UNIVERSITÉ CADI AYYAD
ÉCOLE SUPERIEURE DE TECHNOLOGIE

TP2 : IA Appliquée aux systèmes mécatroniques.

Le travail est élaboré par :

- **Bilal Errabia**
- **Aymen EL haimer**

Le TP est encadrée par :

- **Mme mounir**
- **Mme Hiddane**

Mécatronique et intelligence Artificielle
: MIA

Année universitaire : 2024-2025.

Le lien de project :

https://github.com/bilalerrabia/TP1-/tree/main/tp2_mounir

<u>Premier partie de code</u> :

1. generer dataset en utilisant make_blobs :

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=100, n_features=2, centers=2, random_state=0)
y = y.reshape((y.shape[0], 1))
print('Dimension de X :', X.shape)
print('Dimension de y :', y.shape)
plt.scatter(X[:, 0], X[:, 1], c=y[:, 0], cmap='summer')
plt.show()

```

1. **Imports et génération des données :** Le code importe les bibliothèques nécessaires et génère 100 points en 2D répartis en 2 clusters (X, y).
2. **Affichage des dimensions :** Les dimensions des données (X et y) sont imprimées.
3. **Visualisation :** Les points sont affichés dans un graphe 2D, colorés par classe.

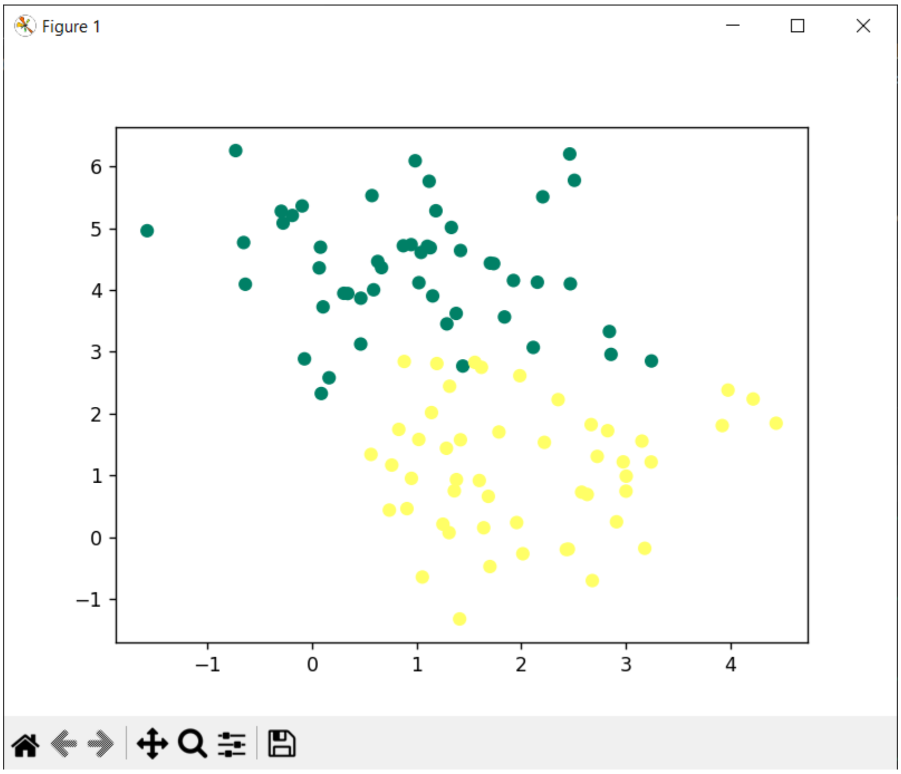

Figure 1 : visualisation de donnes

1. **Initialisation :**

```python
#partie 2:
import numpy as np

def initialisation(X):
    W = np.random.randn(X.shape[1], 1)
    b = np.random.randn(1)
    return W, b

X = np.random.randn(100, 2)
W, b = initialisation(X)
print("W (poids) :", W)
print("b (biais) :", b)

```

La fonction initialisation(X) génère des paramètres aléatoires pour un modèle. Elle crée un vecteur de poids W de dimensions ((nombre des x's dans l'equation),1) et un biais scalaire B, tous deux tirés d'une distribution normale standard via (np.random.randn). Ces paramètres sont retournés pour être utilisés dans un modèle.

Ces valeurs sont aléatoires et varient à chaque exécution, assurant une initialisation différente à chaque fois.

```
PS C:\Users\user\Desktop\tp2_mounir> & "
Dimension de X : (100, 2)
Dimension de y : (100, 1)
W (poids) : [[-0.18162163]
 [ 0.76667175]]
b (biais) : [-0.6504794]
PS C:\Users\user\Desktop\tp2_mounir> & "
Dimension de X : (100, 2)
Dimension de y : (100, 1)
W (poids) : [[-0.96435572]
 [-0.86864072]]
b (biais) : [-1.79728861]
PS C:\Users\user\Desktop\tp2_mounir> & "
Dimension de X : (100, 2)
Dimension de y : (100, 1)
W (poids) : [[-0.40295307]
 [ 0.82492446]]
b (biais) : [-0.15065908]
```

Figure 2 : les diferantes valeur de W et B.

2. **Modèle :**

```python
# modele:

X = np.random.randn(100, 2)
W, b = initialisation(X)

def model(X, W, b):
    z=np.dot(X,W)+b
    A= 1/(1+np.exp(-z))
    return A

A= model(X, W, b)
print(A.shape)
```

La fonction model (X, W, b) calcule z=X·W+b puis applique la fonction sigmoïde A=1/1+e(−z) Elle retourne A, une matrice de probabilités ou activations. La forme de A est (100, 1) si X contient 100 exemples.

3. **La fonction Coût :**

```python
def log_loss (A,y):
    cout = (1/len(y))*np.sum(-y*np.log(A)-(1-y)*np.log(1-A))
    return cout
Cout= log_loss(A,y)
print(Cout)
```

La formule utilisée est :

$$\text{cout} = \frac{1}{N}\sum_{i=1}^{N}\left(-y_i \log(A_i) - (1-y_i)\log(1-A_i)\right)$$

```
PS C:\Users\user\Desktop\tp2_mounir> & "
Dimension de X : (100, 2)
Dimension de y : (100, 1)
(100, 1)
0.7907118288597742
PS C:\Users\user\Desktop\tp2_mounir> & "
Dimension de X : (100, 2)
Dimension de y : (100, 1)
(100, 1)
1.0724274587710774
PS C:\Users\user\Desktop\tp2_mounir> & "
Dimension de X : (100, 2)
Dimension de y : (100, 1)
(100, 1)
0.9072566963300537
PS C:\Users\user\Desktop\tp2_mounir>
```

Figure 3 :les diferants valeur de la fonction cout

Cette valeur dépendra des valeurs spécifiques de A et y générées aléatoirement à chaque exécution.

4. Calcul des gradients :

```python

def gradients (A, X, y):
    dW = (1 / len(y))* np.dot(X.T, A-y)
    db = (1/len(y)) * np.sum(A-y)
    return (dW, db)

print(gradients(A,X,y))


```

Figure 4: la fonction qui va calcule les gradients.

Les formules utilisée pour calculer les gradients :

$$dW = \frac{1}{N} X^T (A - y) \qquad\qquad db = \frac{1}{N} \sum (A - y)$$

```
(array([[-0.0377049 ],
       [-0.00628914]]), np.float64(0.3400926976868648))
PS C:\Users\user\Desktop\tp2 mounir>
```

Figure 5 : les gradients de W1 W2 et b

dW 1= -0.0377049

dW 2= -0.00628914

db = 0.3400926976868648

5.<u>Mise à jour des paramètres :</u>

```python

dW,db=gradients(A,X,y)

def update (W, b, dW, db, alpha):
    W = W - alpha*dW
    b = b - alpha*db
    return (W, b)

W,b=update (W, b, dW, db, 0.1)
print("Nouveaux W :", W)
print("Nouveau b :", b)
#α : C'est le taux d'apprentissage,
```

```
        [ 0.05474918]]), np.flo
Nouveaux W : [[-0.44050984]
 [-0.25091345]]
Nouveau b : [1.56866948]
PS C:\Users\user\Desktop\tp2 m
```

Figure 6 : les Nouveaux valeurs de W et b

La fonction **<u>update (W, b, dW, db, alpha)</u>** ajuste les paramètres WW et bb en fonction de leurs gradients et du taux d'apprentissage α\alpha. Ce dernier détermine la taille du pas pour optimiser les paramètres et minimiser la fonction de coût. Les valeurs de WW et bb sont mises à jour en soustrayant le produit des gradients et du taux d'apprentissage.

I. artificial_neurones class :

code_1.py > ArtificialNeurones > update

```python
import numpy as np
import matplotlib.pyplot as plt
class ArtificialNeurones:
    def __init__(self, alpha=0.1, n_iter=100):
        self.alpha = alpha
        self.n_iter = n_iter
        self.W = None
        self.b = None
    def model(self, X):
        z = np.dot(X, self.W) + self.b
        A = 1 / (1 + np.exp(-z))
        return A
    def log_loss(self, A, y):
        cout = (1 / len(y)) * np.sum(-y * np.log(A) - (1 - y) * np.log(1 - A))
        return cout
    def gradients(self, A, X, y):
        dW = (1 / len(y)) * np.dot(X.T, A - y)
        db = (1 / len(y)) * np.sum(A - y)
        return dW, db
    def update(self, dW, db):
        self.W -= self.alpha * dW
        self.b -= self.alpha * db
    def train(self, X, y):
        self.W = np.random.randn(X.shape[1], 1)
        self.b = np.random.randn(1)
        cout = []
        for i in range(self.n_iter):
            A = self.model(X)
            cost = self.log_loss(A, y)
            cout.append(cost)
            dW, db = self.gradients(A, X, y)
            self.update(dW, db)
        return self.W, self.b, cout
    def plot_cost(self, cost):
        plt.plot(cost)
        plt.xlabel('Itérations')
        plt.ylabel('Coût')
        plt.title('Évolution de la fonction Coût')
        plt.show()
X = np.random.randn(100, 2)
y = np.random.randint(0, 2, size=(100, 1))
alpha = 0.1
n_iter = 100
model = ArtificialNeurones(alpha, n_iter)
W, b, cost = model.train(X, y)
print("Valeur finale de W :", W)
print("Valeur finale de b :", b)
print("le cout ", cost)
model.plot_cost(cost)
```

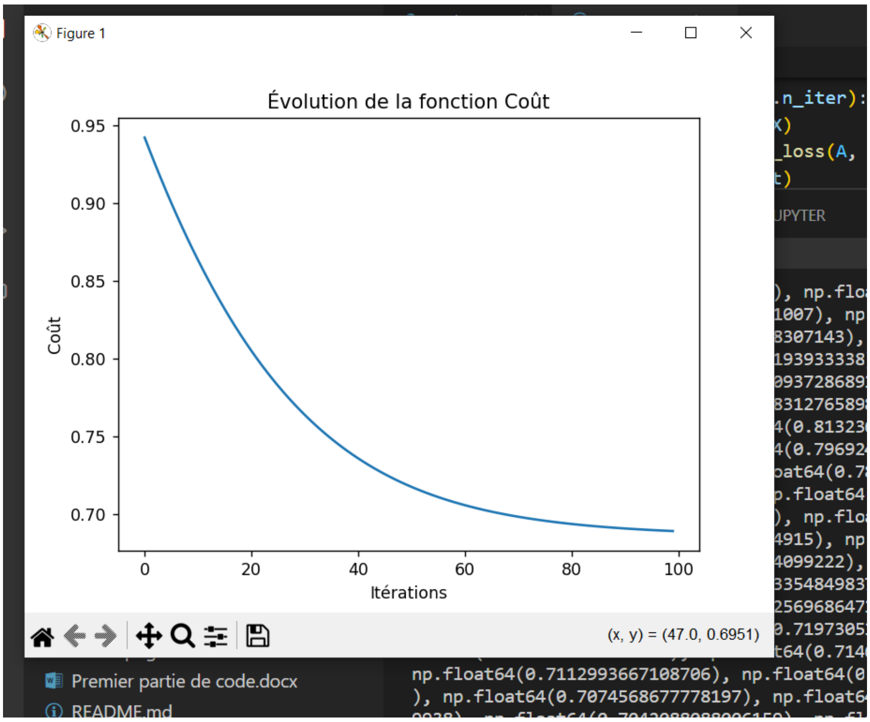

Figure7 : Visualisation de la courbe du coût

```
Valeur finale de W : [[-0.20812566]
 [ 0.35225773]]
Valeur finale de b : [-0.00213232]
```

Figure8 : les Valeur finale des paramètres.

6. **predict fonction :**

```python
def predict(X, W, b):
    A = 1 / (1 + np.exp(-(np.dot(X, W) + b)))  # Calcul de la sigmoide
    return A >= 0.5
```

La fonction predict(X, W, b) calcule les prédictions du modèle en appliquant la sigmoïde, puis compare les résultats à 0.5. Si A≥0.5A \geq 0.5, elle retourne True (classe 1), sinon False (classe 0). Elle est utilisée pour la classification binaire.

7. **accuracy_score :**

```python
from sklearn.metrics import accuracy_score

y_pred = predict(X, W, b)
print("Accuracy:", accuracy_score(y, y_pred))

X_nouveau = np.array([[2, 1], [2, 5], [3, 1.5], [2.5, 1.5]])

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='summer')
plt.scatter(X_nouveau[:, 0], X_nouveau[:, 1], c='red')
plt.show()

predictions = predict(X_nouveau, W, b)
print("Prédictions pour les nouveaux exemples:", predictions)

X0 = np.linspace(-1, 4, 100)
X1 = (-W[0] * X0 - b) / W[1]
plt.plot(X0, X1, c='orange', lw=3)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='summer')
plt.scatter(X_nouveau[:, 0], X_nouveau[:, 1], c='red')
plt.show()
```

Le code calcule l'accuracy du modèle sur les données d'entraînement en utilisant accuracy_score, ce qui évalue la performance globale. Ensuite, il crée des nouveaux exemples (XnouveauX_{nouveau}Xnouveau), les affiche en rouge sur le graphique et prédit leurs classes avec la fonction predict. La frontière de décision est tracée, séparant les deux classes. Les prédictions des nouveaux exemples sont vérifiées par rapport à cette frontière pour évaluer leur cohérence.

8. **Exécuter le programme, compter le nombre d’erreur et interpréter :**

```python

# Calcul des erreurs
errors = np.sum(y_pred.flatten() != y.flatten())
print(f"Nombre d'erreurs: {errors}")
```

```
Valeur finale de W : [[ 0.03228834]
 [-0.12574177]]
Valeur finale de b : [0.18951315]
Accuracy: 0.54
Prédictions pour les nouveaux exemples: [[ True]
 [False]
 [ True]
 [ True]]
Nombre d'erreurs: 46
PS C:\Users\user\Desktop\tp2_mounir>
```

Figure 7: nombre d'erreurs=46

9. Interprétation :

Avec **46 erreurs**, ton modèle montre une performance faible, ce qui signifie qu’il n’a pas bien appris les relations entre X et y. Cela peut être dû à un taux d’apprentissage (α) trop bas ou à un nombre d’itérations (nitern_) trop faible. Si les données ne sont pas linéairement séparables, un modèle linéaire comme celui-ci peut avoir du mal à bien classer les points. nous pourrons essayer d’ajuster les hyperparamètres.