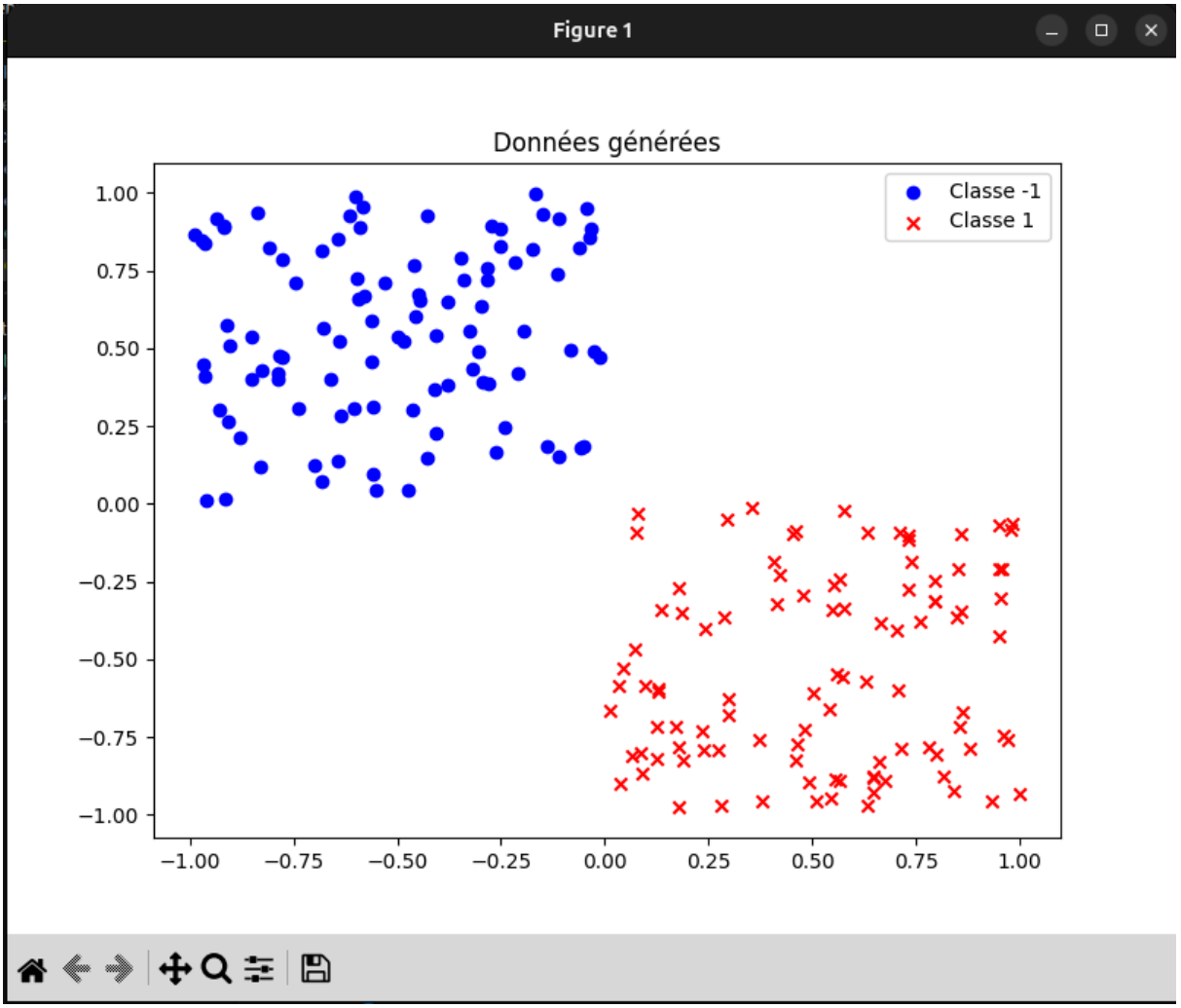
Anass EZZINE
4A FISA

TP-Perceptron

1. Algorithme du perceptron pour la classification binaire

1.1. Commençons par générer et afficher un jeu de données en utilisant la fonction `generateData` fournie :



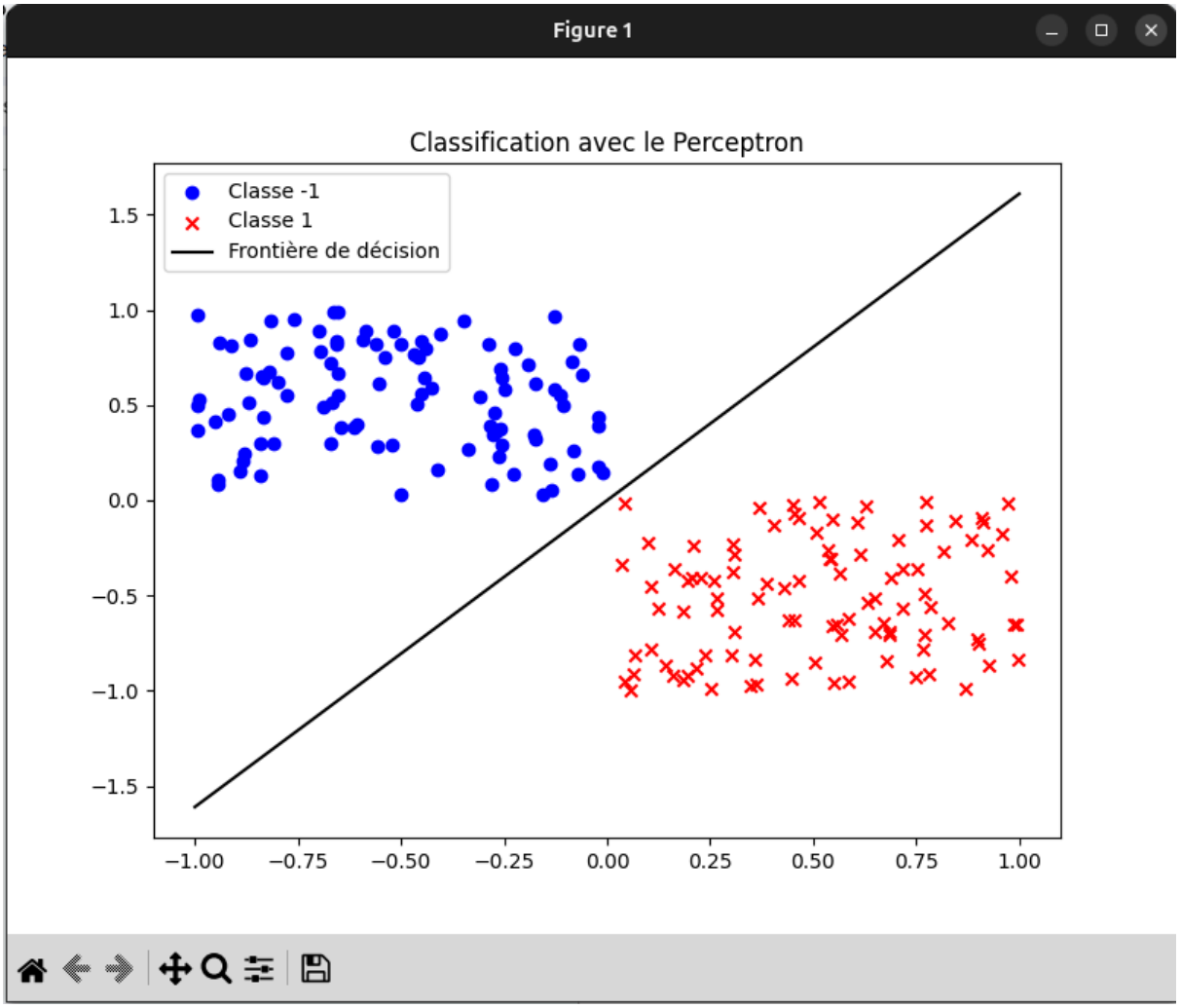
1.2.

Alors, voilà comment ça marche :
Au début, on part d'un vecteur de poids w initialisé à zéro, Ensuite, on répète une boucle : à chaque tour, on regarde chaque exemple de données. Si un point est mal classé (par exemple, il devrait être rouge mais le modèle le voit comme bleu), on ajuste w pour corriger l'erreur. On s'arrête seulement quand tous les exemples sont bien classés (plus d'erreurs !).
Pour montrer le résultat :

- Les points bleus sont ceux de la classe -1,
- Les points rouges, la classe 1,

La ligne noire, c'est la frontière (hyperplan) que le perceptron a apprise pour séparer les deux groupes. on voit qu'il n'y a aucune erreur car le jeu de donnée est linéairement

séparable, centré sur l'origine. Donc ce perceptron simple est particulièrement éfficace sur ce type de données.

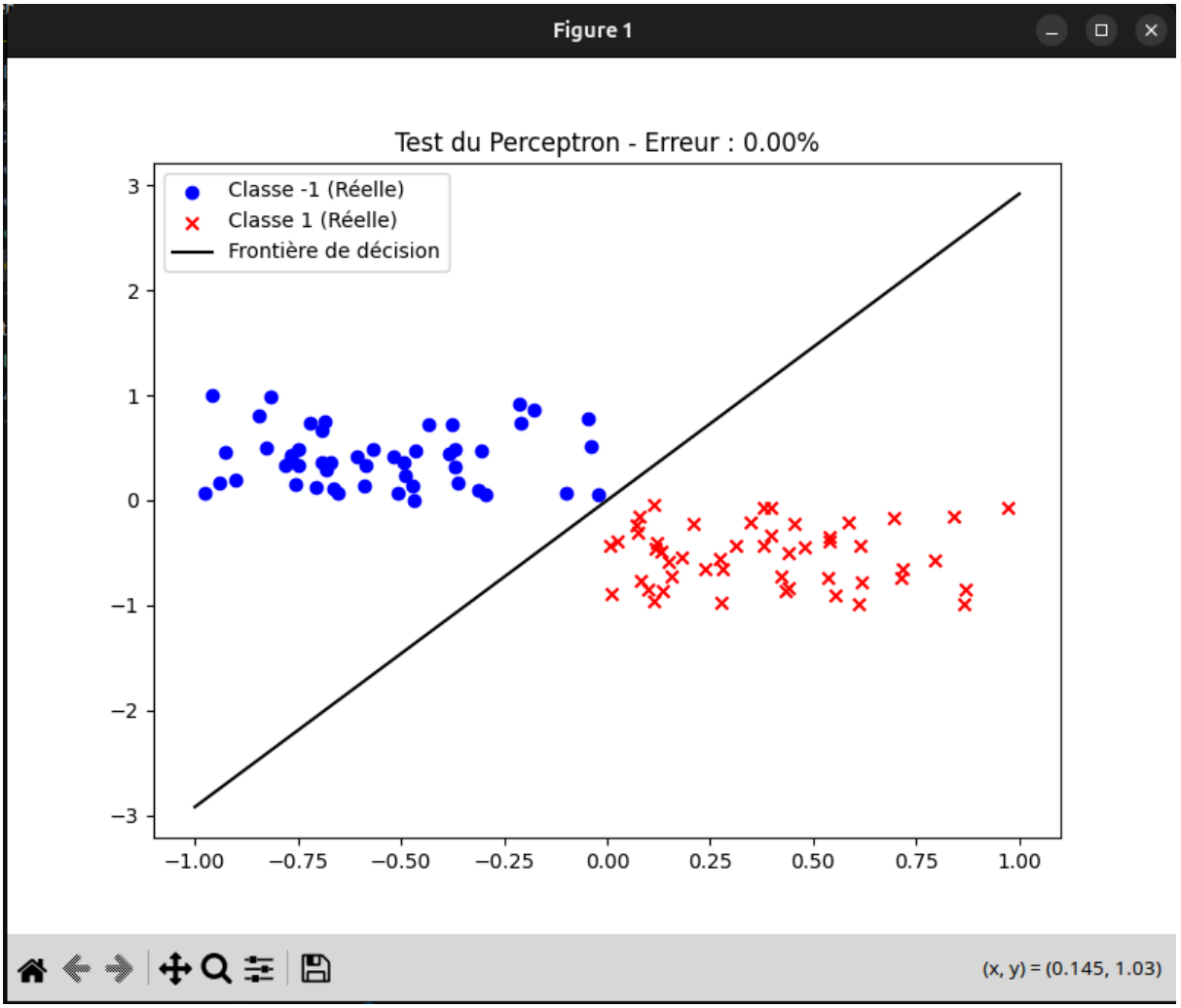


1.3.

Si les données sont bien linéairement séparables, l'erreur doit être proche de 0%.Sinon, l'erreur peut être plus élevée.

```
# Calcul de l'erreur de classification
error_rate = np.mean(Y_pred != Y_test)  # Pourcentage d'exemples mal classés
```

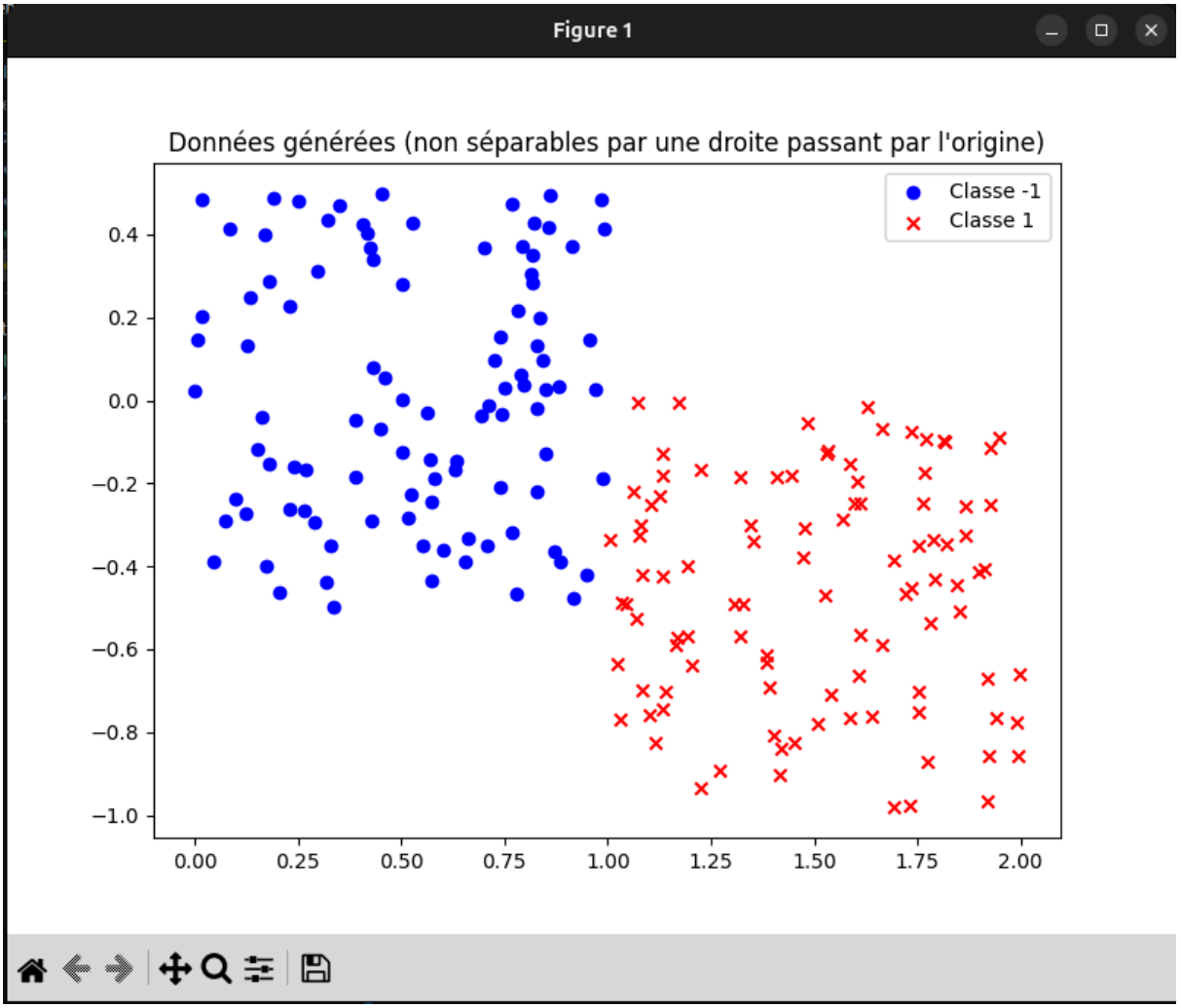
Anass EZZINE
4A FISA



1.4.

Maintenant, passons à l'extension avec un biais, c'est-à-dire le cas où les données ne sont pas séparables par une droite passant par l'origine. L'équation est donc maintenant ax+by+c = 0 avec c représentant le décalage par rapport à l'origine. Ainsi nous avons besoin de compléter nos données avec une troisième colonne, remplis de 1 afin d'avoir nos 3 coefficients a, b, c dans le vecteur de pondération w, en effet cette colonne permet d'ajouter une dimension. La fonction add_bias() permet de concaténer une colonne de 1 à nos données.. Je modifie la fonction de génération des données (generateData2) pour qu'elles soient toujours séparables, mais par une droite qui ne passe pas par l'origine.

```python
def add_bias(X):
    n_samples = X.shape[0]
    bias_column = np.ones((n_samples, 1))  # Colonne de 1
    return np.hstack((X, bias_column))  # Concaténation avec X
```

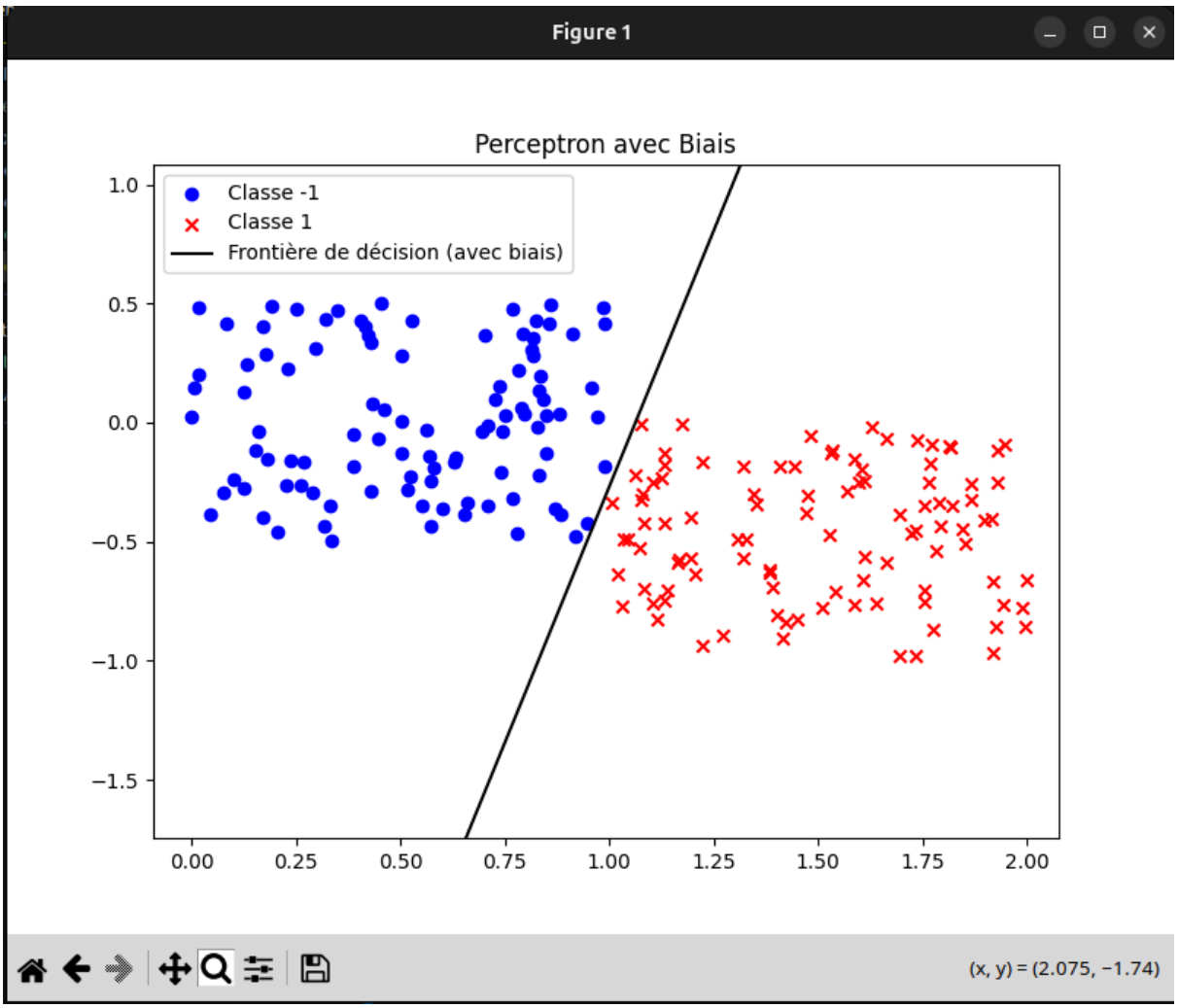


procède de la même façon, à la différence qu'elle applique le perceptron aux données concaténées avec la colonne de 1 qui ajoute une coordonnée à chaque exemple de l'échantillon et les deux points y1, y2 sont déterminés avec la nouvelle équation de l'hyperplan

$$ax + by + c = 0$$

$$y = -\frac{(c + (a * x))}{b}$$

On obtient ainsi le résultat ci-dessous :

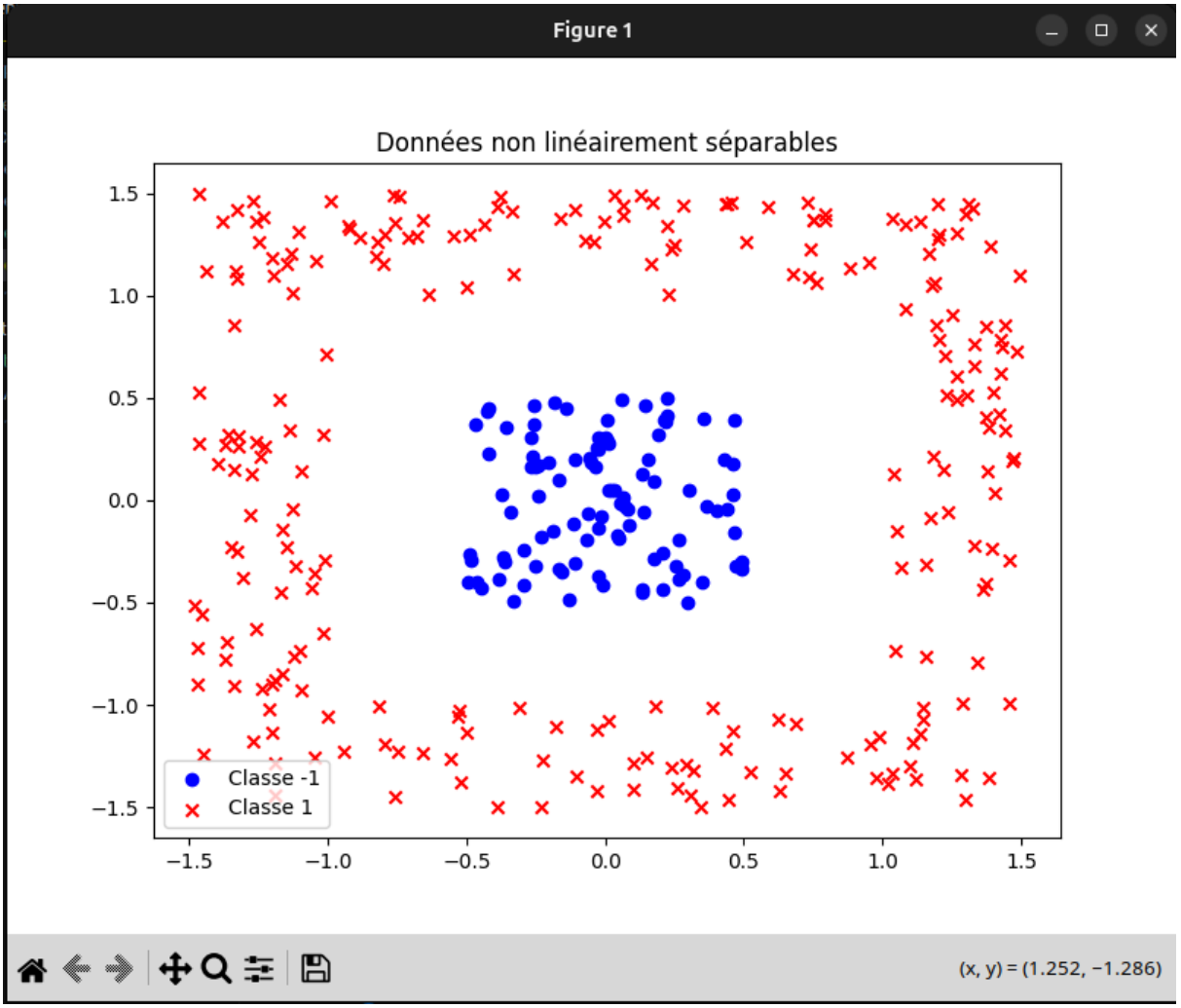


2. Perceptron `a noyau

2.1.

On utilise maintenant des données non linéairement séparables, la fonction genererData3() permet de créer des données comprises dans les zones rouges et bleu de la figure ci-dessous. Ces données ne peuvent être séparés avec les fonctions précédentes

Anass EZZINE
4A FISA



2.2.

Tout d'abord, il est nécessaire d'adapter notre échantillon afin d'obtenir un vecteur de pondération capable de séparer les données à l'aide d'une courbe. Pour cela, l'utilisation d'un plongement est indispensable. Ainsi, nous définissons la transformation Φ par :

$$\Phi(x,y)=(1,x,y,x2,xy,y2)\Phi(x, y) = (1, x, y, x^2, xy, y^2)$$

La fonction ci-dessous applique ce plongement pour convertir l'échantillon. Concrètement, elle génère une matrice de six colonnes et d'un nombre de lignes égal à la taille de l'échantillon, en remplissant chaque ligne avec les valeurs transformées correspondant à chaque exemple de l'échantillon.

```
def polynomial_feature_mapping(X):
    X_poly = np.c_[np.ones(X.shape[0]), X[:, 0], X[:, 1], X[:, 0]**2, X[:, 0] * X[:, 1], X[:, 1]**2]
    return X_poly

# Transformation des données
X3_poly = polynomial_feature_mapping(X3)
```

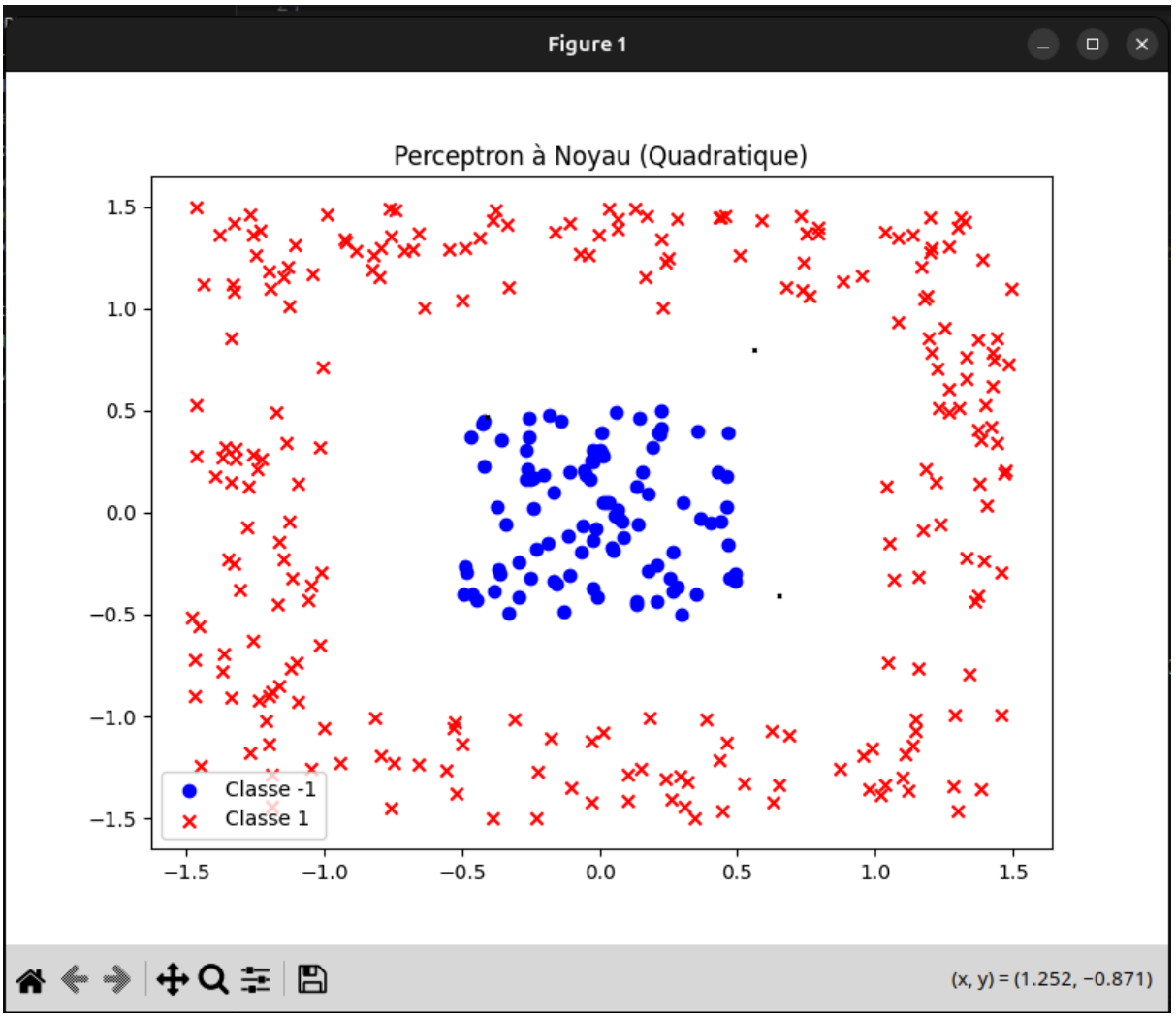
2.3.

Grâce à cette transformation, l'apprentissage de l'échantillon avec le perceptron peut se faire directement, sans modification du modèle. En effet, l'ajout d'une colonne de 1 (nécessaire pour introduire une coordonnée supplémentaire) est déjà pris en charge par le

plongement, qui génère cette colonne automatiquement dans la matrice. Le vecteur de pondération obtenu permet ensuite de définir la fonction correspondant à la courbe séparatrice. Il suffit pour cela de calculer le produit scalaire entre le vecteur (1,x,y,x2,xy,y2)(1, x, y, x^2, xy, y^2) et le vecteur w. Le perceptron standard stocke un vecteur de poids w, mais avec un noyau, on stocke plutôt un ensemble de coefficients α associés aux échantillons support

2.4.

On remarque que les données non séparables linéairement et que la frontière de séparation est courbée pour mieux séparer les classes:



2.5.

Pour passer de la fonction k :R2×R2→R à f :R2→R, la fonction ff romk est l'application de la formule :

$$\sum_{(x_i,y_i)\in VS} (c_i y_i k(x_i, x))$$

```
def f_from_kernel(coeffs, support_set, kernel, x):
    return sum(coeffs[i] * Y3[i] * kernel(support_set[i], x) for i in range(len(coeffs)))
```

Nous pouvons maintenant implémenter l'algorithme `perceptron_kernel()`, qui permet de sélectionner une fonction adaptée à un problème spécifique. Cet algorithme accepte une fonction k, qui peut être aussi bien une fonction polynomiale qu'une gaussienne.

Dans ce contexte, la variable coeff représente l'expression w0x0+w1x1w_0 x_0 + w_1 x_1, tandis que la variable support set stocke l'ensemble des valeurs d'apprentissage après l'application du perceptron.

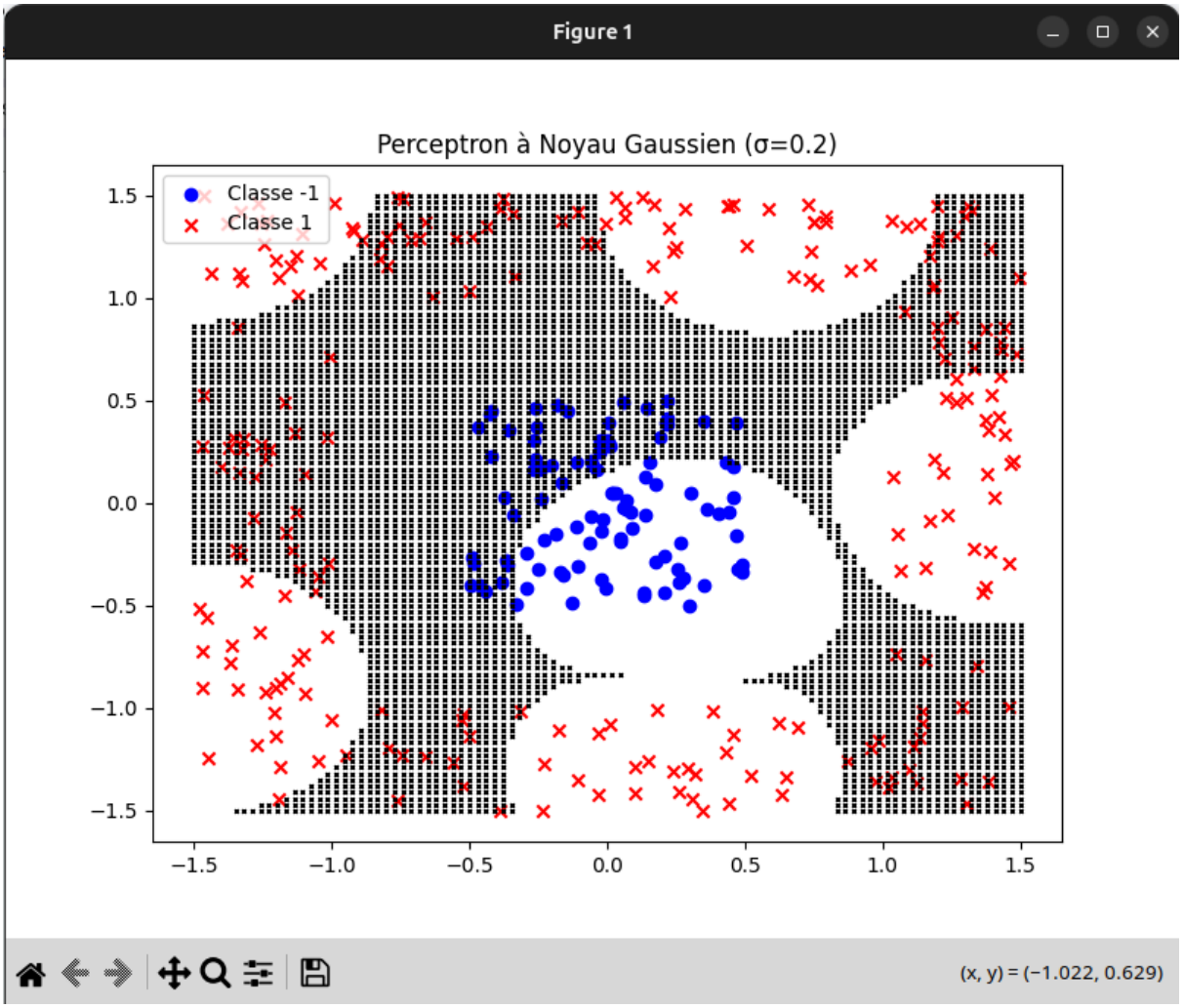
```
def perceptron_kernel(X, Y, kernel, max_iter=1000):
    n_samples = len(Y)
    coefficients = np.zeros(n_samples)  # Coefficients alpha
    support_vectors = X.copy()

    for _ in range(max_iter):
        error = False
        for i in range(n_samples):
            sum_kernel = sum(coefficients[j] * Y[j] * kernel(X[j], X[i]) for j in range(n_samples))
            if Y[i] * sum_kernel <= 0:  # Mauvaise classification
                coefficients[i] += 1  # Mise à jour de alpha
                error = True
        if not error:
            break

    return coefficients, support_vectors
```

On peut aussi tester un noyau Gaussien sur plusieur valeur sigma et on peut voir comment la séparation change pour chaque valeur :

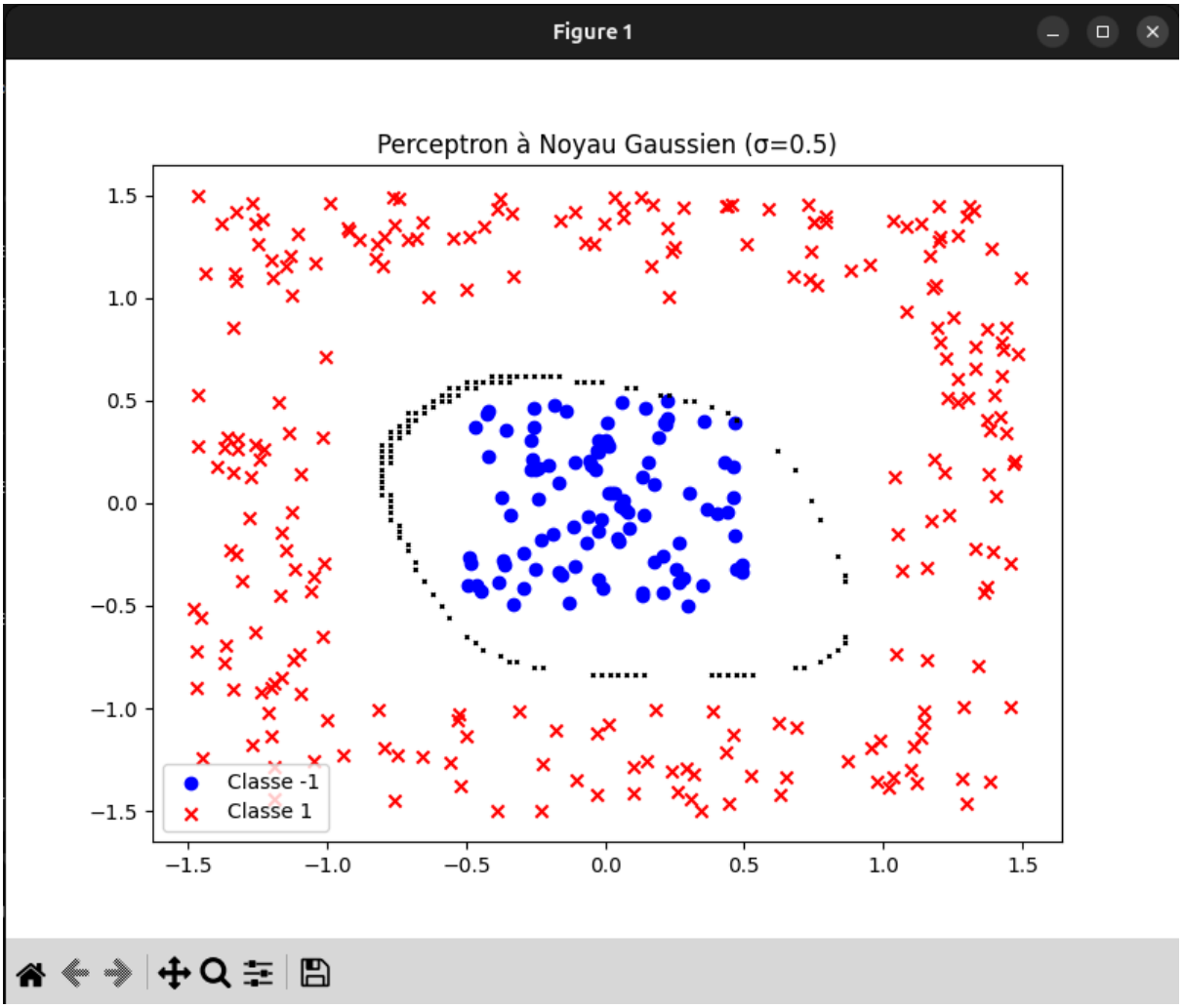
```
def kernel_gaussian(x, y, sigma=1):
    return np.exp(-np.linalg.norm(np.array(x) - np.array(y))**2 / (2 * sigma**2))

# Entraînement du perceptron avec noyau Gaussien
alphas_gauss, support_vectors_gauss = perceptron_kernel(X3, Y3, kernel_gaussian)
```

```
# Expérimentation avec différents sigma
for sigma in [0.2, 0.5, 1, 2, 5]:
    alphas_gauss, support_vectors_gauss = perceptron_kernel(X3, Y3, lambda x, y: kernel_gaussian(x, y, sigma))

    plt.figure(figsize=(8, 6))
    plt.scatter(X3[Y3 == -1, 0], X3[Y3 == -1, 1], color='blue', marker='o', label='Classe -1')
    plt.scatter(X3[Y3 == 1, 0], X3[Y3 == 1, 1], color='red', marker='x', label='Classe 1')

    # Tracé de la frontière de séparation
    res = 100
    x_vals = np.linspace(-1.5, 1.5, res)
    y_vals = np.linspace(-1.5, 1.5, res)

    for x in range(res):
        for y in range(res):
            if abs(f_from_kernel(alphas_gauss, support_vectors_gauss, lambda x, y: kernel_gaussian(x, y, sigma), [x_vals[x], y_vals[y]])) < 0.01:
                plt.plot(x_vals[x], y_vals[y], 'kx', markersize=2)  # Points proches de la frontière

    plt.legend()
    plt.title(f"Perceptron à Noyau Gaussien (σ={sigma})")
    plt.show()
```

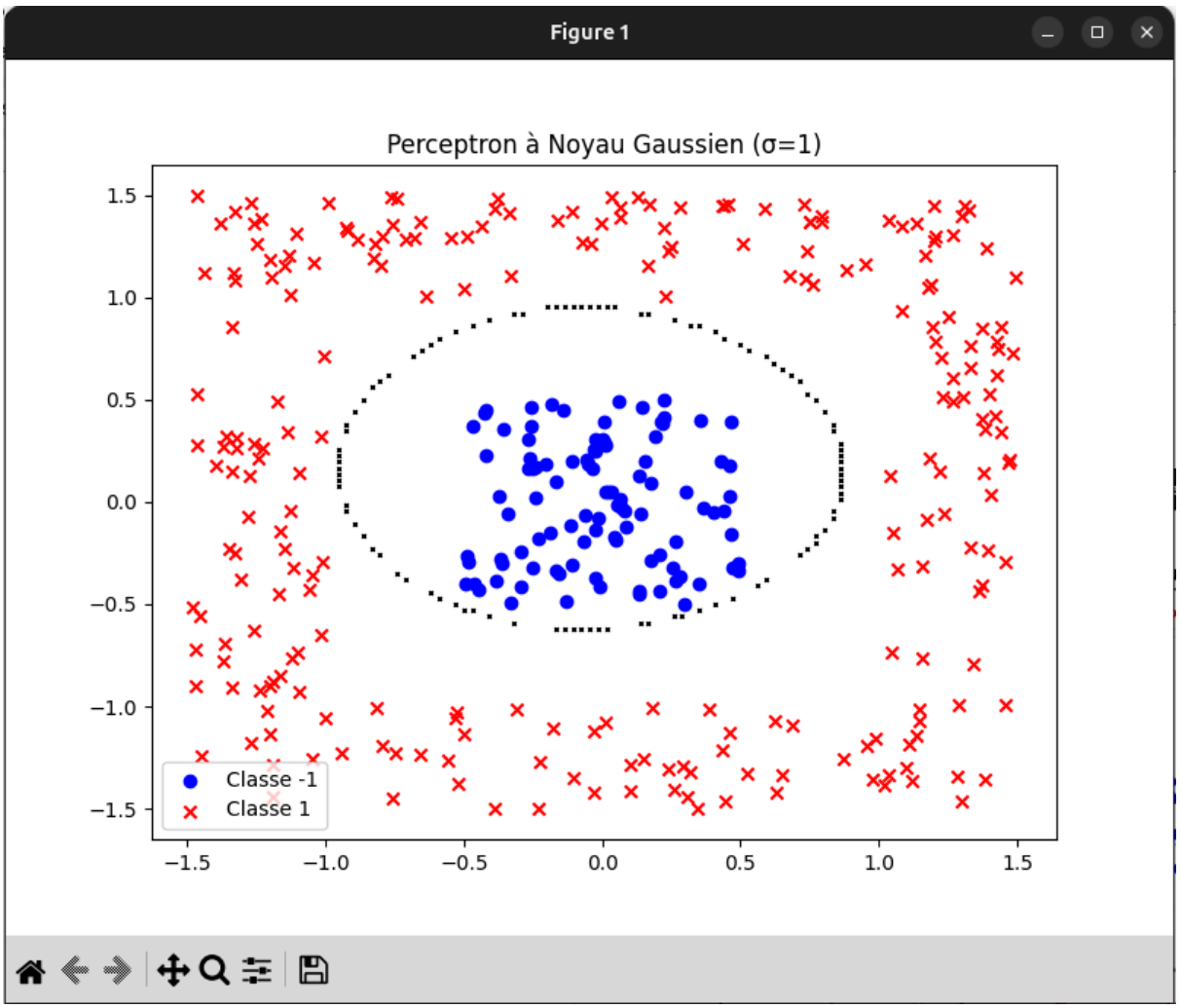
Anass EZZINE
4A FISA



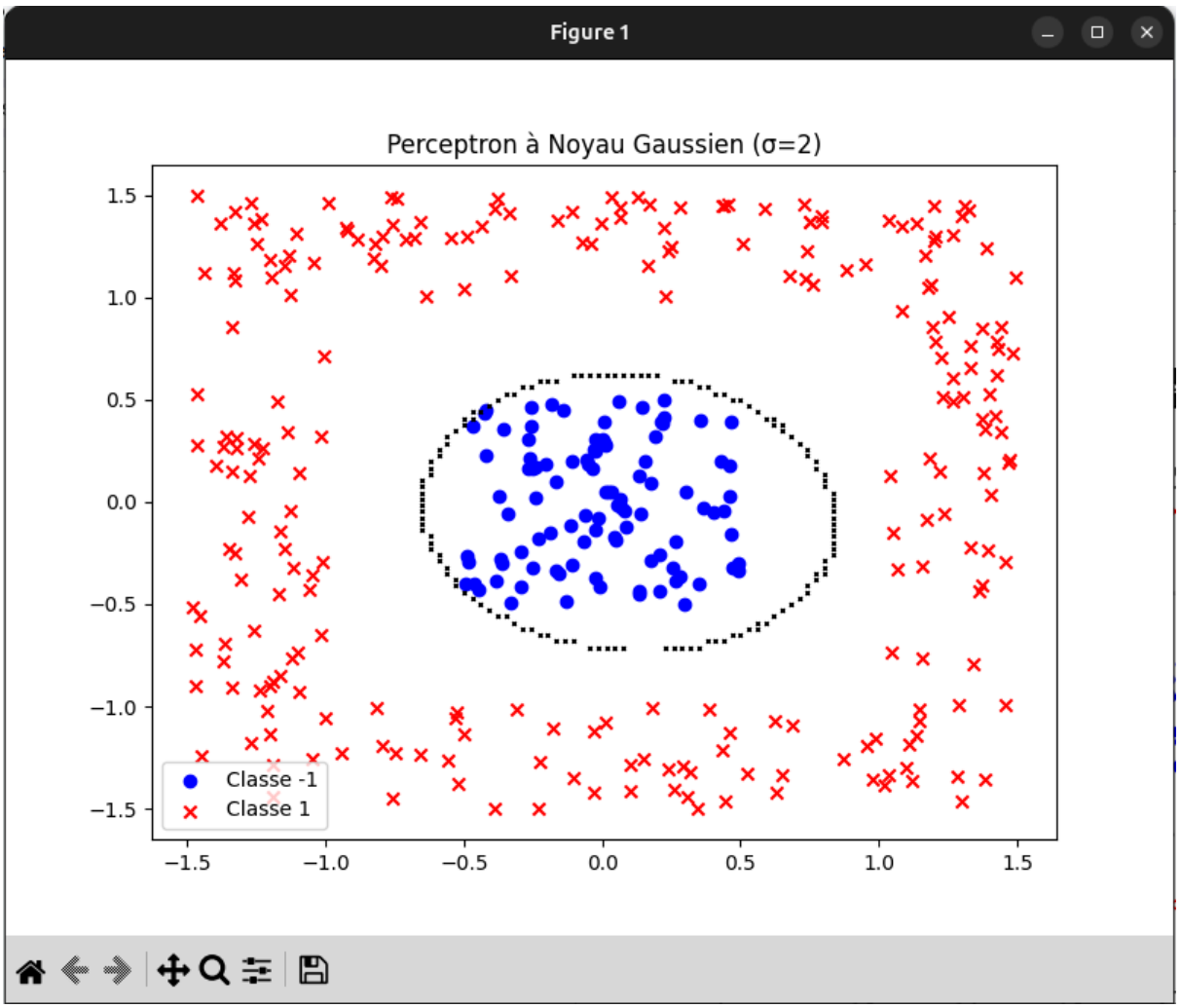
Anass EZZINE
4A FISA

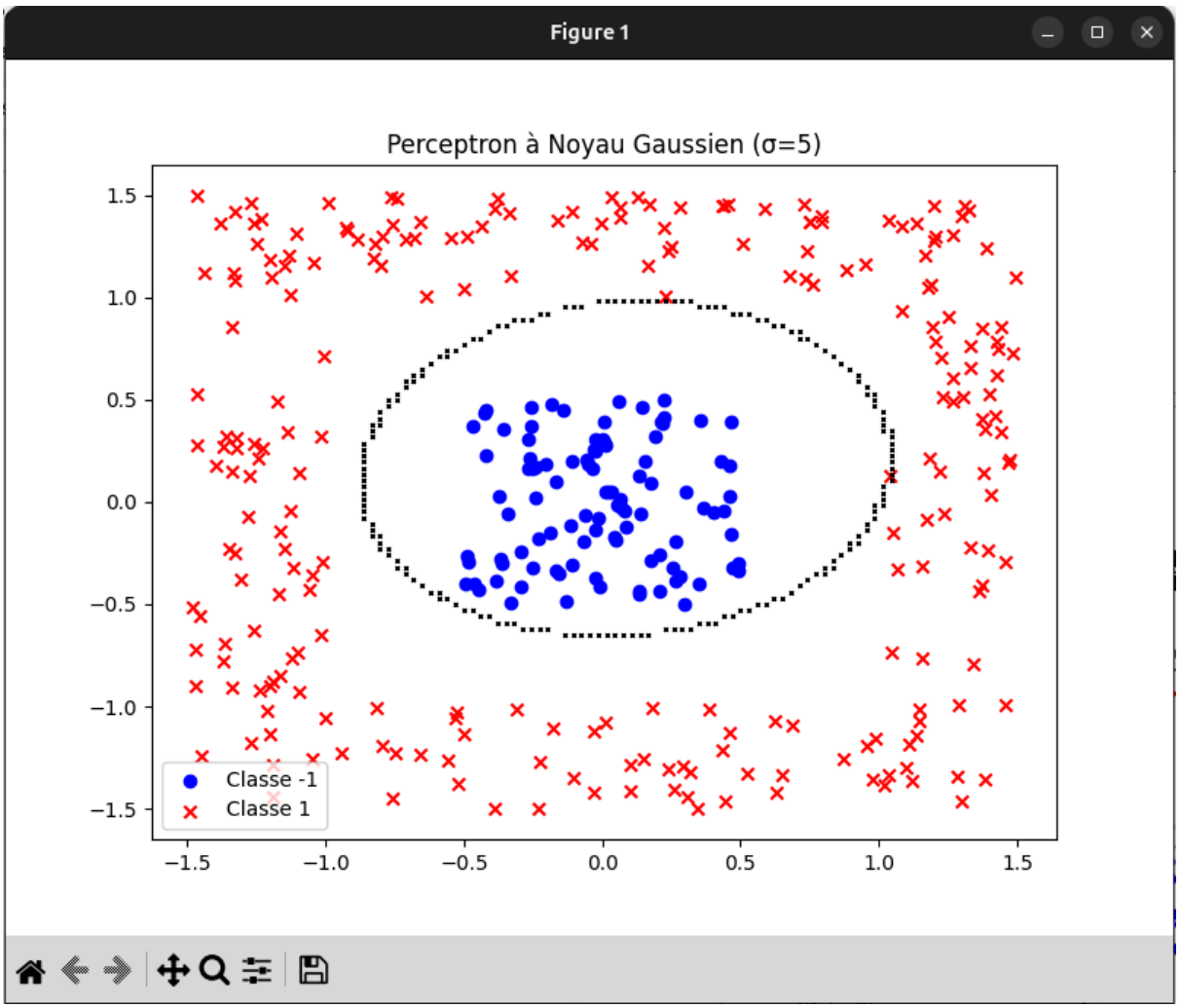


Anass EZZINE
4A FISA



Anass EZZINE
4A FISA





On observe ici l'effet de sigma pour différentes valeurs (0.2,0.5,1,2,5) :

- Pour sigma = 0.2 et 0.5, le modèle peine à faire une bonne prédiction et s'appuie excessivement sur la courbe.
- Lorsque sigma > 5, les temps de calcul deviennent trop longs, sans réelle amélioration par rapport aux résultats obtenus avec sigma compris entre 1 et 5.

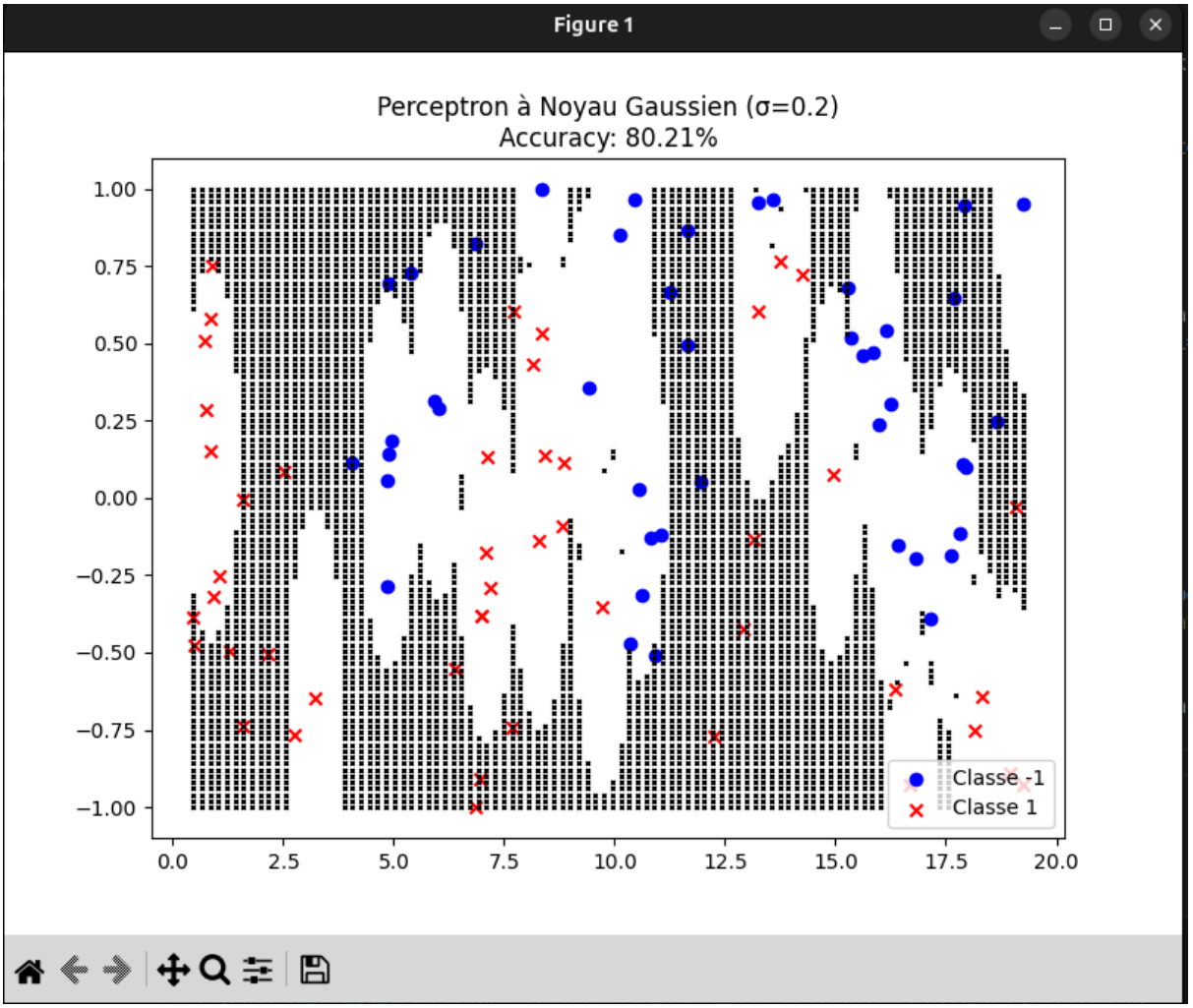
3. Exercice

Pour séparer efficacement les différentes données, il est essentiel de choisir la méthode la plus adaptée. Dans ce cas, les données ne sont pas linéairement séparables, ce qui nécessite l'utilisation d'une approche alternative. Le perceptron à noyau gaussien s'avère être le choix le plus pertinent, car il offre une bonne précision.

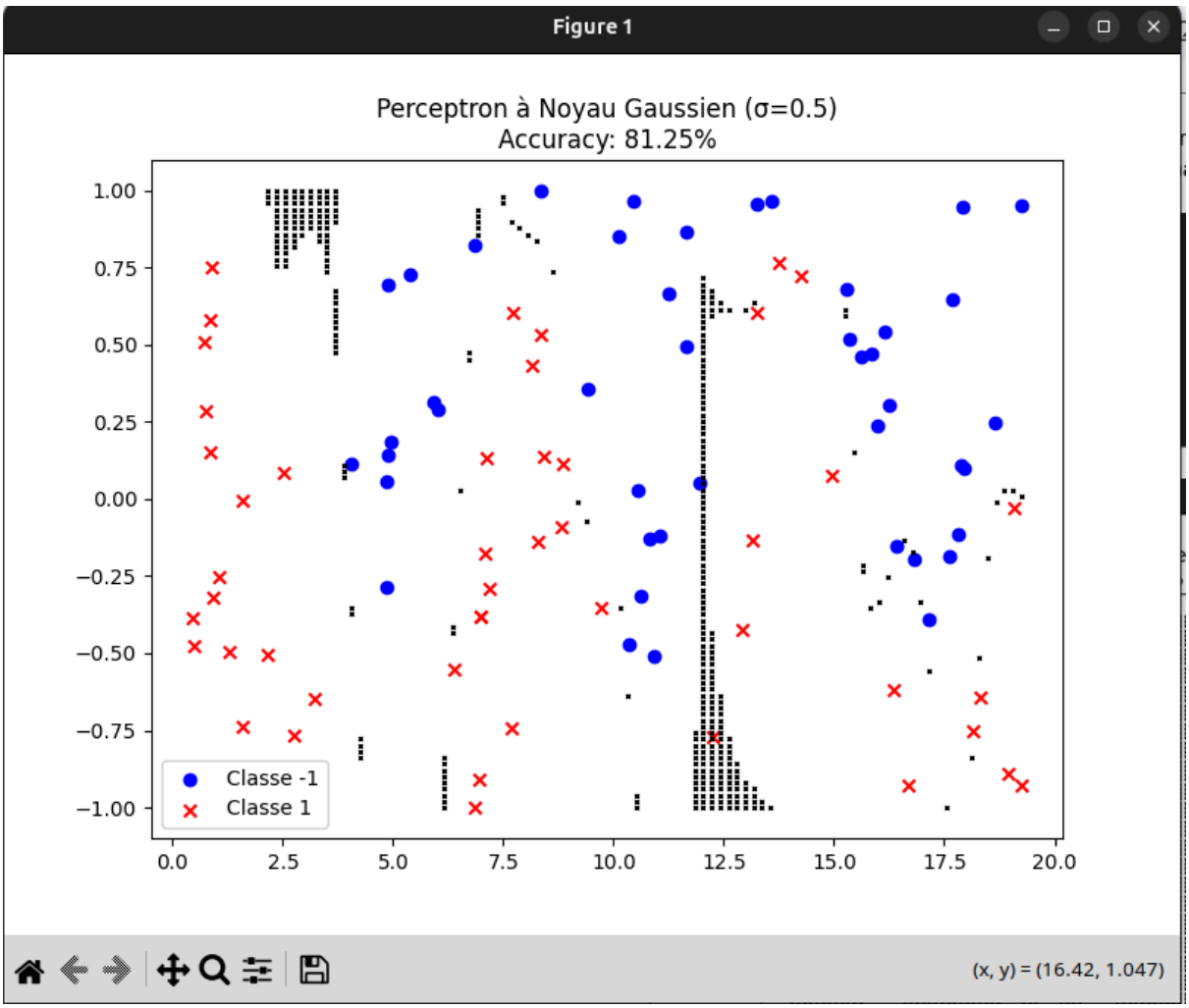
Une fonction score est utilisée pour identifier la meilleure valeur de sigma parmi celles testées. Cette fonction mesure la précision d'un modèle avancé en s'appuyant sur plusieurs éléments : les coefficients (`coeffs`), un ensemble de supports (`support_set`) et une fonction `k`, qui peut agir comme un noyau dans une version améliorée du perceptron. Elle

détermine la classe d'un échantillon en combinant linéairement les supports, puis applique la fonction `np.sign()` pour produire une classification binaire :

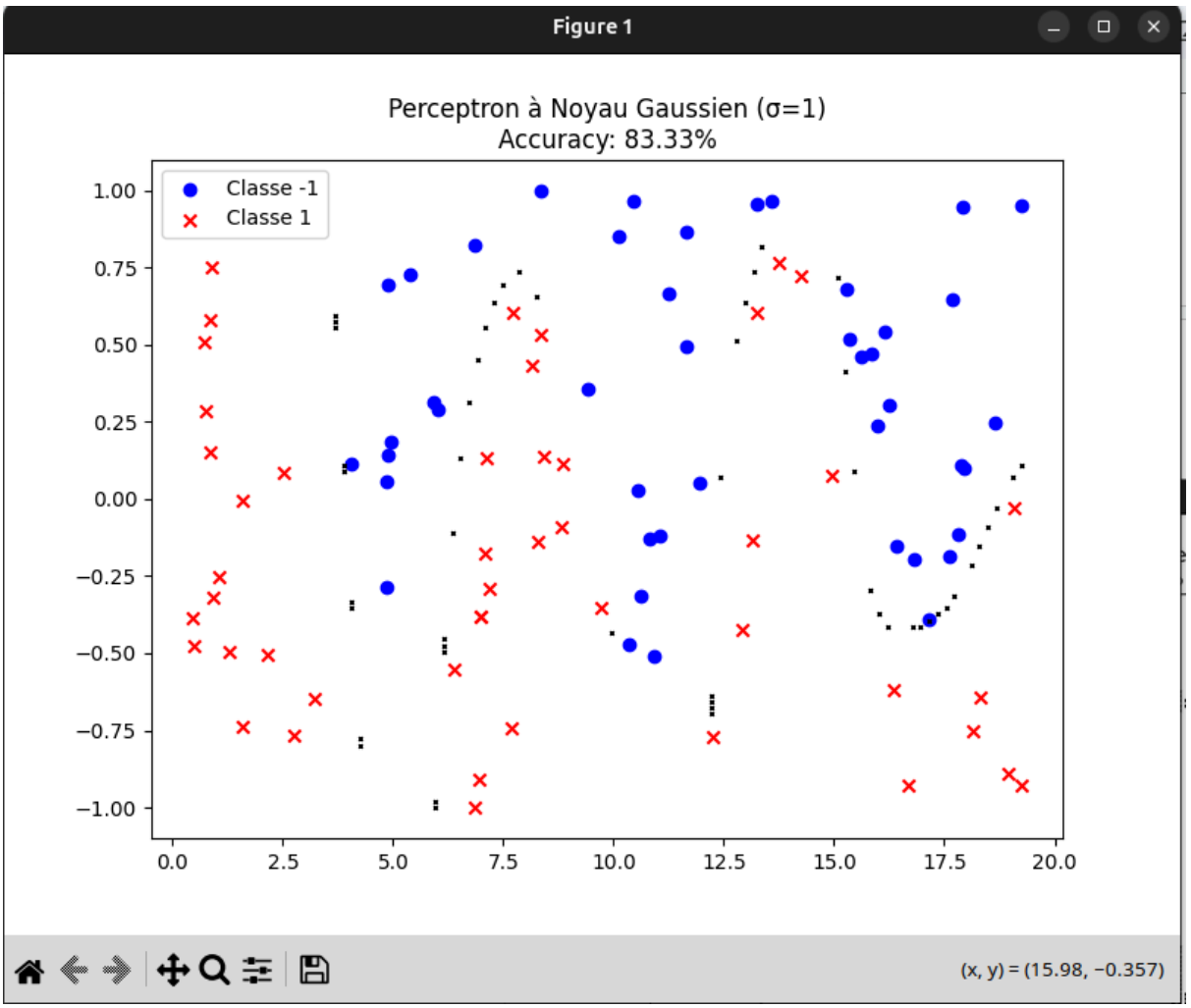
```python
def score(S, coeffs, support_set, k):
    correct_predictions = 0
    total_samples = len(S)

    for x, label in S:
        prediction = np.sign(sum(coeff * k(support, x) for coeff, support in zip(coeffs, support_set)))
        if prediction == label:
            correct_predictions += 1

    return correct_predictions / total_samples
```



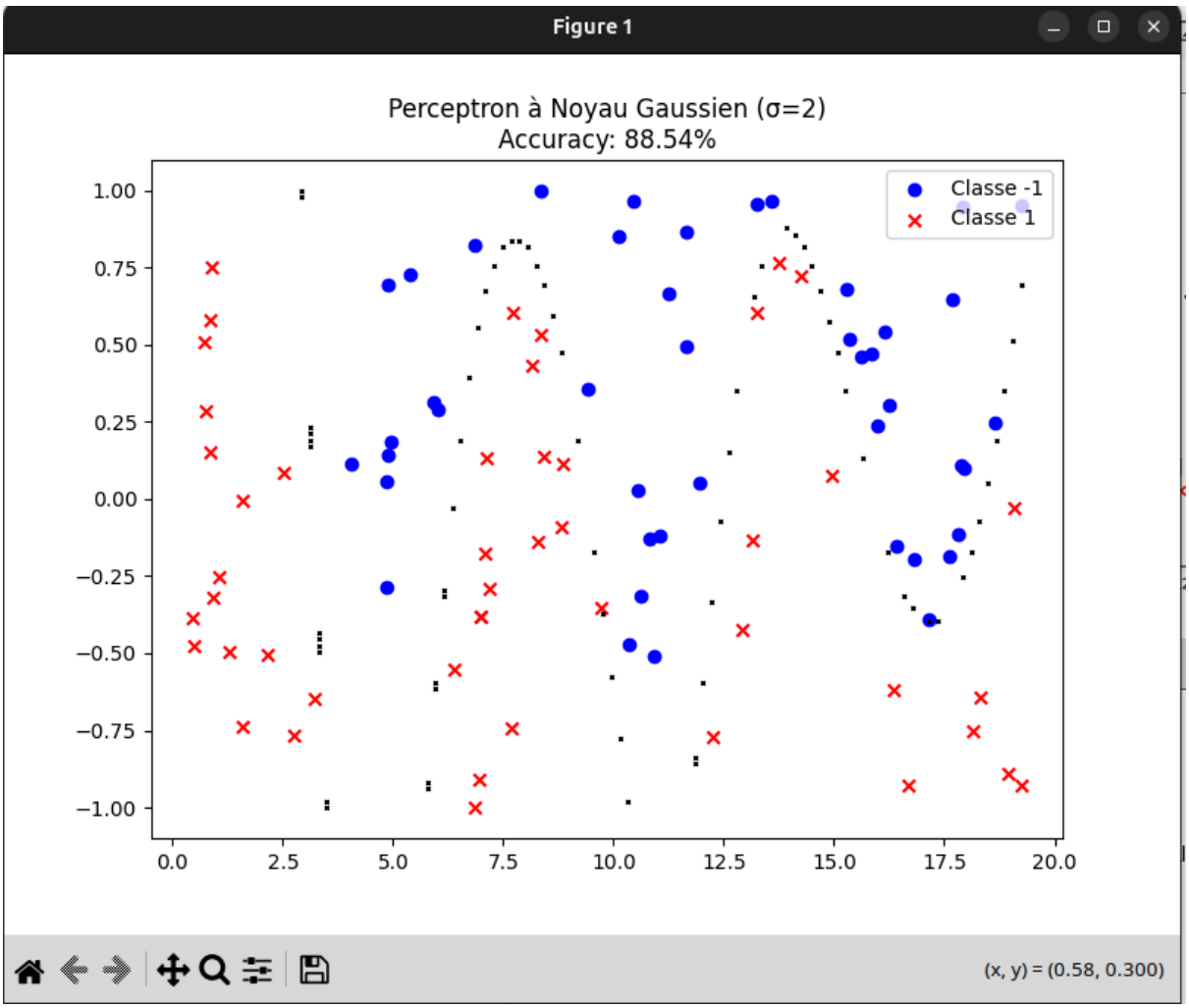
Anass EZZINE
4A FISA



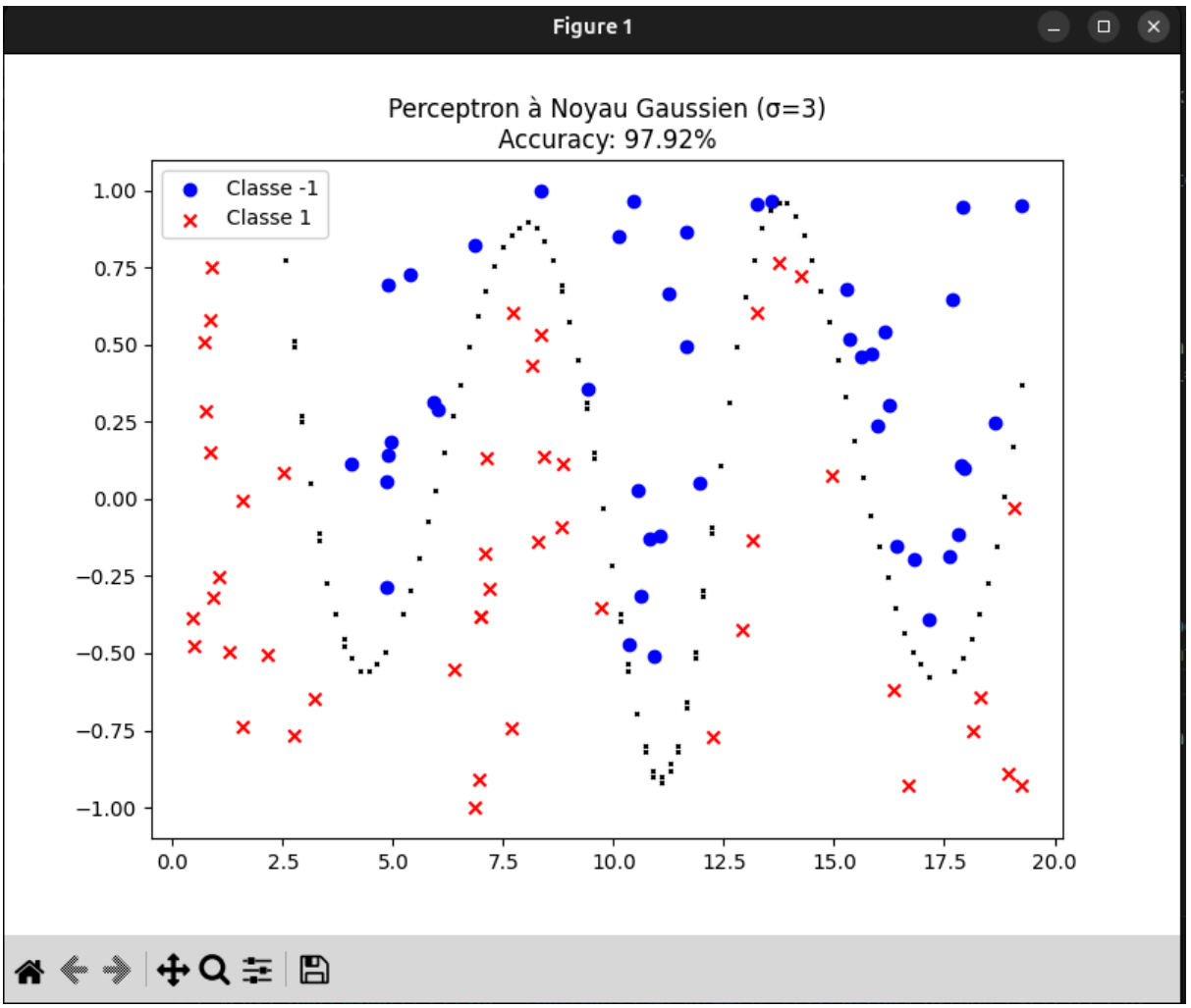
Anass EZZINE
4A FISA



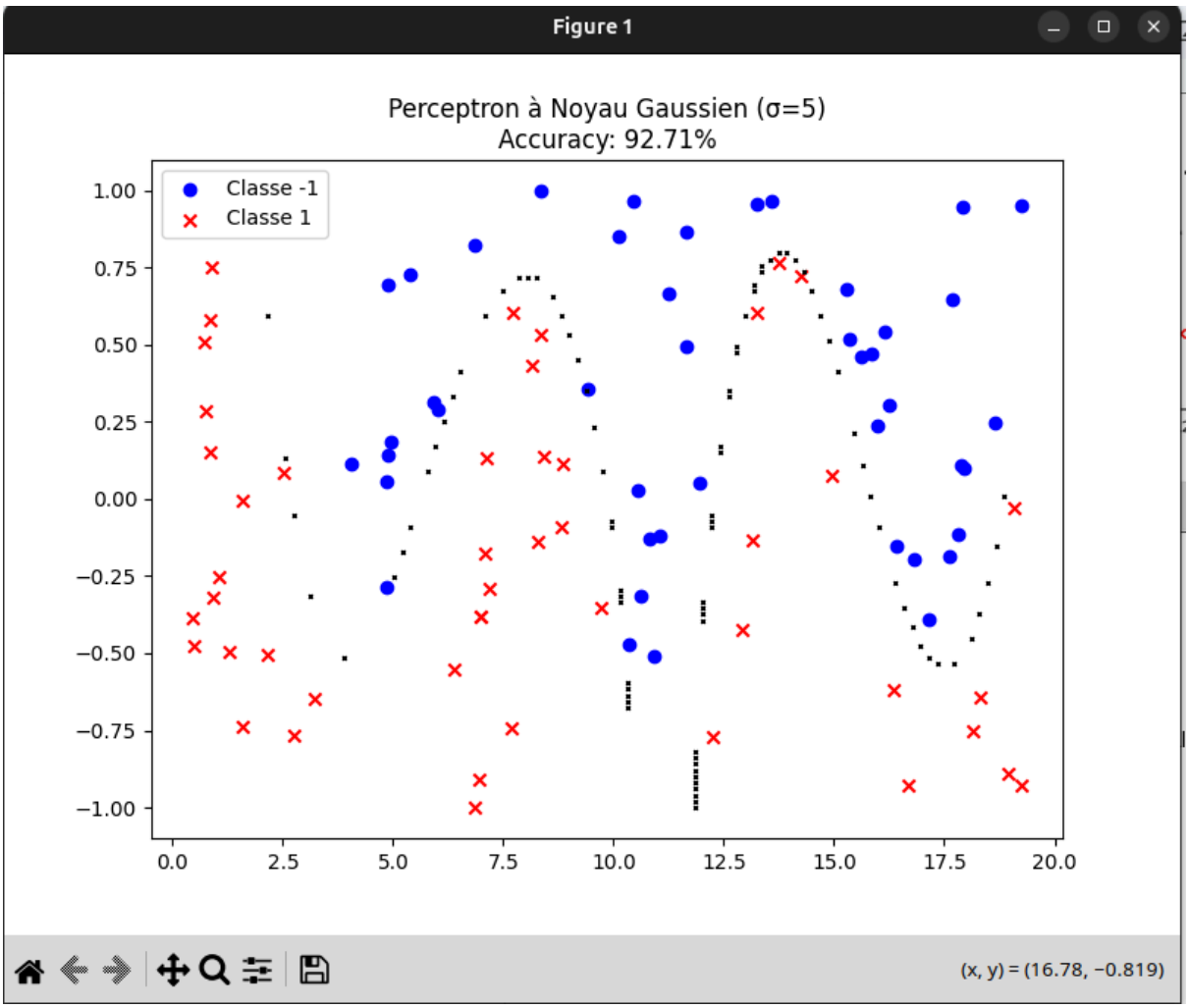
Anass EZZINE
4A FISA



Anass EZZINE
4A FISA



Anass EZZINE
4A FISA



D'après l'analyse des figures ci-dessus, la valeur optimale de sigma est 3, offrant ainsi la meilleure séparation des données.