Compétition Kaggle - IFT-3395

Équipe: Hanz & Al
Hanz Schepens (20189679)
Al-Wahid Bio-Tchane (20137307)

6 novembre 2022

Résumé

Ce document contient toutes les informations concernant la compétition Kaggle, son développement en langage Python, et l'analyse des données. Cette compétition a été réalisé dans le cadre de la première compétition d'apprentissage automatique lancé par le professeur Guillaume Rabusseau. Nous sommes des étudiants du BAC et nous partageons notre expérience enrichissante sur le monde de l'apprentissage automatique.

1 Feature Design

Les méthodes de pré-traitements pour ce projet ont été très minime. Les seul pré-traitements que nous avons fais sur les données ont été d'enlever la dernière colonne des ensembles de données d'entraînement, car elle contenait une colonne "NaN", qui causait des problèmes plus tard dans le code lors d'une multiplication scalaire entre les vecteurs et de seulement sélectionner la deuxième colonne pour les étiquettes. Nous avons décidé de laisser les données plus ou moins tel quel, car les algorithmes d'entraînements vont tout de même fonctionner, même si les données ne sont pas découpés parfaitement. En ce qui concerne la sélection des propriétés, nous les avons tous choisi, car c'était plus simple.

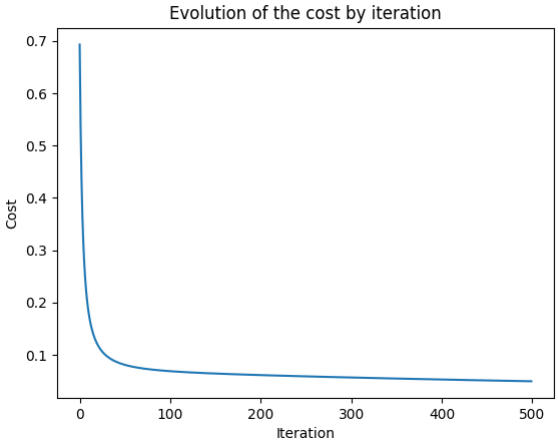
2 Algorithmes

Pour le premier algorithme que nous devions implémenter, nous avons fais la régression logistique tel que demandé. Nous avons utilisé beaucoup de ressources en ligne pour nous aider à accomplir cette tâche, car les algorithmes à implémenter sont assez complexe et nous étions limité dans les librairies que nous avions le droit d'utiliser. Nous avons commencé par faire le petit pré-traitement sur les données et nous voulions normaliser les données, mais nous avons remarqué que les données étaient déjà normalisés. Ensuite, nous avons fais une liste pour chacune des classes différentes de l'ensemble d'entraînement, car nous avons utilisé la stratégie "One vs All" pour déterminer les classes des images. Cette technique consiste a passer une image à chacun des modèles entraînés pour savoir s'il fait parti ou non de cette classe, qui est décidé par la fonction d'activation sigmoïde pour la régression logistique. Pour entraîner les modèles nous avons utilisé l'algorithme qu'il y avait dans les notes. Cela consiste à faire une fonction de coûts, descente de gradient et ensuite de prédire nos résultats. Pour chaque modèles, nous calculons aussi le niveau de précision. Cela peut nous aider à déterminer un modèle s'est bien entraîné sur les données d'entraînement. Nous utilisons aussi le "back propagation" pour calculer les modèles. Une fois tous les modèles entraînés, on lui donne les images de l'ensemble d'entraînement et on prend l'argument maximal pour déterminer la classe de l'image. **Pour le second algorithme**, nous avons utilisé la librairie "sklearn" pour faire un réseaux de neurones. Nous n'avions pas fait de "cross-validation" dans la régression logistique, mais la librairie de "sklearn" nous permettait de le faire relativement facilement, donc nous l'avons fait. Nous lui avons donné plusieurs paramètres pour faire la recherche des meilleurs possibles, mais cela prenait trop de temps, donc nous avons réduit l'espace de recherche des paramètres optimaux et cela a quand même pris environ 6h de recherche. Ensuite, nous n'avions qu'à utiliser ces paramètres pour exécuter notre réseaux de neurones et cela nous à donné 81% de précision.

3 Méthodologie

Pour ce qui concerne l'ensemble d'entraînement de la régression logistique, nous n'avons fait aucune séparation pour un ensemble de validation et nous n'avons pas essayé d'optimiser les hyper-paramètres. Nous avons choisi un alpha de 0.01, car après quelques essaies, est celui qui nous donnait le meilleur résultat. Pour le réseaux de neurones, nous avons utilisé la cross validation pour chercher les meilleurs paramètres pour nos modèles. Nous avons utilisé la cross validation avec k=3 et pleins de paramètres différents pour faire la recherche. Cela a prit environ 5 heures pour trouver les paramètres les plus optimisé. Nous avons terminé avec une activation relu, un alpha de 0.05, les hidden_layer_sizes à (50, 100, 50), un learning_rate constant et le solver adam. Tout ces paramètres nous on donné une précision de 81% sur la compétion publique, étant donné que le résultat final n'est pas encore divulgué. Il y aura un fichier en annexe avec tout les paramètres essayés.

4 Résultats



Ce graphique est pour la régression logistique et il est l'évolution du coût, donc de l'erreur, lorsqu'on augmente les itérations. Ce graphique représente l'évolution du coût pour le modèle 0 seulement, car il sera similaire pour les autres modèles aussi. Clairement, il y aura un sur-apprentissage lorsqu'on augmente le nombre d'itérations, car le coût descend très rapidement et ensuite il décroît lentement. Ceci est aussi observé dans nos prédictions, car nous avons un niveau de précision assez élevé lors de l'entraînement, mais lorsque on lui passe les données de test, la précision est d'environ 16%, ce qui est très mauvais. Nous avons une différence d'environ 80% entre les deux, ce qui veut dire que la régression logistique n'est pas adapté pour résoudre ce genre de problème, du moins, la nature de nos données tel qu'elles le sont, ne nous permet pas de le résoudre efficacement. Pour le réseaux de neurones, nous avons fais une recherche dans l'espace des hyper-paramètres pour essayer de trouver des paramètres optimiser, qui évite le sur-apprentissage ou le sous-apprentissage. Le fichier en annexe contient les niveaux de précision de chacun des modèles essayé avec la cross-validation à 3 couches.

5 Discussion

Les avantages de notre approche pour la régression logistique sont que nous n'avions pas à y passer trop de temps pour arriver à une solution fonctionnel. Pour améliorer notre approche nous devrions séparer les photos en deux et apprendre les modèles sur les chiffres de 0-9, comme sur le jeu de données MNIST traditionnel. Cela nous permettrait de nous approcher du 99% beaucoup plus facilement, car nous aurions beaucoup moins de classes à apprendre et donc plus d'exemple par classe, ce qui nous aiderait avec la précision des modèles. Nous aurions pu aussi implémenter une méthode de validation, mais les résultats n'auraient pas été réellement supérieur à ceux obtenu. Les inconvénients de notre approche seraient que nous avons moins de données sur le "pourquoi" nous n'avons que seulement 16% de précision. Notre approche est un peu plus comme une boîte noire, ce qui nous empêche de clairement comprendre le résultat, car nous n'avons aucune validation. Pour le MLP, les avantages sont que nous pouvions utilisé n'importe quel librairie pour faire ce modèle, et nous avons utilisé "sklearn" ce qui à simplifier la tâche énormément pour faire la validation et l'optimisation des paramètres. Nous avons trouvé des paramètres qui ont environ 81% de précision, ce qui est très bien, considérant que notre approche ne nettoie pas vraiment les données. L'utilisation de la librarie sklearn nous as permis d'accélérer le processus beaucoup, cependant, elle aussi agit tel une boite noire, ce qui nous donne moins de compréhension sur le choix des paramètres. Malgré cela, la documentation est assez riche sur cette librairie, ce qui nous a aidé à mieux comprendre. Le plus gros désavantage pour la partie du MLP, est que nous n'avions pas fais de pré-traitement sur les données, donc le résultat n'était pas optimale. Les améliorations évidentes au projet serait de faire un pré-traitement sur les données, un autre serait de ne pas utiliser la librairie de sklearn et d'implémenter notre propre version de MLP, pour mieux comprendre le processus et nous permette de l'optimiser selon notre cas spécifique. Nous aurions pu aussi utilisé les librairies de tensorflow pour améliorer notre MLP, car Hanz possède une carte graphique assez puissante, qui aurait pu pousser les recherches de paramètres beaucoup plus loin, sauf que sans pré-traitement, nous aurions été limité aussi, de plus, Al n'en possède pas, donc cela l'aurait limiter pour le développement du MLP.

6 Division des contributions

Le rapport à été rédigé par Hanz, ainsi que la définition du problème et le développement de la méthodologie. Nous avions convenu tout les deux que nous n'aurions pas assez de temps avec tout les autres devoirs concomitant pour viser le top 5 de la compétition. Hanz à donc pensé a comment faire le projet dans les temps, ce qui nous a mené à ne pas faire de pré-traitement sur les données, ainsi que d'utiliser la librairie de sklearn pour le deuxième modèle. Nous avons tout les deux participé au développement du code de la régression logistique, ainsi que le MLP, malgré qu'il fût beaucoup plus bref que le premier. Lors de son analyse des données, Al a trouvé qu'il y avait une colonne de "NaN" qui causait des problèmes et l'a corrigé en effacant tout simplement cette colonne des données. Hanz a fait des recherches web pour trouver des ressources en ligne et des exemples de tâches similaires pour nous aider à faire la régression logistique. Les sites web sont dans la section des références. “Nous déclarons par la présente que tous les travaux présentés dans ce rapport sont ceux des auteurs”.

Références

[BOU20] HAMZA BOULAHIA. Logistic regression mnist classification. `https://dhirajkumarblog.medium.com/logistic-regression-in-python-from-scratch-5b901d72d68e`, 2020.

[K20] Dhiraj K. Logistic regression machine learning algorithm in python from scratch. `https://www.kaggle.com/code/hamzaboulahia/logistic-regression-mnist-classification`, Sep 5, 2020.

[Ver21] Suraj Verma. Logistic regression from scratch in python. `https://towardsdatascience.com/logistic-regression-from-scratch-in-python-ec66603592e2`, Apr 8, 2021.

7 Annexe

Voici tout les paramètres essayés pour le perceptron a multi-couches

```
Best parameters found:
 {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.724 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.734 (+/-0.013) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.720 (+/-0.003) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.734 (+/-0.019) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.727 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.764 (+/-0.005) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.731 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.779 (+/-0.018) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.719 (+/-0.004) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'sgd'}
0.688 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}
0.714 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.683 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.719 (+/-0.015) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.744 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.722 (+/-0.035) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.744 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.738 (+/-0.023) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.778 (+/-0.017) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.737 (+/-0.026) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.768 (+/-0.016) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.720 (+/-0.010) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'sgd'}
0.705 (+/-0.009) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}
0.719 (+/-0.000) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.705 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.732 (+/-0.010) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.774 (+/-0.021) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.734 (+/-0.017) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.759 (+/-0.018) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.736 (+/-0.019) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.789 (+/-0.026) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.737 (+/-0.028) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.795 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.728 (+/-0.007) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'sgd'}
0.722 (+/-0.006) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}
0.726 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.727 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.708 (+/-0.056) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.761 (+/-0.010) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.714 (+/-0.016) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.752 (+/-0.007) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.719 (+/-0.009) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.787 (+/-0.013) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.725 (+/-0.006) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.777 (+/-0.016) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.709 (+/-0.009) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'sgd'}
0.700 (+/-0.008) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}
0.714 (+/-0.005) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.699 (+/-0.005) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.718 (+/-0.018) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.774 (+/-0.071) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.714 (+/-0.013) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.763 (+/-0.022) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.717 (+/-0.008) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.806 (+/-0.023) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.731 (+/-0.010) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.785 (+/-0.022) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.716 (+/-0.007) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'sgd'}
0.696 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}
0.716 (+/-0.009) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.696 (+/-0.022) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.711 (+/-0.001) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.775 (+/-0.020) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.719 (+/-0.014) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.771 (+/-0.020) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.734 (+/-0.015) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'sgd'}
0.801 (+/-0.020) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
0.723 (+/-0.004) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.803 (+/-0.023) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'solver': 'adam'}
0.717 (+/-0.012) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'sgd'}
0.711 (+/-0.013) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}
0.723 (+/-0.000) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'sgd'}
0.711 (+/-0.006) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'solver': 'adam'}
```