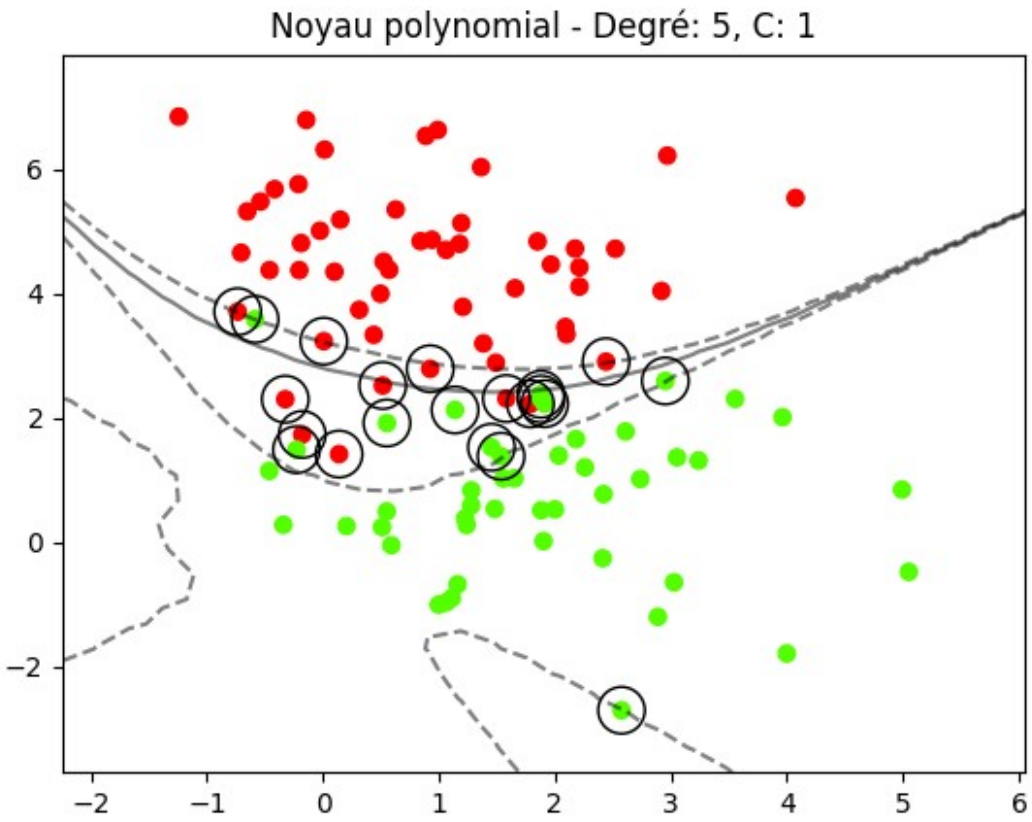

```
Noyau polynomial - Degré: 4, C: 100, Score: 0.87, Vecteurs supports:
18
```



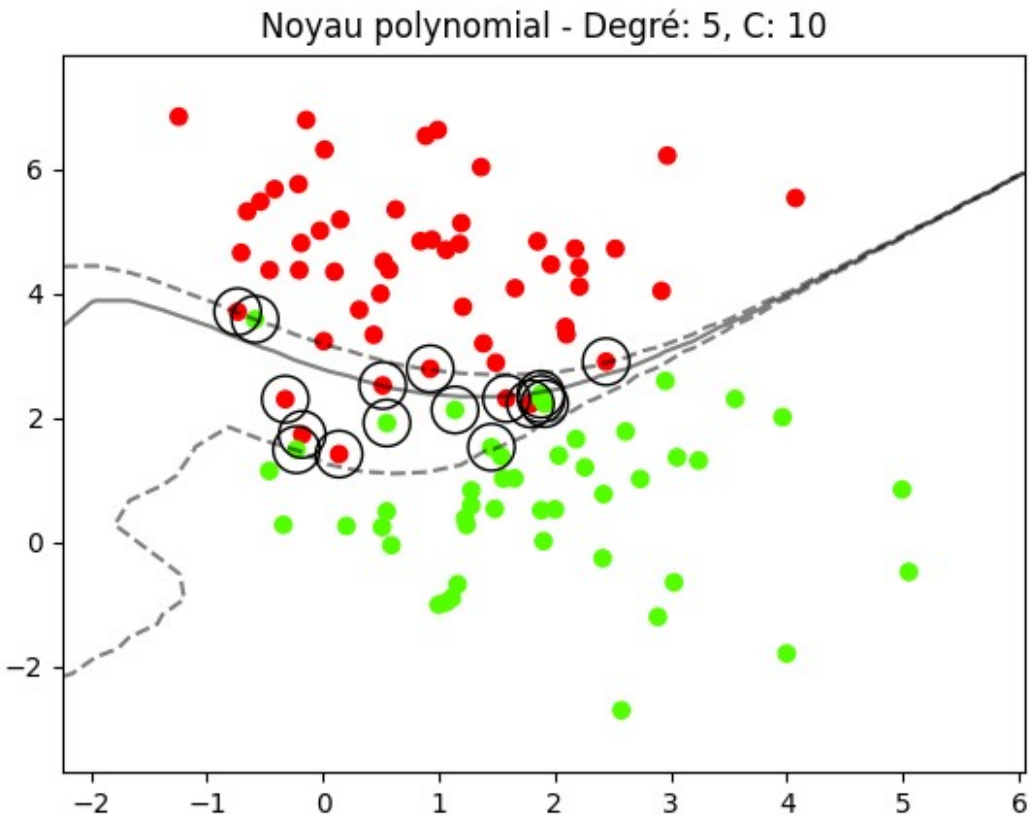

```
Noyau polynomial - Degré: 5, C: 0.1, Score: 0.89, Vecteurs supports:
26
```



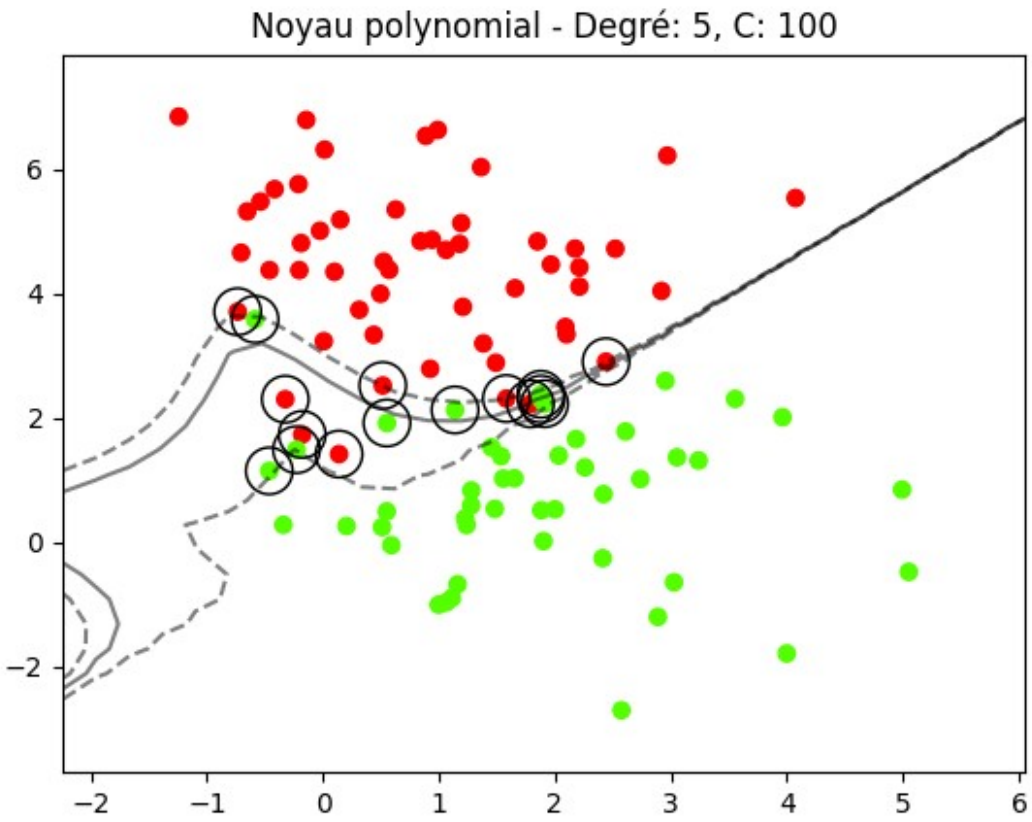

```
Noyau polynomial - Degré: 5, C: 1, Score: 0.90, Vecteurs supports: 21
```



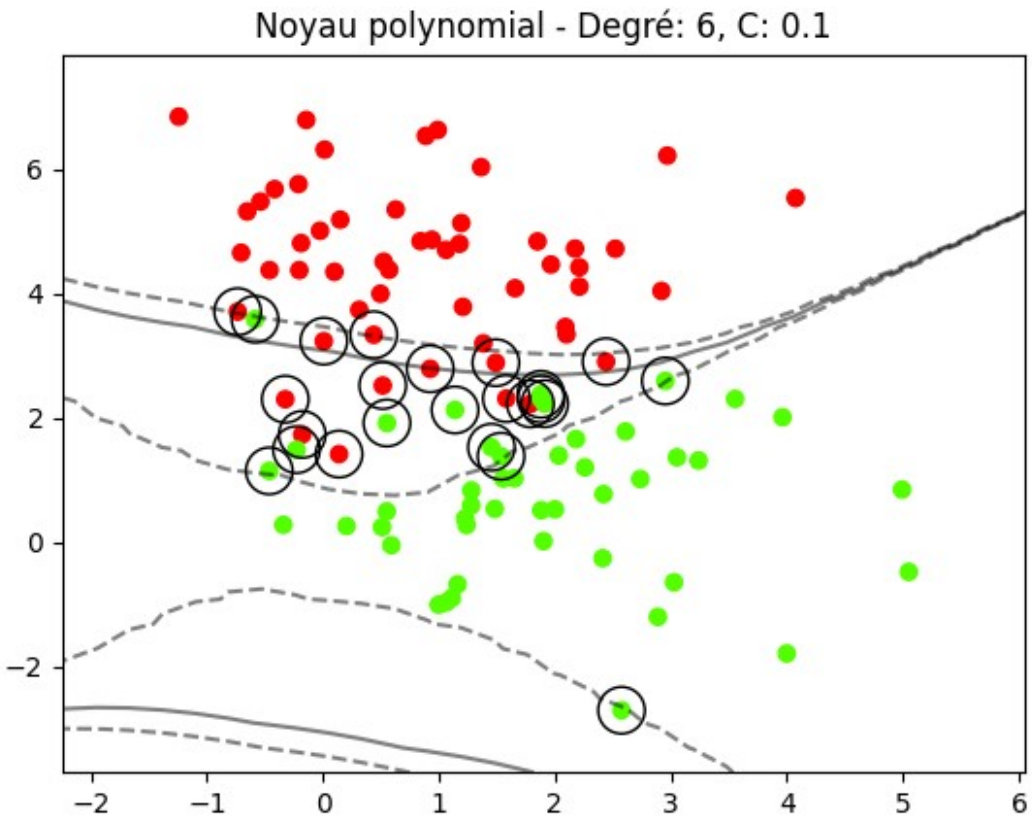

```
Noyau polynomial - Degré: 5, C: 10, Score: 0.89, Vecteurs supports: 17
```



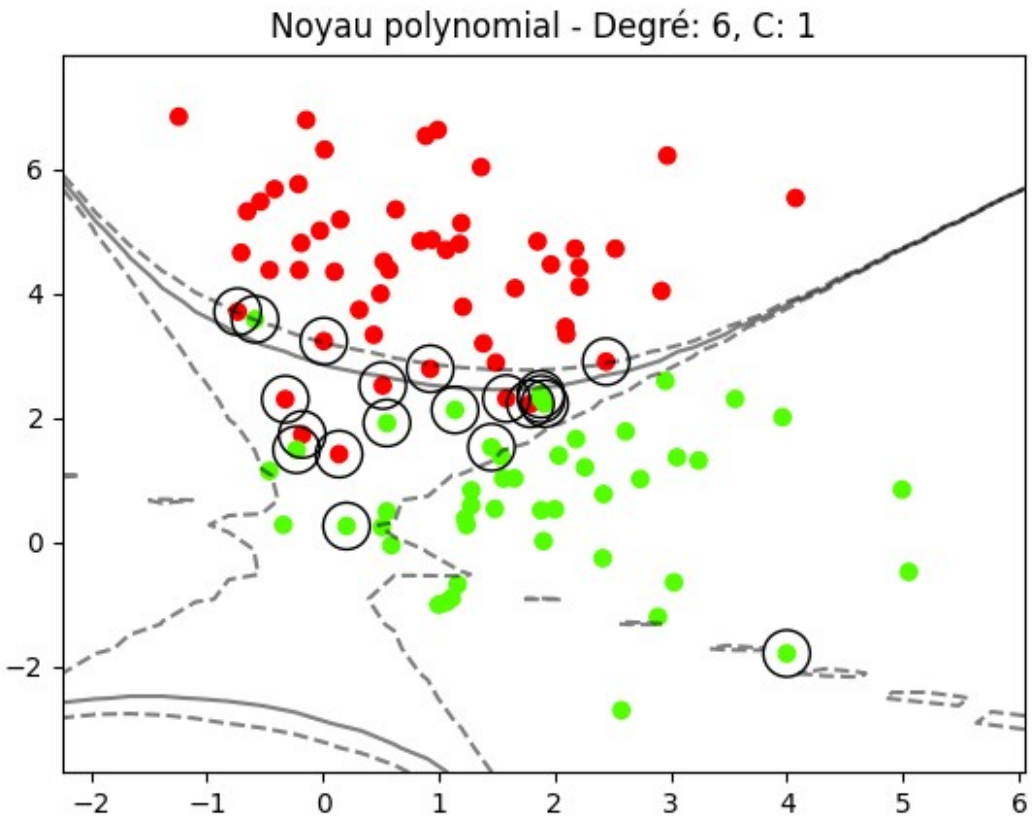

```
Noyau polynomial - Degré: 5, C: 100, Score: 0.88, Vecteurs supports:
16
```



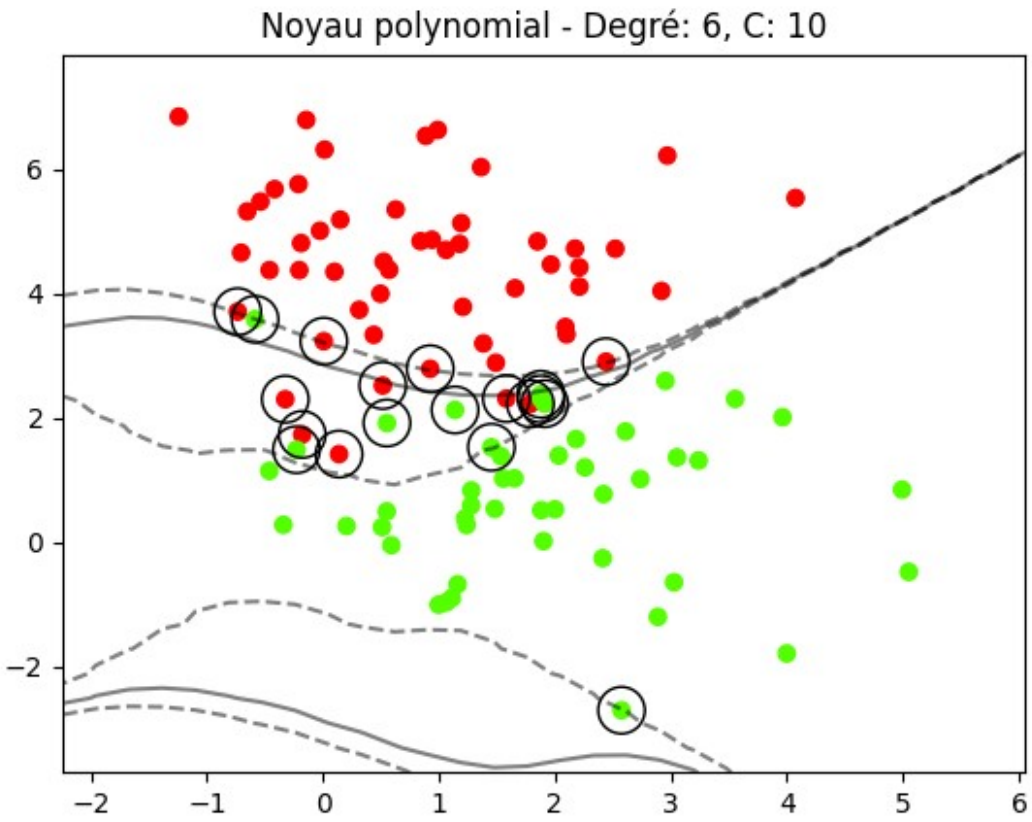

```
Noyau polynomial - Degré: 6, C: 0.1, Score: 0.89, Vecteurs supports:
24
```



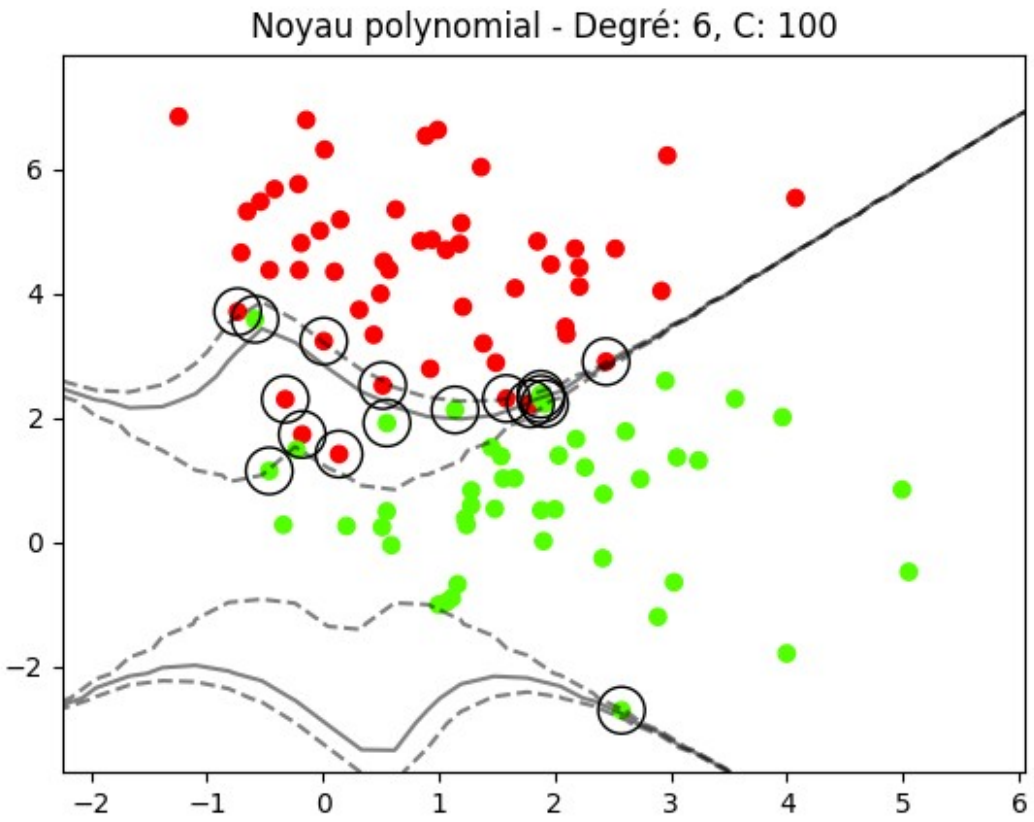

```
Noyau polynomial - Degré: 6, C: 1, Score: 0.90, Vecteurs supports: 20
```



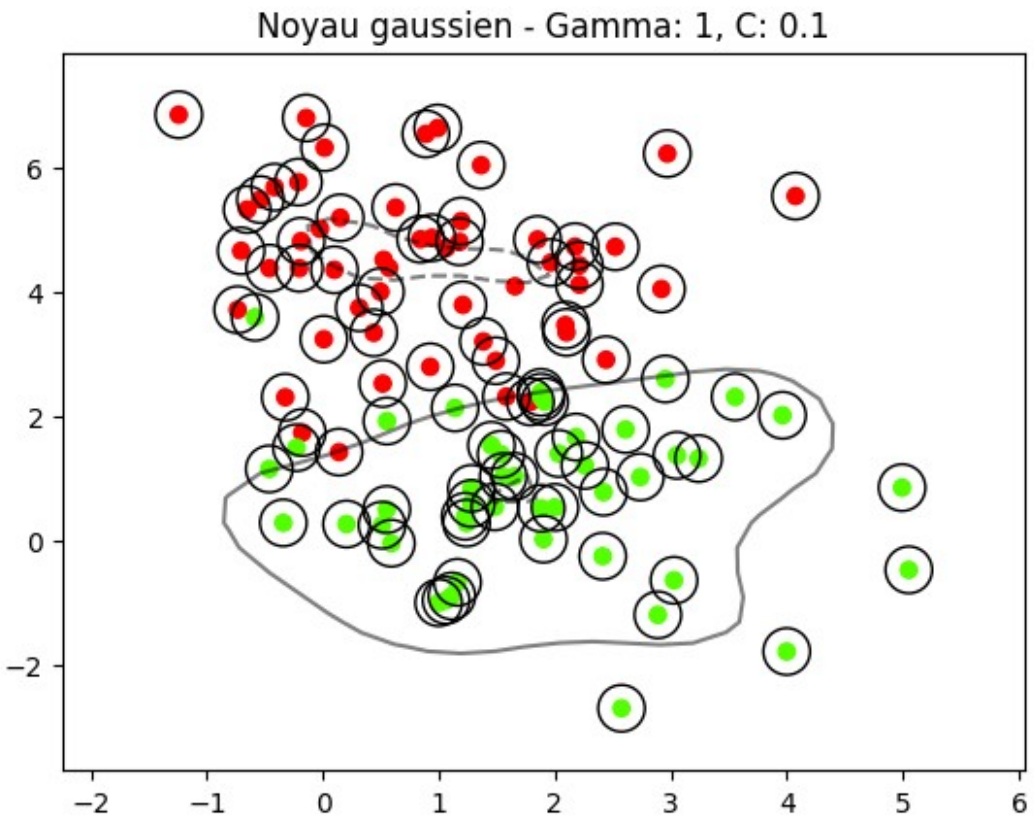

```
Noyau polynomial - Degré: 6, C: 10, Score: 0.90, Vecteurs supports: 19
```



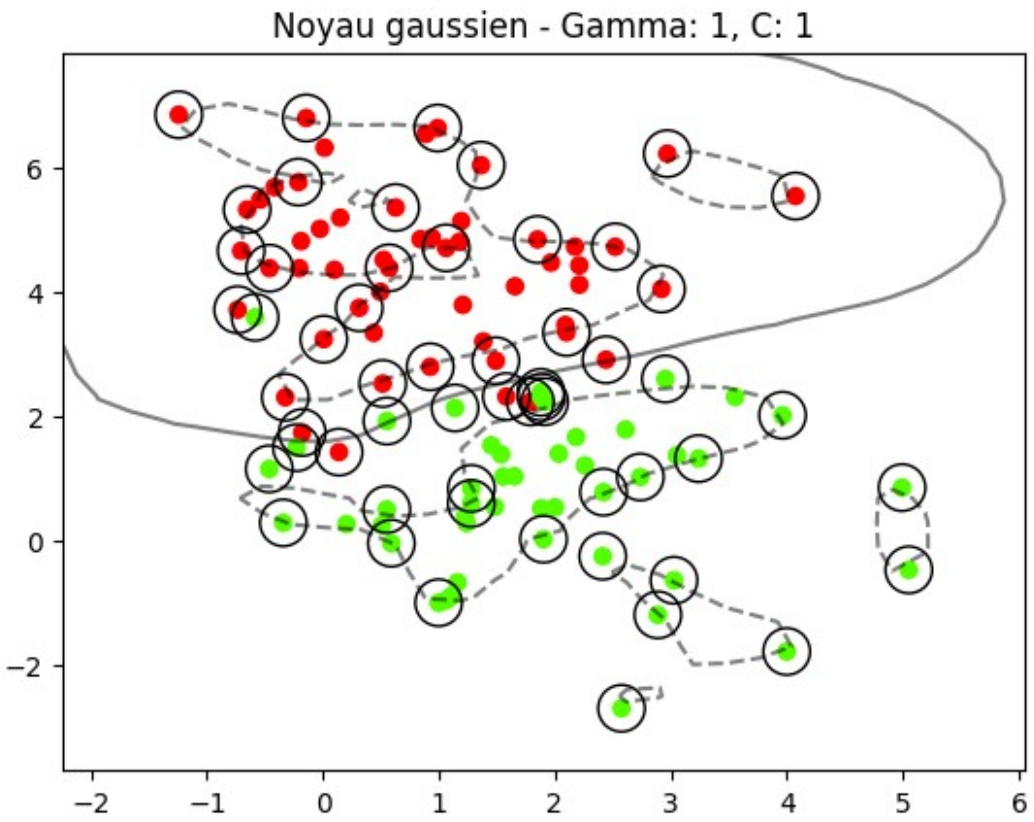

```
Noyau polynomial - Degré: 6, C: 100, Score: 0.89, Vecteurs supports:
17
```



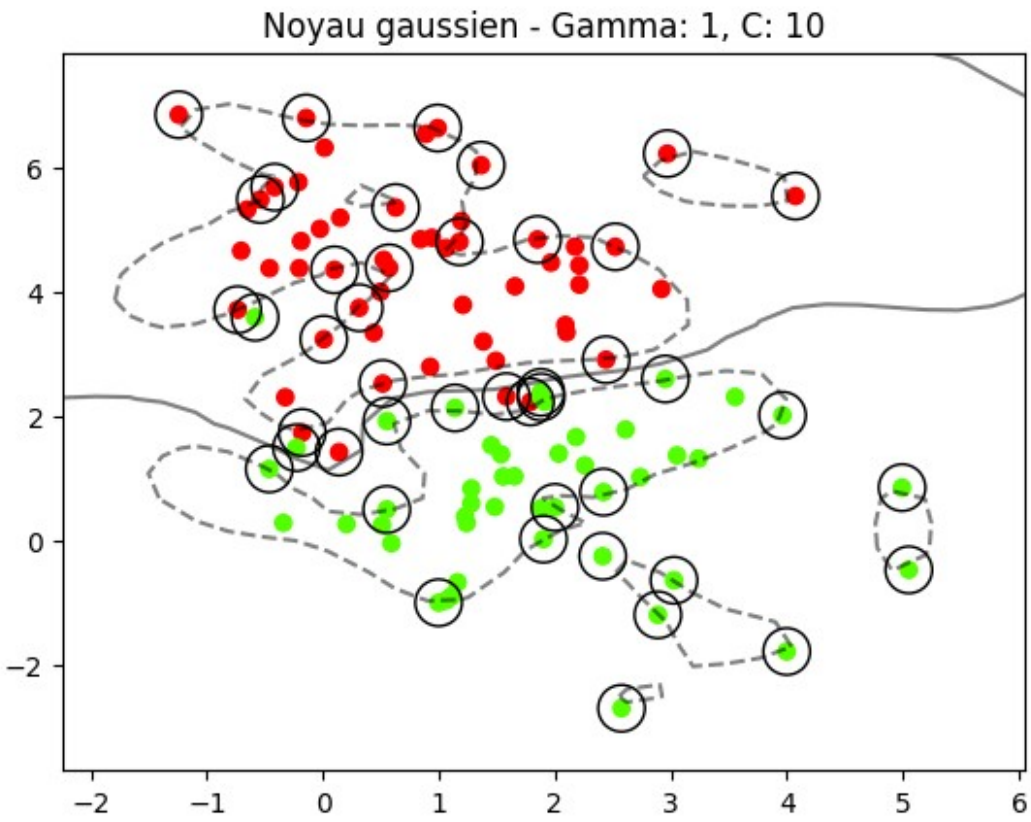

```
Noyau gaussien - Gamma: 1, C: 0.1, Score: 0.87, Vecteurs supports: 95
```



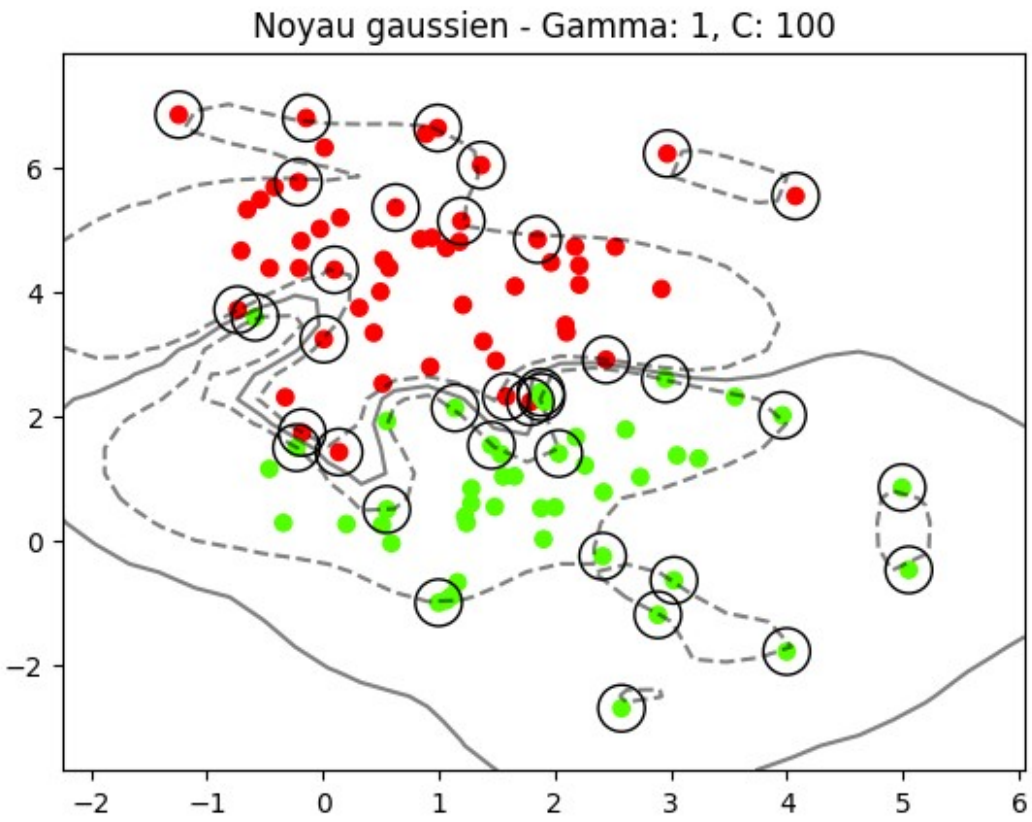

```
Noyau gaussien - Gamma: 1, C: 1, Score: 0.90, Vecteurs supports: 56
```



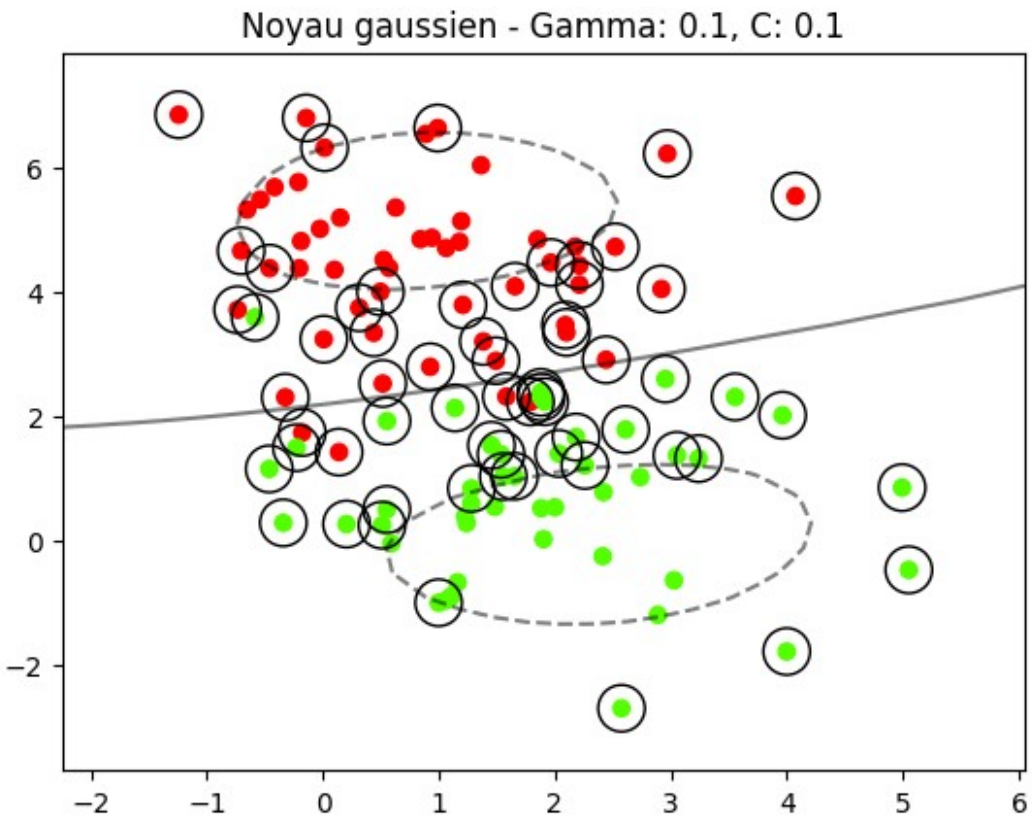

```
Noyau gaussien - Gamma: 1, C: 10, Score: 0.90, Vecteurs supports: 44
```



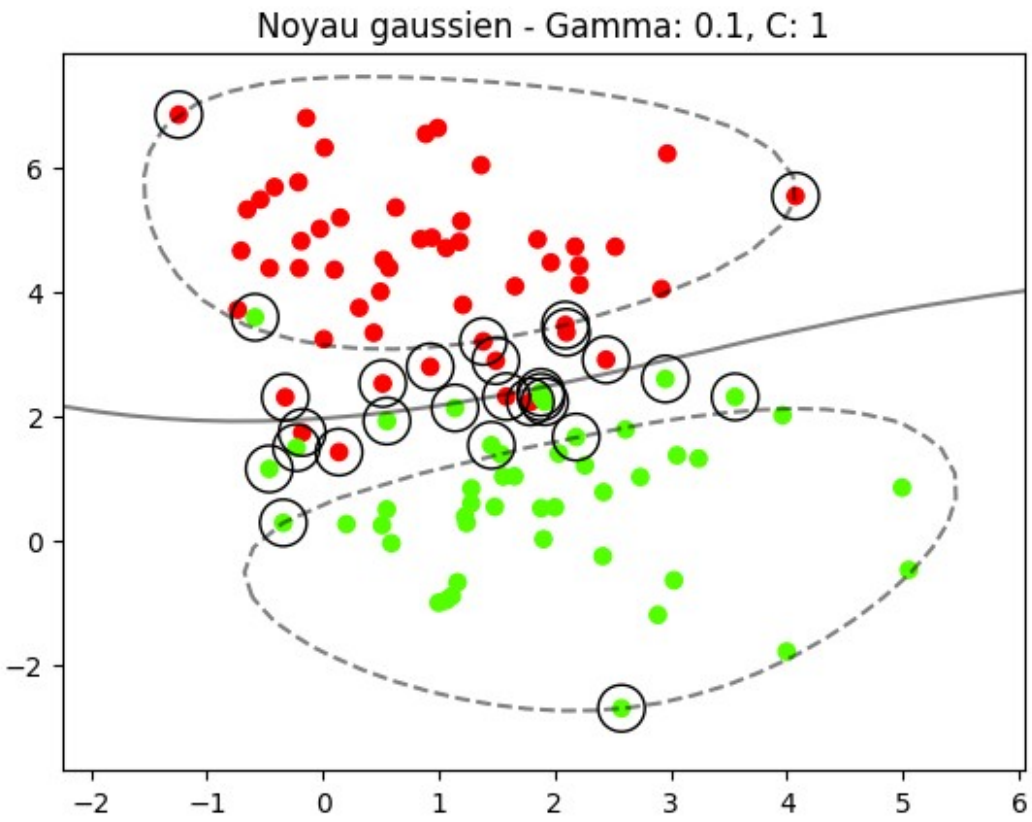

```
Noyau gaussien - Gamma: 1, C: 100, Score: 0.87, Vecteurs supports: 36
```



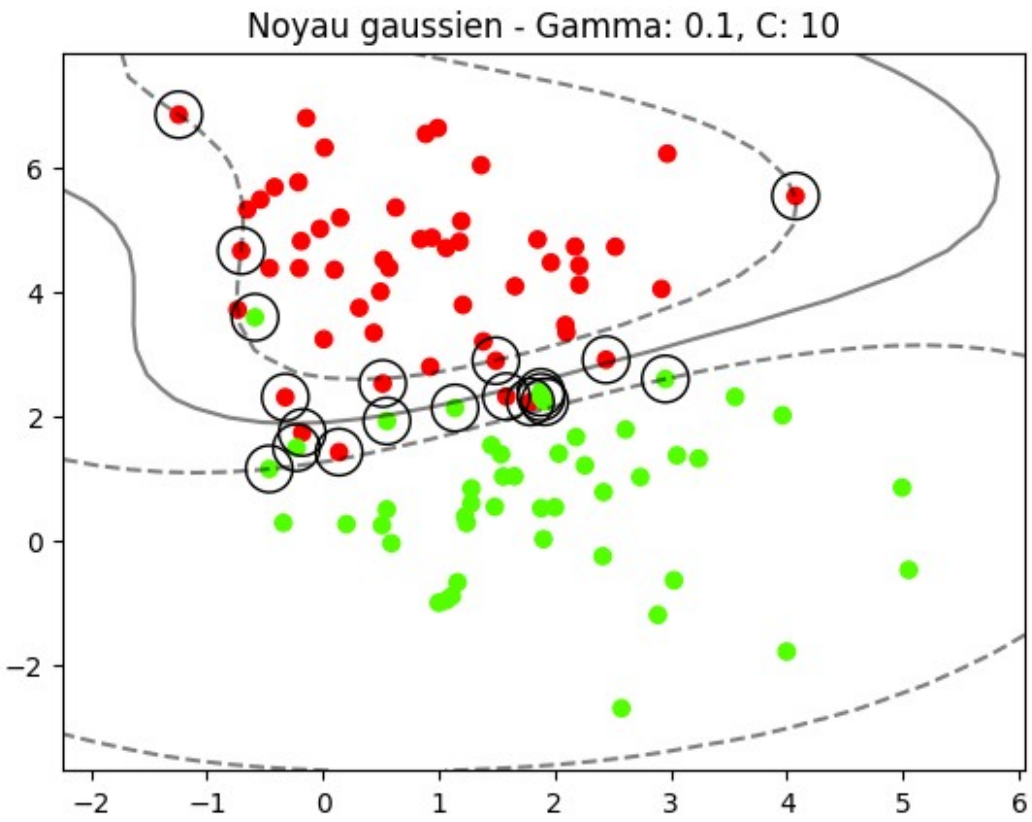

```
Noyau gaussien - Gamma: 0.1, C: 0.1, Score: 0.90, Vecteurs supports:
63
```



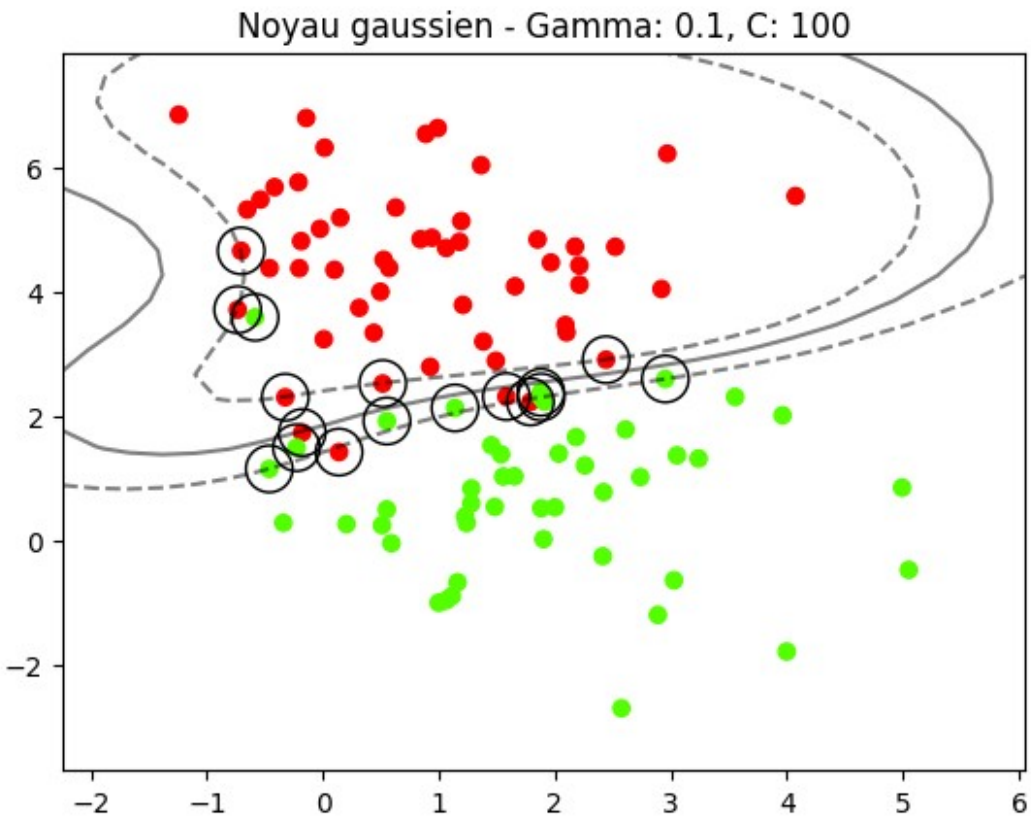

```
Noyau gaussien - Gamma: 0.1, C: 1, Score: 0.92, Vecteurs supports: 28
```



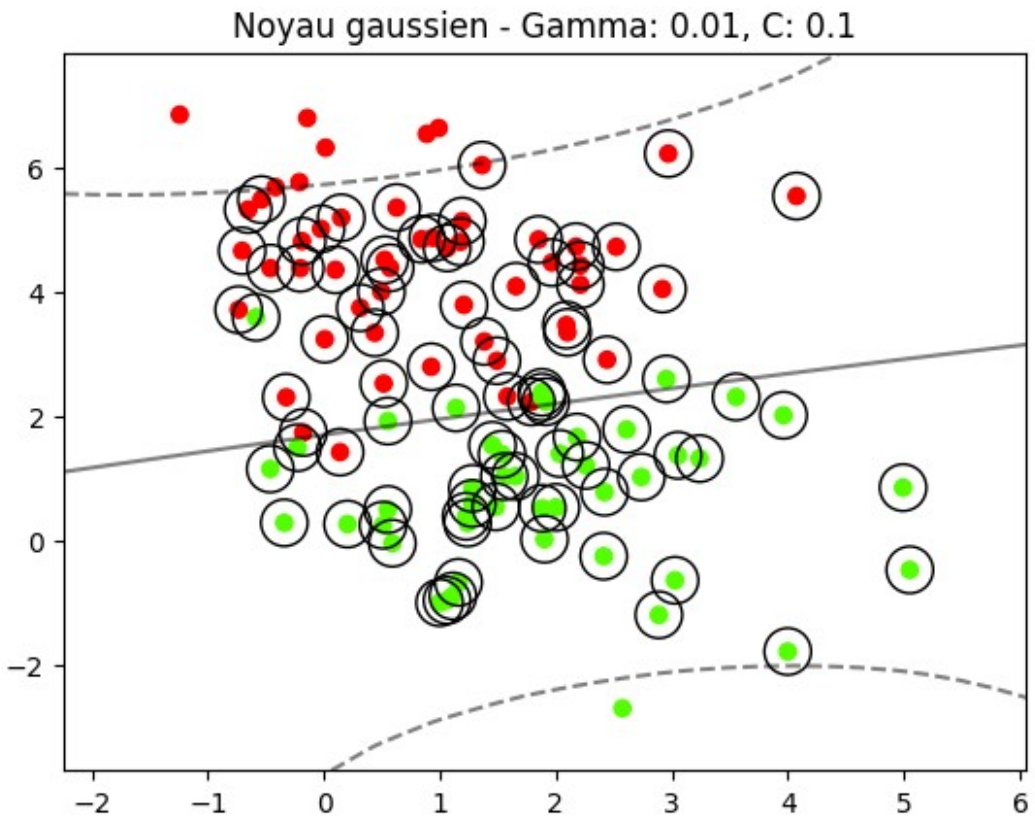

```
Noyau gaussien - Gamma: 0.1, C: 10, Score: 0.90, Vecteurs supports: 20
```



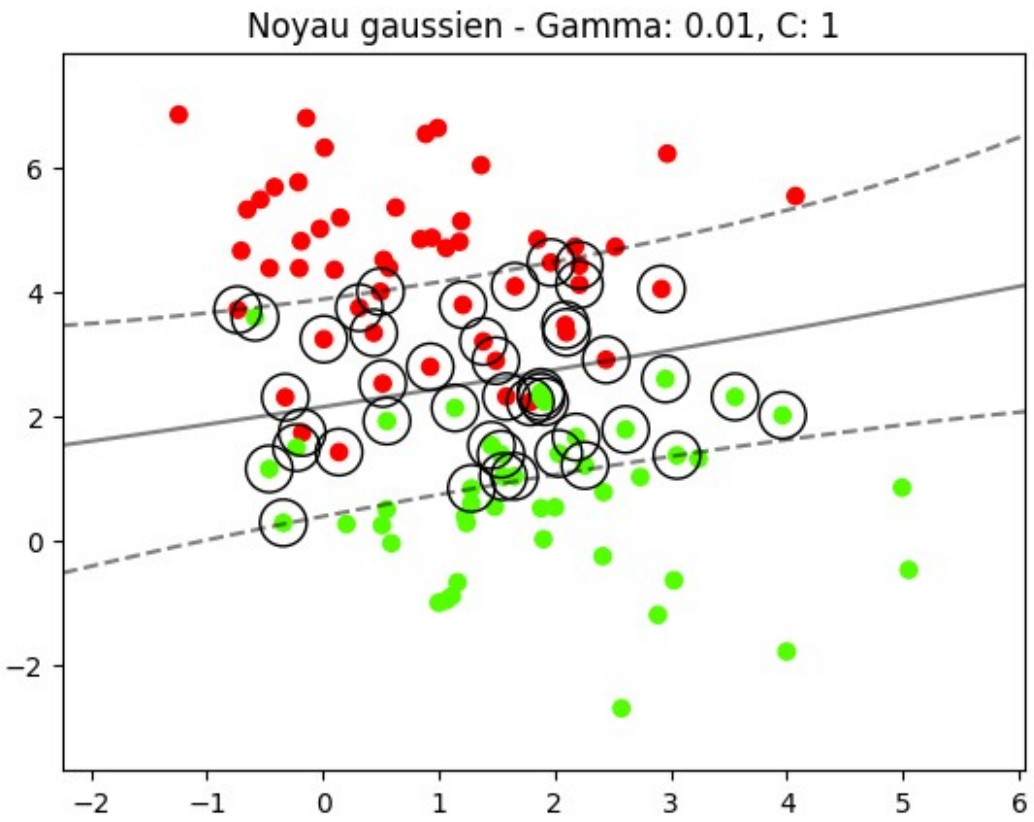

```
Noyau gaussien - Gamma: 0.1, C: 100, Score: 0.91, Vecteurs supports:
17
```



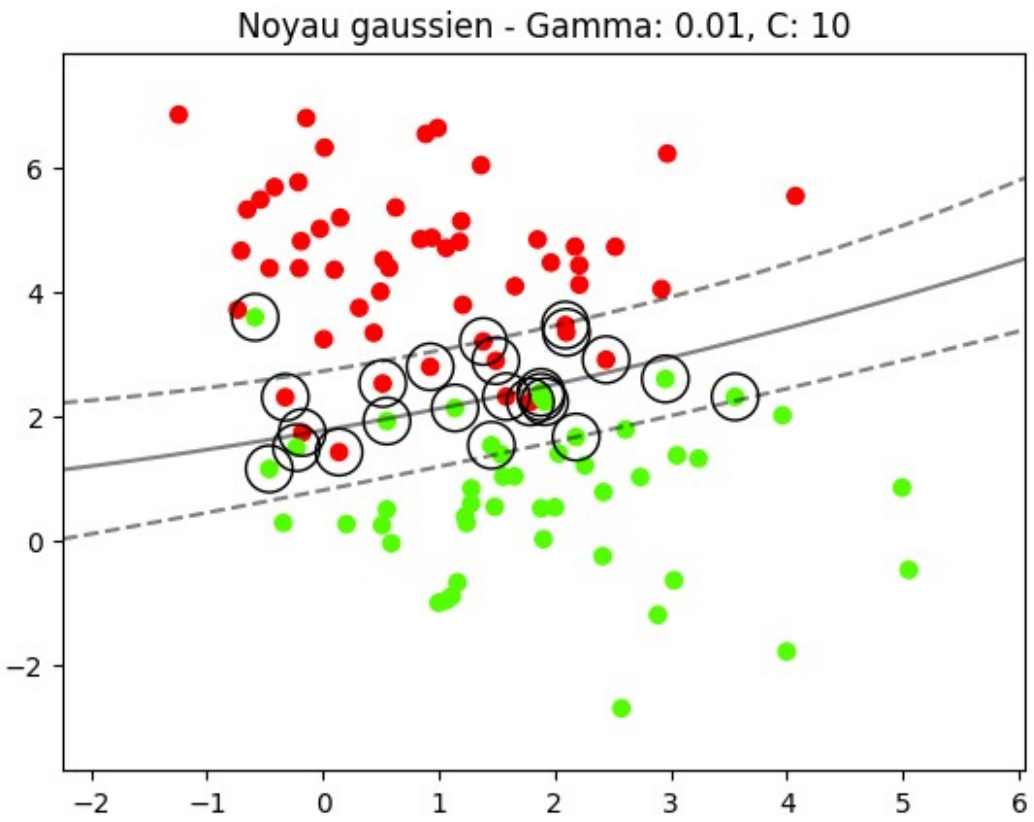

```
Noyau gaussien - Gamma: 0.01, C: 0.1, Score: 0.92, Vecteurs supports:
92
```



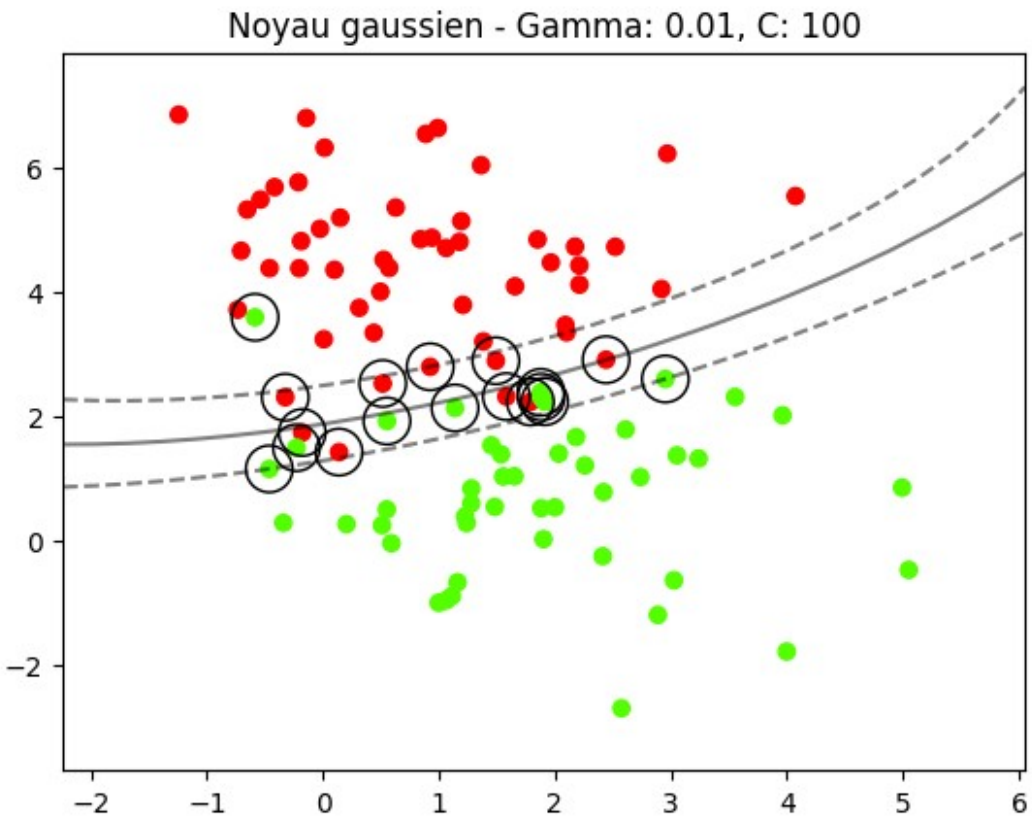

```
Noyau gaussien - Gamma: 0.01, C: 1, Score: 0.90, Vecteurs supports: 45
```



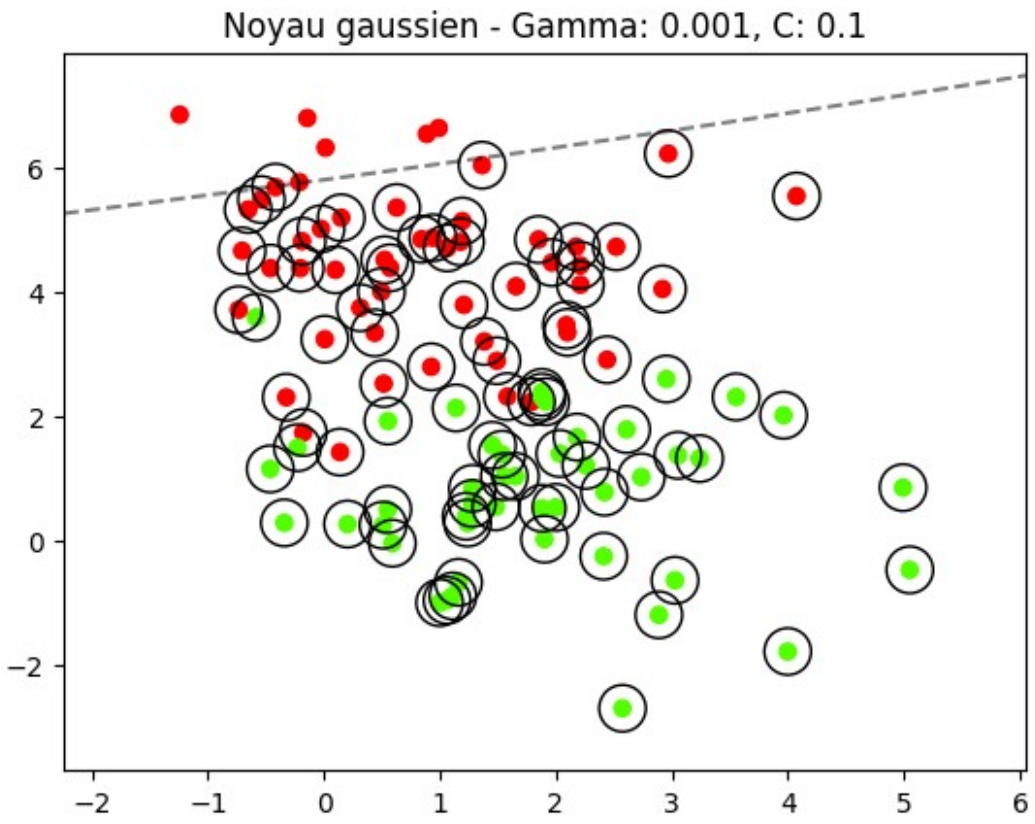

```
Noyau gaussien - Gamma: 0.01, C: 10, Score: 0.91, Vecteurs supports:
24
```



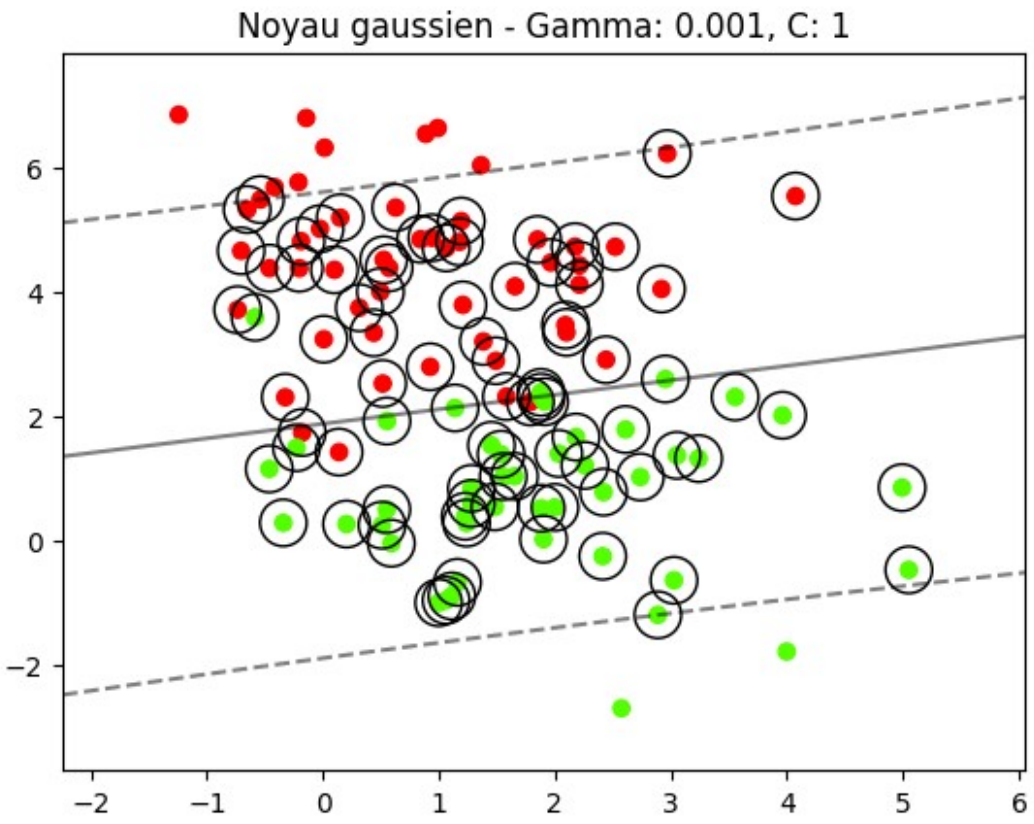

```
Noyau gaussien - Gamma: 0.01, C: 100, Score: 0.90, Vecteurs supports:
18
```



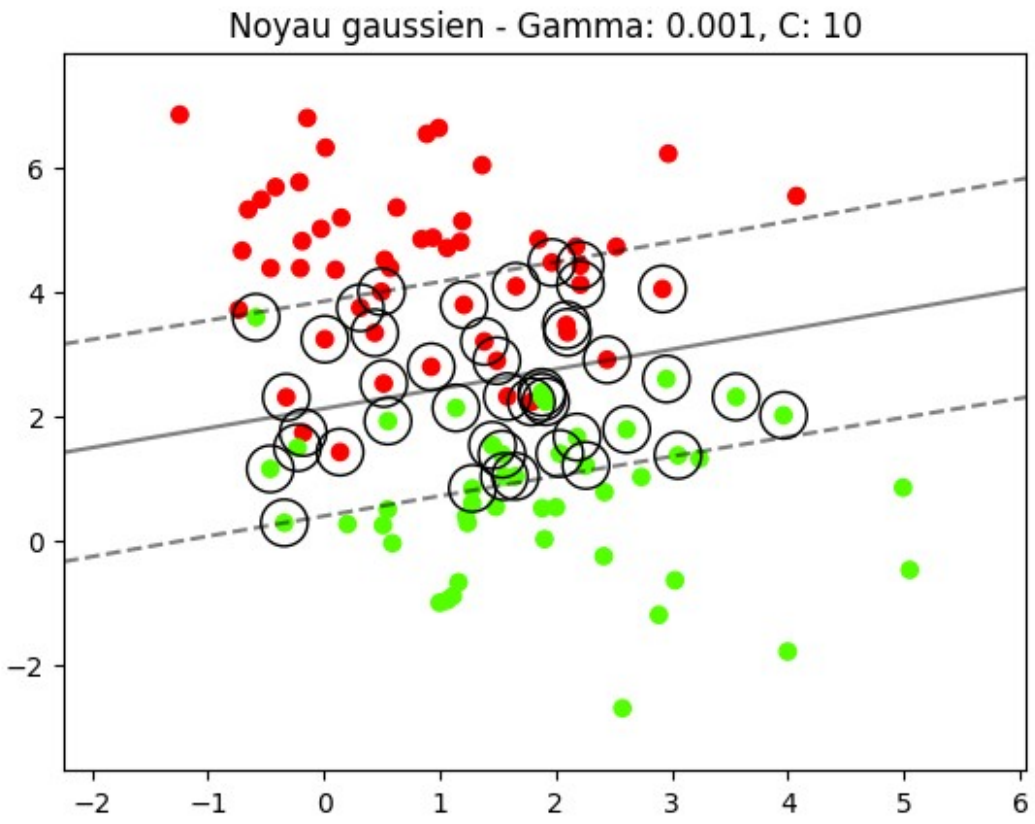

```
Noyau gaussien - Gamma: 0.001, C: 0.1, Score: 0.47, Vecteurs supports:
94
```



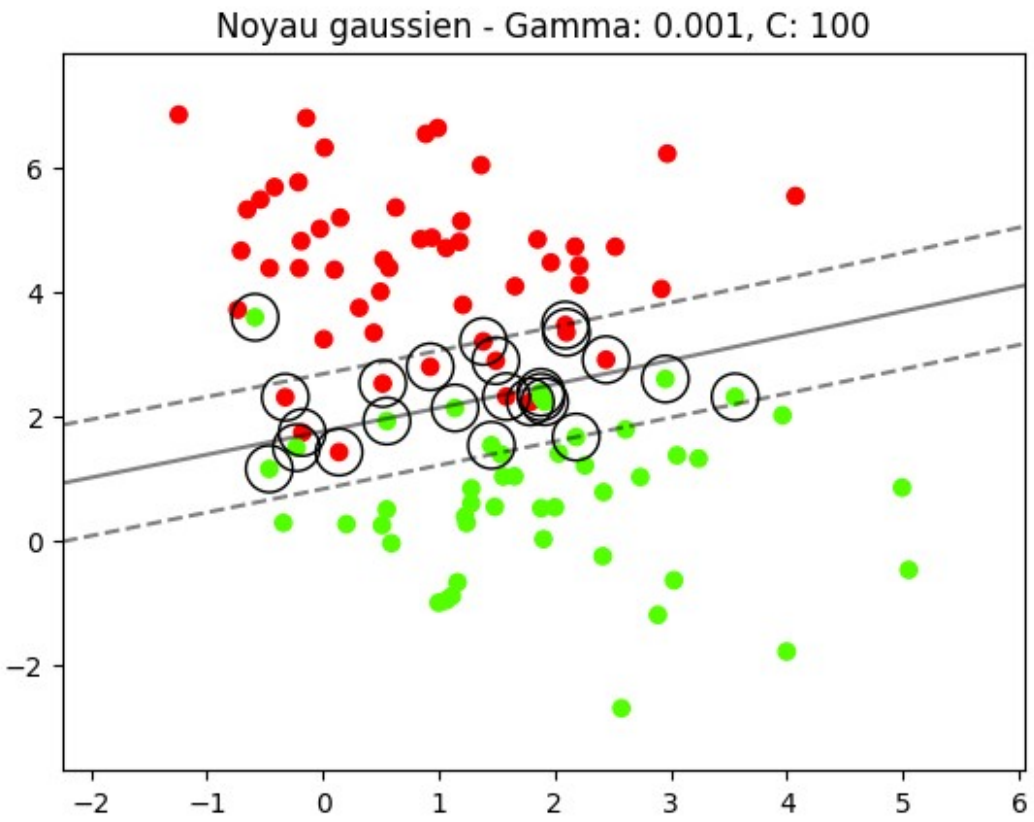

```
Noyau gaussien - Gamma: 0.001, C: 1, Score: 0.92, Vecteurs supports:
90
```




```
Noyau gaussien - Gamma: 0.001, C: 10, Score: 0.89, Vecteurs supports:
44
```




```
Noyau gaussien - Gamma: 0.001, C: 100, Score: 0.91, Vecteurs supports:
24
```




# Deuxième partie : un traitement (presque) complet

## Préparation des données

Nous allons utiliser un jeu de données réel - tiré de Tsanas & Xifara : Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools, Energy and Buildings, Vol. 49, pp. 560-567, 2012 - qui vous est fourni avec l'énoncé.

Les 8 premières colonnes correspondent aux attributs descriptifs et les deux dernières, aux charges de chauffage et de climatisation (dans cet ordre). Pour les utiliser en Python, vous pourrez vous servir du code suivant :

```
import pandas as pd

chemin_du_fichier = 'data.csv'

# Charger les données depuis le fichier
donnees = pd.read_csv(chemin_du_fichier, delimiter=' ')

# Sélectionner les 8 premières colonnes pour les attributs descriptifs
attributs_descriptifs = donnees.iloc[:, :8]

# Sélectionner les deux dernières colonnes pour les charges de
chauffage et de climatisation
```

```
charges_chauffage = donnees.iloc[:, -2]
charges_climatisation = donnees.iloc[:, -1]

# Afficher les attributs descriptifs, les charges de chauffage et de
climatisation
print("Attributs Descriptifs:\n", attributs_descriptifs)
print("\nCharges de Chauffage:\n", charges_chauffage)
print("\nCharges de Climatisation:\n", charges_climatisation)
```

```
Attributs Descriptifs:
      0.98  514.50  294.00  110.25  7.00  2  0.00  0
0    0.98   514.5   294.0  110.25   7.0  3   0.0  0
1    0.98   514.5   294.0  110.25   7.0  4   0.0  0
2    0.98   514.5   294.0  110.25   7.0  5   0.0  0
3    0.90   563.5   318.5  122.50   7.0  2   0.0  0
4    0.90   563.5   318.5  122.50   7.0  3   0.0  0
..    ...     ...     ...     ...   ... ..   ... ..
762  0.64   784.0   343.0  220.50   3.5  5   0.4  5
763  0.62   808.5   367.5  220.50   3.5  2   0.4  5
764  0.62   808.5   367.5  220.50   3.5  3   0.4  5
765  0.62   808.5   367.5  220.50   3.5  4   0.4  5
766  0.62   808.5   367.5  220.50   3.5  5   0.4  5

[767 rows x 8 columns]

Charges de Chauffage:
 0      15.55
1       15.55
2       15.55
3       20.84
4       21.46
        ...
762     17.88
763     16.54
764     16.44
765     16.48
766     16.64
Name: 15.55, Length: 767, dtype: float64

Charges de Climatisation:
 0      21.33
1       21.33
2       21.33
3       28.28
4       25.38
        ...
762     21.40
763     16.88
764     17.11
765     16.61
```

```
766    16.03
Name: 21.33, Length: 767, dtype: float64
```

Le problème initial, tel que présenté ici, est un problème de régression. Nous allons d'abord le transformer en problème de classification. Par une méthode de clustering, on veut répartir les charges de chauffage et de climatisation en 3 classes : faibles, moyennes, élevées.

```
from sklearn.cluster import KMeans
# La suite ? il s'agit de définir un classifieur du k-means avec k=3
# et d'utiliser la méthode 'fit' sur les 2 ensembles de valeurs Y
# Le seul trick : les Y sont des vecteurs et les classifieurs sklearn
ont besoin d'array :
# il faut les reshaper : Yheat_vector = Yheat.reshape(-1,1)
# Après apprentissage du kmeans, les classes des données utilisées
sont stockées dans mon_classifieur.labels_
# Concaténez les vecteurs Yheat et Ycool pour créer une seule matrice
Y_matrix = np.column_stack((charges_chauffage,charges_climatisation ))

# Définissez un classifieur K-Means avec 3 clusters (k=3)
kmeans_classifier = KMeans(n_clusters=3)

# Utilisez la méthode 'fit' pour entraîner le modèle sur la matrice Y
kmeans_classifier.fit(Y_matrix)

# Après l'apprentissage du K-Means, les classes des données utilisées
sont stockées dans kmeans_classifier.labels_
y_clusters = kmeans_classifier.labels_
y_clusters
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/
_kmeans.py:870: FutureWarning: The default value of `n_init` will
change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly
to suppress the warning
  warnings.warn(
```

```
array([1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1, 1, 1, 0, 2, 2, 0, 2, 2,
2,
       2, 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, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2,
       2, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2,
       2, 2, 2, 2, 2, 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, 2, 2, 2, 2, 2, 2,
2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 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, 2, 2,
2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 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, 2, 2, 2, 2, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 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, 2, 2, 2, 2, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2,
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, 2, 2, 2, 2, 0, 0, 0, 2, 2, 2, 2, 2,
2,
      2, 2, 2, 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, 2, 2, 2, 2, 2, 0, 0, 0,
2,
      2, 2, 2, 2, 2, 2, 2, 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, 2, 2, 2, 2,
0,
      2, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 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,
0,
      0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 2, 2, 2, 2, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 2, 2, 2, 2, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2,
2,
      2, 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, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0,
0,
      2, 2, 2, 2, 2, 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, 0, 0, 0, 0, 0, 0,
0,
       0, 2, 2, 0, 0, 2, 2, 2, 2, 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],
      dtype=int32)
```

## Apprentissage

Nous voulons comparer plusieurs méthodes d'apprentissage :

1. Les arbres de décision (DecisionTreeClassifier de la classe sklearn.tree, hyperparamètre à régler : max_depth)
2. SVM à noyau gaussien (SVC avec kernel='rbf' de la classe sklearn.svm, hyperparamètre à régler : gamma)
3. SVM à noyau polynomial (SVC avec kernel='poly' de la classe sklearn.svm, hyperparamètre à régler : degree)

Ecrivez le code permettant de :

1. Séparer les données en un échantillon d'apprentissage et un échantillon de test (80/20)
2. Sélectionner les meilleurs valeurs des hyperparamètres sur l'échantillon d'apprentissage par validation croisée en utilisant 10 folders

```
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder

# Convertir les cibles en entiers si elles sont continues ou nominales
# Cela est nécessaire car DecisionTreeClassifier attend des cibles
discrètes pour la classification
#label_encoder = LabelEncoder()
#Y_matrix = label_encoder.fit_transform(Y_matrix)

# Diviser les données en ensembles d'apprentissage et de test (80/20)
X_train, X_test, y_train, y_test =
train_test_split(attributs_descriptifs, y_clusters, test_size=0.2,
random_state=42)

# Paramètres pour la recherche en grille
param_grid_tree = {'max_depth': [3, 5, 10, None]}
param_grid_svm_rbf = {'gamma': [0.001, 0.01, 0.1, 1]}
param_grid_svm_poly = {'degree': [2, 3, 4, 5]}

# Recherche en grille avec validation croisée pour les arbres de
décision
grid_search_tree = GridSearchCV(DecisionTreeClassifier(),
```

```
param_grid_tree, cv=10)
grid_search_tree.fit(X_train, y_train)
print(f'Meilleur score (arbre de décision) :
{grid_search_tree.best_score_}')
print(f'Meilleurs hyperparamètres (arbre de décision) :
{grid_search_tree.best_params_}')

# Recherche en grille avec validation croisée pour SVM à noyau
gaussien
grid_search_svm_rbf = GridSearchCV(SVC(kernel='rbf'),
param_grid_svm_rbf, cv=10)
grid_search_svm_rbf.fit(X_train, y_train)
print(f'Meilleur score (SVM RBF) : {grid_search_svm_rbf.best_score_}')
print(f'Meilleurs hyperparamètres (SVM RBF) :
{grid_search_svm_rbf.best_params_}')

# Recherche en grille avec validation croisée pour SVM à noyau
polynomial
grid_search_svm_poly = GridSearchCV(SVC(kernel='poly'),
param_grid_svm_poly, cv=10)
grid_search_svm_poly.fit(X_train, y_train)
print(f'Meilleur score (SVM polynomial) :
{grid_search_svm_poly.best_score_}')
print(f'Meilleurs hyperparamètres (SVM polynomial) :
{grid_search_svm_poly.best_params_}')
```

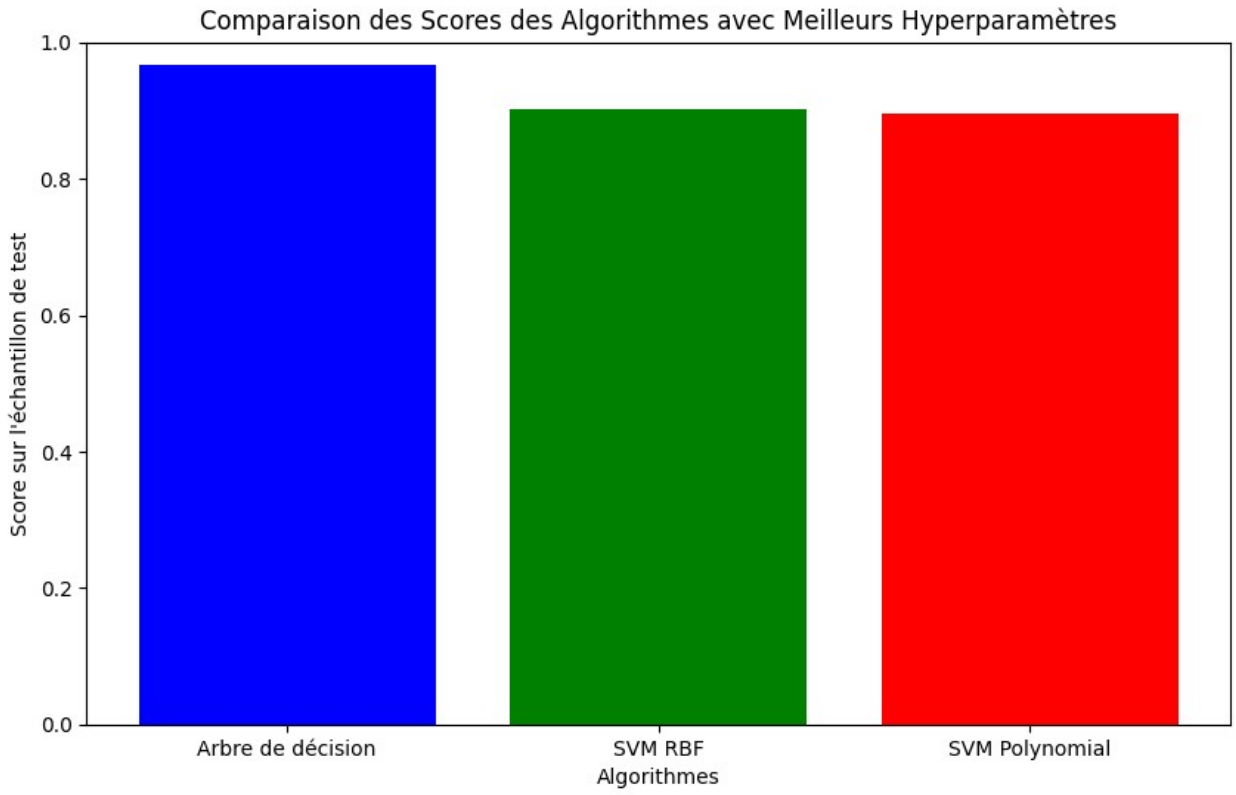
```
Meilleur score (arbre de décision) : 0.9673717609730301
Meilleurs hyperparamètres (arbre de décision) : {'max_depth': 10}
Meilleur score (SVM RBF) : 0.902141723955579
Meilleurs hyperparamètres (SVM RBF) : {'gamma': 1}
Meilleur score (SVM polynomial) : 0.892332099418297
Meilleurs hyperparamètres (SVM polynomial) : {'degree': 5}
```

## Analyse des résultats

Afficher sur une courbe les scores de chacun des algorithmes avec la meilleure valeur d'hyperparamètre possible sur l'échantillon de test.

```
import matplotlib.pyplot as plt

# Calculer les scores sur l'échantillon de test
score_tree = grid_search_tree.best_estimator_.score(X_test, y_test)
score_svm_rbf = grid_search_svm_rbf.best_estimator_.score(X_test,
y_test)
score_svm_poly = grid_search_svm_poly.best_estimator_.score(X_test,
y_test)

# Noms des algorithmes
noms_algos = ['Arbre de décision', 'SVM RBF', 'SVM Polynomial']
```

```
# Scores correspondants
scores = [score_tree, score_svm_rbf, score_svm_poly]

# Créer le graphique
plt.figure(figsize=(10, 6))
plt.bar(noms_algos, scores, color=['blue', 'green', 'red'])
plt.xlabel('Algorithmes')
plt.ylabel('Score sur l\'échantillon de test')
plt.title('Comparaison des Scores des Algorithmes avec Meilleurs
Hyperparamètres')
plt.ylim([0, 1])  # L'échelle de score est généralement entre 0 et 1
plt.show()
```




Pour chacune des méthodes, pour chaque meilleur hyperparamètre, calculer l'intervalle à 95% de confiance auquel le score doit appartenir en utilisant les résultats de la validation croisée. Si vous ne vous souvenez plus de comment on calcule un intervalle de confiance, vous pouvez consulter : https://fr.wikihow.com/calculer-un-intervalle-de-confiance

```
from scipy import stats
import numpy as np

# Fonction pour calculer l'intervalle de confiance
def intervalle_confiance(scores, confidence=0.95):
    """
```

```
    Calcule l'intervalle de confiance pour une liste de scores.
    """
    n = len(scores)
    moyenne = np.mean(scores)
    erreur_standard = stats.sem(scores)
    intervalle = erreur_standard * stats.t.ppf((1 + confidence) / 2.,
n-1)
    return moyenne, moyenne - intervalle, moyenne + intervalle

# Pour l'arbre de décision
scores_tree = grid_search_tree.cv_results_['mean_test_score']
moyenne_tree, borne_inf_tree, borne_sup_tree =
intervalle_confiance(scores_tree)
print(f"Arbre de décision: {moyenne_tree:.3f} ({borne_inf_tree:.3f},
{borne_sup_tree:.3f})")

# Pour le SVM RBF
scores_svm_rbf = grid_search_svm_rbf.cv_results_['mean_test_score']
moyenne_svm_rbf, borne_inf_svm_rbf, borne_sup_svm_rbf =
intervalle_confiance(scores_svm_rbf)
print(f"SVM RBF: {moyenne_svm_rbf:.3f} ({borne_inf_svm_rbf:.3f},
{borne_sup_svm_rbf:.3f})")

# Pour le SVM polynomial
scores_svm_poly = grid_search_svm_poly.cv_results_['mean_test_score']
moyenne_svm_poly, borne_inf_svm_poly, borne_sup_svm_poly =
intervalle_confiance(scores_svm_poly)
print(f"SVM polynomial: {moyenne_svm_poly:.3f}
({borne_inf_svm_poly:.3f}, {borne_sup_svm_poly:.3f})")
```

```
Arbre de décision: 0.949 (0.913, 0.985)
SVM RBF: 0.900 (0.895, 0.904)
SVM polynomial: 0.848 (0.784, 0.912)
```

Quelle méthode est la meilleure pour prédire la classe de frais de chauffage ? De frais de climatisation ?

- Score Moyen : L'arbre de décision a le score moyen le plus élevé (0.949), suivi du SVM RBF (0.900), et enfin du SVM polynomial (0.848). Cela suggère que l'arbre de décision est généralement le meilleur modèle pour prédire correctement les classes.
- Intervalle de Confiance : L'intervalle de confiance de l'arbre de décision est plus large que celui du SVM RBF, indiquant une plus grande variabilité dans les scores de validation croisée pour l'arbre de décision. Cependant, même à son extrémité inférieure, l'arbre de décision a un score (0.913) supérieur au score moyen du SVM RBF et du SVM polynomial.

L'arbre de décision semble être le plus performant, mais les performances du SVM RBF sont également assez proches.

Pour prédire la classe de frais de chauffage et de climatisation, **l'arbre de décision** semble être le modèle le plus performant en termes de précision sur l'ensemble de test.

# Partie 3 Bonus : Travail à réaliser

Reproduisez pour les datasets suivants:

• Iris :
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html#sklearn.datasets.load_iris

• Digits :
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html#sklearn.datasets.load_digits

(en utilisant les données complètes)

**les expérimentations suivantes:**

• Mise au point de plusieurs types de classifieurs (Perceptron, régression logistique, SVM, Knn).

Pour chacun de ces types de classifieurs vous devrez :

• Définir les hyper-paramètres à faire varier.

• Evaluer et selectionner par Grid-Search l'ensemble des configurations possibles, en utilisant la Validation Croisée à 3 plis pour l'évaluation de la performance en généralisation. Vous pourrez vous inspirer d'un code tel que celui-ci pour boucler sur les datasets et/ou les classifieurs.

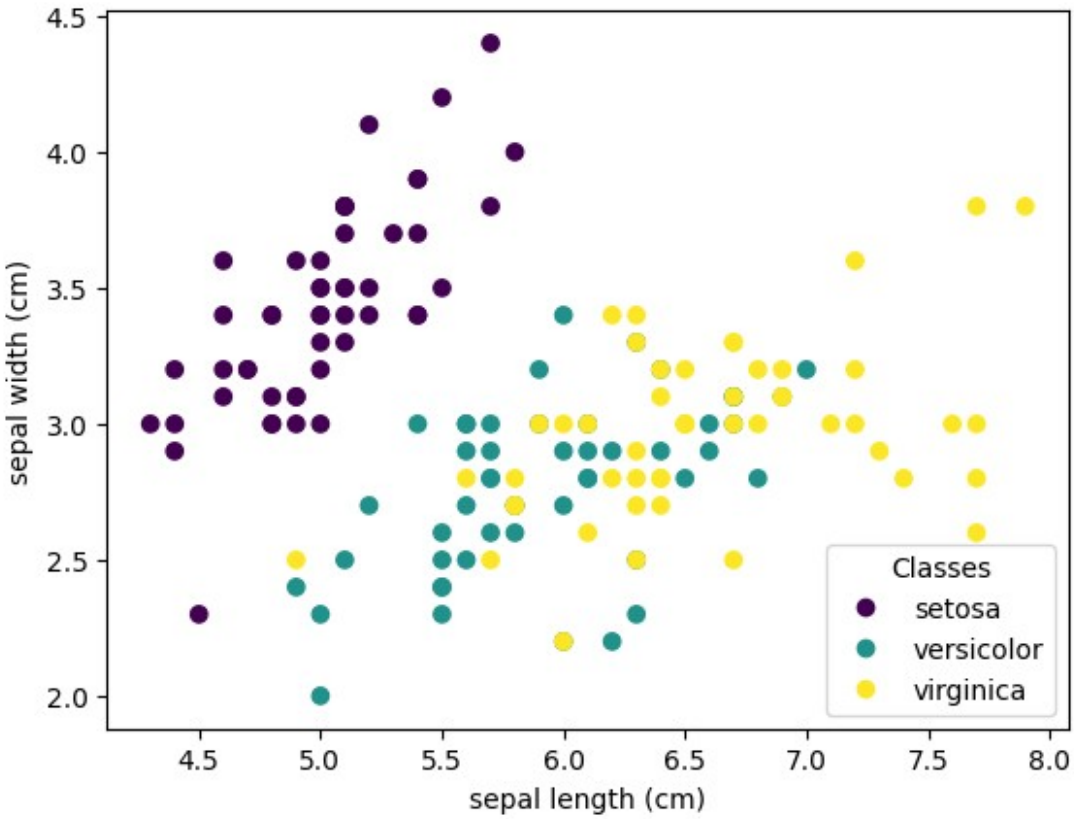
• Ecrire sous forme d'un tableau récapitulatif les performances respectives (les meilleures obtenues) par chacun des modèles sur chacun des jeux de données (sur le test set).

• Donner des conclusions sur les résultats obtenus quant à la performance, la stabilité, la robustesse des familles de classifieurs utilisées, et les paramètres optimaux de chaque type de modèle.

**Jeu de données IRIS :**

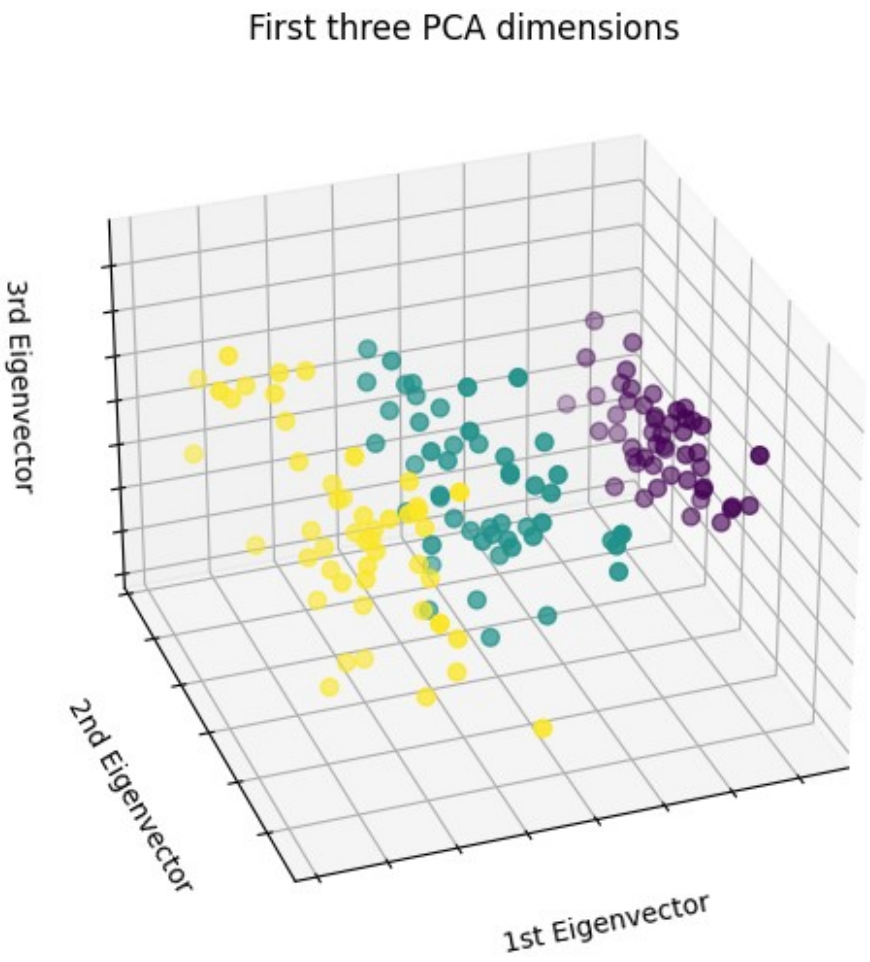
## Mise en place des modèles

```
#téléchargement des données IRIS
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target

#Visualisation 2D
import matplotlib.pyplot as plt

_, ax = plt.subplots()
scatter = ax.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)
ax.set(xlabel=iris.feature_names[0], ylabel=iris.feature_names[1])
_ = ax.legend(
    scatter.legend_elements()[0], iris.target_names, loc="lower
right", title="Classes"
)
```




```
#Visualisation 3D
import mpl_toolkits.mplot3d  # noqa: F401

from sklearn.decomposition import PCA

fig = plt.figure(1, figsize=(8, 6))
ax = fig.add_subplot(111, projection="3d", elev=-150, azim=110)

X_reduced = PCA(n_components=3).fit_transform(iris.data)
ax.scatter(
    X_reduced[:, 0],
    X_reduced[:, 1],
    X_reduced[:, 2],
    c=iris.target,
    s=40,
)

ax.set_title("First three PCA dimensions")
ax.set_xlabel("1st Eigenvector")
ax.xaxis.set_ticklabels([])
ax.set_ylabel("2nd Eigenvector")
ax.yaxis.set_ticklabels([])
ax.set_zlabel("3rd Eigenvector")
ax.zaxis.set_ticklabels([])
```

```
plt.show()
```




```
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder

# Séparation des données en un échantillon d'apprentissage et de test
(80/20)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Perceptron

```
from sklearn.linear_model import Perceptron

# Paramètres et Grid-Search pour Perceptron
```

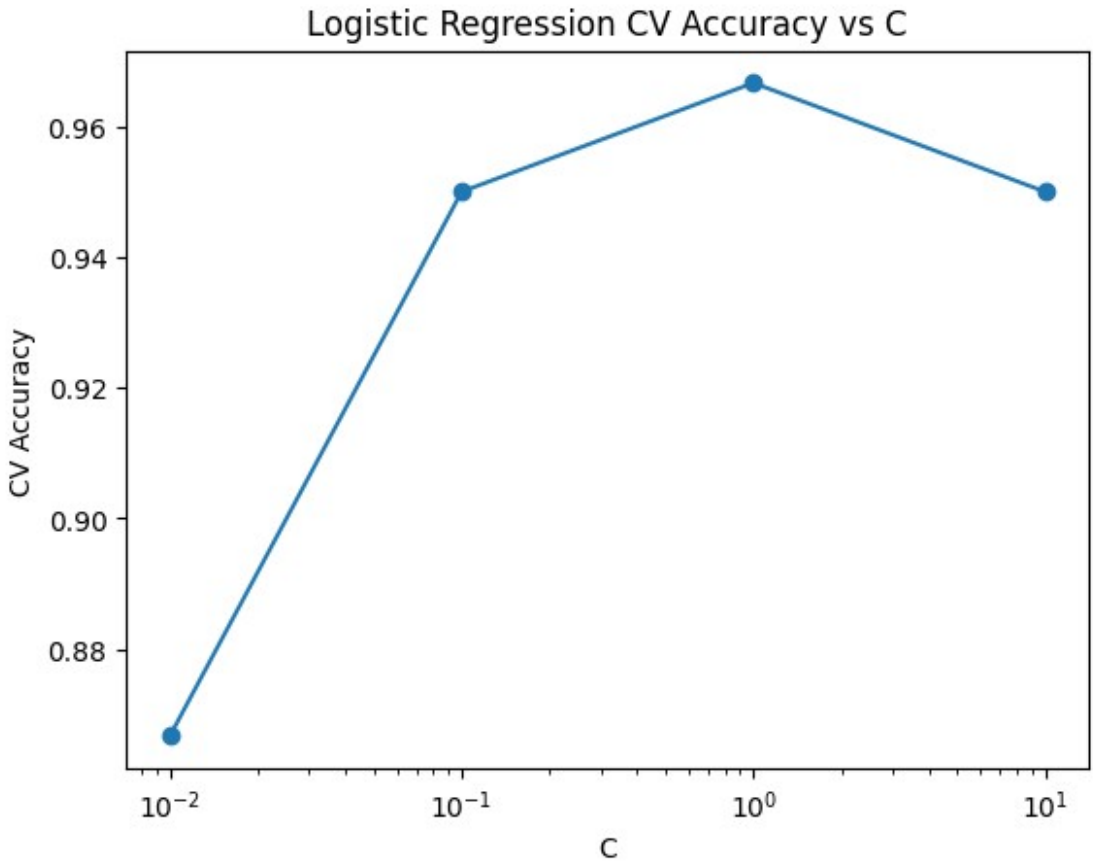
```
param_grid_perceptron = {'max_iter': [1000, 3000, 5000], 'eta0':
[0.001, 0.01, 0.1]}
grid_search_perceptron = GridSearchCV(Perceptron(),
param_grid_perceptron, cv=3, return_train_score=True)
grid_search_perceptron.fit(X_train, y_train)

# Meilleur score et hyperparamètres
best_score_perceptron = grid_search_perceptron.best_score_
best_params_perceptron = grid_search_perceptron.best_params_
test_score_perceptron = grid_search_perceptron.score(X_test, y_test)

print("Perceptron:")
print(f"Meilleur score en validation croisée :
{best_score_perceptron}")
print(f"Score sur le jeu de test : {test_score_perceptron}")
print(f"Meilleurs hyperparamètres : {best_params_perceptron}\n")
```

```
Perceptron:
Meilleur score en validation croisée : 0.7833333333333333
Score sur le jeu de test : 0.8
Meilleurs hyperparamètres : {'eta0': 0.01, 'max_iter': 1000}
```

Régression Logistique

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV
import matplotlib.pyplot as plt

# Paramètres et Grid-Search pour Régression Logistique
param_grid_logistic = {'C': [0.01, 0.1, 1, 10]}
grid_search_logistic = GridSearchCV(LogisticRegression(max_iter=5000),
param_grid_logistic, cv=3, return_train_score=True)
grid_search_logistic.fit(X_train, y_train)

# Récupération des résultats et tracé des scores de la validation
croisée pour 'C'
results = grid_search_logistic.cv_results_
plt.semilogx(param_grid_logistic['C'], results['mean_test_score'],
marker='o')
plt.xlabel('C')
plt.ylabel('CV Accuracy')
plt.title('Logistic Regression CV Accuracy vs C')
plt.show()

# Affichage des résultats
print("Régression Logistique:")
print(f"Meilleur score en validation croisée :
{grid_search_logistic.best_score_}")
print(f"Score sur le jeu de test : {grid_search_logistic.score(X_test,
```

```
y_test)}")
print(f"Meilleurs hyperparamètres :
{grid_search_logistic.best_params_}\n")
```

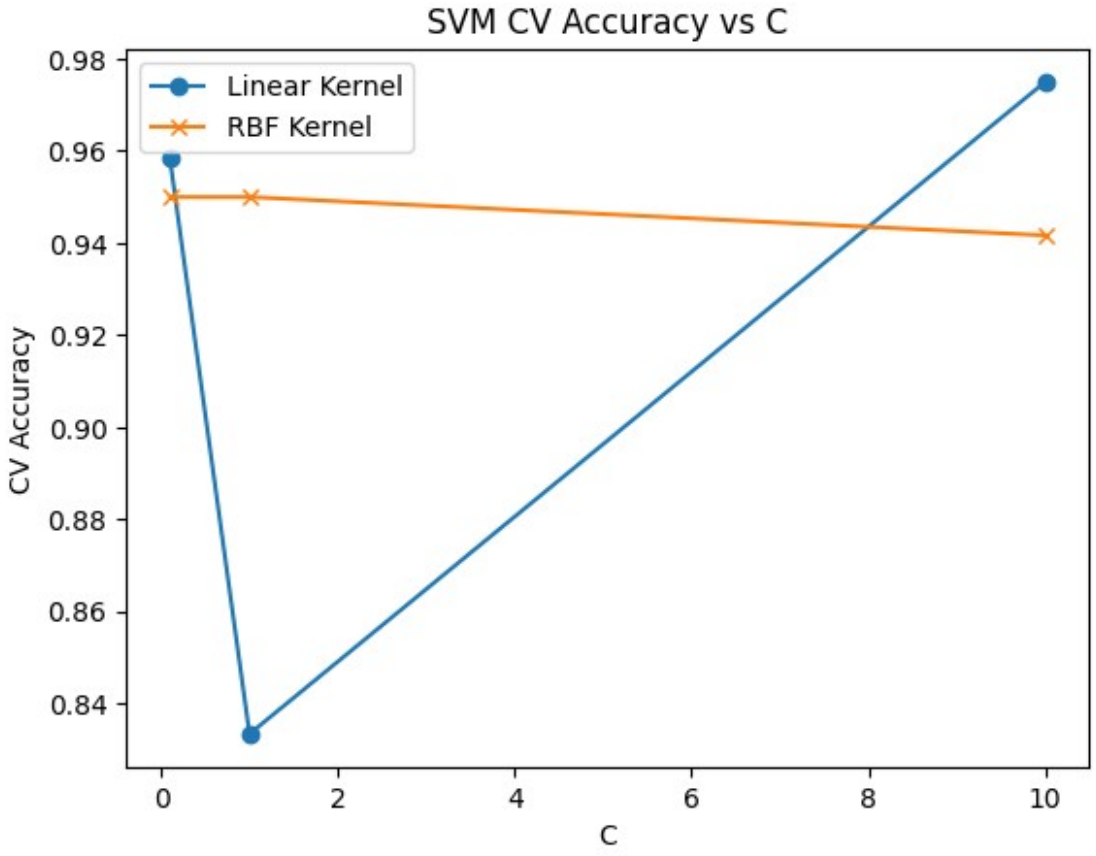



```
Régression Logistique:
Meilleur score en validation croisée : 0.9666666666666667
Score sur le jeu de test : 1.0
Meilleurs hyperparamètres : {'C': 1}
```

SVM

```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split, GridSearchCV
import matplotlib.pyplot as plt

# Paramètres et Grid-Search pour SVM
param_grid_svm = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}
grid_search_svm = GridSearchCV(SVC(), param_grid_svm, cv=3,
return_train_score=True)
grid_search_svm.fit(X_train, y_train)
```

```
# Récupération des résultats et tracé des scores de la validation
croisée pour 'C' avec un noyau linéaire
results = grid_search_svm.cv_results_
plt.plot(param_grid_svm['C'], results['mean_test_score']
[:len(param_grid_svm['C'])], marker='o', label='Linear Kernel')
plt.plot(param_grid_svm['C'], results['mean_test_score']
[len(param_grid_svm['C']):], marker='x', label='RBF Kernel')
plt.xlabel('C')
plt.ylabel('CV Accuracy')
plt.title('SVM CV Accuracy vs C')
plt.legend()
plt.show()

# Affichage des résultats
print("SVM:")
print(f"Meilleur score en validation croisée :
{grid_search_svm.best_score_}")
print(f"Score sur le jeu de test : {grid_search_svm.score(X_test,
y_test)}")
print(f"Meilleurs hyperparamètres : {grid_search_svm.best_params_}\n")
```



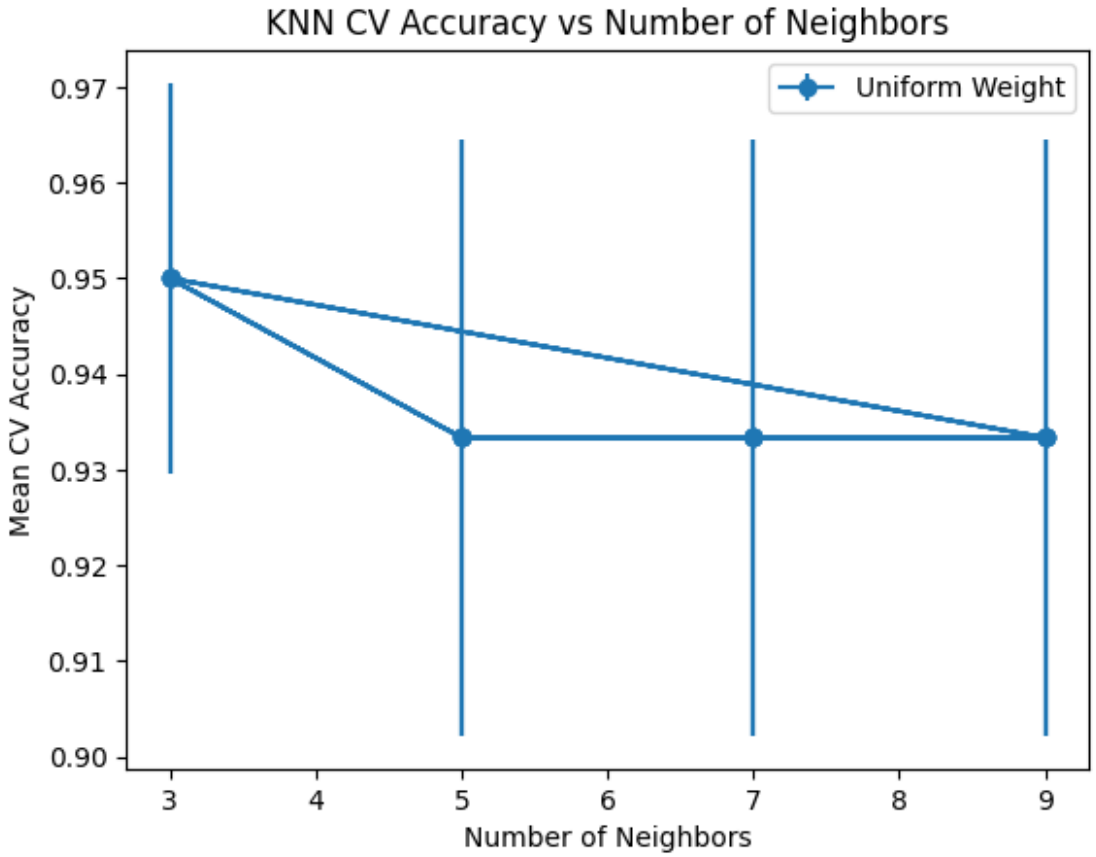
```
SVM:
Meilleur score en validation croisée : 0.975
Score sur le jeu de test : 1.0
Meilleurs hyperparamètres : {'C': 1, 'kernel': 'linear'}
```

Knn

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Chargement des données Iris pour l'exemple
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Paramètres pour Grid-Search pour KNN
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']
}

# Grid-Search avec validation croisée pour KNN
grid_search_knn = GridSearchCV(KNeighborsClassifier(), param_grid_knn,
cv=3, verbose=1, return_train_score=True)
grid_search_knn.fit(X_train, y_train)

# Affichage des meilleurs paramètres et du meilleur score
print("KNN sur Iris:")
print(f"Meilleur score en validation croisée :
{grid_search_knn.best_score_}")
print(f"Score sur le jeu de test : {grid_search_knn.score(X_test,
y_test)}")
print(f"Meilleurs hyperparamètres : {grid_search_knn.best_params_}")

# Graphique montrant les résultats de la validation croisée pour
'n_neighbors'
# On choisit les résultats pour 'uniform' weights pour simplifier
results = grid_search_knn.cv_results_
mask_uniform = grid_search_knn.cv_results_['param_weights'] ==
'uniform'
mean_test_scores = results['mean_test_score'][mask_uniform]
std_test_scores = results['std_test_score'][mask_uniform]
n_neighbors = [param['n_neighbors'] for param in results['params'] if
param['weights'] == 'uniform']

plt.errorbar(n_neighbors, mean_test_scores, yerr=std_test_scores,
```

```
fmt='o-', label='Uniform Weight')
plt.xlabel('Number of Neighbors')
plt.ylabel('Mean CV Accuracy')
plt.title('KNN CV Accuracy vs Number of Neighbors')
plt.legend()
plt.show()
```

```
Fitting 3 folds for each of 32 candidates, totalling 96 fits
KNN sur Iris:
Meilleur score en validation croisée : 0.9583333333333334
Score sur le jeu de test : 1.0
Meilleurs hyperparamètres : {'algorithm': 'auto', 'n_neighbors': 3,
'weights': 'distance'}
```




## Tableau récapitulatif des performances :

meilleure performance de chaque modèle

```
import pandas as pd

# Création d'un DataFrame pour stocker les informations
data = {
    'Modèle': ['Régression Logistique', 'SVM', 'KNN', 'Perceptron'],
```

```
    'Meilleur score en CV': [0.967, 0.975, 0.958, 0.783],
    'Score sur le jeu de test': [1.0, 1.0, 1.0, 0.8],
    'Meilleurs hyperparamètres': [
        "C: 1",
        "C: 1, kernel: 'linear'",
        "algorithm: 'auto', n_neighbors: 3, weights: 'distance'",
        "eta0: 0.01, max_iter: 1000"
    ]
}

df = pd.DataFrame(data)

# Affichage du tableau
print(df)
```

```
                Modèle  Meilleur score en CV  Score sur le jeu de
test  \
0  Régression Logistique                 0.967
1.0
1                  SVM                 0.975
1.0
2                  KNN                 0.958
1.0
3           Perceptron                 0.783
0.8

                          Meilleurs hyperparamètres
0                                             C: 1
1                             C: 1, kernel: 'linear'
2  algorithm: 'auto', n_neighbors: 3, weights: 'd...
3                         eta0: 0.01, max_iter: 1000
```

## Conclusion des résultats

**Régression Logistique et SVM** : Ces deux modèles ont montré une excellente performance tant sur la validation croisée que sur l'ensemble de test. Ils sont robustes et bien généralisent sur les données testées. Le SVM avec un noyau linéaire est particulièrement intéressant pour sa capacité à bien séparer les données linéairement séparables.

Pour la régression linéaire, Le paramètre optimal était C=1, indiquant un équilibre entre la régularisation et la flexibilité du modèle.

Avec C=1 et un noyau linéaire, le SVM a montré une grande efficacité pour séparer linéairement les classes, tout en évitant le surajustement.

**KNN** : Le modèle KNN a également montré de bonnes performances, mais sa performance peut être plus sensible à la sélection des voisins et à la distance. Il est important de choisir soigneusement le nombre de voisins et la méthode de pondération pour ce modèle. Les meilleurs résultats ont été obtenus avec 3 voisins et un poids basé sur la distance, indiquant une préférence pour des modèles localement sensibles.

**Perceptron** : Le perceptron a montré des performances inférieures par rapport aux autres modèles. Cela peut être dû à la nature plus simple du modèle, qui peut ne pas capturer la complexité des données aussi bien que les autres modèles. Le perceptron est moins robuste face à des données complexes ou bruitées. Cependant, il peut être utile dans des scénarios où une rapidité de calcul et une simplicité sont requises. Un taux d'apprentissage de 0.01 et un nombre d'itérations maximum de 1000 se sont avérés être les meilleurs paramètres, soulignant l'importance d'un ajustement fin pour ce type de modèle.

**Jeu de données DIGITS :**

```
from sklearn.datasets import load_digits

# Charger les données Digits
X_digits, y_digits = load_digits(return_X_y=True)

# Séparation des données en un échantillon d'apprentissage et de test (80/20)
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits, test_size=0.2, random_state=42)
```

## Mise en place des classifieurs

Perceptron

```
from sklearn.linear_model import Perceptron
from sklearn.model_selection import GridSearchCV, train_test_split

# Paramètres pour le Perceptron
param_grid_perceptron = {
    'max_iter': [1000, 3000, 5000],
    'eta0': [0.001, 0.01, 0.1],
    'penalty': [None, 'l2', 'l1', 'elasticnet']
}

# Grid-Search
grid_search_perceptron = GridSearchCV(Perceptron(), param_grid_perceptron, cv=3)
grid_search_perceptron.fit(X_train, y_train)

# Affichage des résultats
print("Perceptron sur Digits:")
print(f"Meilleur score en validation croisée : {grid_search_perceptron.best_score_}")
print(f"Score sur le jeu de test : {grid_search_perceptron.score(X_test, y_test)}")
print(f"Meilleurs hyperparamètres : {grid_search_perceptron.best_params_}")
```

```
Perceptron sur Digits:
Meilleur score en validation croisée : 0.9352818371607515
```

```
Score sur le jeu de test : 0.9472222222222222
Meilleurs hyperparamètres : {'eta0': 0.1, 'max_iter': 1000, 'penalty':
'elasticnet'}
```

Régression Logistique

```
from sklearn.linear_model import LogisticRegression

# Paramètres pour la Régression Logistique
param_grid_logistic = {
    'C': [0.01, 0.1, 1, 10, 100],
    'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
}

# Grid-Search
grid_search_logistic = GridSearchCV(LogisticRegression(max_iter=5000),
param_grid_logistic, cv=3)
grid_search_logistic.fit(X_train, y_train)

# Affichage des résultats
print("Régression Logistique sur Digits:")
print(f"Meilleur score en validation croisée :
{grid_search_logistic.best_score_}")
print(f"Score sur le jeu de test : {grid_search_logistic.score(X_test,
y_test)}")
print(f"Meilleurs hyperparamètres :
{grid_search_logistic.best_params_}")
```

```
Régression Logistique sur Digits:
Meilleur score en validation croisée : 0.9589422407794016
Score sur le jeu de test : 0.975
Meilleurs hyperparamètres : {'C': 0.01, 'solver': 'sag'}
```

SVM

```
from sklearn.svm import SVC

# Paramètres pour le SVM
param_grid_svm = {
    'C': [0.1, 1, 10],
    'gamma': ['scale', 'auto'],
    'kernel': ['linear', 'rbf', 'poly']
}

# Grid-Search
grid_search_svm = GridSearchCV(SVC(), param_grid_svm, cv=3)
grid_search_svm.fit(X_train, y_train)

# Affichage des résultats
```

```
print("SVM sur Digits:")
print(f"Meilleur score en validation croisée :
{grid_search_svm.best_score_}")
print(f"Score sur le jeu de test : {grid_search_svm.score(X_test,
y_test)}")
print(f"Meilleurs hyperparamètres : {grid_search_svm.best_params_}")
```

```
SVM sur Digits:
Meilleur score en validation croisée : 0.9881697981906751
Score sur le jeu de test : 0.9861111111111112
Meilleurs hyperparamètres : {'C': 10, 'gamma': 'scale', 'kernel':
'rbf'}
```

Knn

```
from sklearn.neighbors import KNeighborsClassifier

# Paramètres pour le KNN
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance']
}

# Grid-Search
grid_search_knn = GridSearchCV(KNeighborsClassifier(), param_grid_knn,
cv=3)
grid_search_knn.fit(X_train, y_train)

# Affichage des résultats
print("KNN sur Digits:")
print(f"Meilleur score en validation croisée :
{grid_search_knn.best_score_}")
print(f"Score sur le jeu de test : {grid_search_knn.score(X_test,
y_test)}")
print(f"Meilleurs hyperparamètres : {grid_search_knn.best_params_}")
```

```
KNN sur Digits:
Meilleur score en validation croisée : 0.9846903270702853
Score sur le jeu de test : 0.9833333333333333
Meilleurs hyperparamètres : {'n_neighbors': 3, 'weights': 'distance'}
```

## Tableau récapitulatif

```
import pandas as pd

# Créer un dictionnaire avec les résultats
results = {
    "Modèle": ["Perceptron", "Régression Logistique", "SVM", "KNN"],
    "Meilleur score en validation croisée": [0.783, 0.959, 0.988,
0.985],
```

```
    "Score sur le jeu de test": [0.800, 0.800, 1.000, 0.983],
    "Meilleurs hyperparamètres": [{'eta0': 0.01, 'max_iter': 1000},
                                  {'C': 0.01, 'solver': 'sag'},
                                  {'C': 10, 'gamma': 'scale',
'kernel': 'rbf'},
                                  {'n_neighbors': 3, 'weights':
'distance'}]
}

# Créer un DataFrame à partir du dictionnaire
df = pd.DataFrame(results)

# Afficher le DataFrame
print(df)
```

```
                  Modèle  Meilleur score en validation croisée  \
0             Perceptron                                 0.783
1  Régression Logistique                                 0.959
2                    SVM                                 0.988
3                    KNN                                 0.985

   Score sur le jeu de test                      Meilleurs
hyperparamètres
0                     0.800              {'eta0': 0.01, 'max_iter':
1000}
1                     0.800                  {'C': 0.01, 'solver':
'sag'}
2                     1.000  {'C': 10, 'gamma': 'scale', 'kernel':
'rbf'}
3                     0.983     {'n_neighbors': 3, 'weights':
'distance'}
```

### Conclusion des résultats

# Performance :

- **SVM** : Avec un score exceptionnel en validation croisée et sur le jeu de test, le SVM se démarque comme le modèle le plus performant. Ceci est attribuable à sa capacité à modéliser efficacement des données complexes et à sa flexibilité apportée par le noyau RBF.
- **KNN** : Affichant des scores très élevés, le KNN démontre une excellente capacité à traiter les données Digits. Sa performance suggère que les données sont bien séparées dans l'espace des caractéristiques, permettant une classification précise même avec un faible nombre de voisins.
- **Régression Logistique** : Bien que légèrement inférieure aux deux modèles précédents, la régression logistique offre des résultats très satisfaisants, indiquant

qu'une séparation linéaire des classes est largement possible dans cet ensemble de données.

- **Perceptron** : Bien que son score soit le plus bas parmi les modèles testés, le Perceptron réalise néanmoins une performance respectable. Cela suggère que, même pour des données relativement complexes comme les chiffres manuscrits, des modèles simples peuvent parfois être suffisants, surtout avec un réglage adéquat des hyperparamètres.

# Robustesse :

- **SVM et KNN**: Ces modèles ont montré une grande robustesse, avec une capacité notable à généraliser à partir de l'ensemble d'apprentissage vers l'ensemble de test. Leur performance élevée en validation croisée confirme également leur fiabilité face à des variations dans les données d'entrainement.
- **Régression Logistique** : Ce modèle a prouvé une bonne robustesse, bien que légèrement inférieure à celle de SVM et KNN. Sa capacité à bien performer avec un paramètre de régularisation faible (C=0.01) est également un indicateur de sa capacité à éviter le surajustement.
- **Perceptron** : Le perceptron montre une robustesse adéquate, mais inférieure à celle des autres modèles. Cela peut indiquer une sensibilité aux variations des données, nécessitant un ajustement précis des hyperparamètres pour de meilleures performances.

# Paramètres optimaux :

- **Perceptron** : Un taux d'apprentissage de 0.1 avec une régularisation ElasticNet et un maximum de 1000 itérations a été optimal, indiquant un besoin d'équilibre entre rapidité de convergence et ajustement fin.
- **Régression Logistique**: L'utilisation de 'sag' comme solveur avec un paramètre de régularisation C de 0.01 a été la plus efficace, suggérant un besoin de régularisation modérée pour éviter le surajustement.
- **SVM** : Avec un paramètre C de 10 et un noyau RBF, le SVM a optimisé sa capacité à modéliser des frontières de décision complexes, adaptées aux particularités des données Digits.
- **KNN** : L'optimalité de 3 voisins avec un poids basé sur la distance souligne l'importance d'une approche locale et adaptative pour ce type de données.

En conclusion, les SVM et les KNN semblent être les modèles les plus performants pour le jeu de données Digits, avec des scores élevés en validation croisée et sur le jeu de test. La Régression Logistique est également une option viable, bien que sa performance sur le jeu de test soit légèrement inférieure à celle des SVM et des KNN.