Machine Learning TP2: Régression logistique

Melvin DUBEE - Tanguy ROUDAUT FIPASE 24

16 octobre 2023

Vous pouvez trouver nos codes dans les dossiers "code" et "code2" et les résultats dans le dossier "output" qui se trouve à la racine de l'archive.

1 Régression logistique

1.1 Affichage des données

Une première fonction *plotData()*, qui permet d'afficher un graphique 2D avec les axes représentant les deux notes des examens, et les exemples positifs et négatifs affichés avec différents marqueurs.
L'objectif de cette partie sera de construire un modèle de régression logistique pour prédire si un étudiant est admis dans une université en comprenant sa méthodologie.

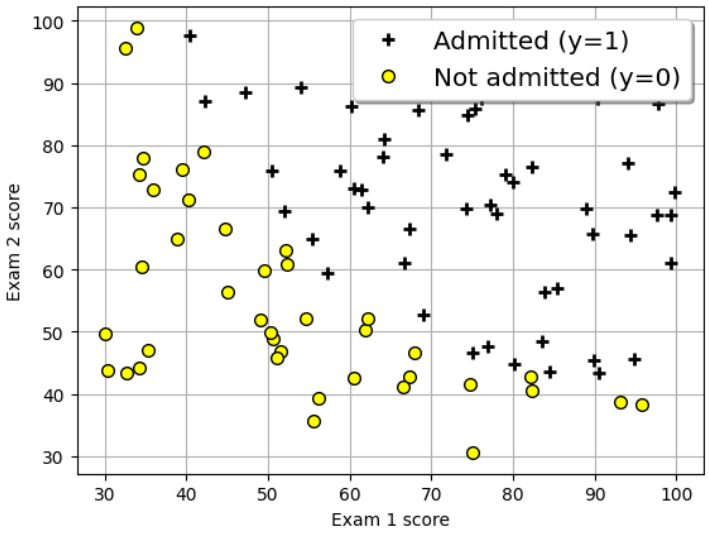

FIGURE 1 – Diagramme de dispersion des données d'entraînement

```
def plotData(X,y):
    pos = X[(y==1).flatten(),:]
    neg = X[(y==0).flatten(),:]
    plt.plot(pos[:,0], pos[:,1], '+', markersize=7, markeredgecolor='black', markeredgewidth=2)
    plt.plot(neg[:,0], neg[:,1], 'o', markersize=7, markeredgecolor='black', markerfacecolor='yellow')
    plt.legend(['Admitted (y=1)', 'Not admitted (y=0)'], loc='upper right', shadow=True,
    ↪  fontsize='x-large', numpoints=1)
    plt.grid()
    plt.xlabel('Exam 1 score')
    plt.ylabel('Exam 2 score')
```

FIGURE 2 – Fonction plotData

1.2 Implémentation

Le modèle de régression linéaire est représenté par l'équation 1. Cette équation nous permet d'obtenir une prédiction en fonction d'une entrée x et de θ.

```
def sigmoid(z):
        g = 1 / (1 + np.exp(-z))

    return  g
```

$$h_\theta(x) = g(x^T\theta) \tag{1}$$

$$g(z) = \frac{1}{1 - e^{-z}} \tag{2}$$

Pour que cette prédiction soit optimale, il est important de déterminer correctement les paramètres de notre modèle : θ. Pour cela, nous devons réaliser deux étapes : Le calcul du coût $J(\theta)$ et une descente de gradient.

1.2.1 Calcul du coût $J(\theta)$

De la même manière que dans le TP1, le calcul du coût $J(\theta)$ permet de mesurer la qualité de la prédiction, si le coût est faible alors notre prédiction est proche des valeurs réelles et inversement si le coût est important. La formule utilisée pour calculer ce coût n'est pas la même pour une régression linéaire que pour une régression logistique.

$$J(\theta) = \frac{1}{m}\sum_{i=0}^{m-1}[-y^{(i)}log(h_\theta(x^{(i)})) - (1 - y^{(i)})log(1 - h_\theta(x^{(i)}))] \tag{3}$$

On remarque ici avec l'équation 6 qui utilise l'équation 1, que la seule valeur qui puisse influencer notre coût est θ. Effectivement, les valeurs restantes : x, y et m ; sont les valeurs de notre problème qui sont déterminées et non modifiables.

Mise en application

```
def costFunction(theta, X, y):
    # Initialize some useful values
    m,n = X.shape
    theta = theta.reshape((n,1))

    predictions = sigmoid(X @ theta)
    J = (1/m) * np.sum(-y * np.log(predictions) - (1 - y) * np.log(1 - predictions))

    return  J

    """ return
    Cost at initial theta (zeros): 0.693147
    Expected cost (approx): 0.693
    """
```

Listing 1 – Fonction computeCost

1.2.2 Descente de gradient

La descente de gradient permet de minimiser le coût et donc d'obtenir les bonnes valeurs de theta pour réaliser une prédiction optimale.

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{4}$$

Cette formule pour calculer le gradient de la régression logistique comme pour le coût n'est pas identique à celle de la régression linéaire, en effet cela est dû aux définitions différentes de $h_\theta(x)$.

Mise en application

```python
def gradientFunction(theta, X, y):
    m = X.shape[0]
    n = X.shape[1]
    theta = theta.reshape((n,1))
    grad = 0
    for i in range(m):
        grad += (1/m) * (sigmoid(X[i] @ theta) - y[i]) * X[i]

    return grad

    """ return
    theta: ['-25.1613', '0.2062', '0.2015']
    Expected theta (approx): -25.161 0.206 0.201

    ---------------------------

    For a student with scores 45 and 85, we predict an admission probability of 0.776291
    Expected Proba (approx): 0.776

    ---------------------------

    Train Accuracy: 89.000000
    Expected accuracy (approx): 89.0%
        """
```

Listing 2 – Fonction gradientFunction

2 Classification multi-classes

Dans cette partie nous allons appliqué une regression logistique *One-Vs-All* dans l'objectif de prédire la valeur numérique d'un chiffre manuscrit.

Le procéder est assez simple, comme le montre la figure 3 :

Nous avons au total 3 classes, on prend une classe et on l'isole des deux autres. Ce qui nous en donne 2, avec lesquels nous allons réaliser une regression logistique. Si on réalise 3 fois cette étapes alors nous aurons 3 limites de décision.
Dans le cas d'une prédiction, nous allons pouvoir calculer les probabilités pour lesquelles notre échantillon se trouve dans une des limites de décision grâce à notre modèle. La probabilité la plus importante nous permettra de déterminer dans qu'elle classe ce trouve notre chiffre manuscrit.

$$h_\theta^{(i)} = P(y = i|x;\theta) \quad (i = 1, 2, 3) \qquad (5)$$

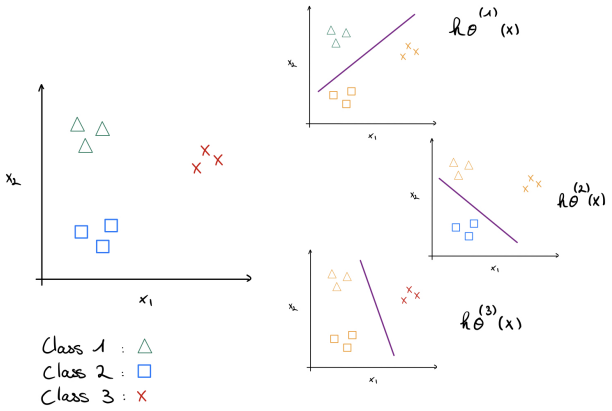

FIGURE 3 – Schéma explicatif de la classification multi-classes

Dans notre cas nous n'aurons pas 3 classes mais 10, il faudra donc réaliser 10 classifieur.

2.1 Visualisation des données

Dans notre matrice X nous avons 5000 échantillon de chiffre manuscrit, où chaque ligne représente une image d'un chiffre en niveau de gris de $20x20$px. Ce qui nous donne finalement une matrice de dimension 5000 sur 400.
Nous allons en afficher une centaine de manière aléatoire à l'aide du script données. Chaque visualisation est différente.

FIGURE 4 – Visualisation des données

2.2 Véctorisation de la régression logistique

Pour cette partie il est important que notre régression logistique soit vectorisé, notre échantilllon de données est important comparé aux exercices précédents.
La vectorisation a de nombreux avantages :

1. Apprentissage plus rapide
2. Code plus lisible
3. Moins d'erreur lié au boucle
4. ...

2.2.1 Vectorisation de la fonction de coût $J(\theta)$

Grâce à python et aux fonctions de numpy la vectorisation s'applique facilement. Il faut coder la fonction de coût comme elle est écrite en retirant les $^{(i)}$ lié à la somme et en la remplacant par *np.sum()*.
La fonction de coût est alors réduite à une ligne, grâce aux multiplications matricielles automatique de python et numpy.

$$J(\theta) = \frac{1}{m}\sum_{i=0}^{m-1}[-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))] \tag{6}$$

```python
def lrCostFunction(theta, X, y, Lambda):
    theta = theta.reshape((n,1)) # (4,1)

    h = sigmoid(X @ theta)
    J = (1/m) * np.sum(-y * np.log(h) - (1 - y) * np.log(1 - h))

    return J
```

FIGURE 5 – Fonction de coût vectorisé

2.2.2 Vectorisation de la descente de gradient

Le principe est le même que pour la fonction de coût $J(\theta)$:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{7}$$

```python
def lrCostGradient(theta, X, y, Lambda):
    theta = theta.reshape((n,1)) # (4,1)

    h = sigmoid(X @ theta)
    grad = (1/m) * (X.T @ (h - y))

    return grad.flatten()
```

FIGURE 6 – Descente de gradient vectorisé

2.2.3 Application d'une régulation

Comme expliqué plus tôt la régulation permet d'éviter le problème d'*overfit* si λ est correctement choisis. Dans ce problème nous avons un nombre important de caratéristiques, il est donc important de réaliser une régulation.

Fonction de coût régularisé et vectorisé

$$J(\theta) = \frac{1}{m}\sum_{i=0}^{m-1}[-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n-1}\theta_j^2 \tag{8}$$

```python
def lrCostFunction(theta, X, y, Lambda):
    theta = theta.reshape((n,1)) # (4,1)
    h = sigmoid(X @ theta)
    J = (1/m) * np.sum(-y * np.log(h) - (1 - y) * np.log(1 - h)) + ((Lambda/(2*m)) *
    ↪  np.sum(theta[1:]**2)) #conventionnellement on n'applique pas la reg sur theta0 d'où theta[1:]

    return J

    """return
    Cost: 0.734819
    Expected cost: 0.734819
    """
```

FIGURE 7 – Fonction de coût régularisé et vectorisé

Descente de gradient régularisé et vectorisé

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{pour} \quad j = 0 \tag{9}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\left(\sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \lambda\theta_j\right) \qquad \text{pour} \quad j \geq 1 \tag{10}$$

```python
def lrCostGradient(theta, X, y, Lambda):
    theta = theta.reshape((n,1)) # (4,1)
    h = sigmoid(X @ theta)
    grad = (1/m) * (X.T @ (h - y))
    grad[1:] += (Lambda/m * theta[1:]) # reg pour j >= 1 (comme pour J(theta)) et theta != 0
    ↪  (conventionnellement)

    return grad.flatten()

    """return
    Gradients:[0.14656137 0.05144159 0.12472227 0.19800296]
    Expected gradients: 0.14656137  0.05144159  0.12472227  0.19800296
    """
```

FIGURE 8 – Descente de gradient régularisé et vectorisé

2.3 Apprentissage d'un classifieur One-Vs-All

Pour réaliser ce classifieur nous avons 10 classes, il nous faut donc 10 étiquettes. Il faut calculer les theta pour chaque classe pour cela on utilise un vecteur d'etiquette de dimension 10 indiquant la classe actuel. Par la suite à l'aide l'algorithme d'optimisation *fmin_tnc* on minimise les theta pour l'étiquette *(la classe)* correspondante puis on les stocks avant de passer à la classe suivante.

```
1  def learnOneVsAll(X, y, num_labels, Lambda):
2      m, n = X.shape
3      all_theta = np.zeros((num_labels, n))
4      initial_theta = np.zeros((n, 1))
5
6      for i in range(1,num_labels+1):
7          print('Optimizing for handwritten number %d...' %i)
8          y_1vsAll = (y == i)*1
9
10         result = fmin_tnc(lrCostFunction, fprime=lrCostGradient, x0=initial_theta, args=(X, y_1vsAll,
           ↪  Lambda), disp=False)
11
12         all_theta[i-1,:] = result[0]
13
14     return all_theta
```

FIGURE 9 – Apprentissage du classifieur

2.4 Prédiction One-Vs-All

La fonction *predictOneVsAll* nous permet de prédire le chiffre manuscrit grâce à classifieur, cette prédiction est réalisé pour tous les chiffres de X, 5000 au total.
Notre classifieur nous permet d'obtenir une prédiction de 96.46%, ce qui est plus que satifaisant sur un total de 5000 chiffres.

Pour ce faire on calcul la probabilité que le chiffre $X[i]$ appartient à l'une des 10 classes, ce qui nous donne 10 probabilités. On prends alors la plus grande, ce qui nous donne le chiffre numérique correspondant.

```
1  def predictOneVsAll(all_theta, X):
2      m = X.shape[0]
3      p = np.zeros((m, 1))
4      p = np.argmax(sigmoid(X @ all_theta.T), axis=1) + 1
5
6      return p
```

FIGURE 10 – Prédiction One-Vs-All