Perrine WURTZ
Simon ANTROPIUS

Rapport de projet
Maths data

Introduction

Ce rapport a pour but de présenter une implémentation de la méthode de régression logistique multi-classes à travers un problème de classification d'images. Pour ce faire, nous avons implémenté notre propre descente de gradient en python puis comparé nos résultats avec la méthode fournie par `scikit-learn`.

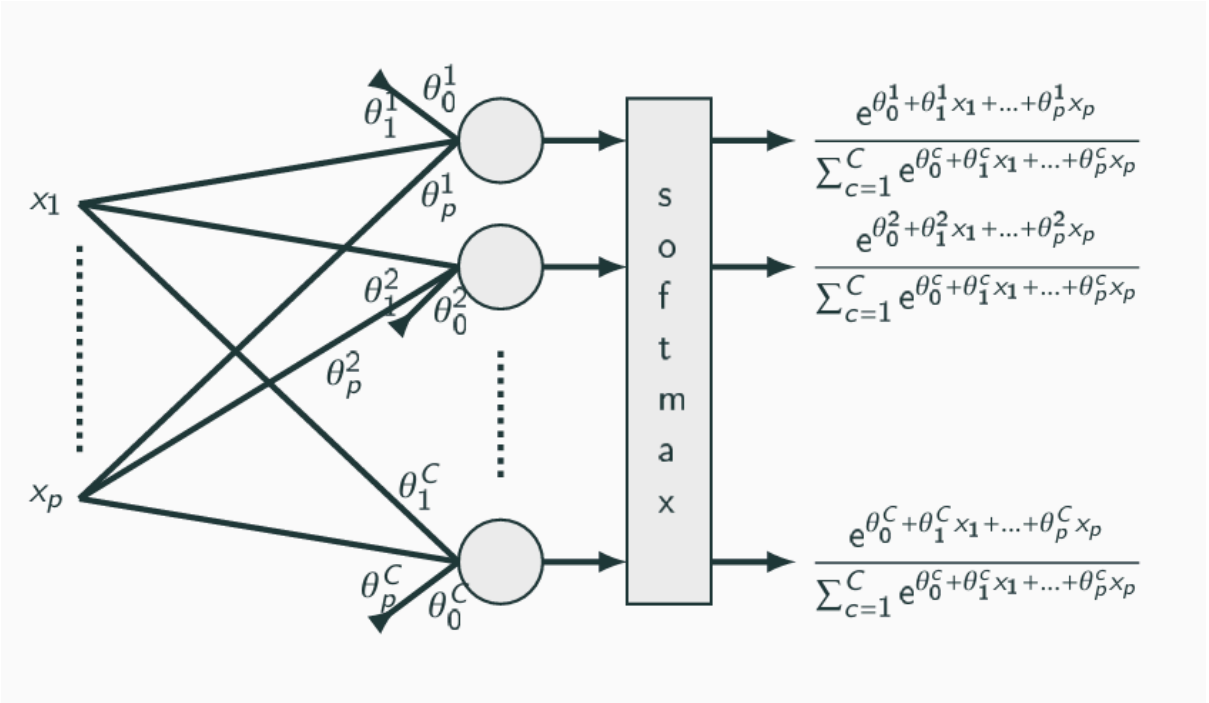
On considère ainsi l'ensemble des données handwritten digits dataset de la bibliothèque `scikit-learn`. Chacune des 1797 données de cet ensemble est une image en niveau de gris (de 0 à 16) de taille 8×8 pixels représentant un chiffre manuscrit, et étiquetée par ce même chiffre.

En utilisant une régression logistique multi-classe, le but est alors d'écrire un programme permettant de classer ces images pixélisées en fonction du chiffre qu'elles représentent, et de comparer le modèle trouvé au modèle déjà présent sur `scikit-learn`.

Description de la méthode

La méthode utilisée pour résoudre ce problème de classification est la descente de gradient. Le principe est le suivant :
On se donne une fonction d'erreur `E` qui va représenter l'erreur entre une prédiction de classe et la réalité. Puis, le gradient d'une fonction donnant la direction de la pente de cette fonction, le but va être de descendre le long de cette pente dans le but de trouver son minimum, minimisant ainsi l'erreur entre prédiction et réalité.

Il faut donc commencer par mettre en place un modèle permettant d'appliquer cette descente. Ce modèle est un réseau de neurones, qu'on peut représenter par ce schéma vu en cours :



Les `xi` représentent les entrées du modèle, donc ici, pour chaque image, 64 pixels. Comme on cherche à classifier ces images en 10 classes, notre réseau possède 10 neurones. Pour chaque `xi`, on associe un poids sur chaque neurone et on ajoute un 65^e poids à chaque neurone représentant le biais.
On aura donc en tout 65*10 = 650 paramètres, qu'on va mettre à jour au fur et à mesure de l'apprentissage.
En sortie de chaque neurone, la fonction d'activation softmax permet d'associer une probabilité d'appartenir à la classe représentée par ce neurone. Chaque sortie sera donc comprise entre 0 et 1 et la somme des probabilités sera égale à 1.

Il ne reste alors plus qu'à choisir la probabilité la plus élevée parmi les 10 pour déterminer la classe de l'image.

Préparation des données

Avant d'appliquer la descente de gradient, il est nécessaire de préparer les données, soit pour optimiser la descente, soit pour obtenir de meilleurs résultats.

- Normalisation des données d'entrée : on utilise un `StandardScaler` python afin d'avoir une moyenne de 0 et un écart-type de 1 (Z-normalisation). Ceci permet d'accélérer la convergence de l'algorithme et d'obtenir une meilleure stabilité numérique en évitant que certaines variables, ayant des échelles très différentes, dominent les autres.

- Encodage OneHot des labels : grâce à un `OneHotEncoder` python, on transforme les labels des différentes classes. Cela est nécessaire, car le modèle pourrait interpréter ces valeurs comme des relations ordinales (par ex. classe 2 > classe 1), ce qui est incorrect et ainsi produire des erreurs. Pour résoudre ce problème, le OneHotEncoder va, par exemple, transformer les classes 0, 1, 2 en :
  0 : [1, 0, 0]
  1 : [0, 1, 0]
  2 : [0, 0, 1]
- Division des données : pour pouvoir tester notre modèle, il faut séparer les données d'entrée en données d'entraînement et de test. En effet, une fois le modèle entraîné, il est incorrect de le tester sur les mêmes données que celles utilisées pour l'entraînement, car si on teste le modèle sur les mêmes données d'entraînement, on ne mesure pas sa capacité à généraliser, mais juste sa capacité à mémoriser ces données.
  Nous avons choisi de garder 80% des données pour l'entraînement et 20% pour le test.

Descente de gradient

Nous avons utilisé le fichier python fourni durant les TD de 'mathématiques pour les data sciences'. Ce fichier contient un objet de la classe `GradientDescent`. On lui fournit une fonction, un delta (learning rate), un epsilon et un nombre maximum d'itérations, puis on peut appliquer la descente de gradient sur cette fonction.
Les deux paramètres importants ici sont le learning rate et le nombre maximum d'itérations.

- Un learning rate trop grand risque de ne pas converger en "sautant par-dessus" le minimum de la fonction au moment de la descente et un learning rate trop petit convergera, mais demandera beaucoup plus d'itérations que nécessaire, ce qui entraînera un temps de calcul supplémentaire.
- Le nombre maximum d'itérations permet de définir un critère d'arrêt supplémentaire pour l'algorithme. En effet, au cas où le critère d'arrêt basé sur epsilon, la tolérance pour l'arrêt de l'algorithme n'atteindrait jamais le niveau requis, il faut pouvoir arrêter l'algorithme par un autre critère.
  Ce critère permet aussi d'avoir un certain contrôle sur la durée de l'algorithme, donc sur la précision du modèle entraîné. Plus on fait d'itérations, plus cela prendra de temps, mais meilleur sera le modèle, jusqu'à un certain point (généralement).

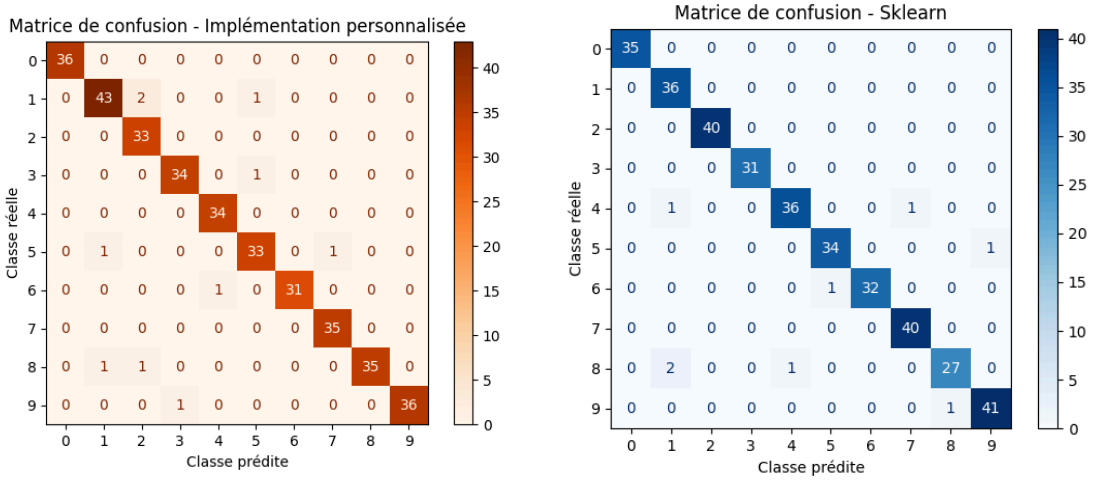
Scikit-learn

Ici, nous avons utilisé la bibliothèque scikit-learn, pour classer les images. Le principe est le même que pour notre descente de gradient.

Résultats et comparaison des modèles

Matrices de confusion :

Pour pouvoir mieux visualiser l'erreur et la précision des modèles, nous avons affiché la matrice de confusion. Celle-ci nous permet de voir combien d'images ont bien été classées, en fonction de la classe réelle. Les cases les plus colorées représentent les classes les plus souvent prédites pour une classe réelle donnée.
Voici les matrices que l'on a obtenues pour une exécution du programme Python :



Précision des modèles :

Nous avons aussi calculé la précision de classement pour chaque modèle. Cette exécution nous a donné une précision de 97,50% pour notre descente, et 97,78% pour sklearn.

On remarque donc que ces deux modèles sont équivalents. Bien que les résultats affichés puissent légèrement varier d'une exécution à l'autre en raison de la répartition aléatoire des données d'entraînement et de test, ces variations restent faibles. En effet, la différence de précision observée ne dépasse jamais 3 %, ce qui prouve la stabilité dans l'apprentissage de notre modèle. Cette similarité entre les résultats de sklearn et ceux de notre implémentation confirme que notre approche est fiable et reproductible dans des conditions similaires.

Erreur sur le modèle personnalisé :

Aussi, pour notre descente de gradient, nous avons décidé de montrer l'erreur du modèle. Avant l'entraînement, nous avons une erreur de l'ordre de plusieurs milliers, alors qu'après, nous avons une erreur de 2,5. Ce qui montre que l'algorithme de descente de gradient a bien rempli son rôle, consistant à diminuer l'erreur.

Conclusion

En conclusion, ce projet nous a permis de mieux comprendre la régression logistique multi-classes et la descente de gradient en l'implémentant nous-mêmes. Nous avons aussi vu l'impact du choix des paramètres et l'importance de l'initialisation des données. En comparant avec scikit-learn, nous avons validé notre approche et constaté que les résultats restent stables.