Implémentation de la Régression Logistique (From Scratch)

OULAD ALI Ayoub

22 octobre 2025

Table des matières

1 Introduction à la régression logistique

La régression logistique est un modèle de classification utilisé pour prédire la probabilité qu'une observation appartienne à une classe binaire $Y \in \{0, 1\}$.

La régression logistique prédit une probabilité à l'aide de la **fonction sigmoïde** :

$$\mathbb{P}(Y = 1 \mid X) = \frac{1}{1 + \exp(-X^\top \beta - b)}$$

où :

— X est un vecteur de variables explicatives,

— β est le vecteur des coefficients,

— b est le biais (intercept).

1.1 Estimation des paramètres

Les paramètres β et b sont estimés par **maximisation de la vraisemblance**. Cela revient à minimiser la **fonction de perte logistique** (ou entropie croisée) :

$$L(\beta, b) = -\sum_{i=1}^{n} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

avec $p_i = \mathbb{P}(Y_i = 1 \mid X_i)$ la probabilité prédite.

2 Préparation des Données

Génération d'un jeu de données synthétiques pour la classification binaire.

```python
import numpy as np
from sklearn.datasets import make_classification
import pandas as pd
from matplotlib import pyplot as plt
from numpy.linalg import norm
from sklearn.metrics import accuracy_score

# Je choisis 100k et 500k comme taille du jeu de donnees
# car empiriquement c'est celui que j'ai trouve le meilleur
X, y = make_classification(n_samples=100000, n_features=5, n_classes=2,
    random_state=42)
df = pd.DataFrame(X, columns=['feature_1', 'feature_2', 'feature_3', 'feature_4'
    , 'feature_5'])
df['target'] = y
```

Listing 1 – Génération des données synthétiques

3 Maximum de vraisemblance et condition d'optimalité

En régression logistique, l'objectif est de trouver les paramètres $(\mathbf{B}, b)$ qui maximisent la vraisemblance du modèle, c'est-à-dire la probabilité d'observer les données données X et y sous le modèle logistique. Cela revient à minimiser la **fonction d'entropie croisée** (ou **négative log-vraisemblance**) définie par :

$$\mathcal{L}(\mathbf{B}, b) = -\sum_{i=1}^{n} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

où $p_i = \sigma(\mathbf{x}_i^\top \mathbf{B} + b)$ est la probabilité prédite pour l'observation i par la fonction sigmoïde σ. La fonction d'entropie croisée est **convexe**, ce qui garantit l'existence d'un minimum global unique (l'argmin).

3.1 Condition du maximum de vraisemblance

Le **maximum de vraisemblance** (ou le minimum de l'entropie croisée) est atteint lorsque la dérivée de la fonction par rapport aux paramètres est nulle. Autrement dit :

$$\nabla_{\mathbf{B}} \mathcal{L}(\mathbf{B}, b) = \mathbf{0}, \quad \text{et} \quad \frac{\partial \mathcal{L}(\mathbf{B}, b)}{\partial b} = 0$$

Ce sont les **conditions du premier ordre**, caractéristiques des points stationnaires (ici, le minimum global unique). En pratique, les algorithmes d'optimisation itératifs comme la descente de gradient cherchent un point où le **gradient est proche de zéro** :

$$\|\nabla \mathcal{L}(\mathbf{B}, b)\| \approx 0$$

Cela constitue un critère d'arrêt courant pour l'optimisation.

4 Fonctions Utilitaires et Gradient

Implémentation des fonctions sigmoïde, prédiction, log-vraisemblance, entropie croisée et calcul des gradients.

```python
def sigmoid(z):
    # Ajout d'un clip pour eviter l'overflow/underflow
    z = np.clip(z, -250, 250)
    return 1 / (1 + np.exp(-z)) # Utilisation de 1 / (1 + exp(-z))

def f(B, b, X_data=X): # Renommer X pour eviter conflit de nom
    z = X_data @ B + b
    return sigmoid(z)

def log_vraisemblance(B, b, X_data=X, y_data=y, eps_clip=1e-15):
    p = f(B, b, X_data)
    p = np.clip(p, eps_clip, 1 - eps_clip)  # evite log(0)
    # Correction: log_vraisemblance est la somme, pas la negative somme
    return (y_data * np.log(p) + (1 - y_data) * np.log(1 - p)).sum()

def entropie_croisee_binaire(B, b, X_data=X, y_data=y):
  # entropie croisee = - log vraisemblance / n
  n_samples = X_data.shape[0]
  if n_samples == 0:
      return 0
  # Utiliser la moyenne pour la stabilite numerique et la comparaison
  return -log_vraisemblance(B, b, X_data, y_data) / n_samples

def grad_B_entropie_croisee_binaire(B, b, X_data=X, y_data=y):
  n_samples = X_data.shape[0]
  if n_samples == 0:
      return np.zeros_like(B)
  # Gradient de l'entropie croisee MOYENNE
  return (f(B, b, X_data) - y_data) @ X_data / n_samples

def grad_b_entropie_croisee_binaire(B, b, X_data=X, y_data=y):
  n_samples = X_data.shape[0]
  if n_samples == 0:
      return 0
  # Gradient de l'entropie croisee MOYENNE
  return (f(B, b, X_data) - y_data).sum() / n_samples
```

Listing 2 – Fonctions de base et gradients

5 Compte-rendu : Difficultés rencontrées lors de l'implémentation

5.1 Génération des données

Les données ont été générées à l'aide de la fonction `make_classification` de `sklearn.datasets` avec 5 features et deux classes. Cette étape ne posait pas de difficulté particulière, mais a nécessité de bien comprendre la structure de la matrice X pour l'analyse suivante.

5.2 Problèmes de convergence

L'implémentation initiale d'une descente de gradient naïve a rencontré plusieurs obstacles :

— **Mauvais choix du pas** ρ : Des valeurs trop grandes ont fait diverger l'entropie croisée ; des valeurs trop petites ont ralenti la convergence ou causé des erreurs numériques (`log(0)`).

— **Critère d'arrêt mal défini** : L'arrêt basé uniquement sur une tolérance ε sur la fonction coût ne suffisait pas. Un nouveau critère basé sur la variation absolue de l'entropie croisée et la norme du gradient a été utilisé :

$$\text{tol} = \max\left(|\mathcal{L}_k - \mathcal{L}_{k-1}|, \|\nabla \mathcal{L}_k\|^2\right)$$

où :

— $\mathcal{L}_k$ est la valeur de l'entropie croisée à l'itération k,

— $|\mathcal{L}_k - \mathcal{L}_{k-1}|$ mesure l'évolution de la fonction de coût,

— $\|\nabla \mathcal{L}_k\|^2 = \|\nabla_B \mathcal{L}_k\|^2 + \left(\frac{\partial \mathcal{L}_k}{\partial b}\right)^2$ est la norme au carré du gradient total.

On continue les itérations tant que tol $> \varepsilon$, avec ε un seuil de tolérance fixé (par exemple $\varepsilon = 10^{-2}$).

— **Gradients non nuls à l'optimum** : Les gradients de B et b restaient parfois élevés, suggérant que la convergence était encore incomplète ou piégée dans un minimum local (ou que la tolérance ϵ était trop grande).

6 Choix de la méthode d'optimisation

Dans ce projet, nous avons choisi d'utiliser une **méthode de descente de gradient** pour minimiser la fonction d'entropie croisée, qui correspond à la négative de la log-vraisemblance dans le cadre de la régression logistique.

Ce choix est motivé par notre **familiarité avec les méthodes de gradient**, développée dans le cadre du **cours d'optimisation**. La descente de gradient est une méthode simple à mettre en œuvre, tout en étant suffisamment robuste pour résoudre efficacement des problèmes d'optimisation différentiable. Elle permet d'approximer le minimum de la fonction objectif en suivant la direction du gradient opposé.

```
def descente_gradient(X_data=X, y_data=y, NitMax=100000, rho=1e-4, eps=1/100, B=
    None, b=None):
  # Initialisation si B et b ne sont pas fournis
  if B is None:
      B = np.ones(X_data.shape[1])
  if b is None:
      b = 1.0 # Utiliser un flottant pour b

  iter = 0
  gradB = grad_B_entropie_croisee_binaire(B, b, X_data, y_data)
  gradb = grad_b_entropie_croisee_binaire(B, b, X_data, y_data)
  B_list = []
  b_list = []
```

```
  entropie_croisee_binaire_liste = [entropie_croisee_binaire(B, b, X_data, y_data)]
  norme_sq = norm(gradB)**2 + gradb**2 # Norme au carre
  # Attention: la tolerance initiale doit etre basee sur la norme et non la valeur de la fonction
  tol = norme_sq # Initialisation avec la norme du gradient au carre
  print(f"Iteration 0: Entropie={entropie_croisee_binaire_liste[0]:.4f}, Norme^2 Grad={norme_sq:.4f}, Tol={tol:.4f}")

  while iter < NitMax and tol > eps:
    B_prev = B.copy() # Garder une copie pour verifier changement
    b_prev = b

    B = B - rho * gradB
    b = b - rho * gradb

    gradB = grad_B_entropie_croisee_binaire(B, b, X_data, y_data)
    gradb = grad_b_entropie_croisee_binaire(B, b, X_data, y_data)

    B_list.append(B)
    b_list.append(b)
    current_entropie = entropie_croisee_binaire(B, b, X_data, y_data)
    entropie_croisee_binaire_liste.append(current_entropie)

    norme_sq = norm(gradB)**2 + gradb**2 # Norme au carre
    # La tolerance est le max entre la variation d'entropie et la norme du gradient au carre
    tol = max(abs(entropie_croisee_binaire_liste[-2] - entropie_croisee_binaire_liste[-1]), norme_sq)

    iter += 1
    if iter % 1000 == 0: # Afficher de temps en temps
        # Verifier si les parametres changent encore significativement
        param_change_B = norm(B - B_prev) / (norm(B) + 1e-8)
        param_change_b = abs(b - b_prev) / (abs(b) + 1e-8)
        print(f"Iteration {iter}: Entropie={current_entropie:.4f}, Norme^2 Grad={norme_sq:.4f}, Tol={tol:.4f}, dParam B={param_change_B:.2e}, dParam b={param_change_b:.2e}")


  B_list = np.array(B_list)
  b_list = np.array(b_list)
  entropie_croisee_binaire_liste = np.array(entropie_croisee_binaire_liste)
  cvg = tol <= eps
  print(f"Fin: Iter={iter}, Entropie={entropie_croisee_binaire_liste[-1]:.4f}, Norme^2 Grad={norme_sq:.4f}, Tol={tol:.4f}, Converged={cvg}")
  return B, b, iter, cvg, B_list, b_list, entropie_croisee_binaire_liste

# Exemple d'appel (peut prendre du temps)
# B_res, b_res, iterations, converged, _, _, cost_history = descente_gradient(NitMax=10000, rho=1e-2, eps=1e-4)
# print(f"B = {B_res}")
# print(f"b = {b_res}")
# print(f"Iterations = {iterations}")
# print(f"Converged = {converged}")
# plt.plot(cost_history)
# plt.title("Evolution de l'entropie croisee")
# plt.xlabel("Iteration")
# plt.ylabel("Entropie Croisee Moyenne")
# plt.show()
```

Listing 3 – Implémentation de la descente de gradient

7 Visualisation de la sigmoïde

— **Objectif** : tracer $P(y = 1 \mid X)$ en fonction d'une seule variable x_i, en fixant les autres.
— **Problème initial** : la fonction sigmoïde était mal tracée (courbes plates ou inversées) car on utilisait directement `sigmoid(X)` au lieu de :

$$f(X) = \sigma(X \cdot B + b)$$

— **Solution** : On a reconstruit un X_{fixed} où seule la feature d'intérêt varie, et les autres sont fixées à la moyenne.

```
# Assurez-vous que B et b sont definis (resultats de descente_gradient)
# B_res, b_res, _, _, _, _, _ = descente_gradient() # Execution reelle si necessaire

# Supposons que B_res et b_res sont disponibles
B_example = np.array([0.598, 0.011, 1.104, 0.895, 1.116]) # Exemple de valeurs
b_example = 0.008

x_vals = np.linspace(X[:, 0].min(), X[:, 0].max(), 200)
X_fixed = np.tile(X.mean(axis=0), (len(x_vals), 1))
X_fixed[:, 0] = x_vals

def predict_proba_fixed(X_data, B_param, b_param):
    # Assurez-vous que f est accessible ou redefinissez la prediction ici
    z = X_data @ B_param + b_param
    return sigmoid(z)

p = predict_proba_fixed(X_fixed, B_example, b_example)

# plt.figure() # Creer une nouvelle figure si besoin
# plt.plot(x_vals, p, label='Sigmoide (feature 1)', color='blue')
# Pour scatter, il faut utiliser les donnees originales X et y
# Afficher un sous-echantillon pour eviter la surcharge
# sample_indices = np.random.choice(X.shape[0], 500, replace=False)
# plt.scatter(X[sample_indices, 0], y[sample_indices], alpha=0.2, color='orange', label='y reel (echantillon)', s=10)

# plt.xlabel('feature_1')
# plt.ylabel('P(y=1)')
# plt.title("Sigmoide en fonction de la feature 1")
# plt.grid(True)
# plt.legend()
# plt.show() # Decommenter pour afficher le graphique
```

Listing 4 – Visualisation de la sigmoïde par rapport à une feature

8 Implémentation de l'algorithme ADAM

L'algorithme ADAM a été testé pour améliorer la descente classique.
— **Erreurs initiales** : les coefficients de moment (β_1, β_2) étaient initialement nuls, neutralisant les avantages de l'algorithme.
— **Gestion des dimensions** : m et v ont été correctement définis comme des vecteurs de même forme que le gradient g.
— **Amélioration constatée** : ADAM a permis une convergence plus fluide, mais un bon réglage de α, β_1, β_2 restait crucial.

```
def ADAM(X_data=X, y_data=y, NitMax=100000, alpha=0.01, eps=1e-4, beta1=0.9, beta2=0.999, B=None, b=None):
  # Initialisation si B et b ne sont pas fournis
```

```
  if B is None:
    B = np.zeros(X_data.shape[1])
  if b is None:
    b = 0.0

  iter = 0
  gradB = grad_B_entropie_croisee_binaire(B, b, X_data, y_data)
  gradb = grad_b_entropie_croisee_binaire(B, b, X_data, y_data)

  m_B = np.zeros(gradB.shape)
  v_B = np.zeros(gradB.shape)
  m_b, v_b = 0.0, 0.0

  B_list = []
  b_list = []
  entropie_croisee_binaire_liste = [entropie_croisee_binaire(B, b, X_data,
    y_data)]

  norme_sq = norm(gradB)**2 + gradb**2
  tol = norme_sq # Initialisation avec la norme du gradient au carre
  print(f"Iteration 0: Entropie={entropie_croisee_binaire_liste[0]:.4f}, Norme^2
     Grad={norme_sq:.4f}, Tol={tol:.4f}")


  while iter < NitMax and tol > eps:
    B_prev = B.copy() # Pour debug
    b_prev = b

    m_B = beta1 * m_B + (1 - beta1) * gradB
    v_B = beta2 * v_B + (1 - beta2) * gradB**2
    m_b = beta1 * m_b + (1 - beta1) * gradb
    v_b = beta2 * v_b + (1 - beta2) * gradb**2

    # Correction du biais
    m_B_hat = m_B / (1 - beta1**(iter + 1))
    v_B_hat = v_B / (1 - beta2**(iter + 1))
    m_b_hat = m_b / (1 - beta1**(iter + 1))
    v_b_hat = v_b / (1 - beta2**(iter + 1))

    # Mise a jour
    B = B - alpha * m_B_hat / (np.sqrt(v_B_hat) + 1e-8)
    b = b - alpha * m_b_hat / (np.sqrt(v_b_hat) + 1e-8)

    B_list.append(B)
    b_list.append(b)
    current_entropie = entropie_croisee_binaire(B, b, X_data, y_data)
    # Verifier si l'entropie augmente (probleme de taux d'apprentissage)
    if len(entropie_croisee_binaire_liste) > 1 and current_entropie >
    entropie_croisee_binaire_liste[-1] + 1e-6:
        print(f"Attention: l'entropie augmente a l'iteration {iter+1}")
        # Option: reduire alpha, arreter, ou revenir en arriere
        # break
    entropie_croisee_binaire_liste.append(current_entropie)


    gradB = grad_B_entropie_croisee_binaire(B, b, X_data, y_data)
    gradb = grad_b_entropie_croisee_binaire(B, b, X_data, y_data)

    norme_sq = norm(gradB)**2 + gradb**2
    tol = max(abs(entropie_croisee_binaire_liste[-2] -
    entropie_croisee_binaire_liste[-1]), norme_sq)

    iter += 1
```

```
    if iter % 100 == 0: # Afficher moins souvent pour ADAM
        param_change_B = norm(B - B_prev) / (norm(B) + 1e-8)
        param_change_b = abs(b - b_prev) / (abs(b) + 1e-8)
        print(f"Iteration {iter}: Entropie={current_entropie:.4f}, Norme^2 Grad
    ={norme_sq:.4f}, Tol={tol:.4f}, dParam B={param_change_B:.2e}, dParam b={
    param_change_b:.2e}")


  B_list = np.array(B_list)
  b_list = np.array(b_list)
  entropie_croisee_binaire_liste = np.array(entropie_croisee_binaire_liste)
  cvg = tol <= eps
  print(f"Fin ADAM: Iter={iter}, Entropie={entropie_croisee_binaire_liste[-1]:.4
    f}, Norme^2 Grad={norme_sq:.4f}, Tol={tol:.4f}, Converged={cvg}")

  return B, b, iter, cvg, B_list, b_list, entropie_croisee_binaire_liste

# Exemple d'appel ADAM (peut etre plus rapide)
# B_adam, b_adam, iter_adam, cvg_adam, _, _, cost_adam = ADAM(alpha=0.01, eps=1e
    -6, NitMax=5000)
# print(f"ADAM B = {B_adam}")
# print(f"ADAM b = {b_adam}")
# plt.figure()
# plt.plot(cost_adam)
# plt.title("Evolution de l'entropie croisee (ADAM)")
# plt.xlabel("Iteration")
# plt.ylabel("Entropie Croisee Moyenne")
# plt.show()
```

Listing 5 – Implémentation de l'optimiseur ADAM

9 Descente de gradient stochastique (SGD / Mini-batch)

Nous avons initialement tenté d'implémenter une descente de gradient stochastique (SGD) pour optimiser notre modèle de régression logistique.

9.1 Principe de la SGD

La SGD (ou sa variante mini-batch) met à jour les paramètres du modèle en utilisant une seule observation (SGD) ou un petit sous-ensemble (mini-batch) à chaque itération, contrairement à la descente de gradient classique qui utilise l'ensemble des données.

Avantages :

- Convergence potentiellement plus rapide en début d'optimisation.
- Possibilité de gérer des jeux de données très volumineux (qui ne tiennent pas en mémoire).
- Peut échapper plus facilement aux minima locaux.

Inconvénients :

- Convergence plus "bruyante" et moins stable vers la fin.
- Nécessité de bien ajuster le taux d'apprentissage (souvent avec une décroissance).
- Critère d'arrêt plus complexe à définir (le gradient sur un batch n'est pas représentatif du gradient global).

9.2 Difficultés rencontrées

L'implémentation de la SGD pure (batch_size=1) s'est avérée trop coûteuse en temps de calcul pour notre jeu de données. Chaque itération nécessitait le calcul du gradient pour une seule observation. La variante mini-batch est un compromis.

9.3 Performances

Les performances finales de la SGD/mini-batch peuvent être bonnes, mais nécessitent un réglage attentif des hyperparamètres (taille du batch, taux d'apprentissage, critère d'arrêt).

```
def descente_gradient_mini_batch(X_data=X, y_data=y, NitMax=1000, rho=1e-2, eps=1e-4, batch_size=128, B=None, b=None):

  if B is None:
    B = np.zeros(X_data.shape[1])
  if b is None:
    b = 0.0

  iter = 0
  n_samples = X_data.shape[0]
  if n_samples == 0:
      print("Erreur: Dataset vide")
      return B, b, 0, False # Retourner False pour convergence

  # Pour Mini-batch, on n'utilise pas la variation de l'entropie comme critere d'arret principal
  # car elle est trop bruitee. On se base sur NitMax ou potentiellement une validation periodique.
  # Le calcul de tol initial est juste indicatif sur le premier batch.

  # Faire le premier calcul hors boucle pour initialiser entropie_liste
  indices = np.random.choice(n_samples, batch_size, replace=False)
  X_batch = X_data[indices]
  y_batch = y_data[indices]

  gradB = grad_B_entropie_croisee_binaire(B, b, X_batch, y_batch)
  gradb = grad_b_entropie_croisee_binaire(B, b, X_batch, y_batch)
  norme_sq_batch = norm(gradB)**2 + gradb**2
  current_entropie_batch = entropie_croisee_binaire(B, b, X_batch, y_batch)
  # entropie_croisee_binaire_liste = [current_entropie_batch] # On ne stocke pas forcement

  print(f"MiniBatch Iter 0: Entropie Batch={current_entropie_batch:.4f}, Norme^2 Grad Batch={norme_sq_batch:.4f}")

  while iter < NitMax: # On s'arrete principalement sur NitMax pour SGD/MiniBatch
     indices = np.random.choice(n_samples, batch_size, replace=False)
     X_batch = X_data[indices]
     y_batch = y_data[indices]

     gradB = grad_B_entropie_croisee_binaire(B, b, X_batch, y_batch)
     gradb = grad_b_entropie_croisee_binaire(B, b, X_batch, y_batch)

     B = B - rho * gradB
     b = b - rho * gradb

     # Optionnel: Affichage periodique et verification convergence sur le set complet
     if (iter + 1) % 100 == 0:
          current_entropie_full = entropie_croisee_binaire(B, b, X_data, y_data)
          gradB_full = grad_B_entropie_croisee_binaire(B, b, X_data, y_data)
          gradb_full = grad_b_entropie_croisee_binaire(B, b, X_data, y_data)
          norme_sq_full = norm(gradB_full)**2 + gradb_full**2
          print(f"MiniBatch Iter {iter+1}: Entropie Full={current_entropie_full:.4f}, Norme^2 Grad Full={norme_sq_full:.4f}")
          # Critere d'arret base sur la norme du gradient sur le set complet
          if norme_sq_full < eps:
```

```
                print(f"Convergence atteinte a l'iteration {iter+1} (norme gradient)")
                break

        iter += 1

    # Calcul final sur tout le dataset pour statut final
    final_entropie_full = entropie_croisee_binaire(B, b, X_data, y_data)
    gradB_full = grad_B_entropie_croisee_binaire(B, b, X_data, y_data)
    gradb_full = grad_b_entropie_croisee_binaire(B, b, X_data, y_data)
    norme_sq_full = norm(gradB_full)**2 + gradb_full**2
    converged = norme_sq_full <= eps # Verification a la fin
    print(f"Fin MiniBatch: Iter={iter}, Entropie Full={final_entropie_full:.4f}, Norme^2 Grad Full={norme_sq_full:.4f}, Converged={converged}")

    return B, b, iter, converged # On retourne les derniers B, b

# B_mb, b_mb, iter_mb, cvg_mb = descente_gradient_mini_batch(NitMax=2000, rho=1e-2, eps=1e-4, batch_size=64)
# print(f"MiniBatch B = {B_mb}")
# print(f"MiniBatch b = {b_mb}")
```

Listing 6 – Implémentation de la descente de gradient Mini-Batch

Remarque : La méthode nécessite souvent un grand nombre d'itérations, et la convergence est plus oscillante.

10 Vérification des résultats

Afin de vérifier la robustesse des résultats, nous générons un vecteur `y_pred` où chaque élément indique si la probabilité prédite par la sigmoïde est supérieure à 0.5. Ainsi, chaque élément de `y_pred` sera égal à 1 si la probabilité est supérieure ou égale à 0.5, et à 0 sinon.

Cela peut être formulé de la manière suivante :

$$y_{\text{pred},i} = \begin{cases} 1 & \text{si } \hat{P}(y_i = 1 \mid X_i) \geq 0.5 \\ 0 & \text{sinon} \end{cases}$$

Cela nous permet de transformer les probabilités continues en prédictions binaires, que nous pourrons ensuite comparer avec les vraies valeurs y pour évaluer la performance du modèle.

10.1 Matrice de confusion

Une des méthodes les plus courantes pour mesurer sa performance est la **matrice de confusion**. Elle fournit une vue d'ensemble de la précision d'un modèle de classification en comparant les prédictions faites par le modèle avec les véritables étiquettes de classe.

Elle contient quatre éléments principaux :

- **Vrai Positif (TP)** : Le modèle a correctement prédit la classe positive (1).
- **Faux Positif (FP)** : Le modèle a prédit la classe positive alors que l'étiquette réelle est négative (0). (Erreur de Type I)
- **Vrai Négatif (TN)** : Le modèle a correctement prédit la classe négative (0).
- **Faux Négatif (FN)** : Le modèle a prédit la classe négative alors que l'étiquette réelle est positive (1). (Erreur de Type II)

		Prédit	
		Positif (1)	Négatif (0)
Réel	Positif (1)	TP	FN
	Négatif (0)	FP	TN

À partir de cette matrice, nous pouvons calculer plusieurs indicateurs :

1. **Exactitude (Accuracy)** : Proportion de prédictions correctes.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

2. **Précision (Precision)** : Proportion des prédictions positives qui sont réellement positives.

$$\text{Precision} = \frac{TP}{TP + FP}$$

3. **Rappel (Recall) ou Sensibilité** : Proportion des vrais positifs détectés par le modèle.

$$\text{Recall} = \frac{TP}{TP + FN}$$

4. **Score F1** : Moyenne harmonique entre la précision et le rappel.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

```python
# Supposons que B_res, b_res sont les resultats de descente_gradient
# et B_mb, b_mb sont les resultats de descente_gradient_mini_batch

# Utiliser de nouvelles donnees de test
X_new, y_new = make_classification(n_samples=1000, n_features=5, n_classes=2, random_state=123)

# Predictions avec la descente classique (exemple)
B_gd_example = np.array([0.512, 0.011, 1.016, 1.026, 1.014]) # Remplacer par vos vraies valeurs
b_gd_example = 0.008

y_pred_gd = f(B_gd_example, b_gd_example, X_new)
y_pred_gd_class = (y_pred_gd >= 0.5).astype(int)
accuracy_gd = accuracy_score(y_new, y_pred_gd_class)
print(f"Precision (Accuracy) pour la descente classique: {accuracy_gd:.3f}")

# Predictions avec Mini-Batch (exemple)
B_mb_example = np.array([0.369, 0.070, 0.853, 1.376, 0.716]) # Remplacer par vos vraies valeurs
b_mb_example = 0.057

y_pred_mb = f(B_mb_example, b_mb_example, X_new)
y_pred_mb_class = (y_pred_mb >= 0.5).astype(int)
accuracy_mb = accuracy_score(y_new, y_pred_mb_class)
print(f"Precision (Accuracy) pour la descente mini-batch: {accuracy_mb:.3f}")

# Fonction Matrice de Confusion
def matrice_de_confusion(y_true, y_pred):
    # Classes sont 0 et 1
    TP = np.sum((y_true == 1) & (y_pred == 1))
    FN = np.sum((y_true == 1) & (y_pred == 0))
    FP = np.sum((y_true == 0) & (y_pred == 1))
    TN = np.sum((y_true == 0) & (y_pred == 0))
    return np.array([[TP, FN], [FP, TN]])

cm_gd = matrice_de_confusion(y_new, y_pred_gd_class)
print("\nMatrice de confusion (Descente Gradient):")
print(cm_gd)

TP_gd = cm_gd[0, 0]
FN_gd = cm_gd[0, 1]
```

```python
FP_gd = cm_gd[1, 0]
TN_gd = cm_gd[1, 1]

precision_gd = TP_gd / (TP_gd + FP_gd) if (TP_gd + FP_gd) != 0 else 0
recall_gd = TP_gd / (TP_gd + FN_gd) if (TP_gd + FN_gd) != 0 else 0
f1_gd = 2 * (precision_gd * recall_gd) / (precision_gd + recall_gd) if (
    precision_gd + recall_gd) != 0 else 0

print(f"Precision (Precision GD): {precision_gd:.3f}")
print(f"Rappel (Recall GD): {recall_gd:.3f}")
print(f"Score F1 (GD): {f1_gd:.3f}")

cm_mb = matrice_de_confusion(y_new, y_pred_mb_class)
print("\nMatrice de confusion (Mini-Batch):")
print(cm_mb)
# Calculer Precision, Recall, F1 pour Mini-Batch de maniere similaire si besoin
```

Listing 7 – Calcul de la précision et de la matrice de confusion

11 Approche d'Optimisation par Moyennage

Cette section détaille une approche d'optimisation alternative explorée dans ce projet, issue d'une réflexion initiée lors d'un travail antérieur sur la "Régression Linéaire et Descente de Gradient from scratch". L'idée est d'exploiter la convexité du problème et les propriétés statistiques pour approcher l'optimum global.

11.1 Principe de la Méthode

L'approche consiste à :

1. Diviser le jeu de données d'entraînement $D = \{(X_i, y_i)\}_{i=1}^{N}$ en M sous-ensembles (splits) $D_1, \ldots, D_M$. Idéalement, ces splits sont disjoints ou tirés aléatoirement.
2. Pour chaque sous-ensemble D_j, effectuer une optimisation complète (par exemple, par descente de gradient) pour minimiser l'entropie croisée $\mathcal{L}_j(B, b)$ calculée sur D_j. Cela donne un jeu de paramètres optimaux $(\hat{B}_j, \hat{b}_j)$ pour ce split.

$$(\hat{B}_j, \hat{b}_j) = \arg\min_{B,b} \mathcal{L}_j(B, b)$$

3. Calculer les paramètres finaux $(\bar{B}, \bar{b})$ en moyennant les paramètres obtenus sur tous les splits :

$$\bar{B} = \frac{1}{M} \sum_{j=1}^{M} \hat{B}_j \quad \text{et} \quad \bar{b} = \frac{1}{M} \sum_{j=1}^{M} \hat{b}_j$$

11.2 Justification Théorique et Convergence vers l'Argmin

Pourquoi cette méthode est-elle susceptible de converger vers l'argmin global (B^*, b^*) ?

1. **Convexité :** Comme mentionné, l'entropie croisée $\mathcal{L}(B, b)$ est convexe. Son minimum global (B^*, b^*) est unique.

2. **Estimateurs Locaux :** Chaque $(\hat{B}_j, \hat{b}_j)$ est le minimum de $\mathcal{L}_j(B, b)$. Si le sous-ensemble D_j est suffisamment grand et représentatif de la distribution globale des données, alors $\mathcal{L}_j$ est une bonne approximation de $\mathcal{L}$, et son minimum $(\hat{B}_j, \hat{b}_j)$ sera un estimateur (bruité) du minimum global (B^*, b^*).

3. **Loi des Grands Nombres :** Considérons $(\hat{B}_j, \hat{b}_j)$ comme une variable aléatoire dont l'espérance est (B^*, b^*) (ou proche de celle-ci si les splits sont représentatifs). En moyennant un grand nombre M de ces estimateurs (idéalement issus de splits indépendants ou faiblement

corrélés), la **Loi des Grands Nombres** stipule que la moyenne $(\bar{B}, \bar{b})$ converge en probabilité vers l'espérance de ces estimateurs :

$$\bar{B} \xrightarrow[M\to\infty]{P} \mathbb{E}[\hat{B}_j] \approx B^*$$

$$\bar{b} \xrightarrow[M\to\infty]{P} \mathbb{E}[\hat{b}_j] \approx b^*$$

Cette convergence est d'autant plus vraie que le **jeu de données initial est grand**, permettant de créer de nombreux sous-ensembles D_j qui restent suffisamment grands pour que $(\hat{B}_j, \hat{b}_j)$ soit un estimateur raisonnable de (B^*, b^*).

4. **Résultats Empiriques :** Les expériences menées (voir code ci-dessous) montrent que les paramètres $(\bar{B}, \bar{b})$ obtenus par cette méthode donnent une bonne précision (*Accuracy* $\approx 85\%$ sur le jeu de données synthétique), ce qui conforte l'idée que $(\bar{B}, \bar{b})$ est une bonne approximation de l'argmin global (B^*, b^*).

11.3 Avantages et Inconvénients

Avantages :

— **Parallélisation :** Les M optimisations peuvent être effectuées en parallèle, réduisant potentiellement le temps total d'exécution sur des architectures multi-cœurs ou distribuées.

— **Robustesse :** Le moyennage réduit la variance des estimateurs individuels due à l'échantillonnage spécifique de chaque split, rendant le résultat final potentiellement plus stable et robuste aux découpages aléatoires.

Inconvénients :

— **Dépendance à la taille des données :** La méthode est plus pertinente lorsque le dataset initial est **grand**, afin que chaque sous-ensemble soit suffisamment représentatif. Sur de petits datasets, les estimateurs locaux peuvent être très bruités et leur moyenne moins fiable.

— **Efficacité en calcul séquentiel :** Sans parallélisation, effectuer M optimisations complètes peut être moins efficace que des méthodes itératives globales comme ADAM ou SGD.

— **Pas un algorithme standard :** Cette approche spécifique n'est pas un algorithme d'optimisation standard nommé dans la littérature, bien qu'elle s'inspire de concepts connus.

```python
# Assurez-vous que X et y sont definis (grand dataset initial)
n_splits = 500
if X.shape[0] < n_splits:
    print("Attention: Moins d'echantillons que de splits demandes.")
    n_splits = max(1, int(X.shape[0]/10)) # S'assurer d'avoir au moins 10 points par split si possible

X_list = np.array_split(X, n_splits)
y_list = np.array_split(y, n_splits)

# Fonction pour estimer sur un seul split
def estimateur_split(X_sub, y_sub):
  # Utiliser la fonction descente_gradient definie precedemment
  # Ajuster les hyperparametres si necessaire pour une convergence raisonnable sur les petits sets
  B_est, b_est, _, cvg, _, _, _ = descente_gradient(X_data=X_sub, y_data=y_sub, NitMax=2000, rho=5e-3, eps=1e-3, B=np.zeros(X_sub.shape[1]), b=0.0) # Params ajustes pour splits plus petits
  if not cvg:
      print("Attention: l'optimisation sur un split n'a pas atteint la tolerance.")
  return B_est, b_est
```

```

Est = []
print(f"Estimation sur {n_splits} sous-echantillons...")
successful_estimations = 0
for i, (X_split, y_split) in enumerate(zip(X_list, y_list)): # Renommer pour clarte
  if X_split.shape[0] > 1: # Verifier qu'il y a assez de donnees
      # print(f"Processing split {i+1}/{n_splits} (taille {X_split.shape[0]}) ...") # Decommenter pour verbose
      try:
          B, b = estimateur_split(X_split, y_split) # Appeler la fonction d'estimation
          if np.isfinite(B).all() and np.isfinite(b): # Verifier si resultats valides
              Est.append((B, b))
              successful_estimations += 1
          else:
               print(f"Resultats non finis sur split {i+1}, ignore.")
      except Exception as e:
          print(f"Erreur sur le split {i+1}: {e}")
  else:
       print(f"Split {i+1} trop petit ({X_split.shape[0]} echantillons), ignore.")
print(f"Nombre d'estimations reussies: {successful_estimations}/{n_splits}")

# Calculer la moyenne des parametres estimes
if Est: # S'assurer qu'il y a des resultats valides
    B_final_mean = np.mean([B for B, b in Est], axis=0)
    b_final_mean = np.mean([b for B, b in Est], axis=0)

    print("\nMoyenne des B estimes sur les splits :")
    print(B_final_mean)
    print("Moyenne des b estimes sur les splits :")
    print(b_final_mean)

    # Evaluer la performance du modele moyen sur de nouvelles donnees
    X_new_eval, y_new_eval = make_classification(n_samples=1000, n_features=5, n_informative=5, n_redundant=0, n_classes=2, random_state=42) # Utiliser un nom different

    y_pred_mean = f(B_final_mean, b_final_mean, X_new_eval) # f doit etre accessible globalement ou redefinie
    y_pred_mean_class = (y_pred_mean >= 0.5).astype(int)
    accuracy_mean = accuracy_score(y_new_eval, y_pred_mean_class)
    print(f"\nPrecision (Accuracy) du modele moyen sur nouveau test set: {accuracy_mean:.3f}")

    # Comparer avec la descente de gradient sur tout le dataset
    print("\nComparaison avec Descente de Gradient classique (execution rapide)...")
    B_full_comp, b_full_comp, _, _, _, _, _ = descente_gradient(X_data=X, y_data=y, NitMax=5000, rho=1e-3, eps=1e-4) # Execution plus courte pour comparaison
    y_pred_full_comp = f(B_full_comp, b_full_comp, X_new_eval)
    y_pred_full_comp_class = (y_pred_full_comp >= 0.5).astype(int)
    accuracy_full_comp = accuracy_score(y_new_eval, y_pred_full_comp_class)
    print(f"Precision (Accuracy) Descente Gradient classique sur nouveau test set: {accuracy_full_comp:.3f}")


else:
    print("Aucune estimation reussie sur les splits.")
```

Listing 8 – Estimation par moyenne sur sous-échantillons

Remarque : L'efficacité et la précision de cette méthode par moyennage dépendent fortement de la taille globale des données, du nombre de splits, et de la bonne convergence de l'optimisation sur chaque split.

12 Application à des datasets "réels"

12.1 Dataset Titanic

On teste l'algorithme sur un dataset concernant le Titanic où la valeur à prédire est 1 (survécu) ou 0 (décédé), en utilisant les features `pclass`, `sex`, `age`.

```
# Import necessaire si non fait avant
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

titanic = fetch_openml(name='titanic', version=1, as_frame=True, parser='auto')
    # Specifier parser
df_titanic = titanic.frame

# Garder un sous-ensemble simple et propre, enlever les lignes avec NaN
df_titanic = df_titanic[['pclass', 'sex', 'age', 'survived']].dropna()

# Encoder la variable 'sex' en 0/1
df_titanic['sex'] = df_titanic['sex'].map({'male': 0, 'female': 1})
# Convertir 'survived' en entier 0/1
df_titanic['survived'] = df_titanic['survived'].astype(int)


# 80% train, 20% test
df_train, df_test = train_test_split(df_titanic, test_size=0.2, random_state=42,
    stratify=df_titanic['survived'])


# Separer features et labels
X_train_titanic = df_train[['pclass', 'sex', 'age']].values
y_train_titanic = df_train['survived'].values
X_test_titanic = df_test[['pclass', 'sex', 'age']].values
y_test_titanic = df_test['survived'].values

# Standardiser les donnees (important pour la descente de gradient)
scaler_titanic = StandardScaler()
X_train_titanic_scaled = scaler_titanic.fit_transform(X_train_titanic)
X_test_titanic_scaled = scaler_titanic.transform(X_test_titanic)
```

Listing 9 – Chargement et préparation du dataset Titanic

12.1.1 Test de la méthode par moyenne (sur Titanic)

```
n_splits_titanic = 50 # Moins de splits car dataset plus petit
X_list_titanic = np.array_split(X_train_titanic_scaled, n_splits_titanic)
y_list_titanic = np.array_split(y_train_titanic, n_splits_titanic)

Est_titanic = []
print(f"Estimation sur {n_splits_titanic} sous-echantillons (Titanic)...")
successful_estimations_titanic = 0
for i, (X_sub, y_sub) in enumerate(zip(X_list_titanic, y_list_titanic)): #
    Renommer pour clarte
  if X_sub.shape[0] > 1:
    # print(f"Processing split {i+1}/{n_splits_titanic} (Titanic)...") #
    Decommenter pour verbose
```

```
    try:
        # Utiliser directement descente_gradient en passant les donnees du split
        B, b, _, cvg, _, _, _ = descente_gradient(X_data=X_sub, y_data=y_sub,
    NitMax=5000, rho=1e-2, eps=1e-3) # Ajuster hyperparams
        if np.isfinite(B).all() and np.isfinite(b):
            Est_titanic.append((B, b))
            successful_estimations_titanic += 1
            # if not cvg:
            #     print(f"Warning: Split {i+1} (Titanic) n'a pas atteint la
    tolerance.")
        else:
            print(f"Resultats non finis sur split {i+1} (Titanic), ignore.")
    except Exception as e:
        print(f"Erreur sur le split {i+1} (Titanic): {e}")
  else:
    print(f"Split {i+1} (Titanic) trop petit, ignore.")
print(f"Nombre d'estimations reussies (Titanic): {successful_estimations_titanic
    }/{n_splits_titanic}")


if Est_titanic:
    B_titanic_mean = np.mean([B for B, b in Est_titanic], axis=0)
    b_titanic_mean = np.mean([b for B, b in Est_titanic], axis=0)

    print("\nMoyenne des B (Titanic) :")
    print(B_titanic_mean)
    print("Moyenne des b (Titanic) :")
    print(b_titanic_mean)

    # Evaluer sur le test set Titanic
    y_pred_titanic_mean = f(B_titanic_mean, b_titanic_mean,
    X_test_titanic_scaled)
    y_pred_titanic_mean_class = (y_pred_titanic_mean >= 0.5).astype(int)
    accuracy_titanic_mean = accuracy_score(y_test_titanic,
    y_pred_titanic_mean_class)
    print(f"\nPrecision (Accuracy) du modele moyen sur test set Titanic: {
    accuracy_titanic_mean:.3f}")

    cm_titanic_mean = matrice_de_confusion(y_test_titanic,
    y_pred_titanic_mean_class)
    print("\nMatrice de confusion (Moyenne, Titanic):")
    print(cm_titanic_mean)
else:
    print("Aucune estimation reussie sur les splits (Titanic).")
```

Listing 10 – Estimation par moyenne sur Titanic

La méthode des moyennes peut moins bien fonctionner sur des datasets plus petits.

12.1.2 Test de la descente de gradient simple (sur Titanic)

```
print("\nDescente de gradient simple sur tout le dataset d'entrainement Titanic
    ...")
# Appeler descente_gradient directement avec les donnees Titanic
B_titanic_full, b_titanic_full, _, cvg_full_titanic, _, _, _ = descente_gradient
    (
    X_data=X_train_titanic_scaled,
    y_data=y_train_titanic,
    NitMax=100000, rho=1e-3, eps=1e-6 # Tenter une tolerance plus faible
)

print("\nB (Titanic Full):")
print(B_titanic_full)
```

```
print("b (Titanic Full):")
print(b_titanic_full)
print(f"Convergence atteinte (Full Titanic): {cvg_full_titanic}")

# Evaluer sur le test set Titanic
y_pred_titanic_full = f(B_titanic_full, b_titanic_full, X_test_titanic_scaled)
y_pred_titanic_full_class = (y_pred_titanic_full >= 0.5).astype(int)
accuracy_titanic_full = accuracy_score(y_test_titanic, y_pred_titanic_full_class)
print(f"\nPrecision (Accuracy) du modele full sur test set Titanic: {accuracy_titanic_full:.3f}")

cm_titanic_full = matrice_de_confusion(y_test_titanic, y_pred_titanic_full_class)
print("\nMatrice de confusion (Full, Titanic):")
print(cm_titanic_full)

TP_full = cm_titanic_full[0, 0]
FN_full = cm_titanic_full[0, 1]
FP_full = cm_titanic_full[1, 0]
TN_full = cm_titanic_full[1, 1]

precision_full = TP_full / (TP_full + FP_full) if (TP_full + FP_full) != 0 else 0
recall_full = TP_full / (TP_full + FN_full) if (TP_full + FN_full) != 0 else 0
f1_full = 2 * (precision_full * recall_full) / (precision_full + recall_full) if (precision_full + recall_full) != 0 else 0

print(f"Precision (Precision Full): {precision_full:.3f}")
print(f"Rappel (Recall Full): {recall_full:.3f}")
print(f"Score F1 (Full): {f1_full:.3f}")
```

Listing 11 – Descente de gradient simple sur Titanic

Les résultats montrent une précision raisonnable, qui pourrait être améliorée (par exemple avec ADAM, régularisation, etc.).

12.2 Dataset Diabète

On applique la même démarche sur un dataset de prédiction du diabète.

```
# Charger le dataset (Assurez-vous que le fichier diabetes2.csv est accessible)
try:
    df_diabetes = pd.read_csv("diabetes2.csv") # Mettre le bon chemin si necessaire
except FileNotFoundError:
    print("Fichier diabetes2.csv non trouve. Assurez-vous qu'il est dans le repertoire.")
    # Creer un dataframe vide pour eviter les erreurs suivantes, ou arreter.
    df_diabetes = pd.DataFrame() # Initialiser comme vide

if not df_diabetes.empty:
    # Separer features (X) et target (y - 'Outcome')
    X_diabetes = df_diabetes.drop('Outcome', axis=1).values
    y_diabetes = df_diabetes['Outcome'].values

    # Split train/test
    X_train_db, X_test_db, y_train_db, y_test_db = train_test_split(
        X_diabetes, y_diabetes, test_size=0.2, random_state=42, stratify=y_diabetes
    )

    # Standardiser
    scaler_db = StandardScaler()
```

```
    X_train_db_scaled = scaler_db.fit_transform(X_train_db)
    X_test_db_scaled = scaler_db.transform(X_test_db)

    # Entrainement avec la descente de gradient
    print("\nDescente de gradient simple sur dataset Diabete...")
    # Appeler descente_gradient avec les donnees Diabete
    B_db_full, b_db_full, _, cvg_db_full, _, _, _ = descente_gradient(
        X_data=X_train_db_scaled,
        y_data=y_train_db,
        NitMax=100000, rho=1e-3, eps=1e-6 # Tenter tolerance plus faible
    )

    print("\nB (Diabetes Full):")
    print(B_db_full)
    print("b (Diabetes Full):")
    print(b_db_full)
    print(f"Convergence atteinte (Full Diabetes): {cvg_db_full}")


    # Evaluation
    y_pred_db_full = f(B_db_full, b_db_full, X_test_db_scaled)
    y_pred_db_full_class = (y_pred_db_full >= 0.5).astype(int)
    accuracy_db_full = accuracy_score(y_test_db, y_pred_db_full_class)
    print(f"\nPrecision (Accuracy) du modele full sur test set Diabete: {
   accuracy_db_full:.3f}")

    cm_db_full = matrice_de_confusion(y_test_db, y_pred_db_full_class)
    print("\nMatrice de confusion (Full, Diabete):")
    print(cm_db_full)
else:
    print("\nDataset Diabete non charge, section ignoree.")
```

Listing 12 – Chargement et préparation du dataset Diabète

Les hyperparamètres (pas, tolérance) ont été ajustés pour tenter d'améliorer la convergence.

13 Prolongement : Régression Logistique Multiclasse avec OvR (One vs Rest)

13.1 Principe

Pour étendre la régression logistique binaire à un problème de classification avec $K > 2$ classes, la méthode OvR (ou OvA - One vs All) consiste à entraîner K classifieurs binaires indépendants. Pour chaque classe $k \in \{0, \ldots, K-1\}$:

- On considère les exemples de la classe k comme positifs (étiquette 1).
- On considère tous les autres exemples (des classes $j \neq k$) comme négatifs (étiquette 0).
- On entraîne un modèle de régression logistique binaire $(\hat{B}_k, \hat{b}_k)$ sur ces nouvelles étiquettes.

Lors de la prédiction pour une nouvelle instance X_{new} :

- On calcule la probabilité de sortie de chaque classifieur k : $p_k = \sigma(X_{\text{new}} \cdot \hat{B}_k + \hat{b}_k)$.
- La classe prédite est celle qui correspond au classifieur ayant donné la plus haute probabilité :

$$\text{Prediction} = \arg\max_k \, p_k$$

13.2 Implémentation et Test sur Iris

- Simple à mettre en œuvre en réutilisant le modèle binaire.
- On teste sur le dataset Iris ($K = 3$ classes).

```python
from sklearn.datasets import load_iris

iris = load_iris()
X_iris = iris.data
y_iris = iris.target

# Standardiser les features
scaler_iris = StandardScaler()
X_iris_scaled = scaler_iris.fit_transform(X_iris)

# Split train/test
X_train_iris, X_test_iris, y_train_iris, y_test_iris = train_test_split(
    X_iris_scaled, y_iris, test_size=0.2, random_state=42, stratify=y_iris
)
```

Listing 13 – Chargement et préparation du dataset Iris

```python
def OVA_train(X_train, y_train, num_classes):
    B_liste = []
    b_liste = []
    for k in range(num_classes):
        # Creer les etiquettes binaires pour la classe k
        y_binaire_k = (y_train == k).astype(int)

        print(f"\nEntrainement du classifieur pour la classe {k} vs Rest...")
        # Entrainer le classifieur binaire k
        # Appeler descente_gradient directement
        # Utiliser une initialisation a zero pour B, b peut aider
        B_k, b_k, _, cvg_k, _, _, _ = descente_gradient(
            X_data=X_train,
            y_data=y_binaire_k,
            NitMax=50000, rho=1e-2, eps=1e-4, # Augmenter rho, reduire eps?
            B=np.zeros(X_train.shape[1]), b=0.0
        )

        B_liste.append(B_k)
        b_liste.append(b_k)
        print(f"Parametres trouves pour classe {k}: Converged={cvg_k}")
        # print(f"B={B_k}, b={b_k}") # Decommenter pour voir les params

    # Retourner des arrays numpy standards
    return np.array(B_liste), np.array(b_liste)

def OVA_predict(X_test, B_liste, b_liste):
    num_samples = X_test.shape[0]
    num_classes = len(B_liste)

    probabilities = np.zeros((num_samples, num_classes))

    for k in range(num_classes):
        B_k = B_liste[k]
        b_k = b_liste[k]
        # Calculer les scores/probabilites pour le classifieur k
        # Utiliser la fonction f globale (ou la rendre accessible)
        # S'assurer que f utilise X_test comme data
        probabilities[:, k] = f(B_k, b_k, X_data=X_test) # Specifier X_data

    # Trouver la classe avec la probabilite maximale pour chaque echantillon
    predictions = np.argmax(probabilities, axis=1)

    return predictions

# Entrainement OvR sur Iris
num_classes_iris = len(np.unique(y_iris))
```

```python
B_liste_iris, b_liste_iris = OVA_train(X_train_iris, y_train_iris,
    num_classes_iris)

# Predictions sur le test set Iris
y_pred_iris = OVA_predict(X_test_iris, B_liste_iris, b_liste_iris)

# Evaluation
accuracy_iris = accuracy_score(y_test_iris, y_pred_iris)
print(f"\nPrecision (Accuracy) OvR sur test set Iris: {accuracy_iris:.3f}")

# Affichage de la matrice de confusion pour multiclasse
from sklearn.metrics import confusion_matrix
cm_iris = confusion_matrix(y_test_iris, y_pred_iris)
print("\nMatrice de confusion OvR (Iris):")
print(cm_iris)

# Visualisation (exemple pour les 2 premieres features)
# plt.figure(figsize=(8, 6))
# plt.scatter(X_test_iris[:, 0], X_test_iris[:, 1], c=y_pred_iris, cmap='viridis
    ', marker='x', label='Predictions OvR')
# plt.scatter(X_train_iris[:, 0], X_train_iris[:, 1], c=y_train_iris, cmap='
    viridis', marker='o', alpha=0.3, label='Entrainement')
# plt.xlabel(iris.feature_names[0])
# plt.ylabel(iris.feature_names[1])
# plt.title('Predictions OvR vs Entrainement (Iris - 2 premieres features)')
# plt.legend()
# plt.show()
```

Listing 14 – Entraînement et prédiction OvR

13.3 Avis sur OvR

Avantages :

— Conceptuellement simple et facile à implémenter si un classifieur binaire existe.

— Modulaire : chaque classifieur est indépendant.

Inconvénients :

— **Lent :** Nécessite l'entraînement de K modèles, ce qui peut être long si K est grand ou si l'entraînement d'un modèle est coûteux.

— Peut souffrir de déséquilibre de classes lors de l'entraînement de chaque classifieur binaire (une classe contre toutes les autres).

— Les probabilités obtenues ne sont pas calibrées (leur somme pour une instance n'est pas nécessairement 1).

Conclusion : La méthode OvR est une approche simple pour la classification multiclasse, mais peut être coûteuse en calcul. D'autres méthodes (comme Softmax ou OvO) existent.

A Sources utilisées

1. Google ML Crash Course - Logistic Regression (FR)
2. DATAtab - Régression logistique
3. SciML Lectures - INRIA (chapitre sur la régression logistique) (Source la plus importante)
4. GRETIS 2019 - Article PDF par Anas Barakat
5. Dremio - One-vs-all Classification