Introduction à l'apprentissage profond:
-
Numpy, Regression Lineaire et Regression Logistique

January 17, 2019

1 Introduction

Comme nous l'avons vu en cours, l'apprentissage profond (Deep Learning) est une manière particulière de faire de l'apprentissage automatique (Machine Learning) c'est à dire faire en sorte qu'une machine puisse prédire certaines informations à partir d'exemples d'apprentissage. Dans ce TP, vous allez implementer deux algorithmes simples d'apprentissage automatique peu profonds dans un premier temps afin de vous familiariser avec la notion de fonction objective et d'apprentissage par descente de gradient. Nous verrons ici deux modèles linéaires d'apprentissage supervisé: la regression linéaire et la regression logistique que vous ferez tourner sur des jeux de données qui vous seront fournis.

Ces modèles d'apprentissage, comme beaucoup d'autres que vous verez (comme les modèles profonds), mettent en oeuvre des opérations sur les données d'entrées telles que des applications linéaire, des non-linéarités, des opérations d'aggrégation etc... Dans votre longue carrière d'expert en Machine Learning, vous serez donc souvent amenés à manipuler des jeux données vectorielles et à effectuer toutes sortes d'opérations et calcul sur celles-ci. C'est pourquoi vous vous familliariserez dans la première partie du TP avec la librairie python Numpy qui implémente de multiples fonctions de calculs scientifiques.

2 Installation de l'environement de travail

2.1 Listes des ressources à récupérer

Placez vous dans un terminal sur un répertoire de travail que vous allez créer pour ce TP. Vous devez ensuite vous procurer les scripts et vous placer dans le repertoire de travail à l'aide des commandes suivantes:

```
git clone --recursive https://github.com/vleveau/DeepLearningTutorial.git
cd ./DeepLearningTutorial
```

Vous devriez retrouver les fichiers suivant:

- Exercice1_Numpy.py
- Exercice2_regression_lineaire.py
- Exercice3_regression_logisitique.py
- addBiasToDataset.py
- computeLoss.py
- computeLossRegLog.py
- data1.npy
- data2_nonlin.npy

- data2.npy
- gradientDescent.py
- gradientDescentRegLog.py
- non_linearities.py
- plotCurve.py
- plotData.py
- predictLinReg.py
- predictLogReg.py
- termcolor.py
- utils.py

Normalement tout devrait bien se passer jusqu'ici. Contrôlez que vous pouvez lancer les exercices "Exercice1_Numpy.py" et "Exercice2_regression_lineaire.py". Signalez toutes bizzareries à votre encadrant de TP. Voila, nous sommes maintenant prêts à mettre les mains dans le tracteur !

3 Rappels de python et utilisation de Numpy

Dans cette première partie, vous allez travailler sur le script python Exercice1_Numpy.py. Cet exercice est divisé en deux parties. La première partie consiste simplement en un rappel des types de base en python (nombres entiers, flotant, booléens, liste) ainsi que des opérations élémentaires que l'on peut effectuer sur ces objets. Prenez un moment pour bien vérifier que ces opérations de bases vous soient bien acquises. La deuxième partie de l'exercice consiste à apprendre à effectuer des opérations sur les matrices et les vecteurs avec l'aide de la bibliothèque Numpy, un outil crucial pour faire du calcul scientifique en python.

Note: Les exercices de ce TP sont principalement des codes à trous. Les questions qui vous sont posées sont donc principalement de remplir les blancs dans le script principal de l'exercice (ces blancs sont en général indiqués par la brave mention "ICI TU DOIS CODER").

Note 2: Les scripts des exercices sont organisés de sorte à ce qu'ils se mettent en pause à la fin de chaque sous-partie de l'exercice (aussi bien celles où vous devez produire du code que celles où vou n'avez rien à fournir). Chaque fois que vous avez fini une partie, lancez la partie suivante du code en appuyant simplement sur ENTREE. Faites des ctrl-c pour quitter le programme en cours chaque fois que vous êtes satisfait du résultat de votre ligne de code pour la question en cours. Puis relancez le script pour tester les questions suivantes, etc... jusqu'à arriver à la fin de la série de question. Pour passer à la suite, appuyer sur ENTREE.

Note 3: Oui, vous allez beaucoup appuyer sur ENTREE pour revenir au résultat de la question en cours. Donc essayez de ne pas trop malmener cette brave touche !

3.1 Opérations et structures numériques de base en python

Dans cette partie, vous n'avez rien à coder. Observez simplement le code ainsi que les résultats qu'il produit pour vous rappeller/familiariser avec les opérations de base de calcul et de manipulation des structures que l'on peut faire avec python. Soyez notamment sûrs de bien maitriser les boucles for pour iterer sur les éléments d'une structure (ici les listes).

3.2 Manipulation des matrices et vecteurs avec Numpy

Dans cet exercice, vous allez vous exercer à effectuer les opérations de bases nécéssaires pour apprendre les paramètres de regression linéaire ainsi qu'effectuer des prédictions à partir du modèle appris. Numpy est une bibliothèque logicielle open source qui fournit de multiples fonctions permettant notamment de manipuler et effectuer des opérations sur des structures matricielles et vectorielles. En Numpy, ces structures peuvent être obtenues à l'aide du même objet générique qui permet d'instancier n'importe quel tableau multidimensionnel : Ndarray.

Ndarray est l'objet principal de NumPy, il s'agit d'une table d'éléments (généralement des nombres), tous du même type, indexés par un tuple d'entiers positifs (commençant par 0, pas comme en Matlab). Les dimensions de la structure sont appelées axes. Ainsi, les vecteurs et les matrices peuvent être instanciés de manière générique avec un Ndarray comprenant respectivement 1 et 2 axes.

3.2.1 Opérations sur les vecteurs

Dans ces premières sous-parties, vous n'avez rien à coder, nous allons apprendre à créer et manipuler des vecteurs avec Numpy. Les premières lignes du code sont simplement à examiner pour comprendre les bases. Vous aurez ensuite à répondre à une série de questions pour apprendre à effectuer certaines opérations sur les vecteurs.

Instancier des vecteurs : Il existe différentes manières de déclarer et instancier un vecteur d'un espace de dimension donnée. Les premières lignes de cette partie du code vous donnent différents exemples.

```
print("Instantiation d'un vecteur R 2 (2 dimension reelles):")
a = np.ndarray(2)
#Contient des valeurs très proche de 0

print("Vecteur nul dans R 2 (2 dimension reelles):")
b = np.zeros(2)
#Affiche "array([ 0.,  0.])"

print("Vecteur remplie de 1 dans R 5:")
c = np.ones(5)
#Affiche "array([ 1.,  1.,  1.,  1.,  1.])"

print("Vecteur dans R 4 contenant des valeurs aléatoires:")  #
d = np.random.random(4)
```

Accès aux valeurs des vecteurs : Les valeurs des tableaux Numpy sont accessibles de façon similaire aux listes simples en python. Gardez bien en tête que le système d'indiçage/indexation pour accéder aux éléments d'une liste/tableau se fait bien dans l'interval $[0; N-1]$ pour un tableau à N éléments. Il est également possible d'accéder et retourner en une seule ligne de commande aux plusieurs valeurs du vecteur en spécifiant un interval d'indices que l'on veut retourner. Par exemple, pour retourner de la composante (dimension) 2 à la composante 4 d'un vecteur:

```
y = x[2:4]
```

Le resultat est retourné sous la forme d'un nouveau vecteur y. On peut aussi retourner les 3 premières composantes du tableau comme suit:

```
y = x[:3]      # ici, ":3" signifie du 1er aux 3 eme élément inclus
```

ou bien retourner de la 3eme composante à la dernière comme ceci:

```
y = x[3:]      # ici, "3:" signifie daux 3 eme au dernier élément
```

Récuperation de la dimension d'un vecteur: examiner les dimensions du tableau numpy : Ici, on montre que l'on peut accéder aux dimensions des axes du tableau multidimensionel avec *shape*. Cette dernière renvoit sous les dimensions sous la forme d'un tuple. Dans notre cas, notre vecteur correspond à un tableau à un seul axe. La dimension d'un vecteur **x** peut donc être obtenue comme suit:

```
dim = x.shape[0]    #La valeur de la dimension du 1er (et seul) axe du tableau
```

Parcours des valeurs du vecteur : Ici, on montre comment on peut boucler sur les éléments d'un vecteur de 3 façons: en itérant sur les indices puis acceder aux valeurs par les indices, en itérant directement sur les éléments du tableau, puis en itérant à la fois sur les indices ET les éléments avec *enumerate*. Cette dernière est souvent très pratique:

```
x = np.random.random(4)
for i, x_i in enumerate(x):
        print("La valeur de la %d eme composante est: %f" % (i,x_i))
```

PARTIE QUESTIONS : A VOUS DE JOUER ! Dans cette partie vous devrez compléter les endroits du code où figure la brave mention "***ICI TU DOIS CODER***". Certaines des questions sont déjà remplies histoire de vous inspirer un peu.

Note : Dans cette partie vous serez amenés à utiliser certaines fonctions de calcul numpy très pratiques pour effectuer des opération d'algèbre linéaire tel qu'un produit scalaire entre deux vecteurs **x** et **y** (avec np.dot(x,y)). Nous verrons par la suite qu'étant donnée que le ndarray est un objet générique cette fonction "produit scalaire" de numpy peut également être utilisé pour les produits matriciels.

3.2.2 Opérations sur les matrices

Cette partie est similaire à la précédente, dans un premier temps on fait quelques points généraux sur la manière d'instancier les objets, puis vous répondrez à une série de questions en implémentant les bouts de codes demandés.

On notra simplement que la différence avec l'instantiation d'un vecteur consiste à préciser les dimensions de deux axes du tableau numpy au lieu d'un seul et que l'accès se fait maintenant comme pour un tableau classique à deux entrées:

```
M = np.random.random((3,3)
shape                  = M.shape
nb_lignes          = shape[0]      # renvoit 3
nb_colones         = shape[1]            # renvoit 3

#Parcours des valeurs de la matrice
for i in range(nb_lignes):
        for j in range(nb_colones):
                print("La valeur de la composante %d%d est:  = %f" % (i, j,
                ↪  M[i,j]))
```

PARTIE QUESTIONS : A VOUS DE JOUER ! Dans les questions, vous aurez à comprendre comment on effectue un produit matriciel entre deux matrices ou bien entre une matrice et un vecteur. Vous verrez deux manières de procéder: avec une boucle, et avec la primitive *dot(,)* de numpy. Le script vous permet de calculer le temps mis pour effecteur ces deux calculs. Pensez bien à écrire votre code entre les deux instructions suivante:

```
start_time = time.time()
# Votre code pour le produit matricielle ici
stop_time =  time.time()
```

Comme vous le verrez, la fonction numpy dot est beaucoup plus rapide que votre implémentation à base de boucles car cette fonction optimise très bien les calculs de nature vectorielle. C'est pourquoi

il sera toujours intéressant d'écrire vos équations en notation vectorielle afin de les implementer simplement et efficacement par la suite.

Note importante: L'important à retenir est qu'on peut considérer une matrice soit comme une application linéaire soit comme un ensemble de vecteurs (chaque ligne correspond à un vecteur d'apprentissage par exemple). Le produit matricielle entre une matrice **M1** de dimensions n x k (n lignes et k colonnes) et une autre **M2** de dimension k x m donnera un résultat **M1M2** (calculé avec np.dot(M1,M2)) de dimensions n x m. Autremment dit le nombre de colonnes de la première matrice doit être égale au nombre de lignes de la deuxième matrice. Un vecteur $\mathbf{x} \in \mathbb{R}^d$ correspond à une matrice de dimension d x 1. On pourra donc calculer le resultat de l'application linéaire décrite par la matrice $\mathbf{M}$ de dimensions n x d sur le vecteur $\mathbf{x}$ de dimensions d x 1 avec np.dot(M,x). Le résultat sera un vecteur $\mathbf{y} \in \mathbb{R}^n$ (représenté par un tableau de dimensions n x 1).

Intuitivement, appliquer linéairement $\mathbf{M}$ de dimensions n x d à $\mathbf{x} \in \mathbb{R}^d$ revient à "*empiler*" les résultats des produits scalaires entre $\mathbf{x}$ et chacun des vecteurs colonne de $\mathbf{M}$. Si il y a n colonnes, il y aura donc n produits scalaires et donc n valeurs dans le vecteur résultats. Les composantes du vecteur résultat correspondent ainsi aux similarités (au sens de la métrique du produit scalaire) entre $\mathbf{x} \in \mathbb{R}^d$ et une base de n vecteurs dans $\mathbb{R}^d$.

4 Votre premier algorithme d'apprentissage supervisé: la Regression Linéaire

4.1 Rappel (vous n'avez rien à produire (ici)

Comme on l'a vu en cours, la régression linéaire est un modèle cherchant à établir un lien linéaire entre des données d'observation et des données à prédire. Plus concrètement, on peut vouloir chercher un lien linéaire entre un vecteur $\mathbf{x} \in \mathbb{R}^d$ et une quantité scalaire (un réel) $y \in \mathbb{R}$ sous la forme:

$$y = \theta_0 + x_1\theta_1 + x_2\theta_2 + x_3\theta_3 + ... + +x_d\theta_d = \theta_0 + \sum_i^d x_i\theta_i \tag{1}$$

que l'on peut aussi écrire en notation vectorielle:

$$y = \theta_0 + \boldsymbol{\theta}^T\mathbf{x} \tag{2}$$

où $\boldsymbol{\theta} \in \mathbb{R}^d$ correspond au vecteur contenant les paramètres de notre modèle que l'on va vouloir apprendre pour prédire la bonne valeur de y en fonction du vecteur $\mathbf{x}$. Une fois ces paramètres appris par notre algorithme d'apprentissage, on pourra utiliser la fonction de prédiction $f_{\boldsymbol{\theta}}(\mathbf{x})$ apprise pour prédire la valeur de y_{new} sur un nouveau vecteur $\mathbf{x_{new}}$ que l'on n'a pas encore observé:

$$y_{new} = f_{\boldsymbol{\theta}}(\mathbf{x_{new}}) = \theta_0 + \boldsymbol{\theta}^T\mathbf{x_{new}} \tag{3}$$

Pour simplifier les calculs en python, on préfère que la fonction de prédiction puisse se calculer à partir d'une notation complètement vectorielle. Ce que l'on fait en pratique, c'est ajouter une composante supplémentaire x_0 au vecteur $\mathbf{x}$ qu'on peut mettre égale à 1

$$\mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ . \\ . \\ . \\ x_d \end{pmatrix} \tag{4}$$

De sorte à ce que la fonction de prédiction linéaire puisse s'exprimer simplement sous la forme du produit scalaire:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T\mathbf{x} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ . \\ . \\ . \\ \theta_d \end{pmatrix}^T \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ . \\ . \\ . \\ x_d \end{pmatrix} = \theta_0 + \sum_i^d x_i\theta_i \tag{5}$$

Dans cet exercice, vous implémenterez le cas simple d'une régréssion linéaire à une seule variable d'entrée et une seule variable de sortie qui pourra donc s'écrire sous la forme:

$$y = f_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x \tag{6}$$

C'est à dire une brave fonction affine dont on pourra afficher la représentation graphique (une droite) sur une figure en 2 dimensions. Par la suite vous aurez donc à implémenter le calcul de la fonction de coût du modèle sur l'ensemble d'apprentissage, le calcul du gradient (en mode batch) de cette fonction de coût ainsi que le code qui met à jour les paramètres du model à partir du gradient et de la valeur des paramètres à l'itération en cours.

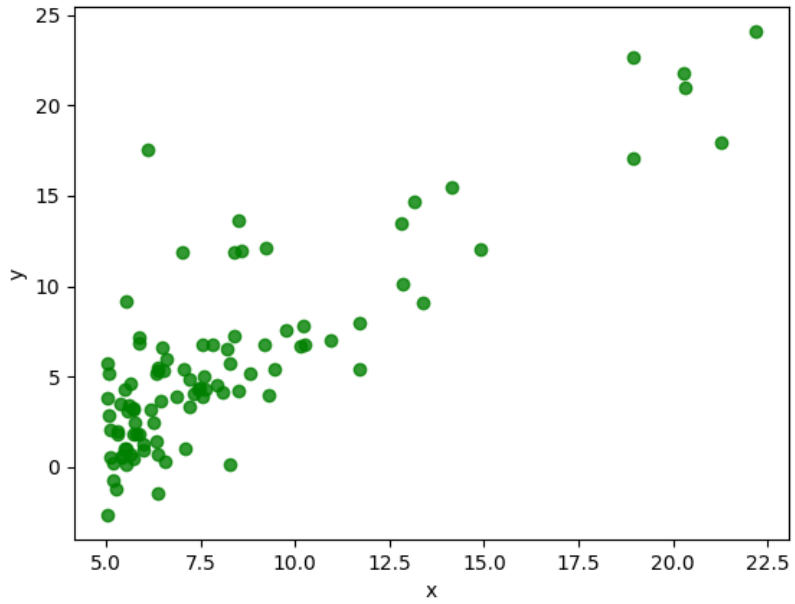

Figure 1: Visualisation du jeu de données pour la régression linéaire. Ici nous voudrions apprendre les paramètres de la droite qui "représente le mieux" la tendance linéaire que l'on voit apparaître dans ce jeu de données.

4.2 Visualisation du jeu de données (rien à faire ici)

On commence gentillement à charger le jeu de données puis à regarder ce qu'il y a dedans. Le tableau numpy retourné contient 97 lignes et 2 colonnes soit 97 exemples d'apprentissages. Chaque exemple correspond à un couple de réels $(x, y) \in \mathbb{R}^2$ dont nous allons devoir avoir à prédire linéairement l'une en fonction de l'autre.

```
#On charge le tableau numpy
dataset = np.load("data1.npy")

#Recuperation du nombre d'exemples d'apprentissage ainsi que la dimension des
↪  vecteurs
n_samples = dataset.shape[0]

#On sépare les valeurs d'entrée et les valeurs à predire dans deux tableaux
↪  différents
X = dataset[:,0]
y = dataset[:,1]
```

On visualise ensuite sur un graphique le jeu de données avec la librairie matplotlib et vous devriez voir apparaitre la Figure 1:

```
plt.scatter(X, y, c="g", alpha=0.8, marker='o')
plt.ylabel("y")
plt.xlabel("x")
plt.show(break=False)    #Pour continuer a executer le programme en laissant
↪  l'image s'afficher
```

4.3 Calcul de la fonction de prédiction et de la fonction objective (10 minutes)

Dans cette partie, **vous devrez compléter les fichiers predictLinReg.py et computeLoss.py** pour calculer la valeur de la fonction de coût du modèle à partir du jeu de données. Vous pouvez l'implémenter avec une boucle comme ce qui est suggéré dans le code, ou bien vous pouvez dès à présent penser à écrire tout cela en notation vectorielle afin de pouvoir effectuer plus rapidement les calculs et avoir une fonction générique qui pourra fonctionner sur n'importe quel dataset d'une taille arbitraire (nb exemples et nb dimensions).

Rappel : Pour la régression linéaire, on peut utiliser une fonction de coût $J(\boldsymbol{\theta})$ qui consiste en la somme des carrés des erreurs de prédictions pour chaque exemple d'apprentissage:

$$J(\boldsymbol{\theta}) = \frac{1}{2N}\sum_{n}^{N}(f_{\boldsymbol{\theta}}(x_n) - y_n)^2 \tag{7}$$

où $f_{\boldsymbol{\theta}}(x_n) = \theta_0 + \theta_1 x_n$. Faites tourner la suite de l'exercice 2 après avoir completé la boucle dans le script computeLoss.py. Vous devriez voir afficher une loss initiale d'environs 33.18. Note: On a initialisé la valeur du vecteur de paramètres à $[0.0, 0.0]$.

Note : Notation vectorielle de la régression linéaire : On peut aussi exprimer ce calcul avec une simple équation en notation vectorielle. Pour cela, on exprime dans un premier temps le résultat de la fonction de prédiction en notation vectorielle:

$$f_{\boldsymbol{\theta}}(\mathbf{X}) = \mathbf{X}^T\boldsymbol{\theta} \tag{8}$$

où $\boldsymbol{\theta} \in \mathbb{R}^d$ est une matrice de dimensions d x 1 et $\mathbf{X}$ est une matrice de dimensions N x d dont les N vecteurs lignes correspondent aux vecteurs d'apprentissage d'entrée. Dans notre cas de la régression linéaire à 1 variable la matrice prend la forme suivante:

$$\mathbf{X}^T = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ . & . \\ 1 & x_n \\ . & . \\ 1 & x_N \end{pmatrix} \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix} \tag{9}$$

La fonction de coût peut ainsi s'exprimer:

$$J(\boldsymbol{\theta}) = \frac{1}{2N}(\mathbf{X}^T\boldsymbol{\theta} - \mathbf{y})^T(\mathbf{X}^T\boldsymbol{\theta} - \mathbf{y}) \tag{10}$$

que l'on peut réécrire:

$$J(\boldsymbol{\theta}) = \frac{1}{2N}(\hat{\mathbf{y}} - \mathbf{y})^T(\hat{\mathbf{y}} - \mathbf{y}) = \frac{1}{2N}||\hat{\mathbf{y}} - \mathbf{y}||_2^2 \tag{11}$$

où $\mathbf{y} \in \mathbb{R}^N$ est le vecteur dont chacune des composantes y_n sont les valeures à prédire à partir de leur x_n correspondant, et $\hat{\mathbf{y}} \in \mathbb{R}^N$ correspond au vecteur de valeurs prédites par le modèle.

4.4 Descente de gradient (20 minutes)

La descente de gradient est une méthode d'optimisation numérique permettant de trouver les valeurs des paramètres d'une fonction qui la minimise. Dans notre cas, nous voulons minimiser l'erreur de prédiction moyenne de notre modèle, nous avons donc défini une fonction objective dépendant des données d'apprentissage et des paramètres du modèle. Cette méthode d'optimisation va consister à calculer le gradient de cette fonction objectif par rapport aux paramètres du modèles et de déplacer le vecteur des paramètres courant par une "petite" translation dans la direction du gradient.

Définition générale du gradient d'une fonction à plusieurs variables : Il s'agit simplement du vecteur contenant les dérivées partielles de la fonction, c-a-d les dérivées de la fonction par

rapport à chaque variable indépendamment des autres:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ . \\ . \\ . \\ \frac{\partial J(\theta)}{\partial \theta_D} \end{pmatrix} \tag{12}$$

En descente de gradient (en mode batch), la mise à jour de chaque paramètre θ_j du modèle à l'itération t se fait donc avec la règle suivante:

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \rho \frac{\partial J(\theta^{(t)})}{\partial \theta_j} \tag{13}$$

ou bien, en notation vectorielle:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \rho \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})^{(t)} \tag{14}$$

où ρ est le learning rate (pas d'apprentissage).

Question : Calculez les dérivées partielles $\frac{\partial J(\theta)}{\partial \theta_0}$ et $\frac{\partial J(\theta)}{\partial \theta_1}$ de la fonction de coût de notre modèle de régréssion linéaire et implémentez la mise à jour des paramètres dans le fichier gradientDescent.py. Vous devrez aussi retourner l'historique des valeurs de la fonction de coût pour les différentes itérations dans un tableau numpy afin de pouvoir tracer ensuite son évolution au fil des itérations d'apprentissage.

Lancez ensuite la suite du programme, ce dernier utilise par défaut un learning rate de 0.01 et un nombre d'époques (itérations) de 1500. Vous devriez obtenir une figure similaire à la Figure 3 et observez que le fonction de coût décroit au fil des itérations et converge après un certain temps.

4.5 Visualisation du modèle linéaire appris (rien à faire ici)

Ici on trace simplement la droite affine apprise par notre algorithme. Comme le montre la Figure 2, on constate avec émerveillement qu'elle "fit" convenablement notre jeu de données.

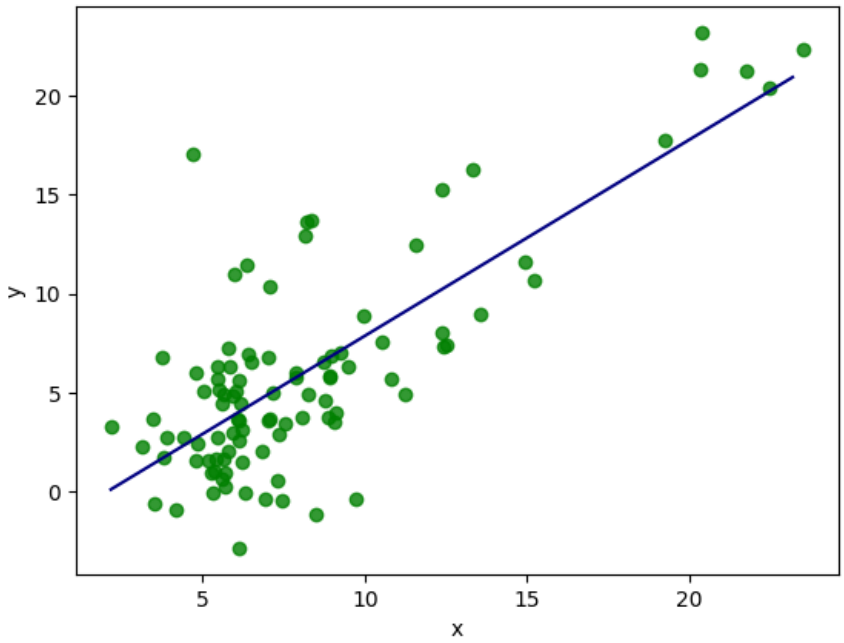

Figure 2: Modèle linéaire appris sur notre jeu de données

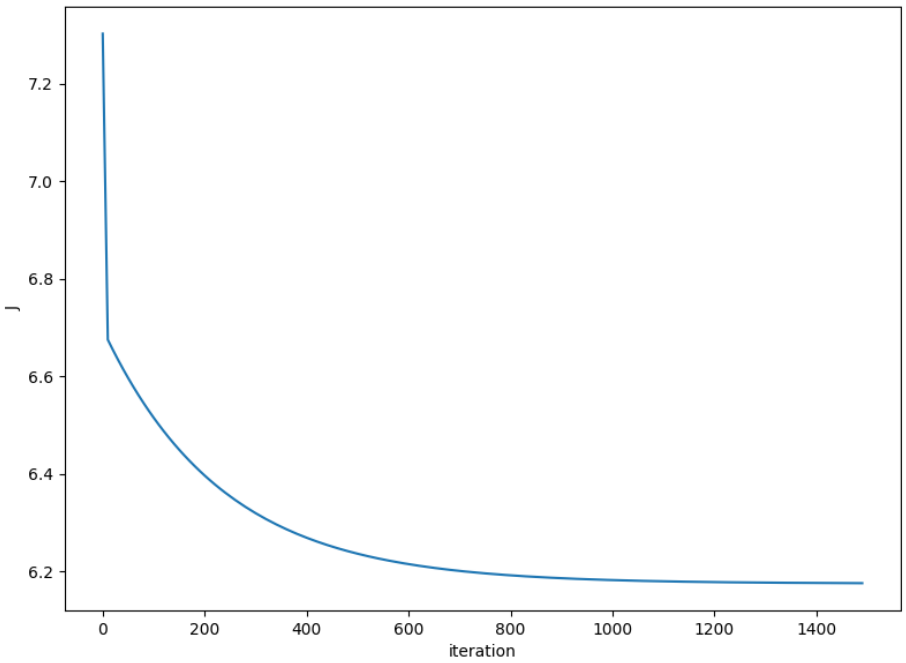

Figure 3: Evolution de la fonction de coût en fonction des itérations.

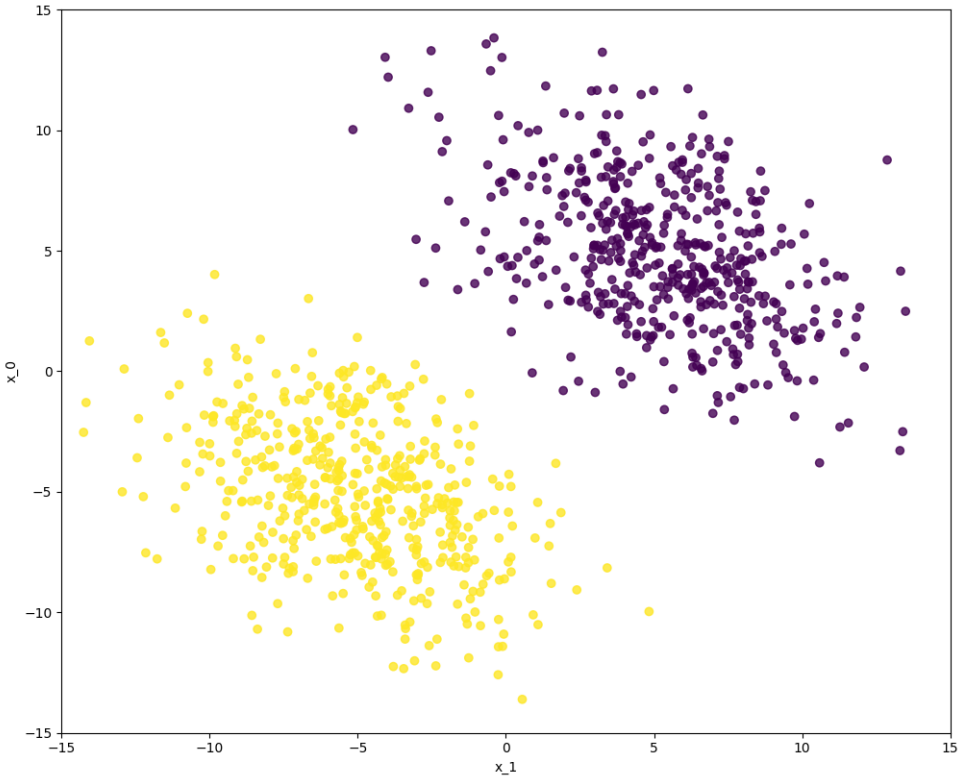

Figure 4: Visualisation du jeu de données pour la régression logistique. Ici nous voudrions apprendre les paramètres de la droite permettant de séparer nos deux classes.

5 Classification supervisée: la Régression Logistique (30 minutes)

Dans cette partie, nous allons implémenter un algorithme de classification supervisée. Contrairement à la régression linéaire qui consiste à prédire une valeur scalaire, la régression logistique a pour but de prédire à partir de données vectorielles d'entrée une variable dite catégorielle. C'est à dire, une variable correspondant à un nombre entier compris entre 1 et K pour un problème à K classes d'objets. Nous considererons ici un cas simple à deux classes linéairement séparables. **Attention, dès à présent vous n'avez plus le droit de faire des boucles. Toutes vos implémentations devront se faire sous forme complètement vectorielle**.

5.1 Classification sur 2 classes

Dans cette première partie nous allons mettre en place un algorithme qui va apprendre l'hyperplan séparateur de ce jeu de données.

5.1.1 Calcul de la fonction de prédiction et de la fonction objective (15 minutes)

La fonction de prédiction de la régression logistique est similaire à celle de la régression linéaire mais on applique en plus une fonction d'activation non linéaire permettant de "simuler une distribution de probabilité sur la sortie de la fonction:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \mathbf{x}) \tag{15}$$

avec $\sigma(.)\mathbb{R} \Rightarrow \mathbb{R}$ une fonction d'activation non linéaire (appellée fonction sigmoid) qui s'applique indépendament sur la sortie du produit scalaire. Cette fonction est définie par:

$$\sigma(z) = \frac{1}{1 + exp(-z)} \tag{16}$$

Pour la régression logistique, on utilisera comme fonction de coût l'entropie croisée:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n}^{N} y_n \log(f_{\boldsymbol{\theta}}(\mathbf{x_n})) + (1 - y_n)(1 - \log(f_{\boldsymbol{\theta}}(\mathbf{x_n})) \tag{17}$$

où $y_n \in 0; 1$ d'où l'appellation de classification binaire.

Questions :

- Complétez dans le fichier non_linearities.py la fonction sigmoid de sorte à ce qu'elle puisse s'appliquer sur n'importe quel type de structure d'entrée de dimension arbitraire (un scalair, vecteur ou une matrice). Cette dernière devra donc s'appliquer sur chaque éléments de la structure. Numpy sait très bien faire de telles applications "element-wise" naturellement.
- Complétez les fichiers predictLogReg.py et computeLossLogReg.py pour implémenter en notation vectorielle les fonctions de prédiction et de coût du modèle.

5.1.2 Descente de gradient (15 minutes)

Ici, on vous fait grâce de calculer les dérivées partielles de cette fonction de coût (bien que ça ne soit pas si compliqué que ça). En réalité, les dérivées partielles sont très similaires à celles de la régression linéaire:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{N} \sum_{n}^{N} (f_{\boldsymbol{\theta}}(\mathbf{x_n}) - y_n) x_n^j \tag{18}$$

Comme vous devrez bien sûr coder en notation vectorielle, on ne le repète pas assez souvent.

Questions : Remplissez le fichier gradientDescentRegLog.py pour compléter la mise à jour des poids. Déroulez ensuite l'exercice pour constater vos prouesses comme pour la partie précédente. Vous devriez notament visualiser l'hyperplan séparateur appris comme sur la Figure 5.

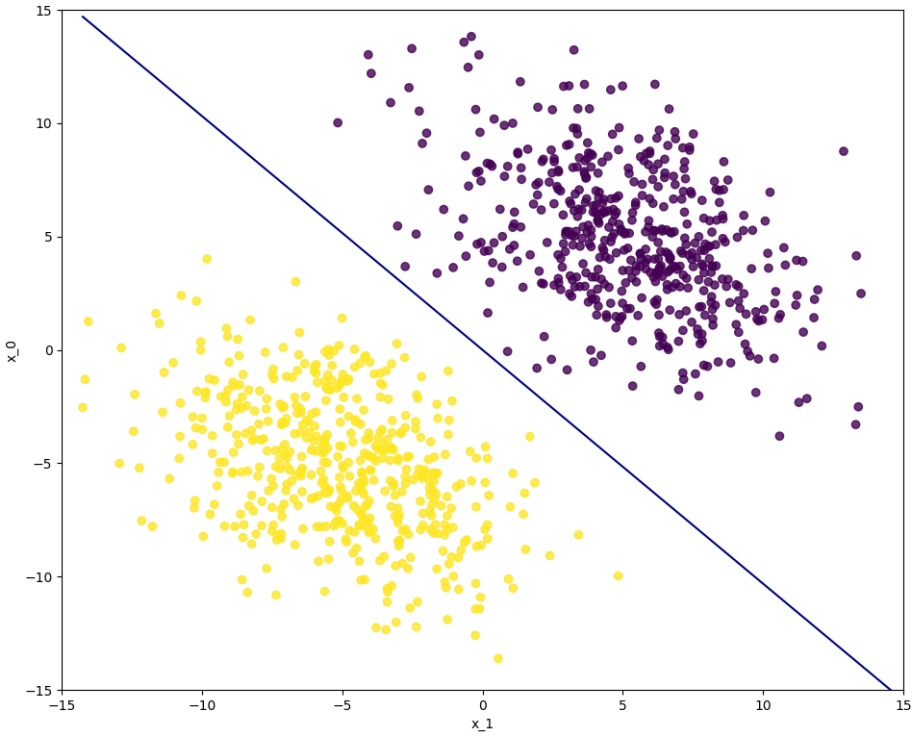

Figure 5: Hyper plan séparateur appris par la régression logistique