

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

Premier partie de code :

1. Générer un dataset en utilisant make_blobs :

```
code_1.py x README.md
code_1.py > ...
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.datasets import make_blobs
4 X, y = make_blobs(n_samples=100, n_features=2, centers=2, random_state=0)
5 y = y.reshape((y.shape[0], 1))
6 print('Dimension de X :', X.shape)
7 print('Dimension de y :', y.shape)
8 plt.scatter(X[:, 0], X[:, 1], c=y[:, 0], cmap='summer')
9 plt.show()
10
```

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.

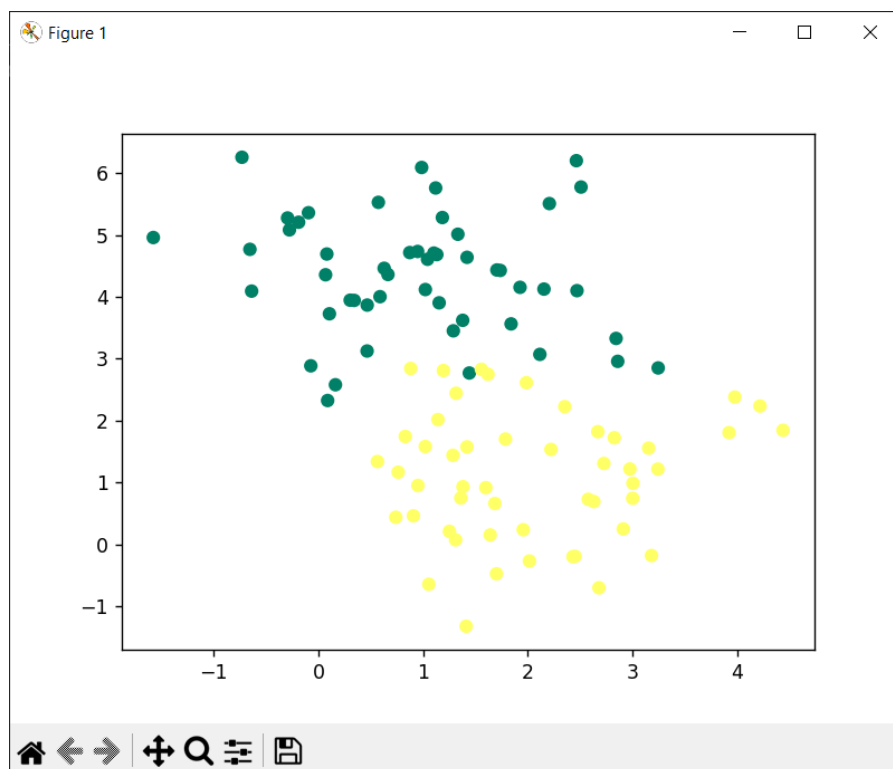


Figure 1 : visualisation de données

1. Initialisation :

```
11 #partie 2:
12 import numpy as np
13
14 def initialisation(X):
15     W = np.random.randn(X.shape[1], 1)
16     b = np.random.randn(1)
17     return W, b
18
19 X = np.random.randn(100, 2)
20 W, b = initialisation(X)
21 print("W (poids) :", W)
22 print("b (biais) :", b)
23
```

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'équation),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> & "C:\Users\user\Desktop\tp2_mounir\initialisation.py"
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> & "C:\Users\user\Desktop\tp2_mounir\initialisation.py"
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> & "C:\Users\user\Desktop\tp2_mounir\initialisation.py"
Dimension de X : (100, 2)
Dimension de y : (100, 1)
W (poids) : [[-0.40295307]
 [ 0.82492446]]
b (biais) : [-0.15065908]
```

Figure 2 : les différentes valeurs de `W` et `B`.

2. Modèle :

```
18
19     # modele:
20
21     X = np.random.randn(100, 2)
22     W, b = initialisation(X)
23
24     def model(X, W, b):
25         z=np.dot(X,W)+b
26         A= 1/(1+np.exp(-z))
27         return A
28
29     A= model(X, W, b)
30     print(A.shape)
31
```

La fonction model (X, W, b) calcule $z=X \cdot W+b$ 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 :

```
31
32     def log_loss (A,y):
33         cout = (1/len(y))*np.sum(-y*np.log(A)-(1-y)*np.log(1-A))
34         return cout
35     Cout= log_loss(A,y)
36     print(Cout)
37
38
39
```

La formule utilisée est :

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

```
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 diférants valeur de la fonction cout

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

4. Calcul des gradients :

```
39
40 def gradients (A, X, y):
41     dW = (1 / len(y))* np.dot(X.T, A-y)
42     db = (1/len(y)) * np.sum(A-y)
43     return (dW, db)
44
45 print(gradients(A,X,y))
46
47
```

Figure 4: la fonction qui va calcule les gradients.

Les formules utilisées pour calculer les gradients :

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

```
(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. Mise à jour des paramètres :

```
46
47  dW,db=gradients(A,X,y)
48
49  def update (W, b, dW, db, alpha):
50      W = W - alpha*dW
51      b = b - alpha*db
52      return (W, b)
53
54  W,b=update (W, b, dW, db, 0.1)
55  print("Nouveaux W :", W)
56  print("Nouveau b :", b)
57  #α : C'est le taux d'apprentissage,
```

```
[ 0.05474918]]), np.float64(0.3400926976868648))
Nouveaux W : [[-0.44050984]
               [-0.25091345]]
Nouveau b : [1.56866948]
PS C:\Users\user\Desktop\tp2 mounir> █
```

Figure 6 : les Nouveaux valeurs de W et b

La fonction **update (W, b, dW, db, alpha)** 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
1  import numpy as np
2  import matplotlib.pyplot as plt
3  class ArtificialNeurones:
4      def __init__(self, alpha=0.1, n_iter=100):
5          self.alpha = alpha
6          self.n_iter = n_iter
7          self.W = None
8          self.b = None
9      def model(self, X):
10         z = np.dot(X, self.W) + self.b
11         A = 1 / (1 + np.exp(-z))
12         return A
13     def log_loss(self, A, y):
14         cout = (1 / len(y)) * np.sum(-y * np.log(A) - (1 - y) * np.log(1 - A))
15         return cout
16     def gradients(self, A, X, y):
17         dW = (1 / len(y)) * np.dot(X.T, A - y)
18         db = (1 / len(y)) * np.sum(A - y)
19         return dW, db
20     def update(self, dW, db):
21         self.W -= self.alpha * dW
22         self.b -= self.alpha * db
23     def train(self, X, y):
24         self.W = np.random.randn(X.shape[1], 1)
25         self.b = np.random.randn(1)
26         cout = []
27         for i in range(self.n_iter):
28             A = self.model(X)
29             cost = self.log_loss(A, y)
30             cout.append(cost)
31             dW, db = self.gradients(A, X, y)
32             self.update(dW, db)
33         return self.W, self.b, cout
34     def plot_cost(self, cost):
35         plt.plot(cost)
36         plt.xlabel('Itérations')
37         plt.ylabel('Coût')
38         plt.title('Évolution de la fonction Coût')
39         plt.show()
40 X = np.random.randn(100, 2)
41 y = np.random.randint(0, 2, size=(100, 1))
42 alpha = 0.1
43 n_iter = 100
44 model = ArtificialNeurones(alpha, n_iter)
45 W, b, cost = model.train(X, y)
46 print("Valeur finale de W :", W)
47 print("Valeur finale de b :", b)
48 print("le cout ", cost)
49 model.plot_cost(cost)
50
```

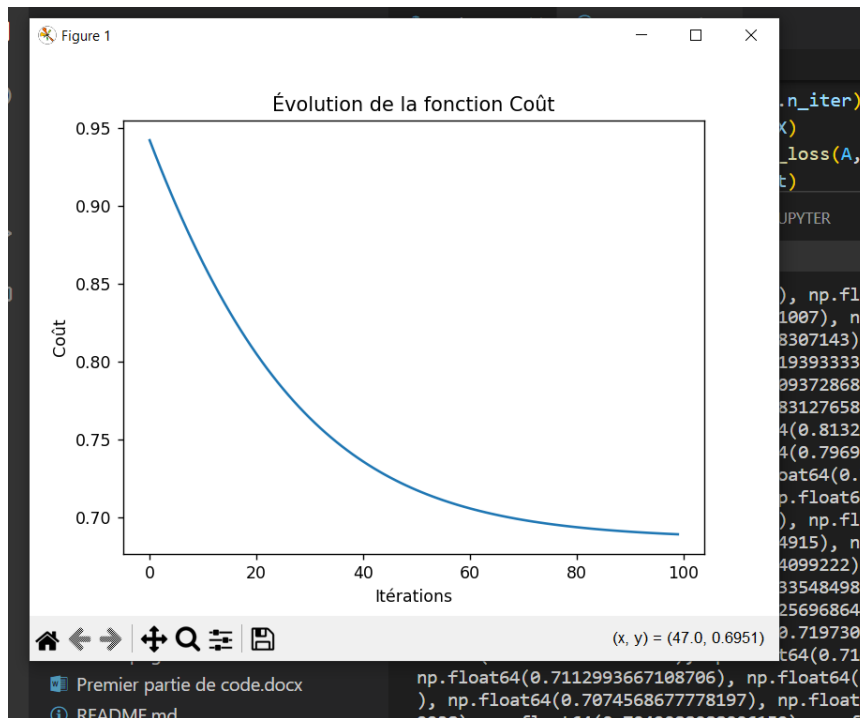



Figure7 : Visualisation de la courbe du coût

```
PS C:\users\user\Desktop\tp2_moulin... & C:/
Valeur finale de W : [[-0.20812566]
 [ 0.35225773]]
Valeur finale de b : [-0.00213232]
```

Figure8 : les Valeur finale des paramètres.

6. predict fonction :

```
57 def predict(X, W, b):
58     A = 1 / (1 + np.exp(-(np.dot(X, W) + b))) # Calcul de la sigmoïde
59     return A >= 0.5
60
61
```

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

7. accuracy_score :

```
62 from sklearn.metrics import accuracy_score
63
64 y_pred = predict(X, W, b)
65 print("Accuracy:", accuracy_score(y, y_pred))
66
67 X_nouveau = np.array([[2, 1], [2, 5], [3, 1.5], [2.5, 1.5]])
68
69 plt.scatter(X[:, 0], X[:, 1], c=y, cmap='summer')
70 plt.scatter(X_nouveau[:, 0], X_nouveau[:, 1], c='red')
71 plt.show()
72
73 predictions = predict(X_nouveau, W, b)
74 print("Prédictions pour les nouveaux exemples:", predictions)
75
76 X0 = np.linspace(-1, 4, 100)
77 X1 = (-W[0] * X0 - b) / W[1]
78 plt.plot(X0, X1, c='orange', lw=3)
79 plt.scatter(X[:, 0], X[:, 1], c=y, cmap='summer')
80 plt.scatter(X_nouveau[:, 0], X_nouveau[:, 1], c='red')
81 plt.show()
82
```

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 (`X_nouveau`), 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 :

```
82
83 # Calcul des erreurs
84 errors = np.sum(y_pred.flatten() != y.flatten())
85 print(f"Nombre d'erreurs: {errors}")
86
```

```
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 :