



ÉCOLE CENTRALE LYON

UE INF

ANALYSE DE DONNÉES ET RECONNAISSANCES DE FORMES
RAPPORT

TD5 - Régression logistique pour la classification

Élèves :

Hugo PUYBAREAU

Enseignant :

Emmanuel DELLANDREA

16 octobre 2023

Table des matières

1	Introduction	2
2	Présentation du jeu de données	2
3	Données d'entraînement et données de test	2
4	Programmation de la régression logistique	3
5	Analyse et conclusion	6

1 Introduction

Le travail consiste à réaliser une régression logistique multi-classes sur un jeu de données à l'aide de la méthode un contre tous. Pour cela, on s'occupera également de séparer le jeu de données en deux sous-ensembles ; un pour les tests et un pour l'apprentissage.

2 Présentation du jeu de données

J'utilise pour ce devoir un jeu de données nommé "Iris". Ce tableau donne les dimensions d'une fleur d'iris pour déterminer quel type d'iris est observé. Les données sont donc regroupées sous 3 classes : Iris Setosa, Iris Versicolore et Iris Virginica. Nous travaillons donc sur un régression logistique à 3 classes. Je modifie le tableau pour que les classes soient nommées 0, 1 et 2 par la suite.

Il existe 4 variables, qui sont toutes continues et qui sont :

- **sepal lenght**, la longueur du sépale
- **sepal width**, la largeur du sépale
- **petal lenght**, la longueur du pétale
- **petal width**, la largeur du pétale

3 Données d'entraînement et données de test

Comme expliqué dans l'énoncé, il faut séparer les données en deux groupes en fonction d'un ratio compris entre 0 et 1. L'implémentation dans mon code est faite à partir de la fonction suivante :

```
def train_test(X, Y, ratio):
    """
    Cette fonction sépare le jeu de données principale en 2 jeu de données,
    un pour l'entraînement et un pour le test du modèle
    """
    indice_train = int(X.shape[0]*ratio) #Je crée l'indice en fonction du ratio
    X_train, X_test, Y_train, Y_test = X[0:indice_train,:], X[indice_train:,:], Y[0:indice_train, :], Y[indice_train:,:]
    return X_train, X_test, Y_train, Y_test
```

FIGURE 1 – Fonction de séparation des données

J'avoue avoir du modifié un peu l'ordre des lignes dans les colonnes parce que le tableau était rangé par ordre de classe et donc, pour le ratio de 0.2 que j'utilise dans mon programme, toutes les fleurs de données d'entraînement appartenaient à la première classe. C'est ensuite utilisé de cette manière :

```
print("Séparation des données pour test et entraînement ...")
ratio = 0.2
X_train, X_test, Y_train, Y_test = train_test(X,Y, ratio)
print("On réalise l'entraînement sur", int(X.shape[0]*ratio), "valeurs")
```

FIGURE 2 – Utilisation de la fonction décrite

4 Programmation de la régression logistique

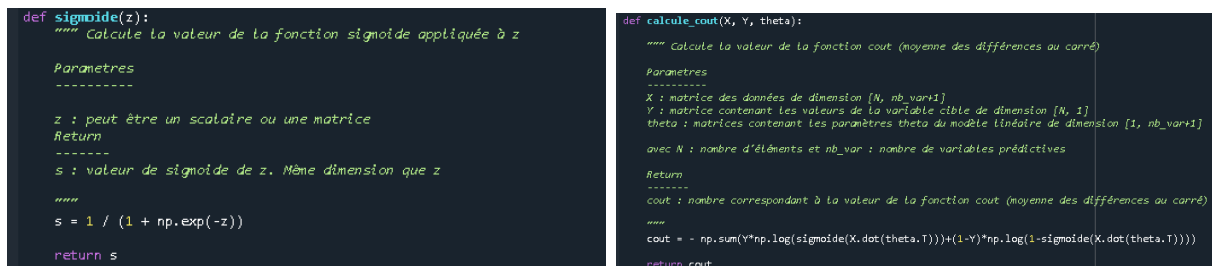
Contrairement à la régression linéaire, la fonction de prédiction utilisée et une fonction sigmoïde. On a :

$$f_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}} \quad (1)$$

On calcule alors le coût à partir de l'expression de la fonction de prédiction selon la formule suivante :

$$J(\theta) = - \sum_{i=1}^N y^{(i)} \log(f_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\theta}(x^{(i)})) \quad (2)$$

Ceci est ajouté dans le code de la manière suivante.



```
def sigmoide(z):
    """ Calcule la valeur de la fonction sigmoide appliquée à z

    Parametres
    -----
    z : peut être un scalaire ou une matrice
    Return
    -----
    s : valeur de sigmoide de z. Même dimension que z

    """
    s = 1 / (1 + np.exp(-z))
    return s
```

```
def calcule_cout(X, Y, theta):
    """ Calcule la valeur de la fonction cout (moyenne des différences au carré)

    Parametres
    -----
    X : matrice des données de dimension [N, nb_var+1]
    Y : matrice contenant les valeurs de la variable cible de dimension [N, 1]
    theta : matrices contenant les paramètres theta du modèle linéaire de dimension [1, nb_var+1]
    avec N : nombre d'éléments et nb_var : nombre de variables prédictives

    Return
    -----
    cout : nombre correspondant à la valeur de la fonction cout (moyenne des différences au carré)

    """
    cout = - np.sum(Y*np.log(sigmoide(X.dot(theta.T)))+(1-Y)*np.log(1-sigmoide(X.dot(theta.T))))
    return cout
```

FIGURE 3 – Implantation dans le code de la fonction de prédiction et de la fonction de calcul du coût

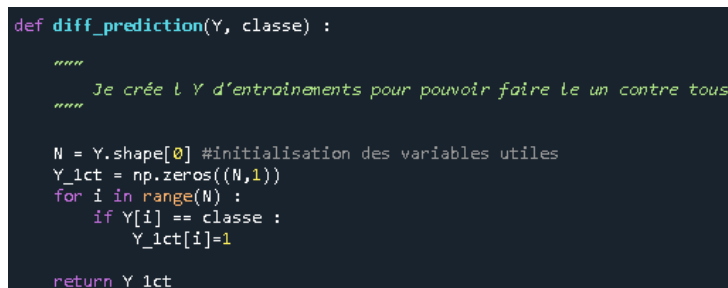
Le code pour la descente du gradient est alors le même que pour la régression linéaire car les changements sont contenus dans la fonction de calcul du coût.

Comme la régression n'est pas binaire, nous utilisons la méthode *"Un contre tous"*. Cette méthode consiste à réaliser 3 régressions logistiques binaires (3 car ici il y a 3 classes). En effet, on réalise les descentes du gradient pour les cas :

- $y = 1$: C'est l'iris 0 et $y = 0$: Ce n'est pas l'iris 0
- $y = 1$: C'est l'iris 1 et $y = 0$: Ce n'est pas l'iris 1
- $y = 1$: C'est l'iris 2 et $y = 0$: Ce n'est pas l'iris 2

Vu comme cela, on comprend pourquoi la méthode est appelée *"Un contre tous"*

Je fais donc une fonction *diff_prediction* qui va me permettre de reconstruire trois matrices Y pour faire les trois régressions logistiques décrites ci-dessus. La fonction est la suivante :



```
def diff_prediction(Y, classe):
    """
    Je crée 3 Y d'entrainements pour pouvoir faire le un contre tous
    """
    N = Y.shape[0] #initialisation des variables utiles
    Y_1ct = np.zeros((N,1))
    for i in range(N):
        if Y[i] == classe:
            Y_1ct[i]=1
    return Y_1ct
```

FIGURE 4 – Fonction *diff_prediction*

On crée alors trois matrices pour les paramètres θ 's. On peut alors réaliser les trois descentes du gradient avec les matrices obtenues :

```
# Initialisation de theta et réalisation de la descente du gradient
# Il y a 3 classes donc on fait 3 matrices de theta
theta0, theta1, theta2 = np.zeros((1,nb_var+1)), np.zeros((1,nb_var+1)), np.zeros((1,nb_var+1))
Y_t0, Y_t1, Y_t2 = diff_prediction(Y_train, 0), diff_prediction(Y_train, 1), diff_prediction(Y_train, 2)
theta0, J_history0 = descente_gradient(X_train, Y_t0, theta0, alpha, nb_iters)
theta1, J_history1 = descente_gradient(X_train, Y_t1, theta1, alpha, nb_iters)
theta2, J_history2 = descente_gradient(X_train, Y_t2, theta2, alpha, nb_iters)
```

FIGURE 5 – Réalisation des descentes du gradient

J'obtiens alors 3 coûts qui convergent de manière satisfaisante pour 10000 itérations. On peut mettre moins mais la bleue ne converge déjà pas très bien à 10 000 :

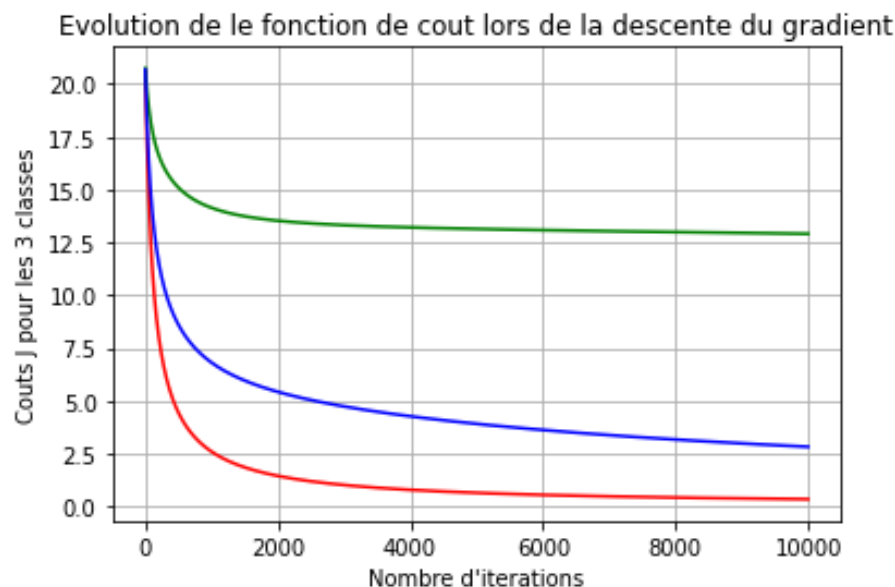


FIGURE 6 – Convergence des coûts

On nous explique dans le cours qu'il faut choisir la classe "k" qui maximise la fonction de prédiction. Je crée donc une matrice de comparaison et modifie la fonction prédiction pour qu'elle puisse fonctionner selon ces indications.

C'est codée comme suit :

```
# Evaluation du modèle

mat_comp_train = np.concatenate((X_train.dot(theta0.T), X_train.dot(theta1.T), X_train.dot(theta2.T)), axis = 1)
print("La matrice de comparaison utilisée pour la prédiction est la suivante : ", mat_comp_train)

Ypred_train = prediction(X_train, mat_comp_train)

print("On obtient alors la matrice de prédiction suivante : ", Ypred_train)
```

FIGURE 7 – Création de la matrice de comparaison

```
def prediction(X,mat_comp):
    """ Predit la classe de chaque élément de X

    Parametres
    -----
    X : matrice des données de dimension [N, nb_var+1]
    theta : matrices contenant les paramètres theta du modèle linéaire de dimension [1, nb_var+1]

    avec N : nombre d'éléments et nb_var : nombre de variables prédictives

    Retour
    -----
    p : matrices de dimension [N,1] indiquant la classe de chaque élément de X (soit 0, soit 1, soit 2)

    """
    N = X.shape[0] #initialisation des variables utiles
    p = np.zeros((N,1)) #initialisation de p

    for i in range(N) :
        for k in range(3) :
            if max(mat_comp[i]) == mat_comp[i][k] :
                p[i] = k
    return p
```

FIGURE 8 – Code de la fonction *prediction*

J'obtiens alors un taux de classification de 93%, c'est satisfaisant pour pouvoir passer au test. Par ailleurs, le code de la fonction *taux_classification* ne change pas :

```
def taux_classification(Ypred,Y):
    """ Calcule le taux de classification (proportion d'éléments bien classés)

    Paramètres
    -----
    Ypred : matrice contenant les valeurs de classe prédites de dimension [N, 1]
    Y : matrice contenant les valeurs de la variable cible de dimension [N, 1]

    avec N : nombre d'éléments

    Retour
    -----
    t : taux de classification (valeur scalaire)

    """
    t = 0 #initialisation de t
    N=Y.shape[0]

    for i in range(N):
        if Ypred[i] == Y[i] :
            t+=1

    t=(t/N)*100 #on ramène t à un pourcentage
    return t
```

FIGURE 9 – Code de la fonction *taux_classification*

Pour les tests, on fait la même chose et on obtient un taux de classification de 85% :

```
print("Entraînement du modèle de regression logistique terminé.")

#Réalisation du test pour le reste des valeurs

mat_comp_test = np.concatenate((X_test.dot(theta0.T), X_test.dot(theta1.T), X_test.dot(theta2.T)), axis = 1)
Ypred_test = prediction(X_test, mat_comp_test)
print("On obtient la matrice de prédiction suivante :", Ypred_test)
print("Taux de classification pour le test : ", taux_classification(Ypred_test, Y_test), "%")

print("Regression logistique terminée")
```

FIGURE 10 – Code pour le test du reste du jeu de données

5 Analyse et conclusion

Lorsque l'on représente la répartition des classes en fonctions des pétales et des sépales. On ne trouve pas de conclusions particulières à faire à mon sens. Les pétales et les sépales sont tout autant influents.

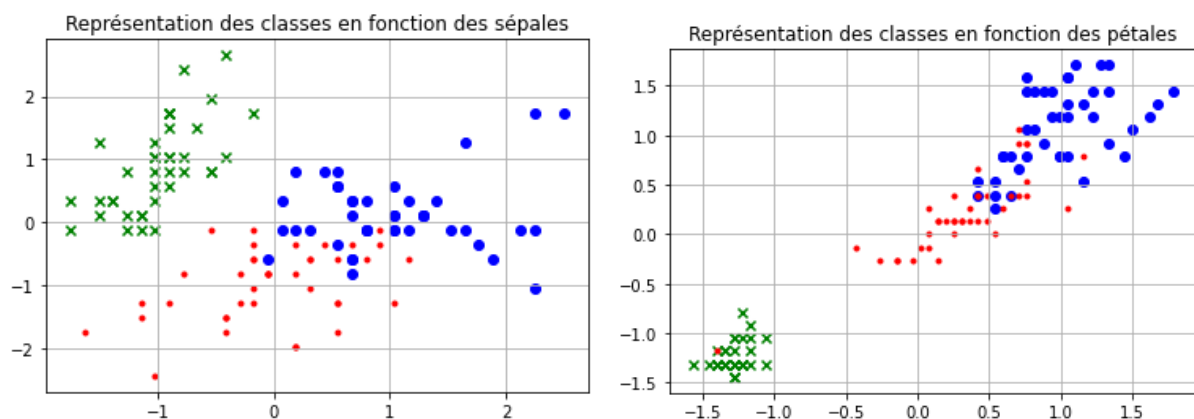


FIGURE 11 – Représentation des classes en fonction des sépales et des pétales

La seule chose à remarquer est qu'il y a une fleur qui a des longueur de pétales un peu aberrante dans la classe rouge. J'ai alors vérifié s'il n'y avait pas de choses similaires dans les valeurs d'entraînement. Si cela avait été le cas, il aurait été préférable d'entraîner à nouveau le modèle sans prendre en compte la valeur aberrante. De plus, la classe bleue et la classe rouge sont plus "proches" entre elles que la classe verte, qui est elle très isolée.

J'en déduis que ces deux types d'iris se ressemblent beaucoup alors que le type vert doit être assez facilement identifiable face aux deux autres. Les classes rouges et bleues correspondent aux iris Versicolores et Virginica. Tandis que la classe verte correspond à l'iris Setosa. Après avoir moi-même comparé quelques photos d'iris sur internet, (je ne suis pas un grand expert en fleur), je pense que cette analyse est plutôt vraie. Les images suivantes aident à convaincre.

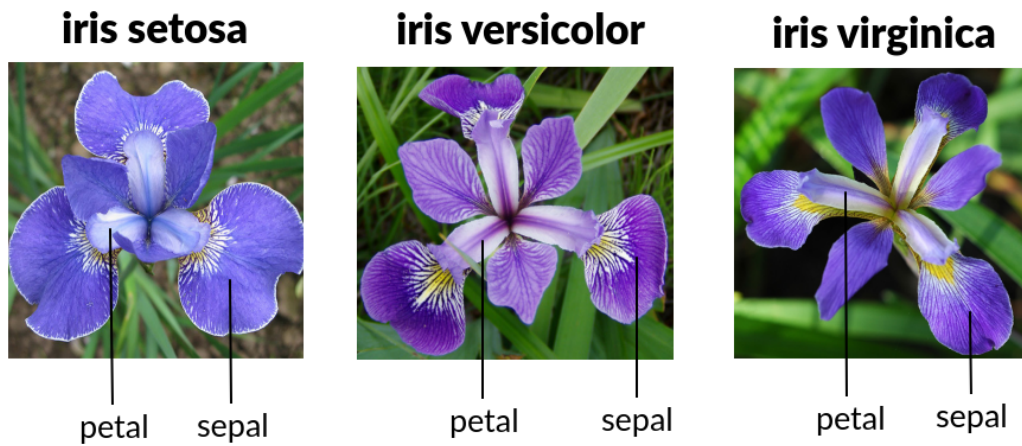


FIGURE 12 – Ressemblances entre les iris virginica et versicolores



FIGURE 13 – Ressemblances schématiques entres les différentes fleurs d'iris