**Premier partie de code** :

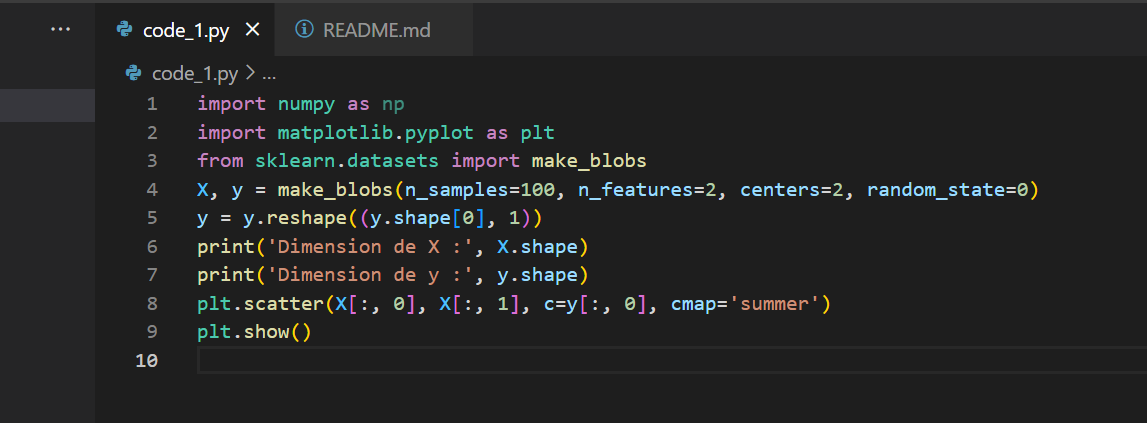
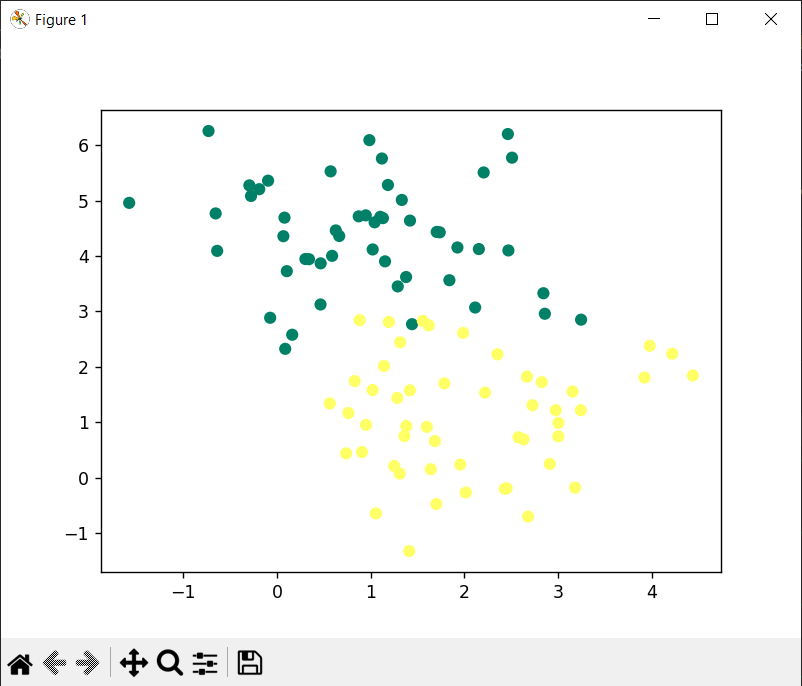
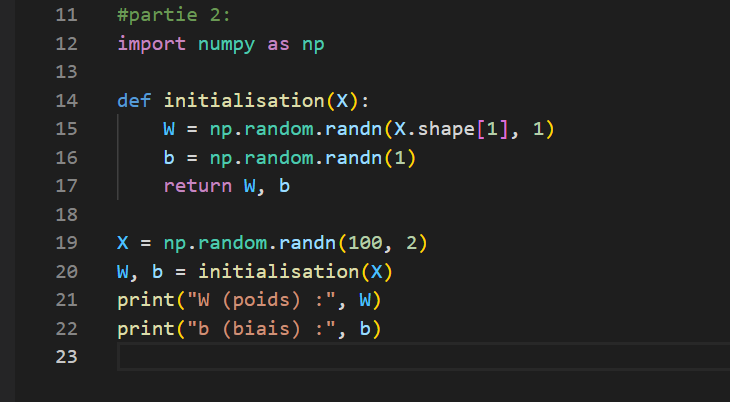
1. **generer dataset en utilisant make\_blobs :**
2. **Imports et génération des données :** Le code importe les bibliothèques nécessaires et génère 100 points en 2D répartis en 2 clusters (X, y).
3. **Affichage des dimensions :** Les dimensions des données (X et y) sont imprimées.
4. **Visualisation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

Figure 1 : visualisation de donnes

1. **Initialisation :**



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.

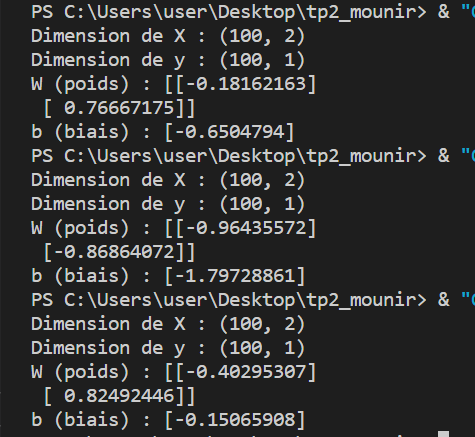
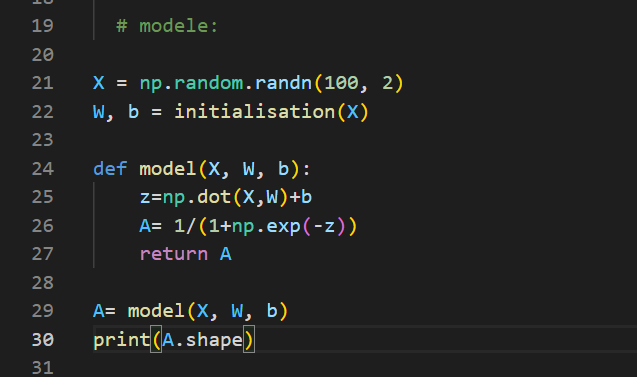
Ces valeurs sont aléatoires et varient à chaque exécution, assurant une initialisation différente à chaque fois.

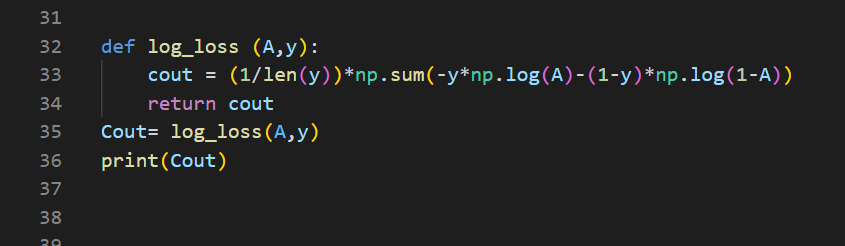
Figure 2 : les diferantes valeur de W et B.

1. **Modèle :**

****

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.

1. **La fonction Coût :**

****

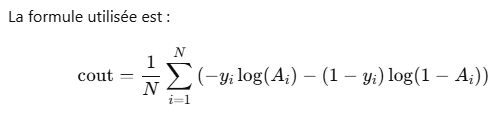
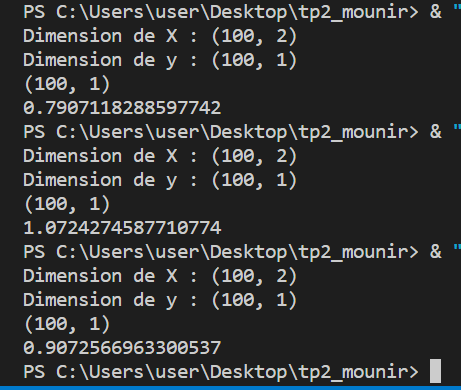
****

Figure 3 :les diferants valeur de la fonction cout

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

1. **Calcul des gradients :**

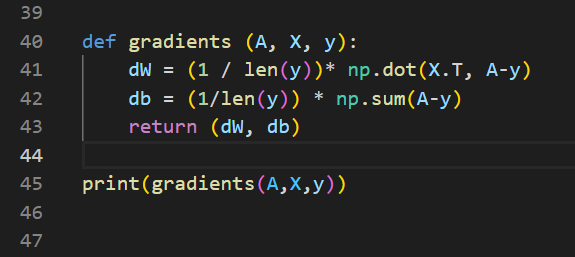
****

Figure 4: la fonction qui va calcule les gradients.

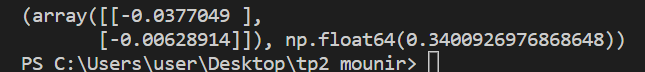
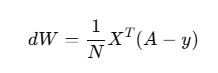
**** Les formules utilisée pour calculer les gradients :

Figure 5 : les gradients de W1 W2 et b

**dW 1= -0.0377049**

**dW 2= -0.00628914**

**db = 0.3400926976868648**

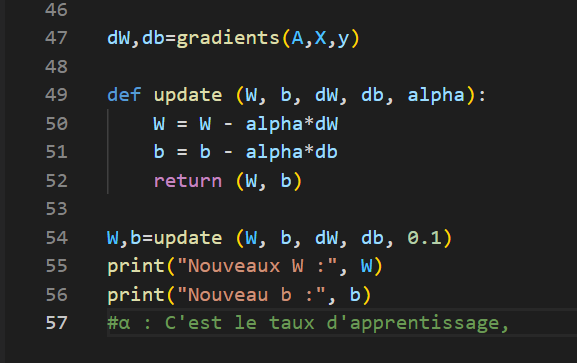
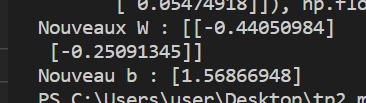
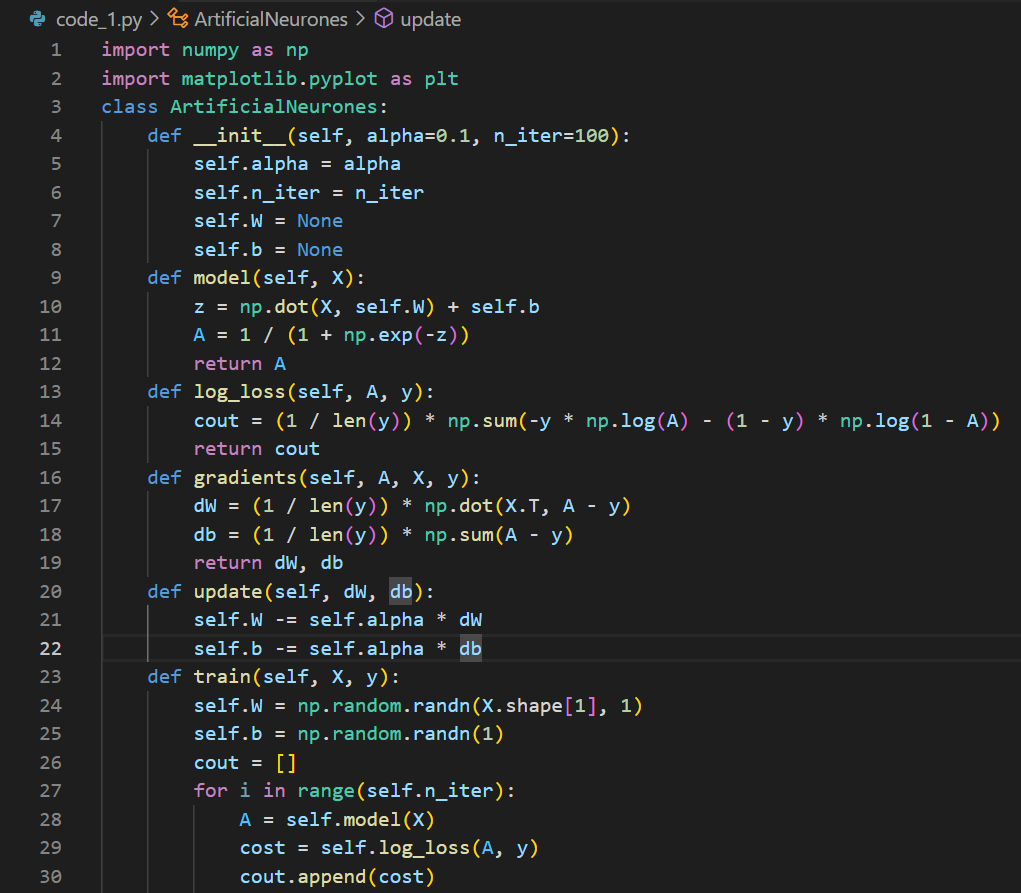
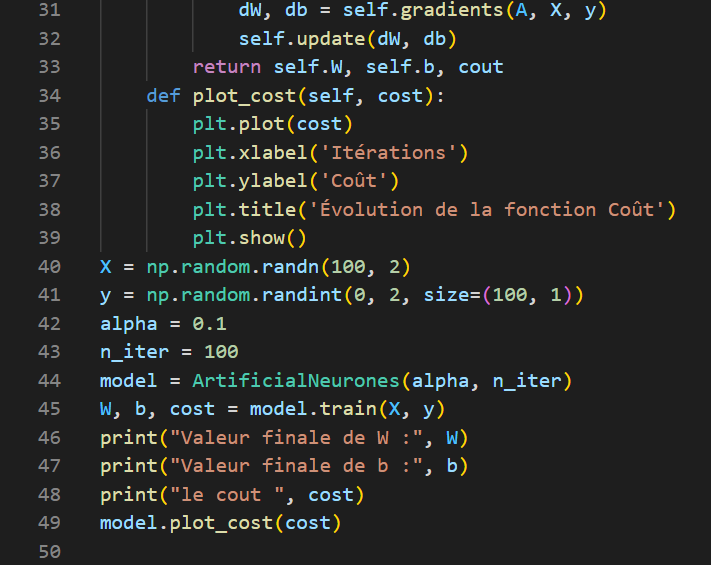
1. **Mise à jour des paramètres :**

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.

1. **artificial\_neurones class :**

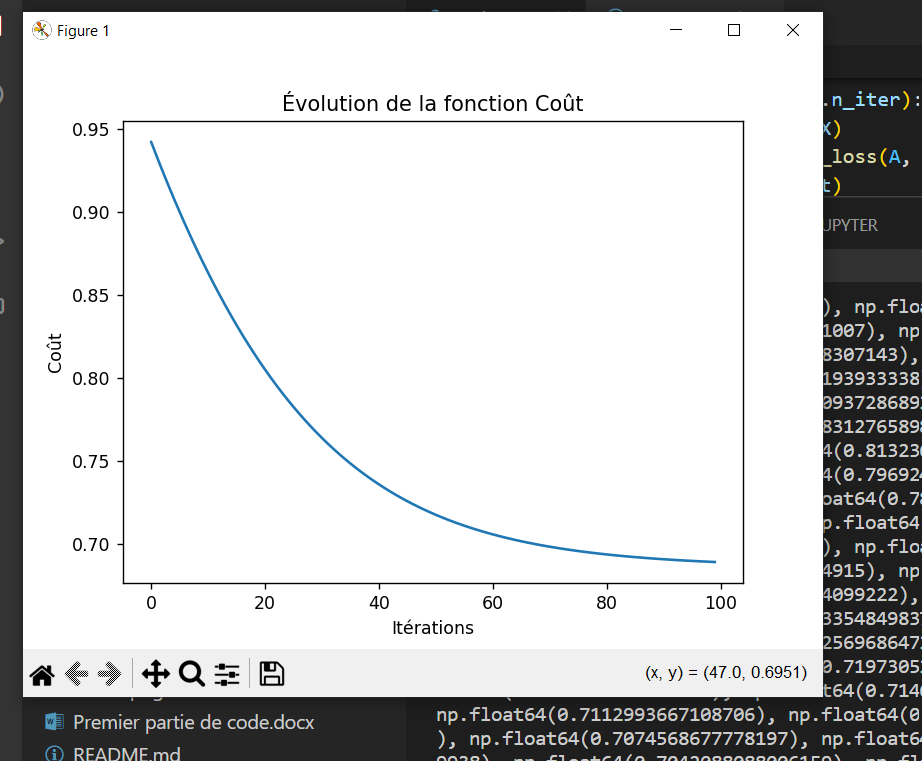
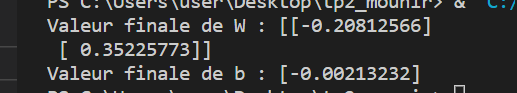
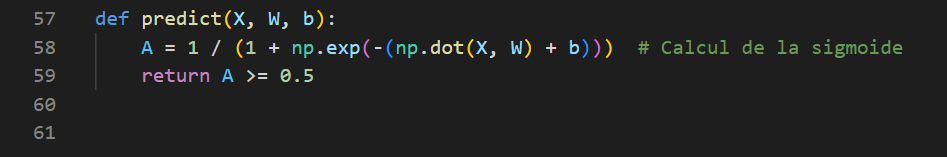
****

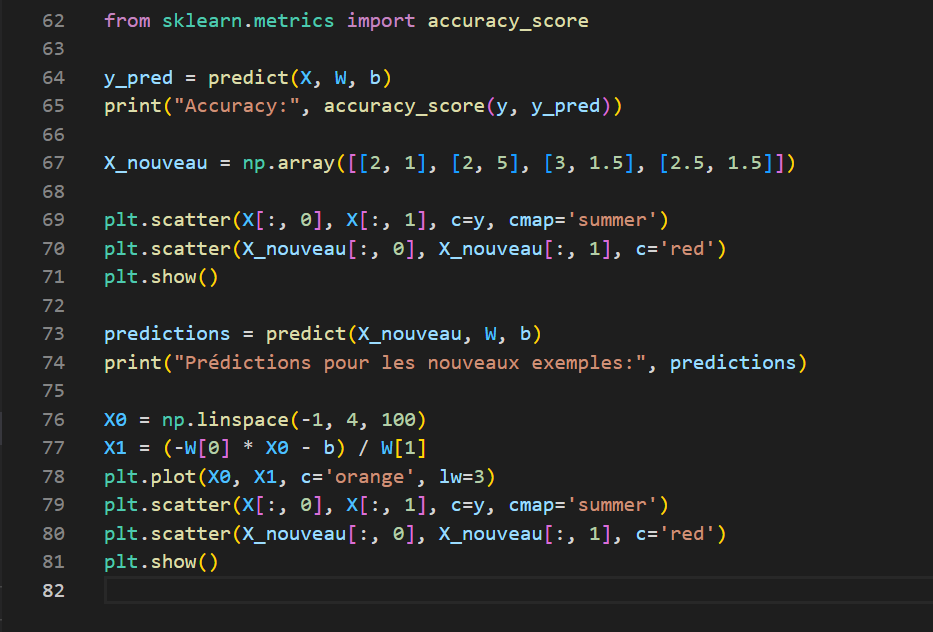
Figure8 : les Valeur finale des paramètres.

Figure7 : Visualisation de la courbe du coût

1. **predict fonction :**

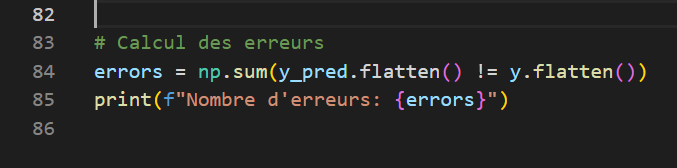


La fonction predict(X, W, b) calcule les prédictions du modèle en appliquant la sigmoïde, puis compare les résultats à 0.5. Si A≥0.5A \geq 0.5, elle retourne True (classe 1), sinon False (classe 0). Elle est utilisée pour la classification binaire.

1.  **accuracy\_score :**

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.

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



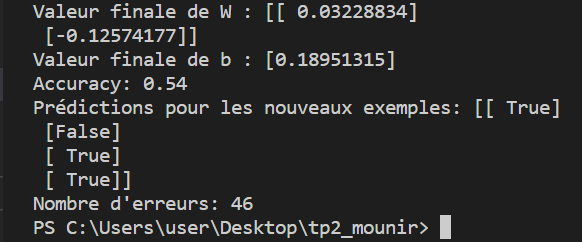
1.  **Interprétation :**

Figure 7: nombre d'erreurs=46

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.